www.uniud.it

Corso di dottorato di ricerca in:

Informatica e Intelligenza Artificiale

Ciclo 37°

On the Role of Graphs in Quantum Computing

Dottorando
Riccardo Romanello

Supervisore
Prof.ssa Carla Piazza

Co-supervisore
Prof. Alberto Policriti

Anno 2025

Institute Contacts
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
https://www.dmif.uniud.it/

© 2025 Riccardo Romanello
This work is shared under the Creative Commons 4.0 License Attribution-NonCommercial-ShareAlike.

Abstract

This thesis investigates the role of graphs in quantum computing, exploring whether their utility in classical computation—where they serve as tools for representation, optimization, and problem-solving—extends meaningfully into quantum computational paradigms. Beginning with the foundational importance of graphs in classical computing, this work examines how graphs are used to model, compress, and solve complex computational problems, particularly in contexts that benefit from their semantic flexibility. Building on this, the thesis addresses the question of how graphs can be encoded and manipulated within quantum computing, with a specific focus on the gate-based model of quantum computation.

Encoding graphs for quantum systems presents unique challenges, particularly the need for unitary transformations that respect quantum mechanics' constraints. Through the exploration of graph-based techniques in quantum settings, this thesis develops and analyzes methods for encoding classical graph structures in ways compatible with quantum requirements. Additionally, it examines the application of graphs in quantum-specific tasks, such as quantum random walks, quantum automata, and quantum circuit synthesis, showing that graph-based approaches can aid in circuit optimization and algorithm design.

The findings contribute to understanding whether classical tools like graphs can provide practical and theoretical advantages in quantum computation, offering a bridge between classical and quantum frameworks. This thesis presents several encoding strategies, practical applications, and a comparative analysis, concluding with insights into future directions where graph theory might further intersect with quantum computing advancements.

Contents

1 Preliminaries 9
1.1 Complex numbers and Linear Algebra: A primer . . . 9
1.1.1 Complex Numbers . . . 9
1.1.2 Linear Algebra . . . 10
1.1.3 Linear operators . . . 12
1.2 Graphs . . . 14
1.3 Answer Set Programming—ASP . . . 18
1.3.1 Syntax . . . 18
1.3.2 Semantics . . . 19
1.3.3 Definite Programs . . . 20
1.3.4 General Programs and the Stable Model Semantics . . . 21
1.3.5 On the Capabilities of Logic Programs . . . 22
1.4 Markov Chains . . . 23
1.4.1 Discrete-Time Markov Chains (DTMC) . . . 23
1.4.2 Continuous-Time Markov Chains (CTMC) . . . 24
1.4.3 Random walks on Graphs . . . 25
1.4.4 Steady-State Distribution . . . 27
1.5 Neural Networks . . . 28
1.5.1 Deep Learning . . . 30
1.6 Quantum Computing and Quantum Mechanics . . . 40
1.6.1 Quantum Bits of Information . . . 41
1.6.2 The Postulates of Quantum Mechanics . . . 42
1.7 Quantum Computing Paradigms . . . 46
1.7.1 Gate-Based Quantum Computation . . . 46
1.7.2 Measurement-Based Quantum Computation . . . 47
1.7.3 Adiabatic Quantum Computation . . . 49

I Graphs and Their Representations 53

2 Graphs in Classical Architectures 57
2.1 Adjacency Matrix . . . 57
2.2 Adjacency Lists . . . 63
2.3 Ordered Binary Decision Diagrams . . . 66
2.3.1 The variable ordering problem . . . 68

2.3.2 Finally, Graphs 70

3 Graphs in Parallel Architectures 75
3.1 Coordinate Encoding—COO 76
3.2 Compressed Sparse Row—CSR 77
3.3 Compressed Sparse Column—CSC 79
3.4 Graph Visits on Parallel Architectures 80

4 Graphs in Quantum Architectures 83
4.1 Graphs in Gate-Based model 84
4.1.1 Graphs and unitary matrices 85
4.1.2 An (Eulerian) Graph-to-Unitary Encoding algorithm 93
4.1.3 The Overall Procedure 99
4.1.4 Some Alternatives to Projectors 101
4.1.5 Experimental Results 105
4.1.6 How Truthful Can an Encoding Procedure Be? 108
4.1.7 Quantum walks 109
4.2 Graphs in Measurement-Based Model 116
4.3 Graphs in Adiabatic Model 118

II Graphs as Semantics 123

5 Automata 127
5.1 Definition of Deterministic Finite Automata 127
5.2 A Brief Recall of a DFA Semantics 130
5.3 Some Problems Related to Automata Theory 133

6 Quantum Automata 135
6.1 Measure Once Quantum Automata 136
6.2 Heisenberg Quantum Finite Automata 139
6.2.1 Heisenberg inspired Automata: Using Memory 143

7 Answer Set Programming and Quantum Computation 153
7.1 Grover's Quantum Search Algorithm 154
7.2 Counting Elements in the Search Space 157
7.2.1 Weighted Model Counting 159
7.2.2 A Quantum Algorithm for WMC 159
7.3 Faceted Navigation among Answer Sets 162
7.3.1 Towards a Quantum Faceted Navigation 164

III Graphs as Reduction Structures 169

8 Bisimulations and Lumpabilities 173
8.1 Bisimulation 175
8.2 Bisimulation and Automata Minimization 177
8.2.1 An algorithm for Incremental DFA Minimization 180

8.2.2 An Incremental Algorithm for NFA Minimization 184
8.3 Lumpability 189

9 Reducing Neural Networks 195
9.1 Weight Sharing 195
9.2 Pruning 196
9.3 Lumping 201
9.4 Quantization 209

10 Quantum Circuit Synthesis 213
10.1 T gate optimality 215
10.2 CNOT minimization 216
10.2.1 An ASP approach 222

Introduction

The foundations of Computer Science trace back to the groundbreaking work of Alan Turing, whose contributions to the field in the 1930s and 1940s established a conceptual framework for computation. Turing's abstraction of a computing machine set the stage for understanding not only how machines could execute algorithms, but also the fundamental limitations of what can be computed. His work provided a rigorous model for exploring problems of computability—whether a problem can be solved using an algorithm—and complexity, which seeks to classify problems based on the resources required to solve them. These concepts laid the foundation of countless theoretical results that now impact diverse areas, ranging from optimization and network theory to artificial intelligence and quantum computing. Central to this lineage of theoretical insights is the notion of graph, a deceptively simple structure that underpins a vast range of computational problems and methods.

Graphs serve as representations for entities and their relationships, with applications that span both practical and theoretical computer science. In classical computing, graphs are instrumental in modelling problems, serving as the core data structure for representing everything from electrical circuits and web links to social networks and molecular structures. Beyond data representation, graphs also enable efficient algorithmic analysis; by transforming computational tasks into graph-based problems, one can often leverage on graph-specific algorithms to simplify and solve complex tasks. This flexibility has positioned graphs not only as a versatile data structure, but also as a semantic framework capable of representing the underlying structure and logic of various domains.

The semantic power of graphs is notable in that they enable computers to interpret problems in ways that mirror human intuition about relationships and structures. In areas such as network analysis, for instance, graphs intuitively model connectivity and flow, while in state-based modelling (such as automata theory), graphs allow each node to represent a specific system state and each edge a possible computation step. This capacity to act as a semantic framework enables graphs to serve as a compression tool: they encapsulate complex relationships and dependencies in a compact form, preserving essential structure hiding details. Through such representations, graphs enable efficient algorithms that address classical problems in optimization, pathfinding, and resources allocation. Graphs thus serve not only as a vehicle for data representation but also as an approach for simplifying and structuring information to make it more computationally manageable. As this thesis investigates, these traits make graphs an appealing model in the context of quantum computing, where resources efficiency is essential.

While classical computing has demonstrated the efficacy of graphs across various fields, the advent of quantum computing introduces both new opportunities and challenges for graph-based methods. Quantum computing harnesses principles from quantum mechanics—such as superposition and entanglement—to perform computations in ways that are fundamentally different from classical approaches. These properties allow quantum systems in some cases to process and represent information with an exponentially greater capacity than classical bits, leading to the promise of quantum advantage for specific tasks.

Quantum advantage implies that certain problems could be solved exponentially faster on quantum computers than on classical ones. For example, Shor's algorithm for integer factorization and Grover's algorithm for database search are famous demonstra-

tions of how quantum systems can outperform classical algorithms. Yet, while much attention has been given to algorithms with proven quantum advantage, the question of how classical tools—such as graphs—might be adapted and optimized for quantum computing remains open. This thesis aims to explore the extent to which graphs can play a similar or even enhanced role in quantum computing as they do in classical computing. Key questions include: Are graphs as integral to quantum computing as they are to classical computing? Can graphs serve as efficient representations within the quantum paradigm, particularly in gate-based models where unitary operations are essential? Additionally, how can we encode graphs effectively within quantum frameworks without losing the advantages they offer in classical settings?

The process of encoding graphs in quantum computing settings introduces distinct challenges. For instance, the requirement for unitary transformations (where operations must be reversible and preserve quantum states) imposes constraints that are not present in classical computing. Encoding classical graphs in a quantum-compliant way requires developing representations that respect such quantum constraints while still preserving the information structure of the original graph. Several approaches to graph encoding in quantum settings are emerging. This thesis will particularly focus on the gate-based model of quantum computation, where computations are executed through sequences of quantum gates. The thesis will explore and expand on existing encoding techniques, such as unitary transformations for performing computations over graphs and quantum walks, which illustrate how certain types of graphs can be integrated into quantum computational processes.

The utility of graphs in quantum computation is not limited to representation; they can also be adopted as tools to capture the operational semantics of more complex models/systems. One example are automata, either classical or quantum ones. In the former case, graphs are actually the underlying structure we are dealing with. In fact, classical automata are usually depicted as labelled directed graphs. For what concerns the latter case, Quantum Automata, graphs still play a fundamental role as in the classical settings. Translating some basic notions from graph-theory to the quantum realm, we can approach the study of Quantum Automata expressiveness in an easier way. In the two aforementioned scenarios, it was pretty straightforward to identify the graph that was related to the given framework. This is not always the case. For example, graphs comes handy when dealing with stable models in Answer Set Programming as well. By carefully formalizing the notion of model semantics, we can show how the problem of identifying a stable model can be reduced to a graph visit. Due to this very result, we can speed up the search for ASP stable models thanks to quantum routines.

Last but not least, graphs can also help whenever we deal with minimization problems. A lot of work has been done tackling the problem of *state explosion* when dealing with complex systems. To overcome this issue, plenty of *equivalence relations* between graphs nodes have been introduced to reduce the sizes of the structures we are manipulating. This very idea of state *behavioural equivalence* can be lifted from the graph realm and adopted in many fields. Two examples are classical automata and neural networks. For what concerns the former, the *automata minimization* problem is a fundamental task in automata theory. Plenty of solutions to this problem have been proposed, but the most effective ones do rely on graph theory. In the latter case—neural networks—it is not straightforward to see how graphs can help in size reductions. How-

ever, in this thesis we will describe in depth a pruning technique based on a equivalence relation coming from Markov Chains. It still remains an open problem how to lift these techniques to quantum-related tasks. For example, applying lumpabilities and/or bisimilarities to Quantum Markov Chains models sounds like a prolific research topic. Another significant application lies in quantum circuit synthesis, where graphs provide a framework for optimizing quantum circuits. In quantum circuit design, minimizing gate count—particularly for expensive gates like the CNOT gate—is critical for enhancing quantum algorithm efficiency and reducing error rates. Graphs offer a structured way to represent circuit paths and dependencies, allowing for optimizations that simplify circuits while preserving functionality. By relying on techniques from graph theory and answer set programming (ASP), one can approach circuit minimization and optimization with greater precision, potentially enhancing their computational tractability. Before presenting a more structured overview of the thesis topics, we want to emphasize the connections among the three main parts.

The first part, focusing on graph encoding, examines the behaviour of these data structures in non-classical contexts. In contrast, the part on using graphs to capture model semantics frames this as a search problem: we start with a given problem, translate its instances into graphs, and then apply search routines to solve it on these newly constructed structures. Finally, the third part is primarily concerned with optimization. This includes tasks like automata minimization, which is inherently an optimization challenge, and neural network pruning, where the goal is to optimally remove nodes to create smaller models. Additionally, the ASP approach to circuit synthesis can also be viewed as a form of optimization by design.

The structure of this thesis is as follows. First, all the required Preliminaries are introduced in Chapter 1. We will describe all the theoretical backgrounds required for the understanding of this thesis. Afterwards, the very core of this manuscript begins. The backbone is composed of three *parts*. Part I deals with the description of techniques and tools for the encoding of graphs in different computational settings. In particular, such part contains three different chapters. Chapter 2 introduces the reader to all the data structures adopted in classical computation to store and manipulate graphs. The very same kind of notions are given in Chapter 3, with the computation environment being set to parallel environments. Last but not least, a deep and exhaustive description of graph encoding techniques in Quantum Computation is the goal of Chapter 4.

The final chapter expands upon a paper I co-authored, titled *Quantum Encoding of Dynamic Directed Graphs*. This work addresses the problem of encoding graphs within a gate-based quantum computing framework, presenting a linear-time procedure for encoding any directed graph into a unitary matrix. Unlike previous approaches, such as [2], which are limited to specific subclasses of graphs—namely, regular graphs—our method enables the encoding of arbitrary directed graphs.

Subsequently, we shift our attention towards the adoption of graphs as tools to capture the semantics of various theoretical models. All the results are provided in Part II. Mimicking what we did for the first part, this one is split in chapters as well. Chapter 5 is devoted to the introduction of basic notions about classical automata. We start from this model since it is between the simpler ones to capture with graphs. The quantum counterpart of automata, namely Quantum Finite Automata (QFA), induces a semantics that can be grasped by means of graphs too. Chapter 6 is devoted to the

introduction of QFA, to the investigation of their semantics, and to the description of one family of automata that rely upon the Heisenberg picture of quantum mechanics.

The class of automata under consideration was introduced in [144], a work I co-authored. This proposal constitutes the core content of Chapter 6. In this chapter, we introduce the concept of the Heisenberg Quantum Automaton and analyze its fundamental properties. Our findings indicate that its expressive power does not exceed that of the Measure-Once Quantum Finite Automaton (QFA). Subsequently, we turn our focus to multi-letter quantum automata. Building upon the definition presented in [148], we extend the theoretical framework of the field by examining their unlimited memory variant. Our results demonstrate that this newly introduced class does not enhance the expressiveness beyond what is already established in the literature [31].

The very last component of this second part is Chapter 7. While automata semantics, either classical or quantum, can be directly encoded into graphs, the same does not hold for the behaviour of ASP frameworks. The stable model semantics, that lie at the core of ASP, seems to be completely unrelated to graphs. The aim of Chapter 7 is to create a relation between stable models and graphs. This link will be enriched by adding a quantum component to the equation.

The aforementioned component is represented by a work I co-authored, which explores the relationship between Answer Set Programming (ASP) and quantum computation [154]. In this study, we propose the use of a *Grover-like* algorithm to accelerate the search for stable models. Specifically, we begin by employing a graph traversal algorithm to identify stable models, utilizing a routing function to navigate the graph. It was proven in [67] that the optimal routing function is classically computationally intractable. To address this challenge, we introduced a quantum algorithm that enables the efficient computation of this function.

Finally, Part III explores the usefulness of graphs as reductions tools. In particular, we will lift the notion of bisimulation and lumpability to different topics. First, we start with Chapter 8. Its aim is twofold. On the one hand, it serves as an introduction to the notion of *behavioural equivalence* when dealing with graph nodes. On the other hand, it is devoted to the in-depth description of a classical automata minimization algorithm which relies on graph colouring as main building block.

I co-authored the paper *Incremental NFA Minimization*, in which this procedure was introduced. Abstracting the problem in terms of graphs played a crucial role in developing efficient algorithms. The distinctive feature of our approach is that the minimization process is performed incrementally, allowing it to be halted at any point while ensuring that the resulting automaton remains language-equivalent to the original one. Notably, algorithms such as Hopcroft's, which address the same problem, do not possess this property. Regarding the computational complexity of the algorithm, in the case of deterministic finite automata (DFA), our approach achieves a complexity that matches the best-known incremental algorithm for deterministic automata minimization. In contrast, for nondeterministic finite automata (NFA), our method improves upon the state-of-the-art complexity.

A closely related concept is exploited in the content of Chapter 9. It starts with the introduction of lumpability relation—roughly, a weighted version of the bisimulation. Subsequently, this very concept is exploited to devise a Neural Network (NN) pruning algorithm based on formal methods.

The procedure for NN reduction via lumpability was introduced in a paper I co-authored. Given the *weighted-graph* structure of neural networks, it is natural to explore whether techniques from performance evaluation—specifically, lumpability—can be employed to reduce network size. However, lumpability cannot be directly applied, as it does not account for the flow of data within the network, a factor absent in traditional formalisms such as Markov Chains. Our findings demonstrate that, since NN weights can be expressed as linear combinations, pruning is possible. Nevertheless, we were able to formally establish this result only for networks using the ReLU activation function.

The closure of this final part is given by Chapter 10. In this case, we are not aiming at reducing any given structure. In this chapter we present an ASP model to minimize the number of CNOT gate in a quantum circuit. The relation to graphs is given by the nature of the model we propose: the minimal length CNOT circuit is depicted as an acyclic directed graph.

This method was introduced in *An ASP Approach for the Synthesis of CNOT Minimal Quantum Circuits*, a paper I co-authored. While the proposed approach does not scale beyond circuits with more than 10 qubits, determining the optimal—i.e., minimal—number of CNOT gates required for synthesizing a linear reversible circuit remains a challenging problem. Therefore, my model should not be regarded as a solution to the CNOT minimization problem. Instead, its purpose is to provide lower bounds against which heuristic algorithms addressing the same problem can be evaluated.

For the sake of comprehension, we introduced a conclusion chapter for each part and a conclusion section for each chapter. Moreover, at the beginning of each chapter, we will clarify its purpose and provide the reader with some insights, or *spoilers*, about what will follow. In this way we hope to make the process of reading this thesis easier.

The questions raised in this thesis aim to bridge the gap between classical and quantum computing, assessing whether classical tools like graphs can maintain their significance in the quantum era. As quantum computing continues to evolve, understanding the role of traditional structures within new paradigms could provide valuable insights not only for theoretical computer science but for practical applications across diverse computational domains. This thesis thus seeks to provide a foundation for future work that uses graph structures as a means of advancing quantum computational models and techniques.

1

Preliminaries

The main and unique goal of this chapter is to introduce all the mathematical preliminaries required throughout this thesis. We want to stress the fact that we will not describe all the notions related to the topics we will discuss, but just the theoretical minimum[1] to be consistent.

Logically speaking, this chapter is divided between notions related to classical computer science and the ones related to quantum setting.

For what concerns the classical part, we start by delving into the basic notions of linear algebra, with a little intermezzo on complex numbers. We then move to the description of the *undisputed* main character of this thesis: graphs. All the required theoretical background will be introduced during the preliminaries. Notions about Answer Set Programming, Markov Chains, and Neural Networks serve as a closure for the classical part of the preliminaries.

On the other hand, we begin the investigation of the quantum settings by introducing the basics postulates of quantum mechanics. We will use them to introduce and explain three different quantum computing paradigms, which will close the preliminaries.

1.1 Complex numbers and Linear Algebra: A primer

1.1.1 Complex Numbers

A foundational idea that Section 1.6 will formalize is that a *quantum computation* sort of reduces to a sequence of manipulations over *vectors* in a *complex-valued vector space*. Let us first recall that a complex number $z \in \mathbb{C}$ is of the form:

$$z = a + ib, \tag{1.1}$$

where $a, b \in \mathbb{R}$ are, respectively, the *real* and *imaginary part* of z; $i \in \mathbb{C}$ is the *imaginary unit* where $i^2 = -1$. The complex number $z^* \in \mathbb{C}$ denotes the *complex conjugate* of z,

[1] Thanks to Leonard Susskind for the name

that is, z with opposite imaginary part:

$$z^* = a - ib. \tag{1.2}$$

The *modulo* of z is denoted as:

$$|z| = \sqrt{a^2 + b^2} = \sqrt{zz^*}. \tag{1.3}$$

Apart from the one given in Equation (1.1), complex numbers may be given two alternative representations. The number $z \in \mathbb{C}$ described in *polar form* is:

$$z = r(\cos\theta + i\sin\theta), \tag{1.4}$$

where $r = |z| \in \mathbb{R}^+$, $a = r(\cos\theta)$ and $b = r(\sin\theta)$ for $\theta \in [0, 2\pi)$.

Recalling Euler's formula $e^{i\theta} = \cos\theta + i\sin\theta$, the *exponential form* of z is easily derived:

$$z = re^{i\theta}. \tag{1.5}$$

1.1.2 Linear Algebra

A vector $\boldsymbol{v}$—conventionally denoted in *bold notation*—in a complex-valued vector space $\mathbb{C}^n$ is of the form:

$$\boldsymbol{v} = \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{pmatrix}, \tag{1.6}$$

where $\alpha_i \in \mathbb{C}$ for $1 \leq i \leq n$. Unless otherwise specified, a vector should be understood to be in column form. The *adjoint* $\boldsymbol{v}^\dagger$ of $\boldsymbol{v}$ is the transposed $\boldsymbol{v}^T$ of $\boldsymbol{v}$, where the coordinates are replaced by their complex conjugates:

$$\boldsymbol{v}^\dagger = (\boldsymbol{v}^T)^* = \begin{pmatrix} \alpha_1^* & \alpha_2^* & \cdots & \alpha_n^* \end{pmatrix}. \tag{1.7}$$

The results presented in this thesis only concern *finite,* complex-valued vector spaces equipped with an *inner product* $(\cdot, \cdot)$.

Definition 1.1. Let V be a vector space over $\mathbb{C}$. An *inner product* over V is any function $(\cdot, \cdot) : V \times V \to \mathbb{C}$ such that, for any $\boldsymbol{x}, \boldsymbol{y} \in V$, the following properties are satisfied:

- *Commutativity.* $(\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{y}, \boldsymbol{x})^*$;
- *First argument linearity.* $(\lambda\boldsymbol{x}, \boldsymbol{y}) = \lambda(\boldsymbol{x}, \boldsymbol{y})$;
- *Positive definiteness.* $(\boldsymbol{x}, \boldsymbol{x}) \geq 0$, with $(\boldsymbol{x}, \boldsymbol{x}) = 0$ if and only if $\boldsymbol{x} = \boldsymbol{0}$.

Because their definitions coincide in the finite case, a vector space satisfying these conditions shall be referred to as *Hilbert space.* Moreover, the equipped inner product

is to be understood as the *dot product.* That is, given $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{C}^n$:

$$(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^\dagger \boldsymbol{y} = \sum_{i=1}^{n} x_i^* y_i. \tag{1.8}$$

The dot product, as well as any other inner product, induces a *norm.* Let $\mathbb{C}^n$ be a Hilbert space, then, the induced norm is the function $\|\cdot\| : \mathbb{C}^n \to \mathbb{R}$ such that, given $\boldsymbol{v} \in \mathbb{C}^n$:

$$\|\boldsymbol{v}\| = \sqrt{(\boldsymbol{v}, \boldsymbol{v})} = \sqrt{\sum_{i=1}^{n} |\alpha_i|^2}. \tag{1.9}$$

where α_i is $\boldsymbol{v}$'s i-th component.

The vector $\boldsymbol{v}$ is said to be a *unit vector* if $\|\boldsymbol{v}\| = 1$.

Dirac notation

For the results yet to be introduced, bold notation shall be abandoned in favour of the more congenial *Dirac notation.* This way, a column vector $\boldsymbol{v} \in \mathbb{C}^n$ is equivalently described by a so-called *ket*: $|v\rangle$. On the other hand, a *bra* allows to describe the adjoint $\boldsymbol{v}^\dagger$ as $\langle v|$.

In turn, the inner product between two vectors $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{C}^n$ is written as:

$$(\boldsymbol{v}, \boldsymbol{w}) = \langle v|w\rangle. \tag{1.10}$$

Basis

A vector space V is fully described by a *basis.* Consider a set $B = \{|v_i\rangle\}_{i=1}^n$ such that $|v_i\rangle \in V$ for $1 \leq i \leq n$. Then, B is said to be a basis of V if it is a minimal set such that vectors of the form:

$$|v\rangle = \sum_{i=1}^{n} \alpha_i |v_i\rangle, \tag{1.11}$$

are *all and only* the vectors laying in V, with $\alpha_i \in \mathbb{C}$ for $1 \leq i \leq n$. Moreover, $\mathsf{dim}(V) = |B|$ is said to be the *dimensionality* of V.

A basis is said to be *orthogonal* if the inner product between any two distinct elements of the basis is zero.

An orthogonal basis is said to be *orthonormal* if its elements are unit vectors. More formally, for an orthonormal basis $B = \{|v_i\rangle\}_{i=1}^n$ it holds that, for any $1 \leq i, j \leq n$:

$$\langle v_i|v_j\rangle = \delta_{i,j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases} \tag{1.12}$$

where $\delta_{i,j}$ is the *Kronecker delta.*

Given an n-dimensional vector space V, we denote the *canonical basis* as:

$$\{|0\rangle, |1\rangle, \ldots, |n-1\rangle\}, \tag{1.13}$$

where, for any $0 \leq i < n$, $|i\rangle$ has a single non-zero entry at the $(i+1)$-th coordinate equal to 1. When no ambiguity concerns the dimensionality of the vector space, the canonical basis shall be denoted simply as $\{|i\rangle\}$.

1.1.3 Linear operators

Let V, W be two vector spaces over $\mathbb{C}$. A *linear operator* is a map $A : V \to W$ which is linear with respect to vectors in V, i.e., :

$$A\Big(\sum_i \alpha_i |v_i\rangle\Big) = \sum_i \alpha_i A |v_i\rangle . \tag{1.14}$$

where $A|v_i\rangle$ denotes the application of A onto vector $|v_i\rangle$.

Matrix representation

Given $n = \mathsf{dim}(V)$ and $m = \mathsf{dim}(W)$, the linear operator A may be given a matrix representation $A \in \mathbb{C}^{m\times n}$:

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} & \cdots & A_{1,n} \\ A_{2,1} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ A_{m,1} & \cdots & \cdots & A_{m,n} \end{pmatrix}, \tag{1.15}$$

where $A_{i,j} \in \mathbb{C}$ for all $1 \leq i \leq m$, $1 \leq j \leq n$. The writings A_i and $A_{,j}$ denote, respectively, the i-th row and j-th column of matrix A. The matrix representation of A varies according to the input and output bases. To this end, it is assumed that, whenever $V = W$, input and output bases coincide. Analogously to the case of vectors, one defines the adjoint $A^\dagger$ of A to be the $n \times m$ matrix

$$A^\dagger = \begin{pmatrix} A^*_{1,1} & A^*_{2,1} & \cdots & A^*_{m,1} \\ A^*_{1,2} & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ A^*_{1,n} & \cdots & \cdots & A^*_{m,n} \end{pmatrix}. \tag{1.16}$$

Outer product representation

Linear operators may also be given an *outer product* representation. This description especially shines when adopting Dirac notation. To see why this is the case, consider the linear operator $A : V \to W$:

$$A = \sum_i |w_i\rangle \langle v_i| , \tag{1.17}$$

where $w_i \in W$ and $v_i \in V$ for all i. Applying A to any $|v'\rangle \in V$ lays out the conveniently expressed the result:

$$|w'\rangle = \sum_i \langle v_i|v'\rangle \, |w_i\rangle \, . \tag{1.18}$$

Definition 1.2 (Identity operator)**.** Let V be a vector space. The action of the *identity operator* $\mathbb{I}_V : V \to V$ is defined as $\mathbb{I}_V \, |v\rangle \equiv |v\rangle$ for any $|v\rangle \in V$. Given an orthonormal basis $\{|i\rangle\}_i$ of V,

$$\mathbb{I}_V = \sum_i |i\rangle \, \langle i| \, . \tag{1.19}$$

$\mathbb{I}_V$ is denoted as either $\mathbb{I}$ or Id whenever no ambiguity concerns the considered vector space.

For the scope of this thesis, an important category of linear operators is that of *normal operators.*

Definition 1.3 (Normal operator)**.** A linear operator A is said to be *normal* if and only if $A^\dagger A = AA^\dagger$.

A relevant subclass of normal operators is that of *unitary operators.*

Definition 1.4. Unitary operators A linear operator U is said to be *unitary* if and only if $U^\dagger U = UU^\dagger = \mathbb{I}$.

A unitary operator $U : V \to V$ preserves:

1. **Angles.** $(U\,|v\rangle\,, U\,|w\rangle) = \langle v|U^\dagger U|w\rangle = \langle v|\mathbb{I}|w\rangle = \langle v|w\rangle$.
2. **Norm.** $\|U\,|v\rangle\,\| = \sqrt{\langle vU^\dagger U|v\rangle} = \sqrt{\langle v|v\rangle} = \|\,|v\rangle\,\|$.

Unitary operators on $\mathbb{C}^n$ are all and only the operators represented by matrices where both columns and rows form an orthonormal basis in $\mathbb{C}^n$.

Hermitian operators define another category of normal operators that intersects that of unitary operators.

Definition 1.5 (Hermitian operators)**.** A linear operator H is said to be *Hermitian* if and only if $H = H^\dagger$.

Intuitively, *hermiticity* may be understood as the complex-valued analogue of *symmetry.*

Definition 1.6 (Direct sum of matrices)**.** Let $M \in \mathbb{C}^{n\times m}$ and $M' \in \mathbb{C}^{n'\times m'}$ be two matrices. The *direct sum* of M and M' is the $(n+n') \times (m+m')$ matrix

$$M \oplus M' = \begin{pmatrix} M & \mathbf{0} \\ \mathbf{0} & M' \end{pmatrix},$$

where $\mathbf{0}$ denotes two matrices with only zero entries.

Tensor product

Tensor product on Hilbert spaces directly builds upon the definition of tensor product on vectors. Let $|v\rangle \in V, |w\rangle \in W$ be two vectors. Their tensor product—*Kronecker product*—$|v\rangle \otimes |w\rangle$ is defined as follows:

$$|v\rangle \otimes |w\rangle = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \otimes \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{pmatrix} = \begin{pmatrix} v_1 |w\rangle \\ v_2 |w\rangle \\ \vdots \\ v_n |w\rangle \end{pmatrix}. \tag{1.20}$$

Often, we will abbreviate $|v\rangle \otimes |w\rangle$ as simply $|v\rangle |w\rangle$ or even $|vw\rangle$. We can now define the tensor product between two spaces. Let V and W be two Hilbert spaces of dimension n and m, respectively. The tensor product of V and W is the nm-dimensional vector space $V \otimes W$. Suppose $\{|v_i\rangle\}_{i=1}^{n}$ and $\{|w_j\rangle\}_{j=1}^{m}$ to be, respectively, orthonormal bases for V and W, then the set:

$$\{|v_i\rangle \otimes |w_j\rangle : 1 \leq i \leq n, 1 \leq j \leq m\}, \tag{1.21}$$

is an orthonormal basis for $V \otimes W$. We conclude our description of tensor product by providing its behaviour on linear operators. Let A, B be two linear operators acting on V and W, respectively. Their tensor product $A \otimes B$ is a linear operator in $V \otimes W$, and it is defined as:

$$A \otimes B = \begin{pmatrix} A_{1,1}B & A_{1,2}B & \cdots & A_{1,n}B \\ A_{2,1}B & A_{2,2}B & \cdots & A_{2,n}B \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,1}B & A_{n,2}B & \cdots & A_{n,n}B \end{pmatrix}$$

where $A_{i,j}$ denotes the entry of A in position i, j.

1.2 Graphs

Graphs are fundamental data structures that encode local, binary relations between entities in a given set.

The very first groundbreaking result in graph theory was made by Euler in [65]. He tackled the *Seven Bridges of Konisberg* problem. Konisberg presents two big islands and two main areas. These four spots are connected by seven bridges—see Figure 1.1.

A debate remained unsolved during centuries: was there a way of having a walk in the city, traversing all the bridges exactly once? Euler solved this problem by mapping it to graph theory. First, he removed all the useless information about the city and only kept the *reachability* information—see Figure 1.2.

After doing so, he abstracted another layer of the problem obtaining what we would call today an *undirected* graph, depicted in Figure 1.3. In such graph, the nodes represents either islands or main areas. On the other hand, edges describe bridges. Using this new structure he was able to introduce the notion of Eulerian path—and Eulerian graph as a consequence—which turned out to be the key idea to solve the Konisberg bridges problem.

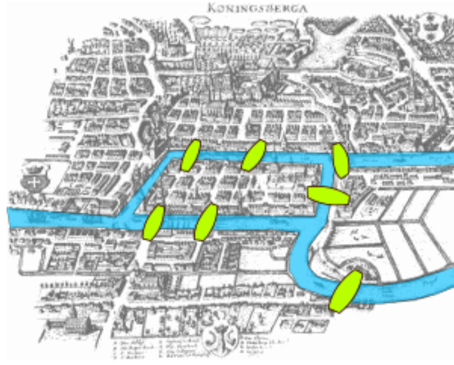


Figure 1.1: Konisberg landscape. Picture taken from Wikipedia.

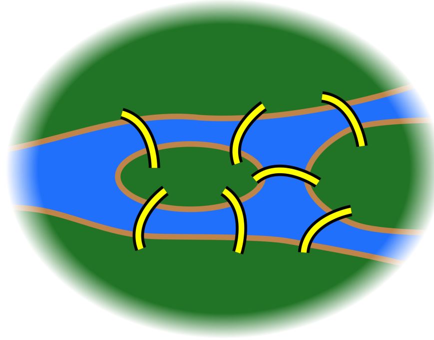


Figure 1.2: Konisberg with only its bridges. Picture taken from Wikipedia.

A primary distinction in graph theory lies between undirected and directed graphs. In an undirected graph, edges can be traversed in either direction, much like crossing a bridge that allows travel both ways, as Euler might have done. In contrast, directed graphs restrict the direction of traversal along edges. A real-world example is a store with separate entrance and exit doors. A store with one door that serves as both entry and exit resembles an undirected graph, while a store with distinct doors for entering and exiting aligns with a directed graph: the entrance is only for entering, and the exit only for leaving. As this thesis mainly focuses on directed graphs, we will now turn our attention to their theoretical framework.

Formally speaking, a directed graph is a pair $G = (V(G), E(G))$ where $V(G)$ is the set of vertices and $E(G) \subseteq V(G) \times V(G)$ is the set of directed edges. When no confusion arises, $V(G), E(G)$ are simply referred to as V, E. Given two vertices $u, v \in V$, there exists a directed edge from u to v if and only if $(u, v) \in E$. Given the directed edge (u, v), u and v denote, respectively, the *source* and *target* of (u, v). Two directed edges $(u, v), (w, z)$ are said to be *consecutive* if and only if $v = w$. G is said to be *undirected* when $(u, v) \in E$ if and only if $(v, u) \in E$. Because this manuscript mostly deals with *directed* graphs, these shall be referred to simply as graphs. The same holds for *"directed*

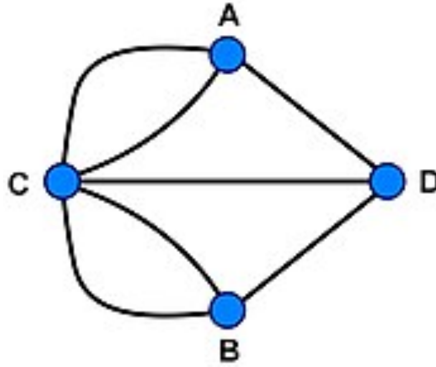


Figure 1.3: Konisberg looking at itself through the graph theory lens. Picture taken from Wikipedia.

edges" and *"edges"*. All the theoretical and algorithmic notions related to encoding graphs will be deeply investigated in the further chapters. However, for the sake of completeness, we briefly anticipate here the notion of *adjacency matrix*. Let $G = (V, E)$ be a directed graph, with $|V| = n$. Its adjacency matrix $M(G) \in \{0, 1\}^{n \times n}$, is defined as follows:

$$M(G)_{u,v} = \begin{cases} 1 & \text{if } (u, v) \in E; \\ 0 & \text{otherwise.} \end{cases} \tag{1.22}$$

Given a vertex $v \in V$, the *out-neighbourhood* of v, $\delta^+(v)$ is defined as the set of all vertices connected via a directed edge *from* v. Analogously, the *in-neighbourhood* $\delta^-(v)$ is the set of all vertices connected via a directed edge *to* v. More formally,

$$\delta^+(v) = \{v' : v' \in V, (v, v') \in E\}; \qquad \delta^-(v) = \{v' : v' \in V, (v', v) \in E\}. \tag{1.23}$$

In turn, the *out-degree* and the *in-degree* of v are defined, respectively, as $d^+(v) = |\delta^+(v)|$ and $d^-(v) = |\delta^-(v)|$. For what concerns graph representation, up to now we only need the adjacency matrices description. Given a set of nodes V, we assume a fixed order over the elements of V, i.e., each node of V can be identified as an integer between 1 and $|V|$. For a graph $G = (V, E)$ with $|V| = n$, the *adjacency matrix* of G is a $(0-1)$-matrix $M(G)$ of size $n \times n$ such that $M(G)_{u,v} = 1$ if and only if $(u, v) \in E$—when G is clear from the context, we will refer to $M(G)$ just with M. We will completely describe different ways of encoding a graph in different computational paradigms in the future chapters.

Part of the following discussion shall require a more general data structure: the *multigraph*. A multigraph is a graph where a source-target pair (u, v) may be connected by more than one edge. Formally, a multigraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a graph where $\mathcal{E}$ is a multiset. Let $\{(u, v)_1, (u, v)_2, \ldots, (u, v)_k\} \subseteq \mathcal{E}$ be k edges sharing source-target pair (u, v). Then, $\mathsf{m}((u, v)) = k$ is said to be the *multiplicity* of edge (u, v). For $(u, v)_x \in \mathcal{E}$, the subscript x allows to unambiguously refer to a specific edge. The entries of the adjacency matrix M of a multigraph $\mathcal{G}$ are, for any $e = (u, v) \in \mathcal{E}$, $M_{u,v} = \mathsf{m}(e)$.

Finally, let $\mathcal{E}, \mathcal{E}'$ be two multisets where m, m' denote the multiplicities in $\mathcal{E}$ and $\mathcal{E}'$, respectively. The *multiset sum* between $\mathcal{E}, \mathcal{E}'$ is the multiset $\mathcal{E} \uplus \mathcal{E}'$ with multiplicity m^+, such that, for any $e = (u, v) \in \mathcal{E} \cup \mathcal{E}'$, $\mathsf{m}^+(e) = \mathsf{m}(e) + \mathsf{m}'(e)$.

Let G be a graph or a multigraph, there need to be as many incoming edges as there are outgoing edges.

Theorem 1. *Let $G = (V, E)$ be a (multi)graph. Then,*

$$\sum_{v \in V} d^+(v) - d^-(v) = 0. \tag{1.24}$$

Given a graph $G = (V, E)$, a *subgraph* $G' = (V', E')$ of G is such that $V' \subseteq V$, $E' \subseteq E$ and, for any $(u, v) \in E'$, $u, v \in V'$. Finally, a *directed n-path* is a graph $P_n = (V, E)$ where $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{(v_i, v_{i+1}) : v_i, v_{i+1} \in V\}$. In such case, we say P_n is a directed path from v_0 to v_n. A *directed n-cycle* is a directed n-path where $(v_n, v_1) \in E$.

We hereby review the most generally known properties related to graphs. In the next chapters, we will also introduce more peculiar ones that are not general but *topic* related—for example, useful for quantum random walks.

Definition 1.7. A graph $G = (V, E)$ is said to be *connected* if and only if, for any $(u, v) \in E$, there exists a directed path either from u to v or vice-versa.

G is said to be *strongly connected* if and only if, for any $(u, v) \in E$ there exists a directed path from u to v.

Definition 1.8. Let $G = (V, E)$ be a graph. A subgraph $H = (V', E')$ of G is said to be a (strongly) connected component of G if H is (strongly) connected.

Definition 1.9. Given a graph $G = (V, E)$, the subgraph $G' = (V', E')$ is said to be a *spanning subgraph* of G if and only if $V' = V$.

Definition 1.10. A graph $G = (V, E)$ is said to be *Hamiltonian* if and only if there exists a spanning graph G' that is a directed cycle.

Definition 1.11. Given $G = (V, E)$, a *tour* is a sequence of consecutive edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$, where each edge occurs exactly once. Then, a graph is said to be *Eulerian*[2] if and only if it contains a tour that traverses all edges in G such that $v_0 = v_n$.

Theorem 2. *A graph $G = (V, E)$ is Eulerian if and only if, for any $v \in V$, $d^+(v) - d^-(v) = 0$.*

Definition 1.12. Given $k \in \mathbb{N}$, a *k-regular directed graph* $G = (V, E)$ is such that, for any $v \in V$,

$$d^+(v) = d^-(v) = k. \tag{1.25}$$

Definition 1.13. Given $n \in \mathbb{N}_{>0}$, the *n-complete graph* is the graph $K_n = (V, E)$, where $|V| = n$ and $E = \{(u, v) : u, v \in V, u \neq v\}$.

The *n-complete graph with self-loops* is the graph $K_n^+ = (V, E)$ where $|V| = n$ and $E = V \times V$.

[2] The name directly comes from the solution to the Konisberg bridges problem

Part of the results in this manuscript shall also require the fundamental notion of *isomorphism.*

Definition 1.14. Let $G = (V, E)$ and $G' = (V', E')$ be two graphs. Then G and G' are said to be *isomorphic* if and only if there exists a bijection $\phi : V \to V'$ such that, for any $u, v \in V$,

$$(u, v) \in E \iff (\phi(u), \phi(v)) \in E'.$$

When G, G' are isomorphic we write $G \cong G'$.

Tools are set to introduce the first - and, arguably, simplest - form of graph encoding: the *line graph.*

Definition 1.15. Given a graph $G = (V, E)$, the respective line graph is $\overrightarrow{G} = (\overrightarrow{V}, \overrightarrow{E})$ where:

- $\overrightarrow{V} = E$. The vertices in $\overrightarrow{G}$ represent the edges from G.
- $\overrightarrow{E} = \{\big((u, v), (v, w)\big) : (u, v), (v, w) \in E\}$. Two vertices are adjacent in $\overrightarrow{G}$ if and only if they represent consecutive edges in G.

1.3 Answer Set Programming—ASP

Logic Programming is a declarative programming paradigm whose foundations stemmed from formal logic. In particular, logic programs are used in the fields of *knowledge representation* and *automated reasoning.*

This section will introduce the bare minimum needed to understand the main topics of this thesis, but without any pretence of being a complete presentation of the topic. The reader may refer to [72] for further details.

1.3.1 Syntax

Logic programs are built on top of 5 main ingredients:

- a finite set C of *constants*;
- a finite set V of *variables*;
- a finite set F of *functions*;
- a finite set P of *predicates*;
- a function $\text{ar} : F \cup P \mapsto \mathbb{N}^+$, known as the *arity* of both functions and predicates.

By combining elements from the sets C, V, and F, *terms* can be built:

- each constant $c \in C$ and each variable $v \in V$ is a term;
- if $t_1, \ldots, t_n$ are terms and $f \in F$ is a function such that $\text{ar}(f) = n$, then $f(t_1, \ldots, t_n)$ is also a term.

A term with no variables is said to be *ground*.

An *atomic formula* – or, for short, an *atom* – is an object of the form:

$$p(t_1, \ldots, t_n) \tag{1.26}$$

where $t_1, \ldots, t_n$ are terms and $p \in P$ is a predicate such that $\text{ar}(p) = n$. An atom is said to be *ground* if it is built using only ground terms. Moreover, we call *literal* an atom or the negation of an atom.

A *rule* ρ is a syntactic object that can be expressed as follows:

$$\alpha \leftarrow \beta_1, \ldots, \beta_n, \neg\gamma_1, \ldots, \neg\gamma_m \tag{1.27}$$

where α, β_i, and γ_j are all atoms. We usually refer to α as the *head* of the rule, and to $\beta_1, \ldots, \beta_n, \neg\gamma_1, \ldots, \neg\gamma_m$ as the *body*.

Some special kind of rules are allowed:

- a *fact* is a rule whose body is empty (i.e. $n = m = 0$);
- a *denial* is a rule without a head;
- a rule whose body does not contain any negated atom (i.e. $m = 0$) is called a *definite clause*[3].

A simultaneous substitution of every variable occurring in a rule ρ with a ground term produces a *ground instance* of ρ. Notice that each rule may have several ground instances.

A *logical program* Π is a finite set of rules. In analogy to rules, a program is said to be *definite* if it contains only definite clauses. The *ground instance* of a program Π is the program defined as the union – for every rule $\rho \in \Pi$ – of all the possible ground instances of ρ.

1.3.2 Semantics

The semantics of a logic program Π is defined with respect to a given set $\mathcal{U}$ – called *universe* – of objects. An *interpretation* is an assignment that binds:

- constants $c \in C$ to objects $\tilde{c} \in \mathcal{U}$;
- function symbols $f \in F$ such that $\text{ar}(f) = n$ to n-ary functions $\tilde{f} : \mathcal{U}^n \mapsto \mathcal{U}$;
- predicate symbols $p \in P$ such that $\text{ar}(p) = n$ to n-ary relations $\tilde{p} \in \mathcal{U}^n \times \{\text{T}, \text{F}\}$.

Moreover, an interpretation I for Π is also a *model* if it satisfies the logical meaning of all the rules $\rho \in \Pi$. An atom α is said to be a *logical consequence* of a program Π if it belongs to every model of Π, and we write it as follows:

$$\Pi \vDash \alpha \tag{1.28}$$

Thanks to Theorem 3, we can however limit ourselves to *Herbrand* universes and interpretations. In particular, fixed a program Π:

[3] Definite clauses are also known as *Horn clauses* in the literature.

- its *Herbrand universe* $\mathcal{U}_\Pi$ is the set of all the ground instances of the terms that occur in Π;
- a *Herbrand interpretation* is an interpretation for Π in $\mathcal{U}_\Pi$. Herbrand interpretations that are also models of Π are called *Herbrand models.*

Theorem 3 (Herbrand Fundamental Theorem)**.** *Let T be a conjunction of clauses. Then T has a model if and only if it has a Herbrand model.*

For convenience, we can also define the *Herbrand base* B_Π of a program Π:

$$B_\Pi \coloneqq \{\alpha : \alpha \text{ is a ground atom}\} \tag{1.29}$$

Observe that any $I \subseteq B_\Pi$ is a Herbrand interpretation for Π.

Remark 1.1. Observe that $\langle 2^{B_\Pi}, \subseteq \rangle$ is a *complete lattice*, i.e., a partially ordered set such that for any pair of its elements both their supremum and infimum are also elements of the lattice.

1.3.3 Definite Programs

The goal of Logical Programming is to be able to compute logical consequences of programs that encode knowledge about some domain. However, it is not always so easy to solve such task.

If we restrict ourselves to definite programs, the following lemma provides the foundations for being able to compute logical consequences:

Lemma 1.1. *Let Π be a definite program, and let I_1 and I_2 be two Herbrand models of Π. Then $I_1 \cap I_2$ is also a model of Π.*

Proof. Let $\alpha \leftarrow \beta_1, \ldots, \beta_n$ be a ground instance of a definite clause $\rho \in \Pi$, and assume that $\{\beta_1, \ldots, \beta_n\} \subseteq I_1 \cap I_2$. Then, $\{\beta_1, \ldots, \beta_n\} \in I_1$ and $\{\beta_1, \ldots, \beta_n\} \in I_2$. Since both I_1 and I_2 are models of Π, it holds that $\alpha \in I_1$ and $\alpha \in I_2$. Thus, $\alpha \in I_1 \cap I_2$. □

Corollary 1.1. *If Π is a definite program, then it has a minimum Herbrand model M_Π that is the intersection of all its models.*

Remark 1.2. Observe that M_Π corresponds to the set of all the logical consequences of the program Π.

In general, there are two different ways in which the minimum Herbrand model of a program can be computed:

- *top-down* methods – such as the *SLD resolution* algorithm implemented in *Prolog* [72, Chapter 12] – usually start from a *goal* clause and try to compute some sort of "derivation" for it by repeatedly applying substitutions;
- on the other hand, *bottom-up* techniques exploit the *immediate consequence operator* $T_\Pi : 2^{B_\Pi} \mapsto 2^{B_\Pi}$, iterating it until fixpoint.

Remark 1.3. Both methods may result in infinite computations under certain circumstances: top-down methods can loop, while bottom-up techniques produce an infinite computation if the universe is infinite.

1.3.4 General Programs and the Stable Model Semantics

Unfortunately, Lemma 1.1 does not hold in the case of *general* programs, i.e. programs that have rules that are not definite clauses. Intuitively, the root cause is that allowing negations in the rules' bodies introduces *non-monotonicity* in the reasoning process. In turn, this weird behaviour is a consequence of the fact that the common-sense meaning of non-definite clauses differs from their logical interpretation. To fix this issue, the *completion* of a program can be computed by replacing implications with double implications[4], in order to strengthen its logical meaning. As a consequence, B_Π can be partitioned in three subsets, each containing:

- atoms that belong to every model of the completion (I^+_Π);
- atoms that do not belong to any model of the completion (I^-_Π);
- all the remaining atoms.

We call *well-founded model* the pair $\langle I^+_\Pi, I^-_\Pi \rangle$, which can be computed in polynomial time with respect to the size of the ground instance of the input program Π. Then, either one of the following two cases applies:

- if $I^+_\Pi \cup I^-_\Pi = B_\Pi$, then the well-founded model identifies the unique, minimum model of Π;
- otherwise, the well-founded model represents some sort of "partial" model of Π.

In order to deal with the possibility of having multiple plausible interpretations, the notion of *stable model semantics* was introduced.

Definition 1.16 (Gelfond-Lifschitz reduct)**.** The *Gelfond-Lifschitz reduct* Π^S of the (ground) program Π with respect to a *candidate model* $S \subseteq B_\Pi$ can be computed by applying the following transformations to Π:

1. first, all the rules $\rho \in \Pi$ whose body contains an occurrence of a literal of type $\neg\alpha$ such that $\alpha \in S$ are removed;
2. after that, any other negated literal is removed from the remaining rules' bodies.

Observe that for any Π and for any $S \subseteq B_\Pi$, Π^S is a definite program, and thus it has a minimum model M_{Π^S}. We say that S is a *stable model* – or, equivalently, an *answer set* – of Π if and only if the following condition holds:

$$S = M_{\Pi^S} \tag{1.30}$$

More in general, a candidate model $S \subseteq B_\Pi$ is a stable model of the program Π if it is a stable model of its ground instance. We denote by $\mathcal{AS}(\Pi)$ the set that contains all the stable models of Π.

[4] To be precise, computing the completion of a program Π involves some additional steps. However, these are not relevant for the purposes of this section.

Example 1.1. Let us consider the following (ground) program:

$$\Pi := \{p \leftarrow \neg q, q \leftarrow \neg p, r \leftarrow p, r \leftarrow q\} \tag{1.31}$$

In addition, consider the candidate model $S := \{p, r\}$. Let us now test whether $S \in \mathcal{AS}(\Pi)$ by applying the definition:

1. the reduct of Π with respect to S is $\Pi^S := \{p, r \leftarrow p, r \leftarrow q\}$;
2. the unique minimum model of Π^S is $M_{\Pi^S} := \{p, r\}$;
3. since $S = M_{\Pi^S}$, we can conclude that $S \in \mathcal{AS}(\Pi)$.

Now we consider another candidate model $S' := \{p, q, r\}$, and apply the same procedure:

1. the reduct with respect to S' is $\Pi^{S'} := \{r \leftarrow p, r \leftarrow q\}$;
2. the unique minimum model of the reduct is $M_{\Pi^{S'}} := \varnothing$;
3. since $S' \neq M_{\Pi^{S'}}$, we can conclude that $S' \notin \mathcal{AS}(\Pi)$.

Remark 1.4. The term *Answer Set Programming*—often shortened as *ASP*—refers to all those Logic Programming techniques whose goal is to compute the stable models of a given input program.

1.3.5 On the Capabilities of Logic Programs

The definition of stable model semantics that we gave in Section 1.3.4 has two important implications:

1. It suggests a method for performing the search for the answer sets of a given program Π: since the definition of reduct applies to the ground instance of Π, then there should be a preliminary step that *grounds* the program. As a matter of fact, all the implemented solvers use this "2-phase" resolution technique (see Figure 1.4). In addition, it is also worth mentioning that grounding turns out to be a quite expensive process, mainly due to the explosion of the size of programs that it implies;

2. The definition of a stable model subsumes a (polynomial-time) procedure for checking whether a given candidate model is an answer set or not. By the well-known *guess & verify* characterization of the NP complexity class, we can easily conclude that finding one stable models of some ground programs Π is a search problem in NP. However, the grounding phase turns out to be *so* hard that computing $\mathcal{AS}(\Pi)$ for some Π that is not grounded is actually a Σ_2-complete problem.

As a closing remark, consider what follows: if a program Π is such that $F \neq \varnothing$ – i.e. there is at least one function symbol occurring in Π – then both the Herbrand universe $\mathcal{U}_\Pi$ and the Herbrand base B_Π are infinite sets. As a consequence, bottom-up methods for computing minimum models will inevitably produce infinite computations. Fortunately, Lemma 1.2 holds.

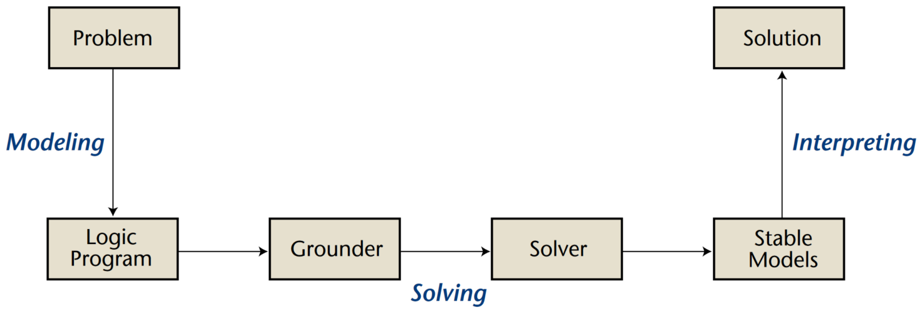


Figure 1.4: Resolution phases of a general logic program under stable model semantics.

Lemma 1.2. *Logic programs without function symbols under stable models semantics are sufficient to capture all* NP *search problems.*

As a consequence, throughout the rest of this thesis we will always consider logic programs without function symbols.

1.4 Markov Chains

In Section 1.2, we described how Euler introduced graph to solve a—literally—walking problem. The notion of *graphs visits*—algorithms that allow to visit all the nodes/edges of a graph—naturally pops out from this very aspect. We usually apply deterministic algorithms to visit a graph. Whenever we adopt Breadth-First Search (BFS) or Depth-First search (DFS), we want to visit and discover every part—node or edge—of the graph. Nevertheless, assume that you want to have a walk in Konisberg but for some reasons your choices are not deterministic but probabilistic. For example, every time you stumble upon a bridge you toss a coin to choose if you want to either cross it or not. Hence, you traverse each bridge with *some probability*. That being the case, the question *if* there exists a path p that crosses all the bridges exactly once is still meaningful but yet not the most important one. On the other hand, we ask ourselves is: if p does exist, what is the probability of walking exactly according to p? In the deterministic model, the existence of p was enough. In the probabilistic settings, the mere existence gives us no certainty that we will actually follow such path. The question we now ask ourselves is: how do we model such behaviour? To answer this question, the theoretical tool we adopt is represented by Markov chains.

Markov chains, also referred to as Markov processes, are stochastic models used to describe systems that undergo transitions from one state to another in a probabilistic manner. The key feature of a Markov chain is that the transition to the next state depends only on the current state (memoryless property). Markov processes are usually split into two main categories: discrete time and continuos time.

1.4.1 Discrete-Time Markov Chains (DTMC)

A *Discrete-Time Markov Chain (DTMC)* is a sequence of random variables $\{X_n\}_{n\geq 0}$, where X_n represents the state of the system at step n. The system switches between

states at discrete time steps, and the probability of transitioning from state i to state j depends only on the current state i, not on the previous states.

Formally, a DTMC is characterized by:

- *State space*: A countable set $S = \{1, 2, 3, \dots\}$.
- *Transition probabilities*: $P_{ij} = \mathbb{P}(X_{n+1} = j \mid X_n = i)$, which form the *transition matrix* P.

The reader may notice that the definition we gave for S is pretty general. Throughout the rest of this thesis, we will assume the set of states of a given Markov Chain to be always *finite*. The transition matrix P is a square matrix where each element P_{ij} represents the probability of moving from state i to state j in one step.

$$P = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n,1} & p_{n,2} & \cdots & p_{n,n} \end{pmatrix}$$

The properties of a DTMC transition matrix are:

- $p_{i,j} \geq 0$, for all $i, j \in S$.
- $\sum_{j \in S} p_{i,j} = 1$, for all $i \in S$ (row sums are 1).

Example 1.2. Consider a simple graph $G = (V, E)$, where V is the set of vertices and E is the set of edges. Suppose we have a random walk on this graph, where at each time step, the walker moves to a neighbouring vertex chosen uniformly at random.

Let $S = \{1, 2, \dots, n\}$ represent the vertices of the graph. The transition matrix P for the random walk is given by:

$$p_{i,j} = \begin{cases} \frac{1}{d(i)} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

where $d(i)$ is the degree of vertex i (the number of edges connected to i).

1.4.2 Continuous-Time Markov Chains (CTMC)

A *Continuous-Time Markov Chain (CTMC)* extends the concept of DTMC to continuous time. In a CTMC, the system transitions between states continuously over time, and the time spent in each state follows an exponential distribution.

Formally, a CTMC $\mathcal{X}(t)$ is characterized by:

- *State space*: A countable set $S = \{1, 2, 3, \dots\}$.
- *Transition rates*: $q_{i,j}$, where $q_{i,j} \geq 0$ represents the rate at which the system transitions from state i to state j, and $q_{i,i} = -\sum_{j \neq i} q_{i,j}$. These rates form the *generator matrix* Q.

For what concerns the set S, the assumption about its finiteness holds for the continuous-time case as well. The *generator matrix* Q is a square matrix where each element $q_{i,j}$ represents the rate of transitioning from state i to state j, with the diagonal elements representing the negative of the total outgoing rate from each state.

$$Q = \begin{pmatrix} q_{1,1} & q_{1,2} & \cdots & q_{1,n} \\ q_{2,1} & q_{2,2} & \cdots & q_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ q_{n,1} & q_{n,2} & \cdots & q_{n,n} \end{pmatrix}$$

As well as in the discrete time case, the entries of Q are all positive or at most zero. The main distinction between the matrix Q and its Discrete-time counterpart is the following property about its diagonal:

$$q_{i,i} = -\sum_{j \neq i} q_{i,j}, \forall i \in S$$

Example 1.3. Consider a graph $G = (V, E)$, where transitions between vertices occur continuously over time, and the transition from vertex i to vertex j follows a Poisson process with rate λ_{ij}. In this case, the generator matrix Q is given by:

$$q_{i,j} = \begin{cases} \lambda_{i,j} & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

with the diagonal elements $q_{i,i}$ determined by:

$$q_{i,i} = -\sum_{j \neq i} Q_{i,j}$$

This describes a CTMC where the system continuously jumps between neighbouring vertices of the graph, with transition rates given by the Poisson process rates $\lambda_{i,j}$.

1.4.3 Random walks on Graphs

Random walks are a fundamental tool in the study of randomized processes. In line with our purposes, this section shall review random walks with emphasis on their algorithmic applications.

Given a graph $G = (V, E)$ and a vertex $v \in V$, a random walk on G consists in the following trivial process: starting from v, walk to a vertex v' chosen from the out-neighbourhood of v uniformly at random. Repeat the process starting from v' for a discrete number of steps. A more formal representation of the process is given by Algorithm 1.

Algorithm 1 The random walk algorithm.

Require: $G = (V, E),\ v \in V,\ T \in \mathbb{N}$.
1: **for** $i := 1$ to T **do**
2: Choose $v' \in \delta^+(v)$ uniformly at random;
3: $v \leftarrow v'$;
4: **end for**

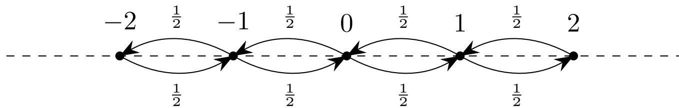


Figure 1.5: Random walk on the infinite line.

Before proceeding any further, let us observe an example that shall be useful also for future parts.

Example 1.4 (Random walk on the infinite line)**.** Consider the graph $G = (\mathbb{Z}, E)$ visualized in Figure 1.5, where $E = \{\{i, j\} : i, j \in \mathbb{Z}, i = j + 1\}$. Assuming the walk to start from vertex 0, the first step may reach either vertex 1 or -1, both with probability 1/2 - a *fair coin toss.*

Once the first step is performed, say towards vertex 1, the second step involves the exact same procedure with respect to vertices 0 and 2.

Let $G = (V, E)$ be a directed graph with $n = |V|$. A random walk over G can be understood in terms of a DTMC where:

- The set of states is V
- The transition matrix P is such that:

$$p_{u,v} = \begin{cases} \frac{1}{d^+(u)} & \text{if } v \in \delta^+(u), \\ 0 & \text{otherwise.} \end{cases} \tag{1.32}$$

With respect to the standard walks on graphs, randomized ones bring probabilities in the playground. Therefore, when we visit a graph we cannot state precisely which state we are currently on. What we must introduce to study the behaviour of a random walk is a so-called *probability vector*. Let $\pi = (\pi_1, \pi_2, \cdots, \pi_n)$ be a vector. Then, π is called a *probability vector* if the following condition holds:

$$\sum_{i=1}^{n} \pi_i = 1$$

Roughly speaking, lifting such class of vector to the notion of random walks, π_i holds the current probability of the walker of being in state i.

Typically, a thorough introduction on random walks and DTMCs would involve a great deal of definitions and results aimed to their asymptotic time-analysis. However, in this thesis we are concerned with the conditions upon which random walks may be extended to the quantum settings, rather than the behaviour of the walks themselves. That being the case, these notions are hereby omitted, and the reader referred to [130] for a more rigorous presentation.

1.4.4 Steady-State Distribution

We assume that the Markov processes we deal with satisfy the following properties: a Markov process $X(t)$ is

- *stationary* if its statistical properties do not change over time, i.e., the family of random variables $(X(t_1), X(t_2), \ldots, X(t_n))$ has the same distribution as the collection $(X(t_1+\tau), X(t_2+\tau), \ldots, X(t_n+\tau))$ for all $t_1, t_2, \ldots, t_n, \tau \in \mathbb{R}^+$.

- *time-homogeneous* if the conditional probability $Prob(X(t+\tau) = s \mid X(t) = s')$ remains constant regardless of t, i.e., the behaviour of the system does not depend on when it is observed. In particular, the transitions between states are independent of the time at which the transitions occur.

- *irreducible* if all states in its state space $\mathcal{S}$ can be reached from all other states, by following the transitions of the process

Within a Markov process, a state is labelled *persistent* or *recurrent* if the probability of the process eventually returning to that state is one. Otherwise, the state is called *transient*. In terms of a system, the recurrent states correspond to the behaviour which is repeatedly exhibited by the system whereas transient states correspond to a behaviour which will be no longer exhibited after a certain time. A recurrent state is termed *positive-recurrent* or *ergodic* if the expected number of steps until the process returns to that state is less than infinity. A Markov process is *ergodic* if all its states are positive-recurrent. In the context of finite Markov chains, irreducibility alone ensures ergodicity.

Ergodicity of a Markov Chain—either Discrete or Continuous—is a necessary and sufficient condition for the existence of a steady-state distribution.

The concept of a *steady-state distribution* (or stationary distribution) is essential in Markov chain theory. It represents a probability distribution over the states that remains unchanged as the system evolves over time.

Steady-State Distribution for Discrete-Time Markov Chains (DTMC)

Let P be the transition matrix of a Discrete-Time Markov Chain (DTMC) with n states. A *steady-state distribution* is a probability vector $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$ such that:

$$\pi P = \pi$$

where π_i represents the long-term probability of being in state i, and the elements of π satisfy the conditions:

$$\sum_{i=1}^{n} \pi_i = 1 \quad \text{and} \quad \pi_i \geq 0 \text{ for all } i.$$

This equation means that if the system starts in the distribution π, it will remain in the same distribution after each transition.

Steady-State Distribution for Continuous-Time Markov Chains (CTMC)

For a Continuous-Time Markov Chain (CTMC) with generator matrix Q, the *steady-state distribution* is a probability vector $\pi = (\pi_1, \pi_2, \dots, \pi_n)$ such that:

$$\pi Q = 0$$

where π_i represents the long-term probability of being in state i. The elements of π must satisfy:

$$\sum_{i=1}^{n} \pi_i = 1 \quad \text{and} \quad \pi_i \geq 0 \text{ for all } i.$$

This equation implies that, in the steady state, the rate of flow into each state equals the rate of flow out, achieving a balance over time. The computation and characterization of steady-state distributions, for both discrete and continuous time cases, is a well-studied topic. However, in this thesis, our primary focus will be on examining the relationships between the steady-state distribution of a Markov chain and that of its reduced version. For a more comprehensive discussion on the properties of steady-state distributions and computational methods, we direct the reader to [105].

1.5 Neural Networks

Artificial Intelligence (AI), Machine Learning (ML) and Deep Learning (DL) are three commonly misunderstood terms. With *Artificial Intelligence* we refer to any technique which enables computers to mimic human behaviours. Both Machine Learning and Deep Learning are proper *subsets* of Artificial Intelligence. The former refers to all the methodologies, algorithms and tools that enable machines to improve with experiences. The latter is, again, a subset of Machine Learning and deals with particular structures called *Neural Networks*.

The relation between AI, ML, and DL is shown in Figure 1.6.

The algorithms coming from both ML and DL have one peculiar property with respect to classical algorithms: they learn from mistakes. In Algorithms and Data Structures classes we are always taught that an algorithm must work in all cases, otherwise it is not correct. The techniques adopted in the field of ML and DL are such that they adjust their behaviour according to what they are doing correctly and what they are not.

At the basics of any ML/DL tool there is a *learning task*. Roughly speaking, the problem we want to solve by *learning* from data. Tasks that are usually tackled can be divided in four major *learning* categories:

- Supervised
- Unsupervised
- Reinforcement
- Generative

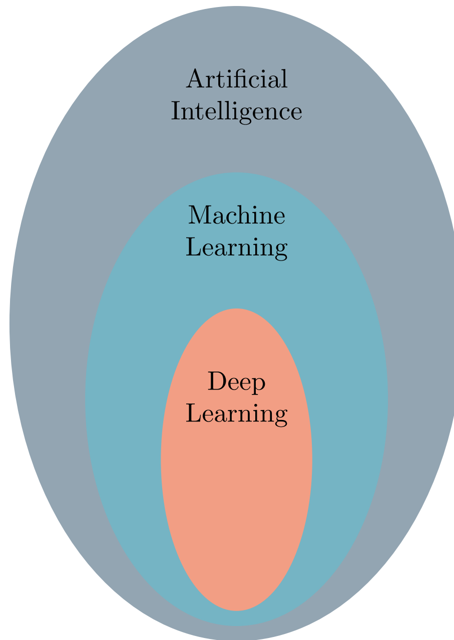


Figure 1.6: Venn diagram depicting the relation between AI, ML, and DL.

The first two are the most famous ones. Supervised Learning refers to *learning tasks* starting from labelled data. A set made of *samples* is provided as input. Each sample is made of *features*. One extra, *special* feature is provided, and it is called *ground truth label*, or simply *label*.

Starting from these data, the supervised learning techniques can solve either *regression* or *classification* tasks. An example of supervised regression task is the house pricing—Figure 1.7. Suppose we are provided with a set of houses, each one labelled with the price it should be sold. The ML algorithm learns from these data and then produces a *model* that can predict the prices of new, unseen houses. Hence, regression tasks are adopted to learn and approximate functions whose outcome is a continuous value.

On the other hand, classification is the task of predicting the class to which each sample belongs among a finite set of possibilities. The ground truth labels that come with the samples define their class and the model we produce must be able to put new samples in the correct class. An example is the tumour classification task. The input dataset is made of samples with two features: age and tumour size. The model will be able to predict if a tumour is either malignant or benign—Figure 1.8.

Both regression and classification are *supervised* learning tasks: input samples must be labelled.

If this is not true—data does not come with labels—the problem we solve is an *unsupervised learning* one. We are still provided with an input set, organized in samples, but in this case we want to cluster them in groups. The aim of the model is to learn a strategy that groups elements according to their properties. In the end, we want ele-

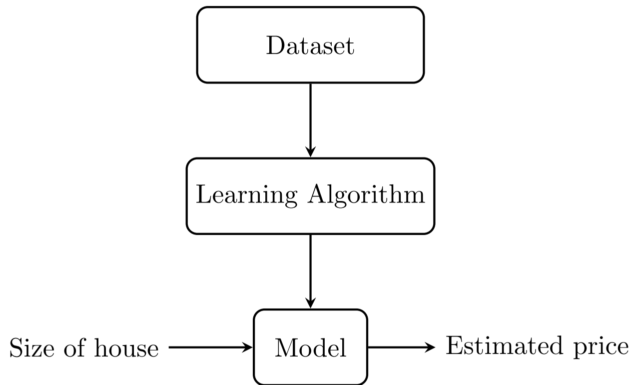


Figure 1.7: House pricing regression task.

ments that are somehow *similar*—according to some similarity measure—to be clustered together. One example may be some geometric distance defined on the samples—notice that a sample with m features can be seen as an m-dimensional vector on which we can define any kind of linear algebraic property.

We refer the reader to [5] for:

- More details on supervised and unsupervised learning techniques
- Introduction to both reinforcement and generative learning

1.5.1 Deep Learning

While all the aforementioned tasks lie in the realm of Machine Learning, we now move to the investigation of deep learning. We do so by focusing on classification, which is the most tackled problem using AI related techniques. As we stated above, classification deals with the assignment of labels to samples where each label represents a class.

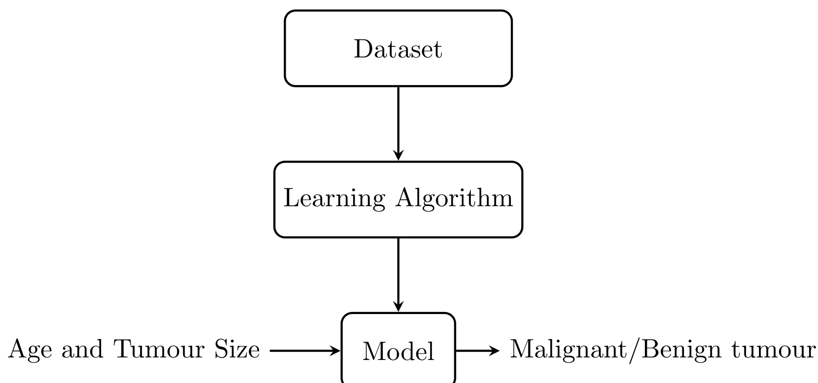


Figure 1.8: Tumour classification task.

The very first model to solve this problem is called *Perceptron*. It was invented at the Cornell Aeronautical Laboratory by Frank Rosenblatt [155].

Before describing how Rosenblatt's perceptron works, we take a little step back to talk about its never mentioned predecessor: The McCulloch-Pitts Neuron [118]. The first important difference is the name given to the tool: neuron. Why is it so? Human beings can process information pre-attentively. In less than the blink of an eye, we can identify elements that come out from an image. On the other hand, if elements are not too visible, we need to make sequential search, which clearly takes much longer. Of course, we solve tasks like classification since we are newborns. Therefore, scientists asked themselves how to take advantage of this prominently human skill to devise algorithms to solve classification systematically. They found out that neurons lie at the very core of the *computations* we carry out in our brain.

A very simplified version of a neuron is depicted in Figure 1.9.

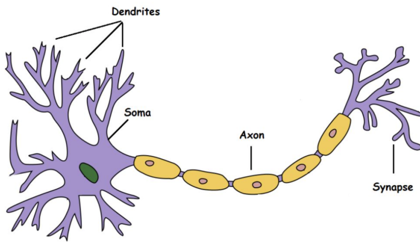


Figure 1.9: An abstract representation of a neuron.

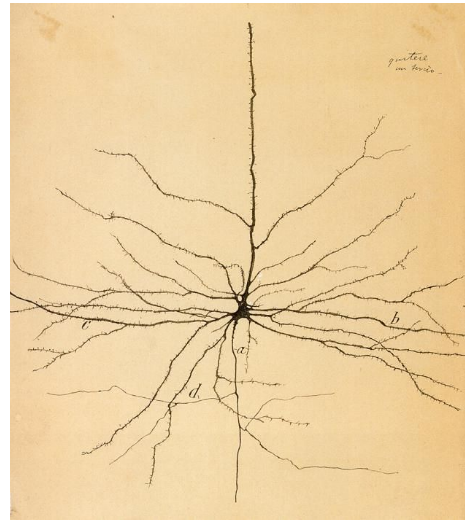


Figure 1.10: Drawing of a brain cell made by Santiago Ramón y Cajal.

On the other hand, Figure 1.10 is a signed drawing of a brain cell made by Santiago Ramón y Cajal[5].

Figure 1.9 highlights four major neuron components:

- Dendrite: receives signals from other neurons
- Soma: the component whose goal is to process the information
- Axon: which serves as a transmitter for the output
- Synapse: the physical connection point between neurons

[5] This was supposed to be a gift from Sheldon to Amy in The Big Bang Theory episode "The Tangible Affection Proof".

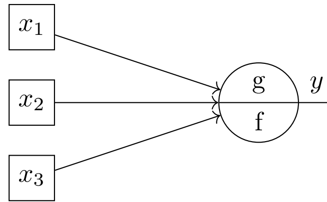


Figure 1.11: A McCulloch-Pitts neuron.

Without delving too much into neuroscientific and biological information, a neuron receives an *input signal* (Dendrite), processes it (Soma) and passes the result to other neurons (Axon and Synapse cooperates in this step).

Imagine you are looking at the picture of a dog. The neuron will receive the information received from the eyes, processes it and then outputs that what is in the photo is actually dog. It has somehow *classified* the dog. Usually neurons activate like a waterfall—one after the other—to obtain this result, but we now focus on a single unit.

The model introduced by McCulloch and Pitts tried to mimic exactly this behaviour. They devised an *artificial neuron* whose structure is depicted in Figure 1.11.

Its behaviour is the following. The first step is to process a series of boolean inputs (dendrites), that are named x_1, x_2, and x_3 in Figure 1.11. We say that the neuron fires if an input takes value 1. The input information is then processed into an aggregating function g—which in the simplest case is a sum—and this is comparable to the soma of a real neuron. The other ingredient f is another function used to check if some conditions on g are met or not. Some examples are:

- f is one if and only if g is one
- f is one if and only if g is greater than zero
- f is one if and only if g is greater than some threshold θ

Suppose g is defined as the sum of the inputs. One possible behaviour of the function f is as follows:

$$f = \begin{cases} 1 \text{ if } g = \sum x_i = 1 \\ 0 \text{ otherwise} \end{cases}$$

The result of the function f is then passed to the next neuron or returned as the result of the classification. In both cases the propagation of the result recalls axons and synapses' job. Notice that f is what nowadays we call *activation function*.

The two main problems of this model are:

1. It cannot process non-boolean inputs
2. Each input is as important as the others

Both these issues has been solved by Rosenblatt's perceptron. Introduced in [155], this model clearly resembles the one by McCulloch and Pitts but with some differences. First, there is no domain constraints on the input: they can hold any value in $\mathbb{R}$.

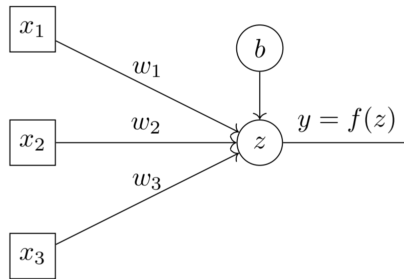


Figure 1.12: Perceptron defined by Rosenblatt.

Moreover, the input are not given to g *as they are*, but they are weighted using some weights assigned to each connection.

In Figure 1.12 each edge from the inputs x_is is labelled with a real number w_i called *weight*. Input values are aggregated according to the following formula:

$$\sum w_i x_i$$

Roughly speaking, a weighted sum of the inputs. With respect to McCulloch-Pitts neuron, in this model there is also the introduction of a *bias* b. An extra input that does not come from other neurons but is considered as well as input to the neuron. Hence, the above sum becomes:

$$z = b + \sum w_i x_i$$

The value z becomes the input of *activation* function which produces the perceptron's output y. Examples of activations functions are:

- Simple threshold: exactly like function f in McCulloch-Pitts neurons, an activation function that just checks if $z > \theta$

- ReLU: this activation function behaves as the identity if $z \geq 0$, otherwise as the constant function 0:

$$\text{ReLU}(z) = \begin{cases} z \text{ if } z \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

- Sigmoid function σ. Its main property is that it maps any input from $\mathbb{R}$ in a continuous range $[-1, 1]$. It is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

  We plot the sigmoid in Figure 1.13.

Notice that this is as far as possible from an exhaustive and complete list of activations functions that may be found in the literature. ReLU, for example, has been tweaked in many ways to deal with different tasks. We refer the reader to [13] for further details.

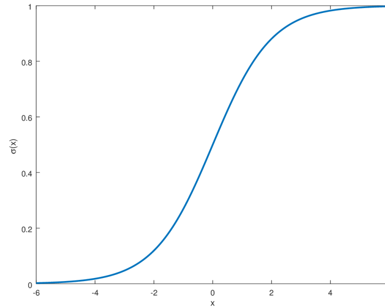


Figure 1.13: Plot of the sigmoid function.

Despite the thousands of activation functions that can be designed, they all must hold two key properties:

- They must be differentiable
- They must be non-linear

The former property will be fundamental when dealing with the *learning algorithm*. While the latter is crucial for the DL algorithms expressiveness.

From the definition of perceptron, a natural question arises: how are weights computed? Rosenblatt innovation was to introduce a *learning* algorithm to choose the best suited values for the weights. Roughly speaking, the perceptron tries to solve some classification task. Any time it makes a mistake by misclassifying an element, the weights are updated to respond to this error. The learning algorithm that was proposed in [155] can be summarized as follows:

- Initialize the weights randomly
- Take one sample $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ with n features and predict $\hat{y}$—the *prediction* is the result of the activation function
- Now compare $\hat{y}$ with the labels y that comes with $\mathbf{x}$.
  - If $\hat{y} = 1$ and $y = 0$, then decrease the weights
  - If $\hat{y} = 0$ and $y = 1$, increase the weights
- Repeat until no sample is wrongly classified

Despite the great success that the perceptron and the learning algorithm had, it was proved that such model could not solve the XOR problem [126]. The XOR problem is a truly simple task: samples are length 2 boolean vectors. Their ground truth is the XOR between the two features—we recall that the XOR between two bits is 1 if and only if the two bits are different, zero otherwise.

Figure 1.14 shows the dataset of the XOR problem. The reason why a perceptron could not solve this problem is that the target classification cannot be obtained via linear

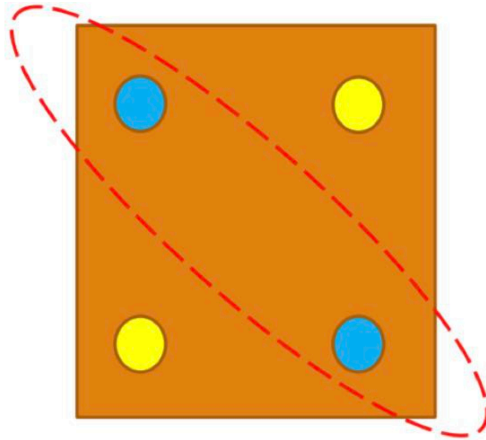


Figure 1.14: The XOR problem

separation using a line, or in general, a hyperplane. In fact, geometrically speaking, whenever a perceptron is trained, it adjusts the weights of a hyperplane that separates the samples. By adapting the weights, we just apply geometric transformation to the plane until it reaches the optimal coordinates to divide the different classes. Using only one hyperplane it is not possible to solve the XOR problem.

To overcome this issue we should be able to use more than one hyperplane to separate data: multi-layer perceptrons is the key.

A *layer* of perceptrons is a set of independent and parallel perceptrons. An example is depicted in Figure 1.15.

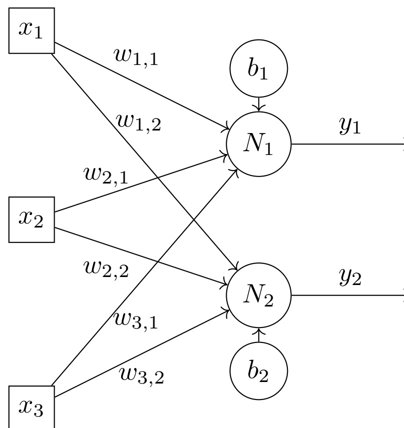


Figure 1.15: A layer of two perceptrons/neurons.

Each perceptron is fed with the same input data, but each neuron has its own weights and bias. Both Neuron 1 (N_1) and Neuron 2 (N_2) work independently. Therefore, given

an input sample $\mathbf{x} = (x_1, x_2, x_3)$, Neuron 1 computes its output as

$$y_1 = f_1(b_1 + \sum_{i=1}^{3} w_{i,1} x_i)$$

Analogously, Neuron 2:

$$y_2 = f_2(b_2 + \sum_{i=1}^{3} w_{i,2} x_i)$$

where f_1 and f_2 are Neuron 1 and Neuron 2 activation functions, respectively. As

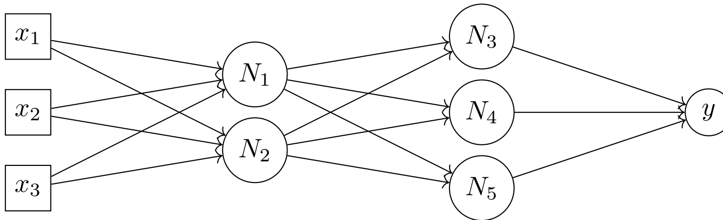


Figure 1.16: Two layers of perceptrons. The naming convention for the weights is that the edge connecting neuron N_i and neuron N_j is $w_{i,j}$

depicted in Figure 1.16, we can also stack different layers of perceptrons. In this case the outputs coming from Neuron 1 and 2 are directed to Neuron 3, 4, and 5—N_1, N_2, N_3, N_4, and N_5, respectively. We omitted the biases to avoid making the picture too heavy to handle.

The layer composed only by N_1 and N_2 is called *input layer* since it is the one that directly interacts with the inputs x_i. The last layer made only of Neuron called y is the so-called *output layer*. The layer in the middle—the one with neurons N_3, N_4, and N_5—is called *hidden layer*. One can decide to add as many hidden layers as he needs to solve the task. The process of giving in input a sample and obtaining an output is called *forward pass*.

Since the amount of weights increases as the number of neuron/layer increases, we need a data structure to formalize the behaviour of a neural network. Matrices and vectors have been adopted as the common standard. Using the neural network depicted in Figure 1.16 as witness, the weights $w_{i,3}$, $w_{i,4}$, and $w_{i,5}$ are stored in a real-valued matrix $W \in \mathbb{R}^{2\times3}$. Where the number of rows is the number of neurons of the input layer (2), and the number of columns is the number of neurons on the hidden layer (3). Hence, element $W_{i,j}$ will contain the weight $w_{i,j}$ of the connection from neuron i to neuron j.

Suppose y_1 and y_2 are the two values obtained by Neuron 1 and Neuron 2 for some input sample. Neurons 3, 4, and 5 now must compute the following quantity:

$$z_j = b_j + \sum_{i=1}^{2} w_{i,j} y_i \text{ with } j \in \{3, 4, 5\}$$

that will be used as input for the activation functions. It turns out that adopting the

vector-matrix representation, the sum $\sum_{i=1}^{2} w_{i,j} y_i$, for any value of j, can be computed with a pre-product between the vector $\overline{y} = (y_1, y_2)$ and the matrix W.

$$\begin{aligned}\overline{y}W &= (y_1, y_2) \begin{pmatrix} w_{1,3} & w_{1,4} & w_{1,5} \\ w_{2,3} & w_{2,4} & w_{2,5} \end{pmatrix} \\ &= (w_{1,3}y_1 + w_{2,3}y_2, w_{1,4}y_1 + w_{2,4}y_2, w_{1,5}y_1 + w_{2,5}y_2)\end{aligned}$$

Which is the vector containing the sums for all the three neurons. Such vector must be added to the *bias vector* $B = (b_3, b_4, b_5)$ obtaining:

$$\begin{aligned}&(w_{1,3}\, y_1 + w_{2,3}\, y_2, w_{1,4}\, y_1 + w_{2,4}\, y_2, w_{1,5}\, y_1 + w_{2,5}\, y_2) + (b_3, b_4, b_5) = \\ &(b_3 + w_{1,3}\, y_1 + w_{2,3}\, y_2, b_4 + w_{1,4}\, y_1 + w_{2,4}\, y_2, b_5 + w_{1,5}\, y_1 + w_{2,5}\, y_2) = (z_3, z_4, z_5)\end{aligned}$$

yielding to the vector containing only the z values. After this step, only the activation functions are missing.

Now that we hold a clear *practical* definition of a Neural Network, we can move to its formal definition: For $k \in \mathbb{N}$, we denote by $[k]$ the set $\{0, 1, \ldots, k\}$, by $(k]$ the set $\{1, \ldots, k\}$, by $[k)$ the set $\{0, \ldots, k-1\}$,and by (k) the set $\{1, \ldots, k-1\}$.

Definition 1.17 (Neural Network)**.** A *Neural Network (NN)* is a tuple $\mathcal{N} = (k, \mathcal{A}ct, \{\mathcal{S}_\ell\}_{\ell \in [k]}, \{W^\ell\}_{\ell \in (k]}, \{b^\ell\}_{\ell \in (k]}, \{A_\ell\}_{\ell \in (k]})$ where:

- k is the number of layers (except the input layer);
- $\mathcal{A}ct$ is the set of activation functions;
- for $\ell \in [k]$, $\mathcal{S}_\ell$ is the set of nodes of layer ℓ with $\mathcal{S}_\ell \cap \mathcal{S}_{\ell'} = \varnothing$ for $\ell \neq \ell'$;
- for $\ell \in (k]$, $W^\ell_{i,j}$ is the weight of the connection between node $i \in S_{\ell-1}$ and node $j \in S_\ell$;
- for $\ell \in (k]$, b^ℓ_i is the of bias of neuron $i \in S_\ell$;
- for $\ell \in (k]$, $A_\ell : \mathcal{S}_\ell \to \mathcal{A}ct$ is the activation association function that associates an activation function with nodes of layer ℓ.

$\mathcal{S}_0$ and $\mathcal{S}_k$ denote the nodes in the input and output layers, respectively.

The operational semantics of a neural network is as follows. Let $v : \mathcal{S}_\ell \to \mathbb{R}$ be a valuation for the ℓ-th layer of $\mathcal{N}$ and $Val(\mathcal{S}_\ell)$ be the set of all valuations for the ℓ-th layer of $\mathcal{N}$. The operational semantics of $\mathcal{N}$, denoted by $[\![\mathcal{N}]\!]$, is defined in terms of the semantics of its layers $[\![\mathcal{N}]\!]_\ell$, where each $[\![\mathcal{N}]\!]_\ell$ associates with any valuation v for layer $\ell-1$ the corresponding valuation for layer ℓ according to the definition of $\mathcal{N}$. The valuation for the output layer of $\mathcal{N}$ is then obtained by the composition of functions $[\![\mathcal{N}]\!]_\ell$.

Definition 1.18. The semantics of the ℓ-th layer is the function $[\![\mathcal{N}]\!]_\ell : Val(\mathcal{S}_{\ell-1}) \to Val(\mathcal{S}_\ell)$ where for all $v \in Val(\mathcal{S}_{\ell-1})$, $[\![\mathcal{N}]\!]_\ell(v) = v'$ and for all $s \in \mathcal{S}_\ell$, $v'(s)$ is defined by the following Equation (1.33):

$$v'(s) = A_\ell(s) \left(\sum_{r \in \mathcal{S}_{\ell-1}} W^\ell_{r,s} v(r) + b^\ell_s \right) . \tag{1.33}$$

The input-output semantics of $\mathcal{N}$ is obtained by composing these one-layer semantics. More precisely, we denote by $[\![\mathcal{N}]\!]^\ell$ the composition of the first ℓ layers so that $[\![\mathcal{N}]\!]^\ell(v)$ provides the valuation of the ℓ-th layer given $v \in Val(\mathcal{S}_0)$ as input. Formally, $[\![\mathcal{N}]\!]^\ell$ is inductively defined by Equations (1.34) and (1.35):

$$[\![\mathcal{N}]\!]^1 = [\![\mathcal{N}]\!]_1 \tag{1.34}$$

$$[\![\mathcal{N}]\!]^\ell = [\![\mathcal{N}]\!]_\ell \circ [\![\mathcal{N}]\!]^{\ell-1} \; \forall \ell \in (k] \tag{1.35}$$

where $\circ$ denotes the function composition.

We are now in a position to define the semantics of $\mathcal{N}$ as the input-output semantic function $[\![\mathcal{N}]\!]$ defined below.

Definition 1.19. The input-output semantic function $[\![\mathcal{N}]\!] : Val(\mathcal{S}_0) \rightarrow Val(\mathcal{S}_k)$ is defined by the following Equation:

$$[\![\mathcal{N}]\!] = [\![\mathcal{N}]\!]^k \,.$$

Before investigating how the learning algorithm introduced in [155] has evolved to cope with the multi-layer perceptrons, we want to stress one particular result about Neural Network capabilities and expressiveness. The behaviour of a neural network primarily consists of a sequence of vector-matrix multiplications, with one key exception: the activation functions. These activation functions introduce non-linearity, which is critical to the network's expressiveness. If the activation functions were linear, they could be incorporated into the matrices, as per basic linear algebra principles. Consequently, the behaviour of individual neurons would be reduced to basic vector-matrix operations, and entire layers could also be represented in this way, resulting in a fully linear algebraic model of the network. In this linear scenario, the entire network could be collapsed into a single neuron, where the weights and biases of this neuron would be aggregates of the weights and biases from the original network. However, this linear approach would fail to capture the complexity required to solve problems like the XOR problem.

Thus, the introduction of non-linearity through activation functions is crucial for the expressiveness and capability of neural networks. Regarding the capabilities of neural networks, a fundamental result underpins many of the techniques using this tool:

Theorem 4 (Universal Approximation Theorem[94])**.** *Any continuous function that maps intervals of real numbers to some output interval of real numbers can be approximated arbitrarily closely by a multi-layer perceptron with one hidden layer.*

We can now move to the description of the learning algorithm that is adopted when *training* a neural network. The algorithm used for a single perceptron is not applicable since by definition it would require the truth value of every neuron. Therefore, *gradient descent* has been adopted.

First, we have to introduce the notion of *cost function*. Until now, we always supposed the *training set*—the set of samples used during the training phase—to be made of only one sample. As the name suggests, this is not true. The first problem we have to tackle is how we can aggregate the classification results obtained from the forward passes of all the samples. For example, in a training set of 4 samples, if the network fails

to classify 2 out of 4, we want to have access to this information without storing all the forward pass results. Roughly speaking, we do not want an *output set* that encodes for each input whether the network worked properly or not. We aim at obtaining a value that somehow encodes how well the network performed. To reach this goal, *cost function* is adopted. A cost function J takes in input the predictions of a network for some samples set X, the ground truth of the samples in X and returns a value encoding how well the network classified X. One example is the cost function of the logistic classifier. Let $\mathbf{y} = (y_1, y_2, \ldots, y_m)$ and $\hat{\mathbf{y}} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m)$ be the ground truth labels and the predicted labels, respectively. The cost function $J(\mathbf{y}, \hat{\mathbf{y}})$ is defined as:

$$J(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{m} \sum_{i=1}^{m} C(\hat{y}_i, y_i)$$

where the function C is:

$$C(\hat{y}_i, y_i) = -y_i \ln(\hat{y}_i) - (1 - y_i) \ln(1 - \hat{y}_i)$$

This function has two peculiar properties:

- Since $\hat{y}_i$ is the result of a NN forward pass, it clearly depends on both the weights and the biases of the function. The function J must derivable with respect to both the weights and the biases
- It behaves in such a way that it is zero if the prediction agrees with the ground truth, while it goes to infinity if $\hat{y}_i$ and y do not agree.

Both these two properties should be enjoyed by any useful cost function one intends to adopt.

Mathematically speaking, the second property is required since we can relate the network performances with the value that the cost function holds. The more the cost function has a high value, the worse the network is performing in the classification task. Hence, our goal is to minimize the value of J. The parameters we can adjust are the biases and the weights of the network: from this very point the first property we named above pops out.

In fact, to minimize the cost function, the gradient descent algorithm is adopted. This algorithm internals are tightly connected to the computation of partial derivatives with respect to the parameters we want to adjust. Roughly speaking, the algorithm works as follows:

1. Starts with some random weights and biases
2. Perform the forward pass for all the input samples
3. Compute the cost function
4. Update weights and biases accordingly.

Up to the third step, it is nothing new. The gradient descent algorithm describes how to update the weights and the biases of the network. Suppose we want to update the

weight $w_{i,j}$ using gradient descent. Its new value will be:

$$w_{i,j} = w_{i,j} - \alpha \frac{\partial J}{\partial w_{i,j}}$$

where α is called learning rate, and it usually defined by the user.

The core idea of gradient descent is to use the landscape of function J as a playground. Suppose the current value of J—with respect to the current set of weights and biases—defines the position of a small marble that is put on J's landscape. Gradient descent applies changes to the values of the parameters so that the marbles will eventually end up in a function minima—either local or global—achieving good performances on the task. As per the marble perspective, the minima is like a pit in which it falls and never comes back from. Always from its perspective, the value of α is a scaling factor that defines how much the marble can move with one single step.

The application of gradient descent is called *backward pass* or *backpropagation*. It is repeated until the results obtained by the network—its *accuracy*—are satisfying.

1.6 Quantum Computing and Quantum Mechanics

During the past few decades, we have been developing, building, and getting used to computing devices that—while often significantly different in shape, purpose, and sometimes also internal architecture—at their core all rely on the rules of the field that goes under the name of *classical computation theory*.

At the fundamental level, stripping down all the convenient abstractions made available by high-level programming languages, any piece of software can be thought as a finite, ordered[6] sequence of basic instructions, which in turn are selected among a fixed-size set of operations that (roughly) represents all the actions that the hardware is capable of performing. Both the fundamental building blocks of computations—the instructions we just mentioned—and the data they manipulate are stored in memory chips, encoded as strings over an alphabet made of only two symbols: 1s and 0s. These are what we call *binary digits* (or, for short, *bits*), and represent the very core of computations as we use them today. In memories made by billions of cells, every single bit carries its own, tiny piece of information, which gets inevitably altered if the value of that bit suddenly changes. Thus, there must be no uncertainty over the value of each bit[7]: it must always be either a 0, or a 1.

The theory of *quantum computations* introduces a new element right at the core of this model, which can be intuitively thought as allowing every bit to be either a 1, a 0, or any linear combination of these two states. More specifically, we refer to the latter situation with the name *states superposition*. To make things even more confusing, the

[6]The informed reader may argue that parallel and distributed computations do not actually fit well with this ordering constraint. For the scope of this thesis though, we will consider a more abstract notion of *classical computation*, in particular referring to computing models such as Turing Machines or analogous devices.

[7]In an ideal world where "random" bit-flips and data corruption do not exist, every bit remains unchanged and retains its piece of information until rewritten on purpose. Unfortunately, the actual memory chips that we use in the real world are not always capable of doing so due to a variety of factors, but this is out of the scope of this thesis and not relevant for the purposes of this chapter.

coefficients of any such linear combination cannot be physically measured, as quantum bits *collapse* to one of the states of the basis—i.e. 0 or 1—every time they are *observed*, with probabilities defined by their inner state.

Even if at first sight it may not look like so, this behaviour dramatically changes the properties of the entire computational model, and requires a deep re-thinking of the algorithms and techniques we use in classical computations in order to exploit the power that derives from quantum properties.

This section is devoted to guiding the reader through the very basics of the theory of quantum computations. Most of the concepts presented here have been adapted from [133] and [161].

1.6.1 Quantum Bits of Information

A classical bit is an elementary object that can assume exactly one out of two possible values: 0 or 1. Quantum bits (or *qubits*, for short) generalize this concept by exploiting linear algebra in the field of *complex numbers*.

Qubits

A *qubit* is a mathematical object defined as a normalized[8] 2-dimensional vector in the complex field. Usually, such vectors are represented with *Dirac's notation*[9]:

- $|\psi\rangle$ is called a *ket* and represents a column vector;
- $\langle\psi| := |\psi\rangle^\dagger$ is called a *bra* and represents a row vector, where the *dagger* indicates an element-wise application of the conjugate operator, followed by a transposition.

A combined use of a *bra* and a *ket* allows to compactly represent the scalar product of two vectors:

$$\langle\psi_1|\psi_2\rangle := \langle\psi_1| \cdot |\psi_2\rangle = |\psi_1\rangle^\dagger \cdot |\psi_2\rangle \tag{1.36}$$

Being qubits 2-dimensional vectors, a basis of size 2 is required to represent them. One of the most commonly used bases is the *computational basis*:

$$|0\rangle := \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad\qquad |1\rangle := \begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{1.37}$$

However, other useful bases do exist, such as the one composed by the following 2 vectors:

$$|+\rangle := \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \qquad\qquad |-\rangle := \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \tag{1.38}$$

It is interesting to notice that any qubit state $|\psi\rangle$ given as a linear combination of the states of some basis, e.g. the computational basis:

$$|\psi\rangle := \alpha\,|0\rangle + \beta\,|1\rangle \qquad \text{with } \alpha, \beta \in \mathbb{C} \,:\, |\alpha|^2 + |\beta|^2 = 1 \tag{1.39}$$

[8] In this context, a vector v is said to be *normalized* if its norm is 1.
[9] We slightly recall the notation that was already introduced in Chapter 1.1

can be rewritten in the following form:

$$|\psi\rangle := e^{\imath\gamma}\left(\cos\frac{\theta}{2}|0\rangle + e^{\imath\phi}\sin\frac{\theta}{2}|1\rangle\right) \tag{1.40}$$

where the term $e^{\imath\phi}$ is called the *relative phase* of z, whereas $e^{\imath\gamma}$ represents its *global phase*. Due to their physical meaninglessness, global phase terms are often factored out and ignored in calculations. Moreover, observe that this rewriting allows for a nice visual representation of a qubit, as shown in Figure 1.17.

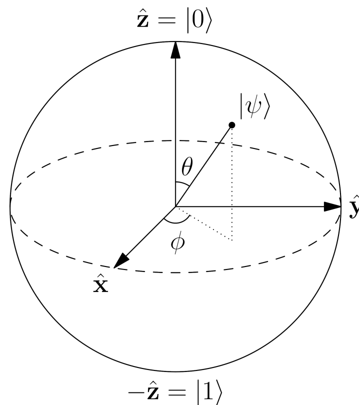


Figure 1.17: The *Bloch sphere* allows to visualize the state of a single qubit as a point on the surface of a 3-dimensional sphere with unitary radius.

1.6.2 The Postulates of Quantum Mechanics

The *quantum mechanical theory* defines 4 fundamental postulates that describe how quantum systems behave and interact.

Description of a Quantum System

The first postulate of quantum mechanics states what follows:

> The state of a quantum mechanical system is completely specified by a function $\Psi(\vec{r}, t)$ that depends on the spatial coordinates $\vec{r}$ of the particle(s) that compose the system and on time t. Such function, called the *wave function*, has the important property that $\Psi^*(\vec{r}, t)\Psi(\vec{r}, t)\, d\tau$ is the probability that the system lies in the volume element $d\tau$ located at $\vec{r}$ at time t.
>
> Since probabilities must add up to 1, the following normalization constraint must hold:
>
> $$\int_{-\infty}^{\infty} \Psi^*(\vec{r}, t)\Psi(\vec{r}, t)\, d\tau = 1 \tag{1.41}$$

When a time instant t is fixed, the postulate directly implies that every isolated physical system is associated with a *separable Hilbert space* in the complex field, and

that the state of such system is entirely described by a unit vector $|\psi\rangle$ called the *state vector*.

Recalling the definition of a qubit given in Section 1.6.1, we can now better understand its meaning: a qubit is nothing less than an algebraic description of a single-particle, isolated (quantum) physical system. Additionally, notice that the constraint reported by Equation 1.41 directly translates to the normalization requirement on the coefficients of the linear combination shown in Equation 1.39.

System Evolution

The second postulate of quantum mechanics defines how quantum systems evolve in time:

> To every *observable* in classical mechanics there corresponds a linear, *Hermitian*[10] operator in quantum mechanics. The wave function of a system evolves in time according to the time-dependent Schröringer equation:
>
> $$\hat{H}\Psi(\vec{r}, t) = \imath\hbar\frac{\partial\Psi}{\partial t} \tag{1.42}$$
>
> where $\hat{H}$ is an observable related to the total energy of the system, and called the *Hamiltonian*.

Despite the general formulation of the postulate, to perform computations it is usually more convenient to consider the evolution of a system in a discrete time framework. Under such condition, the time evolution of a system can be entirely described by a *unitary transformation*. We recall that a matrix is said to be unitary whenever $U^\dagger U = UU^\dagger = I$.

In the discretization process, unitary transformations and Hamiltonians are related according to the following equation:

$$U = \exp\left(-\imath\frac{\hat{H}t}{\hbar}\right) \tag{1.43}$$

where t indicates a time interval. Notice that the constraint on U being unitary is a direct consequence of $\hat{H}$ needing to be Hermitian.

Example 1.5 (Common gates)**.** In the context of quantum computations—where, if we assume to use the circuit computational model, unitary operators are also referred to as *gates*—it is often useful to define a small, well-known set of fundamental operators, which can then be composed in order to apply more complex transformations. Some gates may be more useful than others, but among the most used ones there are *Pauli operators* and the *Hadamard transform*:

$$X := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad Y := \begin{pmatrix} 0 & -\imath \\ \imath & 0 \end{pmatrix} \qquad Z := \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \qquad \mathcal{H} := \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{1.44}$$

[10] A matrix A is said to be *Hermitian*—or, equivalently, *self-adjoint*—if $A = A^\dagger$.

Remark 1.5. One of the main consequences of this postulate is that every quantum operator—and therefore every quantum algorithm—must be *reversible*, i.e., unitary. Among the implications of this constraint, an important result is the *No-Cloning Theorem*, which states that it is impossible to create an independent and identical copy of an arbitrary, unknown quantum state. For more details about the theorem refer to [133].

Measurements

In the introduction to this section we mentioned that the coefficients of a state vector associated to a qubit cannot be measured, as the state itself *collapses* when observed. The third postulate of quantum mechanics formalizes this claim:

> In any measurement of the observable associated with operator $\hat{A}$, the only values that will ever be observed are its eigenvalues $\overline{a}_i$, that satisfy the eigenvalue equation:
>
> $$\hat{A}\,\Psi_i = \overline{a}_i \Psi_i \tag{1.45}$$
>
> If the initial state[11] of the system is:
>
> $$\Psi := \sum_i c_i \Psi_i \tag{1.46}$$
>
> then a measurement of the observable will yield the eigenvalue $\overline{a}_i$ with probability $|c_i|^2$. Moreover, after the measurement, the state of the system *collapses* into the eigenstate Ψ_i corresponding to the measured $\overline{a}_i$.

Remark 1.6. Observe how this *quantization* property of measurements is the one that gives its name to the quantum mechanical theory.

In terms of state vectors, this postulate implies that any measurement operation is associated with a set of operators[12] $\{M_m\}$ such that:

$$\sum_m M_m^\dagger M_m = I \tag{1.47}$$

Then, given a quantum system in a state $|\psi\rangle$:

- the probability of measuring a specific value $\overline{m}$ is given by:

$$p(\overline{m}) := \langle\psi|\, M_{\overline{m}}^\dagger M_{\overline{m}}\, |\psi\rangle \tag{1.48}$$

- after the measurement, the system is left in the state:

$$|\psi'\rangle := \frac{M_{\overline{m}}\,|\psi\rangle}{p(\overline{m})} \tag{1.49}$$

[11] As shown in Equation 1.46, notice that any state Ψ can be always expressed as a linear combination of the eigenstates Ψ_i of $\hat{A}$, because they form a basis of $\mathbb{C}^n$ due to $\hat{A}$ being Hermitian.

[12] Measurement operators are intrinsically different from operators that describe the evolution of a quantum system, even if they are both represented with matrices. In fact, measurement operators aren't constrained to be unitary (and usually they are not).

Remark 1.7. The constraint expressed by Equation 1.47 is equivalent to stating that the probabilities of measuring each value must add up to 1:

$$\sum_m p(m) = 1 \tag{1.50}$$

Example 1.6 (Projective measurements)**.** During computations, it is often useful to design measurements that consist in *projecting* an unknown state onto the vectors of the basis in use. For instance, if we consider the computational basis shown in Equation 1.37, the following two operators implement a *projective measurement*:

$$M_0 := |0\rangle \langle 0| \qquad\qquad M_1 := |1\rangle \langle 1| \tag{1.51}$$

Given a generic state $|\psi\rangle := \alpha |0\rangle + \beta |1\rangle$, the probabilities of measuring 0 or 1 are respectively the following:

$$\begin{aligned} p(0) &= (\alpha^* \langle 0| + \beta^* \langle 1|) \cdot (|0\rangle \langle 0|)^\dagger \cdot (|0\rangle \langle 0|) \cdot (\alpha |0\rangle + \beta |1\rangle) = \alpha^* \alpha = |\alpha|^2 \\ p(1) &= (\alpha^* \langle 0| + \beta^* \langle 1|) \cdot (|1\rangle \langle 1|)^\dagger \cdot (|1\rangle \langle 1|) \cdot (\alpha |0\rangle + \beta |1\rangle) = \beta^* \beta = |\beta|^2 \end{aligned} \tag{1.52}$$

Remark 1.8. More in general, observe that since projectors can be expressed as $M_j := |j\rangle \langle j|$, where j is one of the states of some basis, then:

$$M_j^\dagger M_j = |j\rangle \langle j|j\rangle \langle j| = |j\rangle \langle j| = M_j \tag{1.53}$$

Composite Systems

The fourth postulate of quantum mechanics specifies how to describe a composite quantum system in terms of its components:

> The Hilbert space associated with a composite quantum system is the Hilbert space *tensor product* of the spaces associated with the component systems:
>
> $$|\psi\rangle := \bigotimes_i |\psi_i\rangle \tag{1.54}$$

However, not every composite system is expressible as a tensor product of simpler components. States with such property are said to be *entangled*.

Example 1.7 (Multiple-qubit gates)**.** Equation 1.44 showed some examples of commonly used single-qubit gates. However, since those alone are not sufficient to represent any possible quantum computation, multiple-qubit gates are important as well.

A common example of gates acting on multiple qubits simultaneously are *controlled gates*, i.e. gates that apply some operator U to a quantum state conditionally to the state of some other qubit[13]. For instance, consider the CNOT gate, that is a controlled

[13]This statement is only partially true: the described behaviour applies only when the control qubit is either $|0\rangle$ or $|1\rangle$, but then the operator acts linearly for all other states in superposition.

X gate:

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{1.55}$$

1.7 Quantum Computing Paradigms

This section is devoted to the introduction and description of three different Quantum Computing paradigms. We chose to focus on these particular models:

- Gate-Based
- Measurement-Based
- Adiabatic

Clearly, they are not the all and only paradigms adopted to *achieve* quantum computation. For example, we are not going to talk about Topological Quantum Computation, that nowadays is gaining a lot of popularity due to its intrinsic fault tolerance.

1.7.1 Gate-Based Quantum Computation

This model is by far the most adopted in the manufacturing of physical quantum computers. As it can easily be understood by its name, the main ingredients of this framework are called *gates*. It composes gates to obtain *circuits*, which are direct translations of quantum algorithms.

Formally speaking, we start with a unitary $U \in \mathbb{C}^N \times \mathbb{C}^N$—$N = 2^n$—that describes some particular quantum procedure. We put a *wire* for each one of the n qubits. We suppose each qubit to initially be in state $|0\rangle$. Furthermore, we can now apply *gates* to the qubits to change their state. In this sense, there is a one to one correspondence between gates and unitary matrices. Nevertheless, not all unitaries can be physically manufactured and plugged into quantum circuits. Hence, U must be synthesized in terms of smaller gates called *universal base set* $\mathcal{B}$. For this example, we assume $\mathcal{B}$ is the Clifford+T set which contains the following single qubit gates:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \qquad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}$$

and the two-qubit gate CNOT:

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

With the following effect:

$$CNOT\,(a, b) = (a, a \oplus b)$$

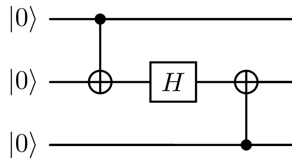


Figure 1.18: A simple Quantum circuit composed by three qubits.

where a, b are the *control* and the *target* of the CNOT gate, respectively.

In Figure 1.18 we depicted a small circuit. All the three qubits are initialized to state $|0\rangle$. Hadamard gate (H) is shown as a box containing the letter H. The other symbol—connecting two different qubits, with a cross on one end and a bullet in the other—is used to introduce a CNOT gate.

All the most known quantum algorithms, like Shor and Grover, can be described with a finite quantum circuit. Moreover, with the process of *synthesis*, any algorithm that has been previously *summed up* to a single unitary, can be unravelled as an ordered set of operations. There exists a tight relation between synthesis and universality in quantum computation. A set of gates S is called universal if and only if we can synthesize any unitary matrix up to any precision

The following theorem characterizes a particular family of gates that are known to be *universal*[14]

Theorem 5. *The set composed by all single-qubit unitary gates and* C-NOT *is universal for quantum computations [133].*

The reader can find more details on the Gate-Based model of computation in [133].

1.7.2 Measurement-Based Quantum Computation

The models of quantum computation that we described in Section 1.7.1–quantum circuits–are direct generalizations of a classical paradigm. In particular, the computation in the model can be split between a fully quantum execution, followed by a measurement. Hence, the translation from quantum to classical information is done only at the end of the computational process. Moreover, the quantum circuit model strongly resembles classical architectures.

On the other hand, measurements are destructive operations and may lead to a great loss of information about a quantum state. Hence, it is pretty peculiar that we can perform universal quantum computation with just measurement operations [150]. Models of quantum computation based only on measurements are interesting for a really fundamental issue: they have no classical counterpart.

Two main Measurement-Based models have been proposed in the literature. The first one, *Teleportation Quantum Computation* has been proposed in [82] and subsequently investigated in [132]. The second one, *One-Way Quantum Computer* (1WQC) or equivalently *cluster state computation* was first introduced in [150]. We will focus on

[14] By means of which we can describe any quantum algorithm.

the second one, since it is considered the *de facto* Measurement-Based model of quantum computation.

From this moment, we will use *One-Way Quantum Computer* and *Measurement-Based Quantum Computer* as synonyms. The input state of a One-Way Quantum Computer is the so-called *cluster state*. The key property of such state is that it is mainly composed of entangled qubits. We now introduce an example of how a cluster state is generated for the 4 qubits case. This procedure can be easily generalized.

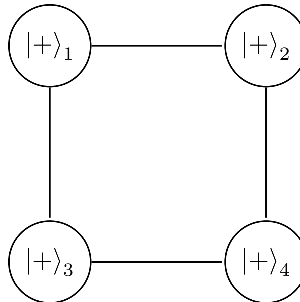


Figure 1.19: A 2×2 cluster state.

Consider a 2×2 square grid composed of 4 qubits as in Figure 1.19—each node is a qubit. Each qubit has been initialized to the state $|+\rangle$ defined as:

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

The state of the qubits is then modified by applying a CZ to each pair of nodes connected by an edge. Considering Figure 1.19, we apply a CZ gate between qubit (1, 2), (1, 3), (2, 4) and (3, 4).

Notice that we explained how the cluster state is generated using a simple 2×2 square. However, a cluster state can be generated from any initial connectivity structure. Let $G = (V, E)$ be an undirected graph, where $n = |V|$. A generic procedure to initialize the cluster state respecting the constraints imposed by G is the following:

- Associate a qubit to each node of the graph.
- Create an initial state $|\psi_0\rangle$ defined as:

$$\bigotimes_{v \in V} |0\rangle_v$$

  where $|0\rangle_v$ means that the qubit assigned to $v \in V$ is set to $|0\rangle$.

- For each edge $\{u, v\} \in E$, apply a CZ gate to the qubits associated with nodes u, v.

Measurement-Based quantum computation is based on the fact that any quantum gate array can be implemented as a set of single qubit measurement operations on a

cluster state[102]. The measurements will be performed on two possible bases. The former is the set $\{|0\rangle, |1\rangle\}$. The latter is the generator $\mathcal{B}(\theta)$ defined as:

$$\mathcal{B}(\theta) = \{|0\rangle \pm e^{i\theta}|1\rangle\}$$

for some angle θ.

The complete explanation of the underlying functionality of Measurement-Based quantum computing is out of the scope of this thesis. We just show an example of how a generic 1-qubit gate U can be applied to a state $|\psi\rangle$ in the Measurement-Based setting. The reader may refer to [102] for further details and examples.

The following example is taken from [150]. Consider a generic 1-qubit gate U. Firstly, we remind the fact that any 2×2 unitary U can be rewritten as:

$$U = R_x(\alpha)R_z(\beta)R_x(\gamma)$$

for some angles α, β, γ. With $R_{\hat{n}}(\theta)$ we denoted a rotation on axis $\hat{n}$ of an angle θ.

We want to apply U to a generic 1-qubit state $|\psi\rangle$.

In Figure 1.20 we depicted the sequence of measurements required to perform such operation. The cluster state is composed by 5 qubits, where the leftmost one is initialized to $|\psi\rangle$ and the remaining ones are all set to $|+\rangle$. The horizontal line connecting the qubit denotes a CZ operation as described above. Under each one of the nodes we explicitly defined the measurements applied to each qubit—with s_i we denote the result of the measurement on the i-th qubit.

At the end of the computation, the rightmost qubit is in state $X^{s_2+s_4}Z^{s_1+s_3}U|\psi\rangle$.

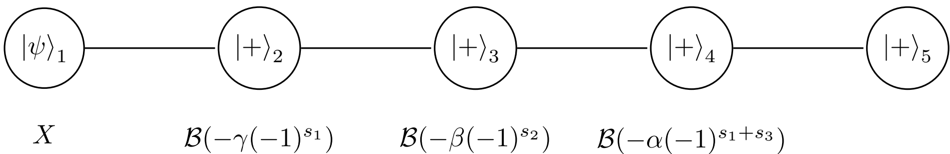


Figure 1.20: Apply 1-qubit gate U to state $|\psi\rangle$.

1.7.3 Adiabatic Quantum Computation

Adiabatic Quantum Computation has been taken into account as a computational model since [66]. Authors adopted a novel quantum algorithm, based on adiabatic effect, to solve SAT—a classical optimization problem. The overall behaviour of such algorithm relied on the *adiabatic theorem*, which can be roughly stated as follows [104]:

A physical system will remain in its lowest-energy state under sufficiently slow changes of external parameters

A computation in this model is described by a pair of Hamiltonian—Hermitian matrices—named H_{init} and H_{final}. Let v_λ be the eigenvector of H_{init} with the smallest eigenvalue. v_λ is denoted as *ground state*, and it is usually the initial state of the computation. Hence, it should be easy to prepare. The output of the computation is

the *ground state* of H_{final}. Therefore, the choice of H_{final} must be done in order to assure that its ground state is the solution to the problem we are tackling.

With *spectral gap* we refer to the difference between the lowest and the second-lowest eigenvalue of a Hamiltonian. The running time of an adiabatic quantum computation is defined as the minimal spectral gap between all the possible Hamiltonians generated as a linear combination of H_{init} and H_{final}:

$$H(s) = (1-s)H_{\text{init}} + sH_{\text{final}} \ \text{ with } s \in [0,1]$$

The adiabatic computation is polynomial if this minimal spectral gap is at least inverse polynomial. Practically, the quantum system is usually initialized to the ground state of H_{init}, which is also applied to such state. The Hamiltonian is then modified (slowly) in the line towards H_{final}. Thanks to the adiabatic theorem, if these changes are performed slowly enough, at the end of the computation the state of the system will be the ground state of H_{final}.

The overall computational power of Adiabatic Quantum Computation is the same as standard quantum computational models. In [164] it was proved that any adiabatic quantum computation can be efficiently simulated by means of standard quantum techniques. The other way round was proved in [3]. The result they obtained is what follows:

The model of adiabatic quantum computation is polynomially equivalent to the standard model of quantum computation

Problems to be solved on Adiabatic quantum models are usually defined via *Quadratic Unconstrained Binary Optimization* (QUBO) encodings. A QUBO problem is defined using *quadratic* functions of *binary* variables x_i. Constraints for such variable are not *strict* as for example in Integer Linear Programming. On the other hand, they are encoded as penalty terms in a matrix Q. Such matrix is the objective function of the optimization (The reader may refer to [110] for further details).

The final task of the quantum annealing process is to find a value x^* such that:

$$x^* = \min_x x^T Q x$$

where with x we indicate a vector of binary variables, while Q is an upper triangular real valued matrix. The set of problems solvable via QUBO are NP-hard, as shown in [141, 21]. Moreover, they share the same structure/computational complexity with a famous problem coming from physics: spin glass Ising models [124]. This equivalence allows a quasi immediate translation of QUBO objective functions into the adiabatic model.

Quantum Annealing is a meta-heuristic search algorithm that can be used to tackle QUBO problems [14]. It usually considered to be the quantum counterpart of the classical simulated annealing optimization technique. Quantum Annealing works by mean of a pair of Hamiltonian H_0 and H_1—usually referred to as initial and final hamiltonian, respectively—that do not commute. Using a control function $\lambda(t) : \mathbb{R} \to [0,1]$—where t describes a time dependency—a *total* Hamiltonian can be defined as follows:

$$H(t) = (1-\lambda(t))H_0 + \lambda(t)H_1$$

If the landscape of the function λ is such that the hypothesis of the adiabatic theorem are respected, Quantum Annealing algorithms are in fact adiabatic quantum computations.

Graphs and Their Representations

Introduction to Graphs and Their Representations

Most students entering a Computer Science program are, or should be, aware that they will eventually encounter an Algorithms and Data Structures course. At University of Udine, this course is taught by my current supervisor, who introduces students each year to foundational topics such as complexity, arrays, sorting algorithms, trees, and heaps, to name a few. Part of this course are graphs as well. The very first question that is answered when introducing them to the students is the following:

How do we store them?

Without an answer to this question, no graph algorithm could ever have been devised.

Thus, this part is devoted to answering this fundamental question. We will start within the realm of classical computation, exploring familiar data structures such as adjacency matrices and adjacency lists. Next, we shift to parallel computation, where we examine specialized encodings suited for parallel environments. Finally, we will turn to quantum computation, where encoding a graph is far less straightforward than in classical settings. In such chapter, we will show that entire classes of graphs cannot be directly encoded within the quantum computing framework without some form of manipulation[15].

[15] Oh, unitarity, thou art heartless...—(partially)Sheldon Cooper grappling with gravity and a couch.

2

Graphs in Classical Architectures

The purpose of this chapter is to review foundational definitions and techniques about graphs manipulation in the classical settings. By doing so, we will reinforce knowledge that will serve as a core building block for the more advanced discussions and applications reviewed throughout the rest of the thesis. By establishing a solid base on classical graph storage and manipulation, we set the stage for more sophisticated treatments.

Let $G = (V, E)$ be a graph as defined in Section 1.2. The most common representations adopted to store and manipulate G in classical architectures are the following:

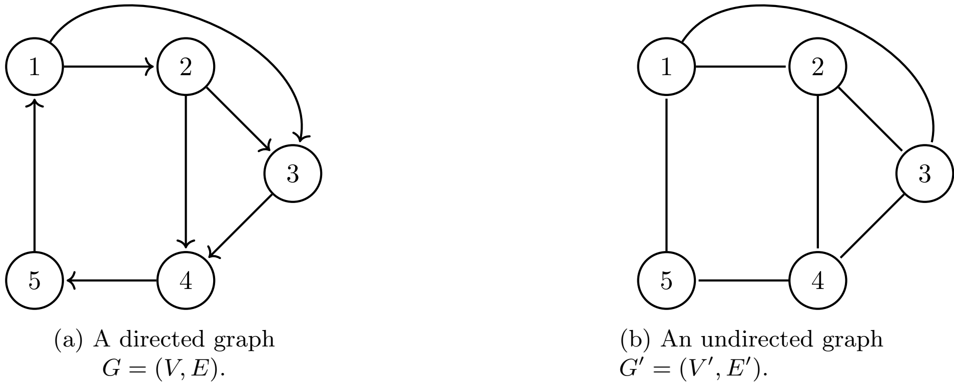
- Adjacency matrices
- Adjacency lists
- Ordered Binary Decision Diagrams

2.1 Adjacency Matrix

We begin this section by recalling a definition we already provided in 1.22 when first introducing theoretical notions of graphs. A graph $G = (V, E)$ with $n = |V|$ can be exhaustively described by its *adjacency matrix* $M(G) \in \{0, 1\}^{n \times n}$, where

$$M(G)_{u,v} = \begin{cases} 1 & \text{if } (u, v) \in E; \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

Unless otherwise specified for a given graph $G = (V, E)$ it is assumed that vertices are labelled $V = \{1, 2, \ldots, n\}$ and, in turn, that the order of rows and columns of $M(G)$ be following the order given by the nodes labels. Hence, in Equation 2.1 the names of the nodes u, v can be considered to be natural numbers in the range $\{1, 2, \ldots, n\}$. Moreover,



(a) A directed graph $G = (V, E)$.

(b) An undirected graph $G' = (V', E')$.

Figure 2.1: A graph G and its undirected version G'.

when no ambiguity concerns the graph under consideration, we refer to $M(G)$ simply as M.

Using Definition 2.1, the adjacency matrix of the graph G depicted in Figure 2.1a is as follows:

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

In the case that G is undirected, edges have no direction. Hence, they can be traversed back and forth. This property is encoded as follows in the adjacency matrix description:

$$M_{i,j} = M_{j,i} \quad \forall i, j \in V$$

which is, in *linear algebra* terms, that M is symmetric. Using graph G' depicted in Figure 2.1b as a witness, we get that its adjacency matrix is:

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

To store the matrix M we require exactly one bit for each pair (i, j), with $i, j \in V$. Hence, a total of n^2 bits which is $\Theta(n^2)$ space in asymptotic notation.

Let us now investigate how the basic information about a node $v \in V$ can be retrieved from M. In the first case, let G be an undirected graph. The out-neighbourhood of v is equivalent to:

$$\delta^+(v) = \{u : M_{v,u} = 1\}$$

Roughly speaking, is the set of nodes u for which the element in position (v, u) of M is non-zero. This is clearly true from the definition of M.

Since G is undirected, it holds that $\delta^+(v) = \delta^-(v)$. Hence, the set $\{u : M_{v,u} = 1\}$ is also the *in-neighbourhood* of v. From this definition, it is simple to see that $d^+(v) = |\delta^+(v)|$ is the number of ones in the v-th row of M.

Again, we consider G' from Figure 2.1b to give an example of the notions we just described. Let $v = 2$ be the node we take into account.

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.2}$$

In matrix 2.2 we highlighted in blue the row associated to v. Interpreting the vector we obtained, we can say that the set of nodes to which v is connected is $\{1, 3, 4\}$. In the same matrix, we highlighted in red the column associated to v. In this case we obtain v's in-neighbourhood. By the way, since the graph is undirected, column and row associated to v are equal.

If we shift our attention to a directed graph G, its adjacency matrix is no more symmetric. Let v be a node of G. Its out-neighbourhood is defined as:

$$\delta^+(v) = \{u : M_{v,u} = 1\}$$

while its in-neighbourhood is:

$$\delta^-(v) = \{u : M_{u,v} = 1\}$$

Differently from the undirected case, the two sets cannot be supposed to be identical. Therefore, the out-degree and the in-degree of v are the number of ones in the v-th row and v-th column, respectively.

As for the undirected case, we now use an example to clarify what we just described. Consider graph G depicted in Figure 2.1a and let $v = 2$ be the node we take into account.

$$M = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{2.3}$$

In matrix 2.3 we highlighted the row and the column associated to node $v = 2$. The out-neighbourhood of v is identified by the non-zero entries of the blue row—nodes 3 and 4. On the other hand v's in-neighbourhood is just node 1 since in the red column only position $(1, 2)$ is non-zero.

Despite being *simple* to retrieve an entire row/column of a matrix, we want to stress what follows. Let v be a node of a graph $G = (V, E)$ and let r, c be the v-th row and column of $M(G)$, respectively. It holds that r is not the set $\delta^+(v)$ and c is not the set $\delta^-(v)$: this is due to the presence of the 0s in the matrix. To obtain the actual sets one have to go through the vectors r, c and keep only the entries with value 1. The time

complexity of this operation is $\Theta(|V|)$.

We saw how to turn a graph into a $\{0,1\}$ matrix and how to retrieve basic information about nodes and edges. Dealing with matrices, a natural question arises: if I apply a *linear algebraic* operation on an adjacency matrix $M(G)$, what is the effect of such operation in the graph G?

For example, consider two graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$ and let $M(G_1), M(G_2)$ be their adjacency matrices. Let $M(G_3) := M(G_1) + M(G_2)$. What we obtain is the adjacency matrix of the multigraph $G_3 = (V_3, E_3)$ resulting from the union of G_1 and G_2. It is easy to see that a generic element of $M(G_3)$, namely $M(G_3)_{u,v}$, can hold only 3 values [1]:

- $M(G_3)_{u,v} = 0$. Since both $M(G_1)$ and $M(G_2)$ have elements only in $\{0,1\}$, then it holds that $M(G_1)_{u,v} = M(G_2)_{u,v} = 0$. In graph terms we have that $(u,v) \notin E_1 \wedge (u,v) \notin E_2 \leftrightarrow (u,v) \notin E_3$
- $M(G_3)_{u,v} = 1$. In this case, it is easy to see that $(u,v) \in E_1 \oplus (u,v) \in E_2 \leftrightarrow (u,v) \in E_3$
- $M(G_3)_{u,v} > 1$. Due to this peculiar circumstance, G_3 is defined as a *multigraph*—it can have more a copy of the same edge. If both $(u,v) \in E_1$ and $(u,v) \in E_2$, then $M(G_1)_{u,v} + M(G_2)_{u,v} = M(G_3)_{u,v} = 2$. Therefore, E_3 is a multiset with two copies of the edge (u,v)

When defining $M(G_3)$, we supposed V_1 and V_2 to be of the same size (otherwise the dimension of the matrices would not fit for a sum operation). If this precondition does not hold, then it is true that $|V_1| < |V_2|$—the other case is symmetric. Let $\Delta = |V_2| - |V_1|$ be the difference in size between the nodes of the two graphs. Perform a *normalization* by adding Δ all-zero columns and Δ all-zero rows to $M(G_1)$. After this step, $M(G_1)$ and $M(G_2)$ have the same size.

Example 2.1. We use this small example to give a better explanation of what we just discussed.

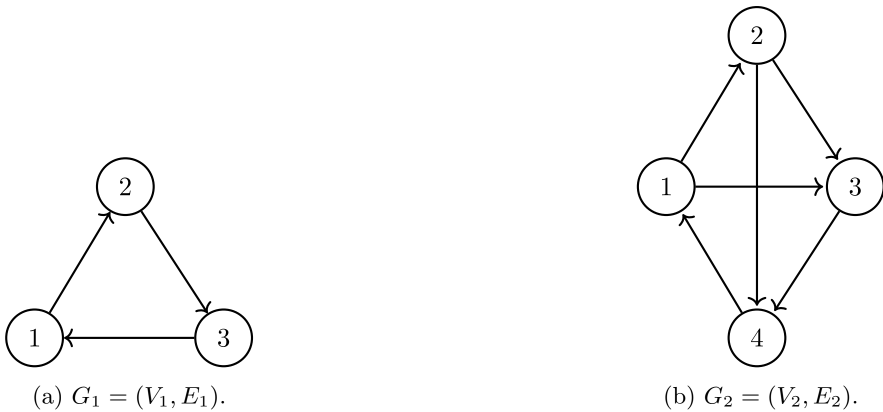
The adjacency matrices of the two graphs depicted in Figure 2.2a and Figure 2.2b are as follows:

$$M(G_1) = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \quad M(G_2) = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Since $|V_1| < |V_2|$, we have to normalize it before summing. What we obtain is:

$$M(G_1) = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

[1] We describe the possible results on directed graphs. Nevertheless, the same conditions hold for the undirected case too.



(a) $G_1 = (V_1, E_1)$.

(b) $G_2 = (V_2, E_2)$.

Figure 2.2: Two directed graphs G_1 and G_2.

where we highlighted the *portion* of matrix we had to add to normalize $M(G_1)$. We can now compute $M(G_3)$ as $M(G_1) + M(G_2)$:

$$M(G_3) = \begin{pmatrix} 0 & 2 & 1 & 0 \\ 0 & 0 & 2 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

The graph G_3 obtained by *interpreting* $M(G_3)$ is depicted in Figure 2.3. We coloured in red the edges coming from G_1 while in blue the ones from G_2.

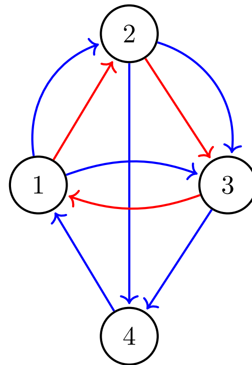


Figure 2.3: Graph $G_3 = (V_3, E_3)$ obtained by *interpreting* $M(G_3)$

.

We now investigate the effect of a matrix product. In this case, we consider only one graph $G = (V, E)$ together with its adjacency matrix M. The question we pose is the following: what is the graph obtained by interpreting the result of $M^2 = M * M$? The answer is very peculiar: the value of $M^2_{u,v}$, with $u, v \in V$ represents the number of paths of length 2 starting from u and ending in v.

Example 2.2. Consider the adjacency matrix $M = M(G_2)$ where G_2 is the graph depicted in Figure 2.2b. The result of the product between M and itself is:

$$M^2 = \begin{pmatrix} 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

The (ordered) pair of nodes u, v that have a length two paths connecting them are: (1, 3), (1, 4), (1, 4), (2, 1), (2, 4), (3, 1), (4, 2), (4, 3). Which are exactly the indexes of the non-zero elements of M^2.

This kind of behaviour is not isolated to the case of M^2. Computing the i-th power of M, namely M^i, then element (u, v) represents the number of paths of length i connecting u and v—notice that in the case of directed graphs, the number of paths from u to v may differ from the number of paths from v to u.

Clearly, sum and product are not the only operations one can perform on matrices. Let $G = (V, E)$ with $|V| = n$ be a graph and M its adjacency matrix. Consider a column vector $x \in \{0, 1\}^n$ and let $y = Mx$. What is, in graph terms, the meaning of y? Assume x has only one non-zero element in position u. Then it holds that:

$$y_v = 1 \leftrightarrow v \in \delta^-(u)$$

Roughly speaking, y is a boolean vector encoding the in-neighbourhood of node u.

Example 2.3. Consider the adjacency matrix $M = M(G_2)$ where G_2 is the graph depicted in Figure 2.2b. Let $x = (0\,0\,1\,0)^T$ be a vector encoding that we are currently visiting node 3—$x_3 = 1$. If we compute the product $y = Mx$, what we obtain is:

$$y = Mx = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \{1, 2\} = \delta^-(3)$$

The case where x has more than a single non-zero element is straightforward. The vector $y = Mx$ represents the union of the in-neighbourhoods of the nodes described by x—notice that y may have elements that are neither 0 nor 1.

On the other hand, the post multiplication $z = qM$ behaves as follows. Let $q \in \{0, 1\}^n$ be a row vector with only one non-zero element in position u. Then, the following property holds for $z = qM$:

$$z_v = 1 \leftrightarrow v \in \delta^+(u)$$

Example 2.4. Consider the adjacency matrix $M = M(G_2)$ where G_2 is the graph depicted in Figure 2.2b. Let $q = (0\,1\,0\,0)$ be a vector encoding that we are currently visiting node 2—$x_2 = 1$. If we compute the product $z = qM$, what we obtain is:

$$z = qM = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}^T \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \{3, 4\} = \delta^+(2)$$

Hence, the pre multiplication allows moving from a node u to the elements of its out-neighbourhood. The case where q has more than a single non-zero value is handled in the same way as for the post multiplication.

Many more operations on adjacency matrices may be taken into account to describe their effect on the initial graph. Such topic is usually referred to as *Spectral Graph Theory*. We refer the reader to [53] for further details.

2.2 Adjacency Lists

We now focus on the second most common representation for graphs in classical architectures: adjacency lists.

We saw that with adjacency matrices, the encoding of a graph is:

- simple, since the definition of adjacency matrix is pretty straightforward
- information about nodes neighbourhood can be easily retrieved via matrices rows and columns
- moving to the *realm* of linear algebra opens up a series of peculiar aspects

Nevertheless, we mentioned only once the computational complexity of dealing with such a representation. In particular, to simply store a graph $G = (V, E)$ with n nodes, exactly n^2 bits are required—$\Theta(n^2)$ space in asymptotic notation. Let v be a node of G with adjacency matrix $M = M(G)$. Retrieving $\delta^+(v)$ requires time $\Theta(n)$: we need to obtain the v-th row of M and then search those u for which $M_{v,u} = 1$. The case for $\delta^-(v)$ is symmetric—goes through columns instead of rows.

Nevertheless, a lot of information is *useless*: if the graph is sparse, then a lot of elements in its adjacency matrix will be 0. Broadly speaking, to spare some space and time complexity, one approach is to get rid to the 0 elements and just store information about edges that are actually part of the graph.

This particular idea is exploited when considering adjacency list to store a graph $G = (V, E)$. An adjacency list for G is an array L with $|V| = n$ elements, one per node. Each cell $L[u]$, with $u \in V$, is a list that contains only the nodes in the out-neighbourhood of u.

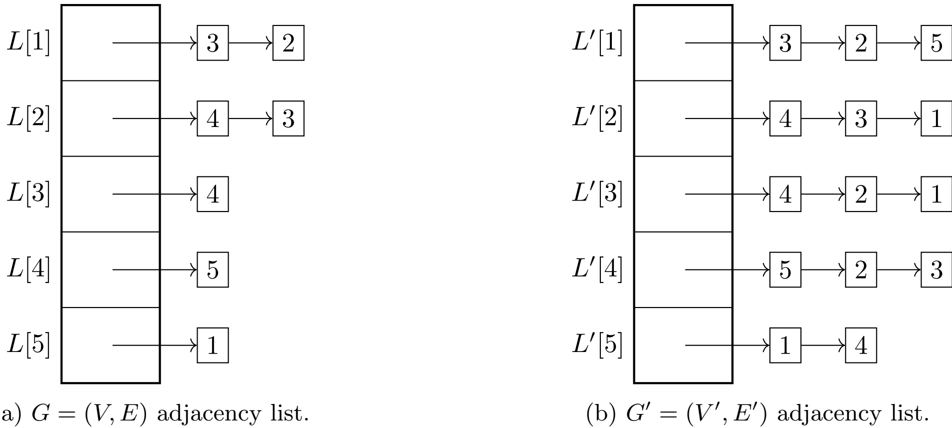
Example 2.5. Consider graphs G, G' depicted in Figure 2.3. Their adjacency lists are as follows:

We can use Figure 2.4a and Figure 2.4b to introduce some properties about adjacency lists.

We start with the undirected case—Figure 2.4b. Considering that edges are not oriented and can be traversed in both directions, the following property holds about the elements $L'[u]$:

$$v \in L'[u] \leftrightarrow u \in L'[v]$$

Due to this particular property, the list $L'[u]$ describes both $\delta^+(u)$ and $\delta^-(u)$—the out-neighbourhood and the in-neighbourhood. Therefore, the size of u's neighbourhood is the length of the list $L'[u]$. Using this description, the space complexity of storing L' is:



(a) $G = (V, E)$ adjacency list.

(b) $G' = (V', E')$ adjacency list.

Figure 2.4: G, G' adjacency lists.

- $|V|$ space is used to store the array
- $2|E|$ space is allocated for the elements inside the lists. Notice that the factor 2 is necessary since every edge is repeated.

Therefore, we obtain a total of $\Theta(|V|) + \Theta(|E|) = \Theta(|V| + |E|)$. Since in a graph with n nodes there can be at most n^2 different edges, then the overall space required is $\mathcal{O}(n^2)$. Notice that in this case, if the graph is highly sparse, we will need much less space with respect to the adjacency matrix.

We now move to the directed case—Figure 2.4a. Since edges in this case cannot be traversed back and forth, they are not repeated when stored in L.

This being the case, the computation of $\delta^+(u)$ differs from the computation of $\delta^-(u)$, with $u \in V$. In particular, $L[u] = \delta^+(u)$ by definition. Therefore, it can be obtained in time $\Theta(1)$. On the other hand, computing $\delta^-(u)$ requires checking inside the whole array L. More precisely, Algorithm 2 has to be applied.

Algorithm 2 Find the in-neighbourhood of a given node in a graph

```
1: function In-Neighborhood(L, u)
2:     δ⁻(u) ← ∅
3:     for all v ∈ V do
4:         if u ∈ L[v] then
5:             δ⁻(u) ← δ⁻(u) ∪ {v}
6:         end if
7:     end for
8:     return δ⁻(u)
9: end function
```

The complexity of such algorithm is mainly due to two factors:

- Running over all the nodes v of the graph at line 3
- Checking for the presence of u inside each list $L[v]$, at line 4

The complexity of running over all nodes in V is $\Theta(|V|)$. On the other hand, checking for the presence of a node u in $L[v]$ has to be carefully handled. In particular, one may *overestimate* that each time the check is performed, it requires $\mathcal{O}(|E|)$ time—we need to go through the whole list $L[v]$ which has length $\mathcal{O}(|E|)$. With this result, the overall complexity of Algorithm 2 is $\mathcal{O}(|V||E|)$ which is in the worst case cubic on the number of nodes.

However, rethinking about the check at line 4, we see that its actual complexity can be better estimated. Let v be the current node in the for loop at line 3. Then we can answer the question $u \in L[v]$ in time $\mathcal{O}(|\delta^+(v)|)$—it was the quantity $|\delta^+(v)|$ to be overestimated in our previous analysis. Moreover, consider one particular edge $e = (w, z) \in E$. It is easy to see that e can belong only to $\delta^+(w)$—where with *belong* we mean that only the presence of node z in $\delta^+(w)$ can prove the existence of e. Hence, every edge can be *seen* only once.

Now, suppose to unravel the for loop at line 3, creating $|V|$ different copies of the if statement at line 4. Each copy will be bound to one particular node. Therefore, we will have n different copies of such if. The overall pseudocode can be found in Algorithm 3.

Algorithm 3 Find the in-neighbourhood of a given node in a graph

```
 1: function In-Neighborhood-Unravelled(L, u)
 2:     δ⁻(u) ← ∅                                    ▷ Let V = {1, 2, ··· , n}
 3:     if u ∈ L[1] then
 4:         δ⁻(u) ← δ⁻(u) ∪ {1}
 5:     end if
 6:     if u ∈ L[2] then
 7:         δ⁻(u) ← δ⁻(u) ∪ {2}
 8:     end if
 9:     ···
10:     if u ∈ L[i] then
11:         δ⁻(u) ← δ⁻(u) ∪ {i}
12:     end if
13:     ···
14:     if u ∈ L[n] then
15:         δ⁻(u) ← δ⁻(u) ∪ {n}
16:     end if
17:     return δ⁻(u)
18: end function
```

If we now study its complexity, we notice that each if statement costs $\mathcal{O}(|\delta^+(i)|)$, where i is the current node index. Since each edge *belongs* to only one δ^+ we get that:

$$\sum_{v \in V} |\delta^+(i)| = |E|$$

Hence, the overall complexity of the for loop at line 3 of Algorithm 2 is $\mathcal{O}(|E|)$.

The reader may notice that using this very idea, whenever an algorithm requires going through all the edges of a graph:

for all $e \in E$ **do**

in the adjacency list paradigm this can be rewritten as:

for all $v \in V$ **do**
  for all $u \in L[v]$ **do**

We recall that in the adjacency matrix approach, sets δ^+ and δ^- of a given node are computed in time $\mathcal{O}(|V|)$.

While with the adjacency matrix approach we could use matrix operations to obtain information about the initial graph, this is not possible in the adjacency list case.

2.3 Ordered Binary Decision Diagrams

As we mentioned before, graphs are usually encoded either with *adjacency matrix* or *adjacency lists*. In this section we will present an encoding technique based on OBDDs for graphs.

Binary Decision Diagrams (*BDDs*, for short) and their ordered counterpart (*OBDDs*) were introduced in 1986 by Bryant [43] as a data structure for representing boolean functions, taking inspiration from *Ordered Binary Decision Trees* but using graphs instead of trees in order to maintain a compact structure.

Definition 2.1 (Ordered Binary Decision Tree)**.** *[73]* The *Ordered Binary Decision Tree* for the boolean function $f : \{0,1\}^n \mapsto \{0,1\}$, with respect to the variable ordering $\pi = \langle x_1, \dots, x_n \rangle$, is the complete labelled binary tree in which:

- each internal node v, at depth i, is labelled $var(v) = x_i$ and has two outgoing edges labelled 0 (dotted edge) and 1 (not dotted edge), respectively;
- each leaf is labelled with a value $l \in \{0,1\}$;
- for all $\bar{x} \in \{0,1\}^n$ and $z \in \{0,1\}$, $f(\bar{x}) = z$ if and only if the only path of edges labelled $\bar{x}$ in the tree ends on a leaf whose label is z.

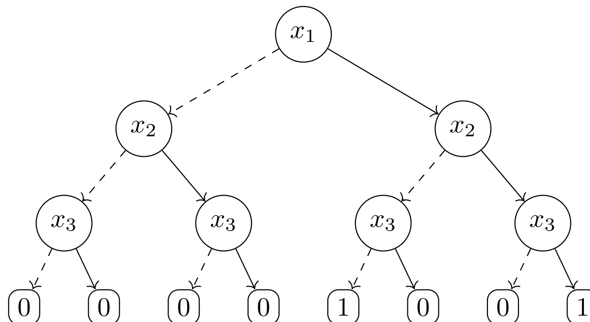


Figure 2.5: An example of Ordered Binary Decision Tree for $f = x_1 \wedge (x_2 \iff x_3)$, $\pi = \langle x_1, x_2, x_3 \rangle$.

Definition 2.2 (Ordered Binary Decision Diagram)**.** An *Ordered Binary Decision Diagram* for the boolean function $f : \{0,1\}^n \mapsto \{0,1\}$, with respect to the variable ordering $\pi = \langle x_1, \dots, x_n \rangle$, is a rooted acyclic directed labelled graph in which:

- the nodes set is $V = D \cup \{0,1\}$, where D is the set of *decision nodes*, i.e. nodes v_j labelled $var(v_j) = x_i$;
- each decision node has exactly two outgoing arches called the *low* and *high* edges, labelled with 0 and 1 respectively;
- for each path p starting from the root node and ending on a node $t \in \{0,1\}$, the following properties hold:
  - each variable x_i appears only once in the path, hence its length is at most n;
  - moreover, the variables appear in p in the order given by π;
  - for each possible assignment $\bar{y}$ of the variables $\{x_1, \dots, x_n\} \setminus \{var(p_j)\}$, where p_j are the decision nodes traversed by p, it holds that $f(\bar{x}, \bar{y}) = t$, where $\bar{x} \in \{0,1\}^k$, $k \leq n$ is the sequence composed by the labels of the edges traversed by p.

In other words, an OBDD is very similar to an Ordered Binary Decision Tree, except that its structure is a directed acyclic graph rather than a tree, and there is a strict total order placed on the occurrence of variables as one traverses the graph from root to leaf [46]. Also, notice that with Definition 2.2 we allow the existence of multiple OBDDs representing the same boolean function. In what follows, we will present a method for obtaining the most compact one, given a variable ordering π.

For what concerns the construction of an Ordered Binary Decision Tree $\mathcal{T}$, it can be built from a boolean formula f just by choosing a variable ordering and looking at the truth table for f. In order to get a valid OBDD, we can just replace all $\mathcal{T}$'s leaves with two single nodes – labelled 0 and 1 – and then redirect all the needed edges to the new nodes.

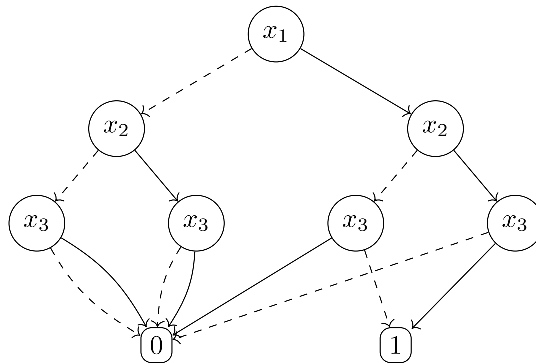


Figure 2.6: An OBDD directly derived from the tree in Figure 2.5.

However, notice that most paths can be simplified by eliminating redundant nodes and edges. In order to obtain a minimal OBDD, one can iterate the application of two basic rules until fixed point [73]:

1. *Remove Duplicate Vertices*

   If two vertices u and v are such that $var(u) = var(v)$, $low(u) = low(v)$, and $high(u) = high(v)$, then eliminate one of the two and redirect all incoming edges to the other node.

2. *Remove Redundant Tests*

   If a decision node v has $low(v) = high(v)$, then eliminate v and redirect all incoming edges to $low(v)$.

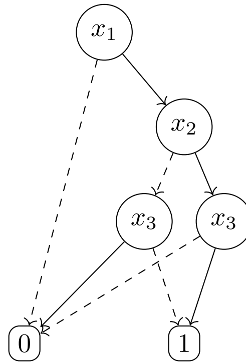


Figure 2.7: The reduced form of the OBDD in Figure 2.6.

It easily follows that given a boolean formula $f : \{0,1\}^n \mapsto \{0,1\}$ and a variable ordering $\pi = \langle x_1, \ldots, x_n \rangle$ it is always possible to build an OBDD, and there exists one that is minimal (i.e. number of nodes, Figure 2.7) [43, 46]. In the rest of this thesis, we will always refer to the minimal OBDD when considering this data structure.

It is also possible to build an OBDD by recursively applying a composition operation that takes two OBDDs for the formulas f and g, and builds another OBDD for $f\langle op\rangle g$, where $\langle op\rangle$ is a boolean operator. The algorithm, called Apply, is explained in detail in [73, Section 1.1.2], and allows to build an OBDD by traversing bottom-up the semantic tree representing a boolean formula.

2.3.1 The variable ordering problem

Even if the OBDD construction is always possible given a boolean formula, in [43] it has been shown that the size of the resulting data structure, measured as the number of its nodes[2], critically depends on the variable ordering π. In order to better analyze this behaviour, consider the following function:

$$f(x_1, x_2, x_3, x_4) = (x_1 \iff x_2) \vee (x_3 \iff x_4)$$

with respect to two different variable orderings:

$$\pi_1 = \langle x_1, x_2, x_3, x_4 \rangle \qquad \pi_2 = \langle x_1, x_3, x_2, x_4 \rangle$$

[2]Notice that this unit is actually a good index for measuring the size of an OBDD. In fact, the number of edges is always linear (specifically double) in the number of decision nodes.

By applying the Remove Duplicated Vertices and Remove Redundant Tests rules, one can build the two respective OBDDs (Figure 2.8).

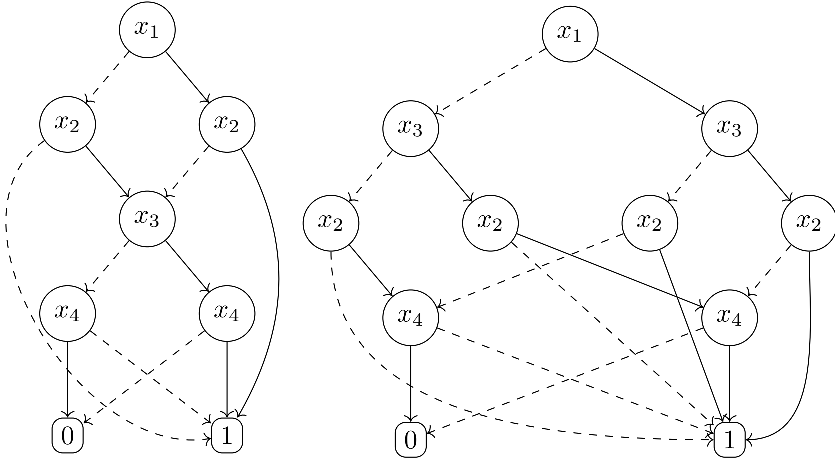


Figure 2.8: The two different OBDDs resulting from considering respectively π_1 and π_2.

Intuitively, we can already notice that the second OBDD has a greater number of nodes since no conclusion can be drawn until a value is assigned to x_2. This results in having a complete binary tree for the first half of the variables, i.e. an exponential blow-up [73].

Theorem 6. *There are infinitely many boolean functions for which there exist at least two different variable orderings that result in OBDDs respectively polynomial and exponential (in size).*

Proof. Consider the family of functions $g_{2n} = (x_1 \iff x_2) \vee \cdots \vee (x_{2n-1} \iff x_{2n})$ and the variable orderings $\pi_{2n} = \langle x_1, x_2, x_3, \ldots, x_{2n} \rangle$ and $\pi_{2n+1} = \langle x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n} \rangle$. When building the OBDD for g_{2n} with respect to π_{2n+1}, the result of the function is completely unpredictable until a value is assigned to x_2. Hence, a complete decision tree of height n must be a subgraph of the OBDD under construction, which means that its size is at least exponential in n. However, when considering π_{2n}, the OBDD can progressively evaluate the result of each factor $g_i = (x_i \iff x_{i+1})$. This means that the OBDD for g_{2n} can be built by concatenating n blocks of constant size – each block evaluates g_i for a certain i: if its result is 1 then the output of the function is 1, otherwise the OBDD can proceed to the following block without the need of keeping any internal state. Thus, the OBDD for g_{2n} with respect to π_{2n} has size $\Theta(n)$. □

As proven again by Bryant, there exist functions that have only exponential-size OBDDs, as well as functions that for any variable ordering have a linear-size OBDD. Bollig and Wegener proved that finding the best ordering with respect to the OBDD size of a given function is NP-complete [73].

2.3.2 Finally, Graphs

Since OBDDs represent boolean functions, and a graph is uniquely defined by the set of its nodes and the set of its edges, one can encode the *characteristic functions* of these sets in order to obtain such a representation.

Definition 2.3 (Characteristic function)**.** The characteristic function of a set A is a function $f_A : X \mapsto \{0,1\}$ such that $f_A(x) = 1 \iff x \in A$.

Basing on these considerations, we can encode a generic graph $\mathcal{G} = \langle V, E \rangle$ with a pair of OBDDs as follows [73]:

- each node $v \in V$ is encoded with a binary string $s_v \in \{0,1\}^{\lceil \log_2 |V| \rceil}$ – the set $\{s_v \ \forall\ v\} \subseteq \{0,1\}^{\lceil \log_2 |V| \rceil}$ can be encoded with an OBDD Ω_V by considering its characteristic function;

- given such encoding for $\mathcal{G}$'s nodes, we can build the characteristic function for the edges – $f_E : \{0,1\}^{2 \cdot \lceil \log_2 |V| \rceil} \mapsto \{0,1\}$ such that $f_E(s_u, s_v) = 1 \iff (u,v) \in E$. Ω_E is the OBDD for f_E.

The OBDD-based representation for $\mathcal{G}$ is given by the tuple $\langle \Omega_V, \Omega_E \rangle$. Hence, considering the results obtained on OBDDs, its size is in the best case proportional to $\log(|V|)$, but explodes to $\Theta(|V|)$ in the worst case.

In order to assess the effectiveness of the OBDD-based representation, we now focus on how the most basic graph operations – namely *visits* – can be implemented on this structure.

As already mentioned above, graphs are usually encoded by building their *adjacency matrix* or the *adjacency lists* for each node. It seems therefore reasonable to investigate how to replicate the querying of such structures on an OBDD-based representation. In particular, it turns out that all these operations can be expressed in terms of a single procedure – which we call *relational product* – that is able to retrieve the set of nodes on the adjacency list of a given set of vertices [73].

Consider a graph $\mathcal{G} = \langle V, E \rangle$ whose set E is represented by an OBDD $\Omega_E(x_1, \ldots, x_n, y_1, \ldots, y_n)$. Moreover, assume to have a set of nodes $A \subseteq V$ represented with an OBDD $\Omega_A(x_1, \ldots, x_n)$. The set of nodes that are on the adjacency list of at least one element in A, which can also be seen as the set of nodes reachable from A by traversing at most 1 arch, is given by the following boolean function:

$$A' = \{(y_1, \ldots, y_n) : \exists x_1, \ldots, x_n(\Omega_A(x_1, \ldots, x_n) = 1 \wedge \Omega_E(x_1, \ldots, x_n, y_1, \ldots, y_n) = 1))\}$$

In general, since we want the set A' to be represented by an OBDD too, it is possible to compute $\Omega_{A'}$ by first building the OBDD for $(\Omega_A(x_1, \ldots, x_n) = 1 \wedge \Omega_E(x_1, \ldots, x_n, y_1, \ldots, y_n) = 1)$, and then quantifying on the required variables. However, notice that introducing the existential quantification as the final step implies that the intermediate OBDD depends on a number of variables that is double with respect to the final result of the whole operation. In order to avoid this overhead, a specific algorithm RelProd was developed.

For instance, to compute the adjacency list of the set of nodes $A = \{000, 010\}$ with respect to the graph shown in Figure 2.9, one would need to calculate Rel-

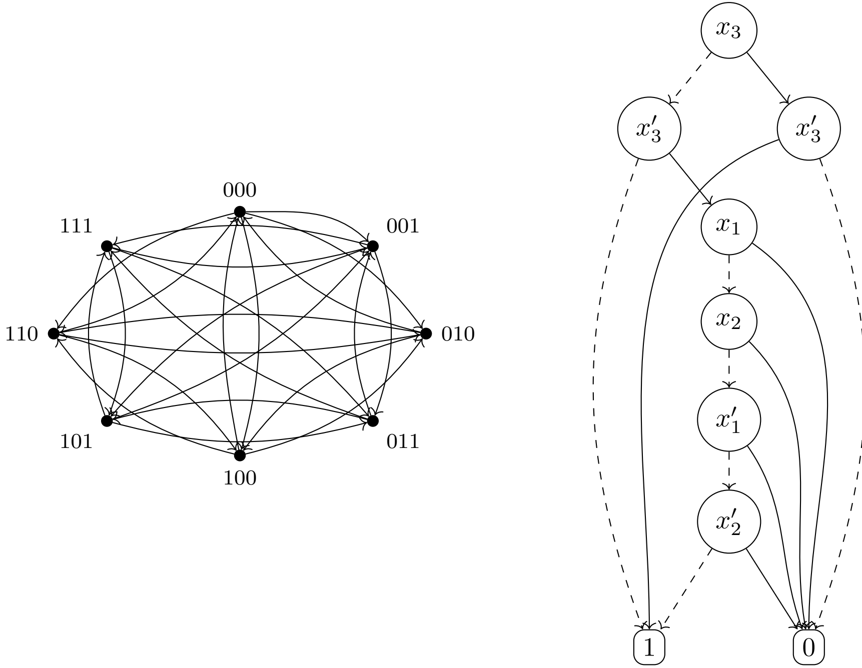


Figure 2.9: Example of a graph and the related OBDD Ω_E that encodes the characteristic function of the set of its edges. The variable ordering used for the OBDD construction is $\pi = \langle x_3, x'_3, x_1, x_2, x'_1, x'_2 \rangle$. Notice that in this case Ω_V would be trivial since all paths would lead to 1.

Prod$(\Omega_A, \Omega_E, \{x_1, x_2, x_3\})$. The result of the procedure is a boolean function represented with an OBDD. In this example case, the resulting function is $\neg x'_3 \vee (\neg x'_1 \wedge \neg x'_2)$, which evaluates to true for the inputs $A' = \{000, 001, 010, 100, 110\}$, i.e. exactly the adjacency list of A.

The computational complexity of the RelProd algorithm can be computed by considering how it builds a new intermediate OBDD at each recursive call. In particular, by using the results obtained in [73, sec. 1.1.2], we can show that given two OBDDs (r_0 and r_1 in the algorithm) with at most i_0 and i_1 variables respectively, composing them into a new OBDD (namely r) costs $\mathcal{O}(2^{i_0} \cdot 2^{i_1}) = \mathcal{O}(2^{i_0+i_1})$. Hence, each recursive call in RelProd costs $\mathcal{O}(2^{2(m-k)})$, where m and k are respectively the total number of variables and the number of quantified ones in the final OBDD. A bound on the number of recursive calls can be simply represented by m. Thus, since $m \in \mathcal{O}(\log |V|)$ and $k = m/2$ in an OBDD-represented graph, RelProd requires $\mathcal{O}(\log |V| \cdot 2^{\log |V|}) = \mathcal{O}(|V| \cdot \log |V|)$ computational steps in total. This result is sometimes presented in literature by reporting $\mathcal{O}(|V|)$ symbolic steps, without further analysing the computational complexity of the symbolic steps.

Maintaining the leitmotiv of the previous sections on matrices and lists, we now briefly investigate the impact of OBDDs in the implementation of basic visits algorithms: BFS and DFS. We begin with the former. A BFS pass on a graph can be achieved in

Algorithm 4 RelProd

```
function RelProd(f, g, X)                              ▷ f, g: OBDDs, X: variable set
    if (f = 0 ∨ g = 0) then return 0
    end if
    if (f = 1 ∧ g = 1) then return 1
    end if
    if (f, g, r) is in the computed table then return r
    end if
    let x be the topmost variable between f and g top variables
    r0 ← RelProd(f|x=0, g|x=0, X)
    r1 ← RelProd(f|x=1, g|x=1, X)
    if x ∈ X then r ← r0 ∨ r1
    else r ← (x ∧ r1) ∨ (¬x ∧ r0)
    end if
    insert (f, g, r) into the computed table
    return r
end function
```

a OBDD-based encoding by iterating the RelProd procedure. We repeatedly apply such function until a fixpoint is reached:

$$\Omega_{A_{i+1}} = \textsc{RelProd}(\Omega_{A_i}, \Omega_E, \{x_1, \ldots, x_n\})$$

where A_0 is the singleton containing only the starting node. At each iteration the OBDD Ω_{A_i} represents the set of nodes at distance i from the root. Before moving to the investigation of DFS visit, we take a little detour. We briefly show how a OBDDs can be applied to easily obtain a procedure for *reachability* problem[3]. Let $G = (V, E)$ be a graph and let $u \in V$ be a node. The reachability problem asks to compute the set of nodes that are reachable from u in G. This very definition of the problem can be lifted to sets of nodes as well. Let $S \subseteq V$ be a set of nodes in G. Solving reachability on S means computing S': the union between all the reachabilities of the nodes composing S. It is straightforward to see that the set of nodes S' reachable from another set S is basically the union of all the nodes visited during a BFS. Therefore, an OBDD for S' can be computed by iterating $\textsc{RelProd}(\Omega_S, \Omega_E, \{x_1, \ldots, x_n\})$ until fixed point and then performing the union of all the intermediate Ω_i. Both problems require iterating RelProd a number of times that is bounded by $|V|$ (specifically, the length of the longest path in $\mathcal{G}$). This means that both algorithms have computational complexity $\mathcal{O}(|V|^2 \cdot \log |V|)$.

While implementing BFS is straightforward, the same does not hold for Depth-First-Search (DFS). By traversing the graph depth-first, we lose the ability to perform the computation in parallel by working with sets of nodes. In fact, by using RelProd the computational complexity for computing the adjacency list of either a single node or a set of nodes is exactly the same. This problem has quite important consequences, as other algorithms based on DFS (e.g. Strongly Connected Components computation) turn out to be inefficient too. We refer to [73] for more details. In the same PhD thesis they also present an alternative algorithm to DFS, which they call *spine-sets* and use for symbolic computation on OBDD-based graphs.

[3] I could not have completed this thesis without referencing my supervisor's favourite problem at least once.

Conclusions on Graphs in Classical Architectures

In this chapter, we reviewed foundational concepts from classical computer science related to graph encoding. The three data structures we discussed are (i) Adjacency Matrices, (ii) Adjacency Lists, and (iii) Ordered Binary Decision Diagrams (OBDDs).

For adjacency matrices, we highlighted how encoding graphs within a linear algebraic structure enables the redefinition of many graph operations in terms of linear algebra. This connection forms the basis of spectral graph theory (SPT), a field dedicated to studying this relationship. As noted in the section on matrices, a deeper exploration of SPT is beyond the scope of this thesis.

Transitioning to adjacency lists, we presented them as an improvement over matrices in terms of space efficiency. Unlike matrices, which store data for edges that may not exist in the graph, adjacency lists avoid this redundancy. However, while adjacency lists are more efficient for sparse graphs, they lose the linear algebraic properties available with matrices. Nonetheless, adjacency lists often lead to time- and space-efficient algorithms for non-complete graphs.

Lastly, we introduced Ordered Binary Decision Diagrams (OBDDs). These structures aim to compactly represent the truth values of Boolean formulas, with their connection to graphs achieved through functions f_V and f_E, encoding the graph's nodes and edges, respectively. OBDDs offer an intuitive basis for implementing algorithms such as BFS. However, selecting an optimal ordering for the variables in f_V and f_E can be challenging, as it may lead to a state explosion.

3

Graphs in Parallel Architectures

We now turn our attention to the study of Parallel Architectures. While parallel computation remains part of the broader field of classical computation, we have chosen to separate our discussion of graph encoding techniques from the foundational concepts of classical computing.

Although matrices, lists, and OBDDs can be used also over parallel architectures for graph-related tasks, they may not fully exploit the capabilities of GPUs, which are designed to perform numerous operations concurrently. Instead, to take full advantage of parallel processing, specialized encoding techniques are often required. These approaches aim to optimize both space and performance by exploiting the GPU's strengths.

In this chapter, we will investigate graph encoding methods tailored for parallel computation, with a particular focus on space-efficient representations derived from adjacency matrices. This approach allows us to review techniques that make graph processing more efficient in terms of both memory usage and computational speed when deployed on parallel architectures.

Parallel Architectures have become more and more prominent in the past decades. Their growth in fame is mostly due to their key contributions in the development in AI-related tasks.

Despite the advancements in computational power that central processing units (CPUs) have achieved, classical computers can still perform only one operation per time step[1].

To address this *architectural* limitation, parallel architectures were introduced, enabling multiple operations to be executed simultaneously. This possibility is mainly due to the introduction of GPUs, the parallel counterpart of CPUs. The capabilities that GPUs enjoy have made them a fundamental tool for a lot of computational tasks. For example, their adoption has dramatically improved algorithms dealing with linear algebra operations. This very speed up they provided, have been fundamental in the development of Artificial Intelligence / Machine Learning. In particular, the step of propagating a dataset towards a neural network, reduces to a tensor product. Such

[1] usually defined by the system's *clock*.

kind of operation can be parallelized and executed efficiently using GPUs. This reason directly leads us to the following question:

Can we use this particular property to speed up algorithms on graphs that are represented as matrices?

To answer comprehensively, we first have to split our attention in two cases: dense and sparse graphs.

The former require dense matrices to be represented. In this case, we cannot apply any particular technique to somehow *lighten* the representation, hence, the algorithms. On the other hand, the latter induce sparse matrices. In parallel computation, different encodings have been proposed to exploit such sparsity. The first goal is to keep using matrices while trying to save memory space—in some sense keeping the *pros* of matrices, while removing the *cons* of the 0 entries. The second goal is to develop faster algorithms on graphs using such encodings.

3.1 Coordinate Encoding—COO

Let $M \in \{0,1\}^{m \times n}$ be a matrix with m rows and n columns. Moreover, let k be the number of non-zero entries in M.

Using the Coordinate Encoding (COO), M is a represented with 3 arrays: R, C, and V. The meaning of these three arrays is the following. Let us assume we require to access the i-th non-zero element of M. Let r, c, v be the i-th element of R, C, and V, respectively. Then the following holds:

$$M_{r,c} = v$$

Roughly speaking:

- The i-th position of R contains the row index of the i-th non-zero element of M
- The i-th position of C contains the column index of the i-th non-zero element of M
- The i-th position of V contains the value of the i-th non-zero element of M

Example 3.1. Let M be the following matrix:

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

M has exactly 7 non-zero elements, therefore R, C, and V have length 7. The vector R will hold values:

$$0,\ 1,\ 2,\ 3,\ 3,\ 4,$$

that represents the (sorted) row indexes of non-zero values. Analogously, vector C contains:

$$0,\ 4,\ 2,\ 1,\ 4,\ 5.$$

Since M is boolean, all the non-zero elements have value one. Therefore, array V seems useless. Nevertheless, lifting this idea to a non-binary example, would require V to contain different values.

Figure 3.1 highlights the basic idea behind COO encoding. In yellow, we highlighted a non-zero element in position (1, 4) and its correlated entries in R, C, and V.

1	0	0	0	0	0
0	0	0	0	1	0
0	0	1	0	0	0
0	1	0	0	1	0
0	0	0	0	0	1

(a) The boolean matrix M.

0	1	2	3	3	4

0	4	2	1	4	5

1	1	1	1	1	1

(b) Vectors R, C, and V.

Figure 3.1: An example of matrix M encoded using COO.

3.2 Compressed Sparse Row—CSR

Let $M \in \{0, 1\}^{m \times n}$ be a matrix with m rows and n columns. Moreover, let k be the number of non-zero entries in M.

The Compressed Sparse Row (CSR) encoding is similar to the COO. The unique difference is that vector R from COO is replaced by vector $\overrightarrow{R}$. The meaning of this new array is to substitute the set of position described by R with a set of *offsets* stored in $\overrightarrow{R}$. Besides this difference of the value contained, R and $\overrightarrow{R}$ have also different lengths. While the former has length k, the latter has length $m + 1$—the number of rows plus one.

Given a row index j, let r, r' be the j-th, $(j+1)$-th element of $\overrightarrow{R}$, respectively. With CSR encoding it holds that:

- the r-th, $(r + 1)$-th, $\cdots$, $(r' - 1)$-th elements of C contain the column indexes of the non-zero element in row j

- the r-th, $(r + 1)$-th, $\cdots$, $(r' - 1)$-th elements of V contain the values the non-zero element in row j

Example 3.2. We now introduce a small example to better explain the workings of CSR. Let M be the matrix depicted in Figure 3.2.

1	0	0	0	0	0
0	1	1	0	1	0
0	0	1	0	0	0
0	1	0	0	1	0
0	0	0	0	0	1

Figure 3.2: A boolean matrix M.

With the CSR encoding, we represent M using three vectors $\overrightarrow{R}, C$, and V. The last two—C and V—are defined as for the COO case, while $\overrightarrow{R}$ is now an *offset* vector. All the three of them are depicted in Figure 3.3.

0	1	2	4	2	1	4	5

(a) Vector C.

1	1	1	1	1	1	1	1

(b) Vector V.

0	1	4	5	7	8

(c) Vector $\overrightarrow{R}$ of M's CSR encoding.

Figure 3.3: Vectors C, V, and $\overrightarrow{R}$ of M's CSR encoding.

We focus on the $j = 1$ row of M, that has 2 non-zero values in position (1, 1), (1, 2), and (1, 4). This information is encoded in $\overrightarrow{R}, C$, and V as follows. Since we seek information about row with index $j = 1$, we focus on $\overrightarrow{R}_j = \overrightarrow{R}_1 = 1$. Moreover, we also store the element next to that, namely: $\overrightarrow{R}_{j+1} = \overrightarrow{R}_2 = 4$. We look at the portion of C between index 1 (included) and 4 (excluded), highlighted in yellow in Figure 3.4a. It contains values 1, 2, and 4, which means that elements in positions (1, 1), (1, 2), and (1, 4) are non-zero in M. The same approach can be applied also to retrieve the values stored in such cells. In particular, the portion of V between index 1 (included) and 4 (excluded)—highlighted similarly to the C case in Figure 3.4b—contains the values of M in positions (1, 1), (1, 2), and (1, 4).

0	1	2	4	2	1	4	5

(a) Interval $[1, 4)$ of C.

1	1	1	1	1	1	1	1

(b) Interval $[1, 4)$ of V.

Figure 3.4: Portions of C and V induced by [1, 4].

Despite showing the internals of CSR, the above example should also serve as a witness for the memory that is spared when using CSR. In fact, using COO encoding,

the vector R would have been as in Figure 3.5. While R has as many elements as the number of non-zero entries in M, $\overrightarrow{R}$ has one element per row of M.

What is peculiar is R's *redundancy*—that is fixed with CSR $\overrightarrow{R}$ approach. In fact, since there are 3 non-zero elements in row 1, R contains exactly 3 repetitions of the number 1. The same happens also for 3: since in row 3 there are two non-zero entries, then R contains 2 copies of 3. This waste of space is fixed by using the *offset* encoding proposed in CSR.

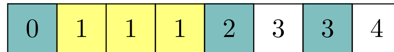


Figure 3.5: Vector R stored if COO was adopted.

3.3 Compressed Sparse Column—CSC

Compressed Sparse Column (CSC) is the third matrix encoding method we describe. As it can be easily derived from its name, it is the *column-based* version of CSR.

Hence, instead of describing this version theoretically, we introduce it practically by showing how CSC encodes a matrix similar to the one we used in Example 3.2 but focusing on columns.

We want the reader to notice that adopting CSC on a matrix M is the same as adopting CSR on M^T.

Example 3.3. Let M be the matrix depicted in Figure 3.2.

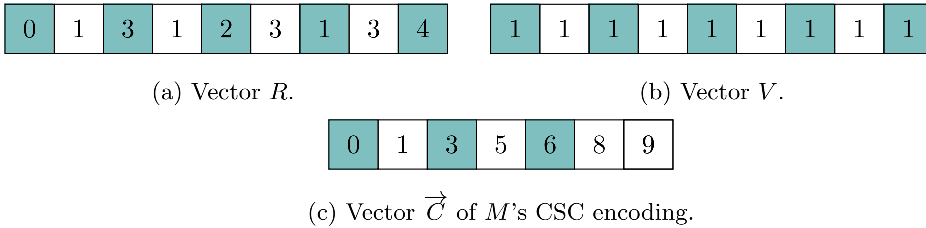
With the CSC encoding, we represent M using three vectors R, $\overrightarrow{C}$, and V. Vectors R and V are depicted in Figure 3.7a and Figure 3.7b, respectively. Vector R contains the row indexes of the non-zero elements, read column by column, left to right and top to bottom. Vector V contains the values of the non-zero elements.

The *offset* vector $\overrightarrow{C}$ is depicted in Figure 3.7c. Analogously to the $\overrightarrow{R}$ case, $\overrightarrow{C}$ has length *number of columns in M + 1*.

Suppose we want to retrieve information about the column $j = 1$ of M. It contains two non-zero elements in position (1, 1) and (3, 1).

1	0	0	0	0	0
0	1	1	0	1	0
0	0	1	0	0	0
0	1	0	1	1	0
0	0	0	0	0	1

Figure 3.6: A boolean matrix M.

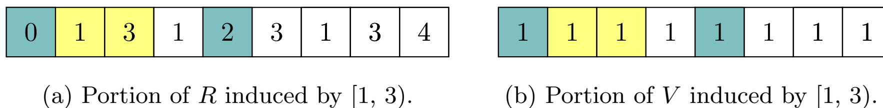


(a) Vector R.

(b) Vector V.

(c) Vector $\overrightarrow{C}$ of M's CSC encoding.

Figure 3.7: Vectors R, V, and $\overrightarrow{C}$ of M's CSC encoding.



(a) Portion of R induced by [1, 3).

(b) Portion of V induced by [1, 3).

Figure 3.8: C's and V's portions induced by the interval [1, 3)

This fact is encoded in R, $\overrightarrow{C}$, and V as follows. We start by storing the values of $\overrightarrow{C}_j = \overrightarrow{C}_1 = 1$ and $\overrightarrow{C}_{j+1} = \overrightarrow{C}_2 = 3$.

Therefore, we focus on the slice of R induced by the interval [1, 3)—we highlighted it in Figure 3.8a— that contains values 1 and 3. Hence, positions (1, 1) and (3, 1) are non-zero. Their values are stored in the portion of V induced by the interval [1, 3)—as for the R case, we highlighted this particular slice in Figure 3.8b.

Concluding, element in position (1, 1) has value 1 and element in position (1, 3) has value 1.

As for the other cases, we stress the fact that vector V only contains 1s since we are dealing with a boolean matrix. What has been introduced can be clearly used also for matrices with other domains.

Notice that, what we said for CSR about the redundancy of R that is fixed using $\overrightarrow{R}$ holds also for CSC. In this case, we spare memory by removing duplicated elements from C instead of reducing R.

Other storing methods that we can mention are Sliced Ellpack (SELL), Block Sparse Row (BSR) and Blocked Ellpack (BLOCKED-ELL). We refer the reader to the API reference guide for cuSPARSE—the CUDA sparse matrix library—for some examples on these techniques[2].

3.4 Graph Visits on Parallel Architectures

Following the structure established in our classical discussions on adjacency lists/matrices and OBDDs, we now turn our attention to how different graph encodings impact the implementation of graph algorithms. The simplest method to utilize COO, CSC, and CSR formats for a visit routine is to create a procedure that retrieves the neighbourhood of a node according to the chosen parallel encoding. This approach allows us to maintain

[2] cuSPARSE

the overall algorithm framework while addressing the specific methods of accessing node data. A straightforward observation reveals that between Breadth-First Search (BFS) and Depth-First Search (DFS), BFS is more adaptable to parallel processing. BFS operates by exploring nodes level by level using a queue, which means that nodes in layer $l+1$ are only processed after all nodes in layer l have been visited. This level-wise approach naturally lends itself to parallel execution, as multiple nodes within the same layer can be handled simultaneously. In contrast, DFS seeks to delve deeply into a graph before backtracking, relying on recursion to process one node at a time. This inherently sequential nature of DFS makes it less suitable for parallelization. Consequently, much of the research literature emphasizes BFS for parallel implementations. Examples of advanced parallel BFS approaches can be found in [117, 74].

Conclusions on Graphs in Parallel Architectures

In this chapter, we introduced three specialized graph encoding techniques designed for parallel computation: COO (Coordinate Format), CSC (Compressed Sparse Column), and CSR (Compressed Sparse Row). These formats have proven to be highly efficient in terms of space and computational speed when processing graphs in parallel. As discussed earlier, there are numerous other encoding methods beyond these three, yet they share a common goal: minimizing unnecessary space usage. By focusing only on the non-zero or relevant elements within the graph's structure, these encodings help to reduce memory overhead and optimize parallel processing capabilities. Towards the end of the chapter, we briefly investigated the potential of implementing algorithms, such as Breadth-First Search (BFS), directly on graphs encoded in COO, CSC, and CSR formats. This exploration hints at the broader scope of possibilities for developing efficient, scalable graph algorithms within these formats, potentially enabling faster traversal and data manipulation when working in parallel environments.

4

Graphs in Quantum Architectures

In this chapter we delve into the topic of encoding graphs into the quantum paradigm. Quantum computation can be *achieved* adopting different foundational models. The three we are going to study are the following

- Gate-Based Quantum Computation: algorithms are circuits and operations are unitary matrices, that in this setting are referred to as *gates*

- Measurement-Based Quantum Computation: the initial state is a *cluster state* and only measurements can be applied

- Adiabatic Quantum Computation: a pair of *Hamiltonians* are given—initial and final—and the *adiabatic theorem* is exploited to solve problems.

The primary objective of this chapter is twofold. On the one hand, in Section 4.1, we introduce a procedure for encoding graphs into unitary matrices. This section builds upon the works [58, 57], which I co-authored. The algorithm presented takes as input an arbitrary multigraph and outputs a unitary matrix that encodes its structural properties. This resulting unitary representation can, for instance, be utilized to implement a quantum random walk on the original multigraph. Subsequently, we examine the *fidelity* of this encoding, leading to the conclusion that any unitary-based graph encoding inevitably introduces a form of bias in the resulting quantum random walk. On the other hand, this chapter also provides a comparative analysis of different quantum computing architectures. While encoding a graph within the gate-based model necessitates the construction of a unitary matrix, this requirement does not extend to alternative paradigms such as measurement-based and adiabatic quantum computing. Notably, the task of translating a graph into the quantum domain yields fundamentally distinct solutions depending on the quantum architecture under consideration.

4.1 Graphs in Gate-Based model

In this section, we will examine the role of graphs within gate-based models, exploring their properties, associated problems, and potential enhancements. This will be the longest section in this chapter, and it will be structured as follows. We will begin by introducing the essential concepts related to *graphs of unitaries*. Next, we will detail an encoding procedure presented in [58, 57]. Throughout this exploration, we will transition from theoretical foundations to examples, ultimately leading to a discussion of empirical results. As an *Intermezzo*, we will introduce a second encoding algorithm with significant similarities to the first. Finally, after presenting both encoding algorithms, and in parallel with our approach in the classical case, we will investigate the primary traversal algorithm applicable to unitary-encoded graphs: Quantum Random Walks.

The study of graph encoding in the Gate-Based model of computation is tightly linked with quantum walks. The idea is to efficiently encode a graph by means of a unitary operator. The adjacencies are then stored in the unitary matrix associated to the operator. By re-iterating the unitary operator in the quantum circuit, one performs a quantum walk that spans the given graph. For the sake of completeness, we introduce the reader to the problem of unitaries synthesis. Let U be a unitary matrix that encodes in some fashion the neighbourhood relation of a graph. To fruitfully use U on a quantum computer, it must be synthesized in terms of elementary gates. This kind of *rewriting* process is called synthesis. It has become more and more prominent in quantum computation during the last few years [145, 108]. This problem can also be related to graph encoding. For example, in [136] the authors tackle the synthesis problem for unitaries used to solve graph colouring. Before proceeding any further, we introduce quantum walks with a simple example.

Consider again the case of graph G from Figure 4.1, a quantum walk may be defined in terms of operator U. Its construction is analogous to that of a classical Markov chain. The three vertices of G induce the state space $\mathbb{C}^3$, with each vertex v being represented by a column vector $|v\rangle$ of the canonical basis. Transitions are guided by the adjoint of U: $U^\dagger$. The adjoint $U^\dagger$ is here required to abide to the conventional representation of unitary evolution as a matrix-vector multiplication. We adopted a description based on the adjoint $U^\dagger$ since the natural encoding used to produce U—borrowed from graph theory—would require an operation of the form $\langle v| U$ to perform a Quantum Random Walk step.

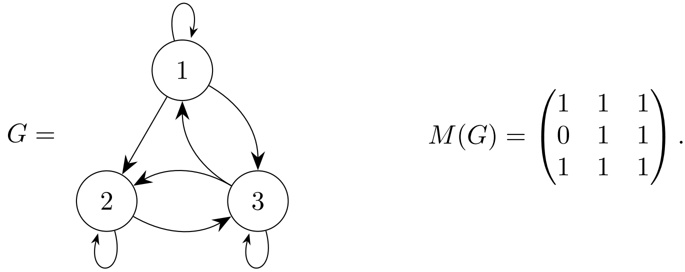
Recalling Example 4.1, let us define a quantum walk on graph G from Figure 4.1. Let the quantum walk start from state $|1\rangle = (1, 0, 0)^T$. Performing the first step then leads to state

$$U^\dagger |1\rangle = \frac{1}{\sqrt{2}} |1\rangle + \frac{1}{2} |2\rangle + \frac{1}{2} |3\rangle .$$

After a single step, the quantum walker finds herself into a superposition of all three vertices of the graph.

Another insightful example has the quantum walk start from state $|\psi\rangle = \frac{1}{2} |1\rangle + \frac{1}{\sqrt{2}} |2\rangle + \frac{1}{2} |3\rangle$. A single step transition leads to state

$$U^\dagger |\psi\rangle = |2\rangle .$$



$$M(G) = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}.$$

Figure 4.1: Graph of a unitary matrix.

The step has produced a rather singular effect: whereas the probability amplitudes for the paths leading to vertex 2 reinforce each other, the paths heading towards vertices 1 and 3 cancel out. These two phenomena are known, respectively, as *constructive* and *destructive interference* and characterize the peculiar behaviour of quantum walks: *more paths heading towards the same vertex do not necessarily imply a higher probability of reaching it.*

The notion of Quantum Random Walk will be further investigated in subsequent sections. However, the key ingredient that we want to stress is the following: to perform a Quantum Random Walk on a graph G, we must be able to encode its neighbourhood properties in a unitary matrix.

4.1.1 Graphs and unitary matrices

Following up on what has just been said, we now provide some foundational results on the class of graphs that are *amenable* for being encoded—walked—in the quantum setting. The most astonishing result we will encounter is the one bonding unitaries and the notion of eulerian graphs. They will turn out to be the widest class of graphs amenable for a unitary encoding.

Definition 4.1. Given a matrix $M \in \mathbb{C}^{n\times n}$, the *support* of M is the matrix $M^S \in \{0,1\}^{n\times n}$ where, for any $1 \le i,j \le n$,

$$M^S_{i,j} = \begin{cases} 1 & \text{if } M_{i,j} \neq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{4.1}$$

The relationship can, thus, be stated in matrix theoretical terms, considering the adjacency matrix of the graph.

Definition 4.2. A directed graph $G = (V, E)$ is said to be *the graph of a matrix M* if and only if M is supported by the adjacency matrix $M(G)$.

Example 4.1. Graph G depicted in Figure 4.1 is the graph of the following unitary

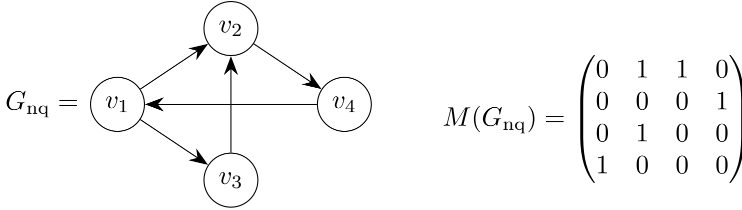


Figure 4.2: Non-quadrangular graph.

matrix U

$$U = \begin{pmatrix} 1/\sqrt{2} & 1/2 & 1/2 \\ 0 & 1/\sqrt{2} & -1/\sqrt{2} \\ -1/\sqrt{2} & 1/2 & 1/2 \end{pmatrix}.$$

In turn, we let $\mathcal{U}$ be the *set including all graphs of unitary matrices.* Before investigating which graphs belong to $\mathcal{U}$, let us briefly outline the ones that certainly *do not.* A unitary matrix does not admit zero-rows or columns. The adjacency matrix of some $G \in \mathcal{U}$ should, then, satisfy the same condition. That is, all vertices in G should own at least one in- and out-neighbour.

Eulerian, Specular and Quandrangular Graphs

Work by Severini [159], established a set of relations between graph properties and unitarity. Two key concepts required to define such rules are specularity and quadrangularity.

Specularity is defined as follows:

Definition 4.3. A graph $G = (V, E)$ is said to be *specular* if and only if, for any pair of vertices $u, v \in V$, the following two conditions are satisfied:

$$\delta^+(u) \cap \delta^+(v) = \varnothing \text{ or } \delta^+(u) = \delta^+(v); \quad \text{and} \quad \delta^-(u) \cap \delta^-(v) = \varnothing \text{ or } \delta^-(u) = \delta^-(v). \tag{4.2}$$

In other words, if two vertices share an *out-neighbour* (*in-neighbour*), then they must share the entire out-neighbourhood (in-neighbourhood).

The following Lemma represents a key result for the purposes of this thesis.

Lemma 4.1. *Let $G = (V, E)$ be a graph, then its line graph $\overrightarrow{G}$ is specular.*

On the other hand, quadrangularity has both a weak and a strong formulation. Let us begin with the former.

Definition 4.4. A graph $G = (V, E)$ is said to be *quadrangular* if and only if, for any pair of vertices $u, v \in V$, where $u \neq v$, it is always the case that

$$|\delta^+(u) \cap \delta^+(v)| \neq 1 \quad \text{and} \quad |\delta^-(u) \cap \delta^-(v)| \neq 1. \tag{4.3}$$

In other words, when a graph is quadrangular, two distinct vertices never share *exactly one* in- or out-neighbour. Graph G_{nq}, displayed in Figure 4.2, violates quadrangularity twice. One violation involves vertices v_1, v_3: v_2 being their only shared out-neighbour.

With Lemma 4.2, we give proof as to why quadrangularity is a necessary condition for graphs of unitary matrices. Because the same result is shown below for strong quadrangularity, Lemma 4.2 trivially follows. However, being quadrangularity a seemingly abstract property, it is useful to gradually observe its effects on adjacency matrices. This also sets us off on our study on the relationship between graphs of unitary matrices and quadrangularity.

Lemma 4.2. *Any directed graph of a unitary matrix is quadrangular.*

Referring to the example in Figure 4.2, observe the first and third row of $M(G_{\mathrm{nq}})$. Any matrix U supported by $M(G)$ shall follow the same pattern of zero-entries. The inner product between rows U_1, U_3 is, then, determined by $U_{1,2} \cdot U_{3,2}$. Being the two entries necessarily non-zero, their product is non-zero.

Let us proceed one step further with the introduction of strong quadrangularity.

Definition 4.5. Given a graph $G = (V, E)$, consider sets of the two following forms:

- $S_{\mathrm{out}} \subseteq V$, where, for any $u \in S_{\mathrm{out}}$ there exists $v \in S_{\mathrm{out}}$ such that $\delta^+(u) \cap \delta^+(v) \neq \varnothing$.
- $S_{\mathrm{in}} \subseteq V$, where, for any $u \in S_{\mathrm{in}}$ there exists $v \in S_{\mathrm{in}}$ such that $\delta^-(u) \cap \delta^-(v) \neq \varnothing$.

Then, G is said to be *strongly quadrangular* if and only if for any set of the form S_{out} or S_{in} it holds that

$$\left| \bigcup_{u,v \in S_{\mathrm{out}}} \delta^+(u) \cap \delta^+(v) \right| \geq |S_{\mathrm{out}}|; \tag{1}$$

$$\left| \bigcup_{u,v \in S_{\mathrm{in}}} \delta^-(u) \cap \delta^-(v) \right| \geq |S_{\mathrm{in}}|. \tag{2}$$

Let us first verify that this version of quadrangularity, indeed, is stronger. G non-quadrangular implies the existence of a subset $S_{\mathrm{out}} = \{u, v\}$ (or S_{in}) where $|\delta^+(u) \cap \delta^+(v)| = 1$. Thus, because $1 < |S_{\mathrm{out}}|$, G is not strongly quadrangular either.

This fact is also supported by the existence of graph G_{nsq} displayed in Figure 4.3. G_{nsq} is quadrangular though not strongly quadrangular. To verify quadrangularity, observe that, in $M(G_{\mathrm{nsq}})$, there exists no pair of rows (or columns) sharing exactly one non-zero entry. On the other hand, detecting non-strong-quadrangularity of G_{nsq} is more involved. Consider the set $S_{\mathrm{out}} = \{v_1, v_2, v_3\}$, and observe that their common out-neighbours are $\{v_1, v_2\}$. Orange and green colours have been used in $M(G_{\mathrm{nsq}})$ to highlight this fact. Because $|\{v_1, v_2\}| < |S_{\mathrm{out}}|$, Rule *1* of strong quadrangularity is violated.

Lemma 4.3. *[159] Any graph of a unitary matrix is strongly quadrangular.*

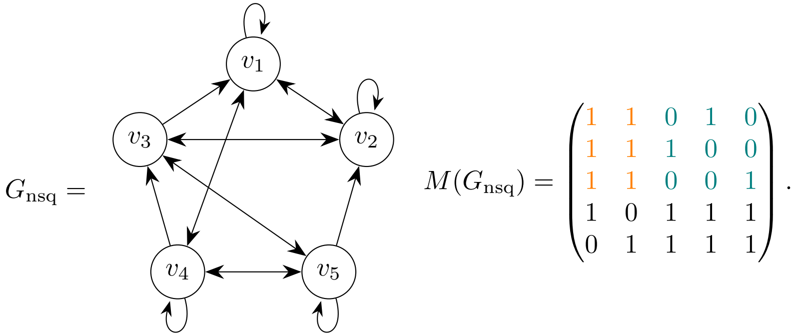


Figure 4.3: Quadrangular graph that is not strongly quadrangular.

The proof for Lemma 4.2 describes a specific case of the one in Lemma 4.3. While the former relies on the non-existence of two orthonormal, one-dimensional vectors, the latter generalizes: for any $s < k$, there exist no k orthonormal s-dimensional vectors.

Having gained acquaintance with strong quadrangularity, it is yet to be shown how, together with specularity, these make sufficient conditions for a graph G to be that of a unitary matrix. A pair of preliminary notions are still due.

Claim 4.1. *For $n > 0$, the n-complete graph with self loops K_n^+ is the graph of a unitary matrix.*

Proof. For any n, the adjacency matrix $M(K_n^+)$ has no zero-entries. For any n, the *Discrete Time Fourier* (DFT) orthonormal basis induces a unitary matrix $\text{DFT}(n)$ without zero entries, where, for any $0 \leq i, j \leq n-1$

$$\text{DFT}(n)_{i,j} = \frac{1}{\sqrt{n}} e^{2ijk/n}. \tag{4.4}$$

□

Definition 4.6. Let M be a $n \times m$ matrix. M' is an *independent full submatrix* of M if and only if, for any $1 \leq i, k \leq n$ and $1 \leq j, l \leq m$, if an entry $M_{i,j}$ is also an entry of M', then, any $M_{i,l}, M_{k,j}$ is either another entry of M' or a zero entry.

To better understand the notion of independent full submatrix, consider the matrix

$$M = \begin{pmatrix} 0 & 0 & 0 & \alpha_{1,1} & \alpha_{1,2} & 0 & 0 \\ \beta_{1,1} & \beta_{1,2} & 0 & 0 & 0 & \beta_{1,3} & 0 \\ 0 & 0 & 0 & \alpha_{2,1} & \alpha_{2,2} & 0 & 0 \\ 0 & 0 & 0 & \alpha_{3,1} & \alpha_{3,2} & 0 & 0 \\ \beta_{2,1} & \beta_{2,2} & 0 & 0 & 0 & \beta_{2,3} & 0 \\ 0 & 0 & \gamma_{1,1} & 0 & 0 & 0 & \gamma_{1,2} \end{pmatrix}, \tag{4.5}$$

where A, B, C are independent full submatrices of M:

$$A = \begin{pmatrix} \alpha_{1,1} & \alpha_{1,2} \\ \alpha_{2,1} & \alpha_{2,2} \\ \alpha_{3,1} & \alpha_{3,2} \end{pmatrix}; \quad B = \begin{pmatrix} \beta_{1,1} & \beta_{1,2} & \beta_{1,3} \\ \beta_{2,1} & \beta_{2,2} & \beta_{2,3} \end{pmatrix}; \quad C = \begin{pmatrix} \gamma_{1,1} & \gamma_{1,2} \end{pmatrix}. \tag{4.6}$$

All is set for the main result of this section. Lemma 4.4 makes the first and hardest step through the endeavour.

Lemma 4.4. [159] *A specular, strongly quadrangular graph is the graph of a matrix composed of square independent full submatrices.*

Lemma 4.5. *The graph of a matrix composed of square independent full submatrices is the graph of a unitary matrix.*

At last, Theorem 7 follows from Lemmata 4.4 and 4.5.

Theorem 7. [159] *A specular, strongly quadrangular graph is the graph of a unitary matrix.*

Strongly quadrangular line graphs Theorem 7 decreed specularity and strong quadrangularity as sufficient conditions for a graph to be that of a unitary matrix. On the other hand, Lemma 4.1 states that all line graphs are specular. Corollary 4.1 trivially follows.

Corollary 4.1. *A strongly quadrangular line graph is the graph of a unitary matrix.*

That being the case, a spontaneous way to characterize part of set $\mathcal{U}$ answers the question: Which graphs give rise to strongly quadrangular line graphs? The following result has been shown by Severini in [159] and exhaustively solves the problem.

Theorem 8. [159] *Let G be a graph. Then, $\overrightarrow{G} \in \mathcal{U}$ if and only if G is Eulerian or the disjoint union of Eulerian components.*

Bridgeless, inseparable directed graphs

Severini has also investigated the relationship between graphs and unitary matrices in the opposite direction [158]. Let us consider G, graph of a unitary matrix. It is possible to define appropriate metrics to assess the *connectivity* of G. As it turns out, having $G \in \mathcal{U}$ requires its connectivity be *not too fragile.*

The following preliminary definitions will help formalize the opening statement. Let $G = (V, E)$ be a graph.

Definition 4.7. A set $E' \subset E$ is said to be a *disconnecting set of directed edges* if and only if the graph $G' = (V, E \backslash E')$ has more connected components than G.

An analogous definition may be given with respect to vertices.

Definition 4.8. A set $V' \subset V$ is said to be a *disconnecting set of vertices* (or a *cut*) if and only if the graph $G' = (V \backslash V', E_{V'})$ has more connected components than G, where $E_{V'}$ is the subset of edges restricted to vertices in V'.

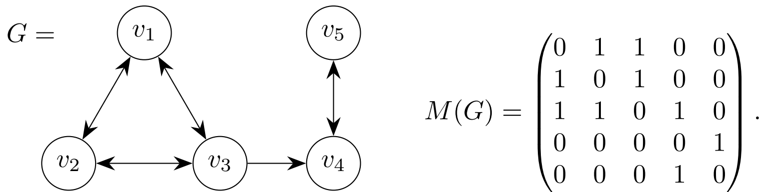


Figure 4.4: Graph including a bridge, a directed bridge and two cut-vertices.

Finally, we can define the primitives that will characterize the results of this section.

Definition 4.9. A *bridge* is a disconnecting set of edges $E' = \{(u, v), (v, u)\}$. In other words, a *bridge* is a disconnecting undirected edge.

Definition 4.10. A *directed bridge* is a disconnecting set of edges $E' = \{(u, v)\}$.

Definition 4.11. A *cut-vertex* is a disconnecting set of vertices $V' = \{v\}$.

Definition 4.12. Let E' be a set of edges such that, were edges E' to be deleted, the given graph would be split in two connected components C_1, C_2. Then, E' is said to be a disconnecting set of directed bridges if and only if, for all $(u, v) \in E'$, $u \in C_1$ and $v \in C_2$.

Figure 4.4 provides an example covering all three definitions. The undirected edge $\{v_4, v_5\}$ is a bridge. Edge (v_3, v_4) is a directed bridge. Vertex v_3 is one of the two cut-vertices in the graph.

Finally, G is said to be *bridgeless* if it contains no bridges; analogously, *inseparable* if it contains no cut-vertices. In connection with the opening statement of this section, one could - rather informally - deem a graph violating any of these three properties to be - *"not so connected."*

At last, let us introduce the central result of this section.

Theorem 9. *[158] Let G be the graph of a unitary matrix U. Then, the following conditions are satisfied:*

1. *G has no directed bridges.*

2. *Either G is bridgeless, or all bridges belong to connected components that are K_2 or K_2^+.*

3. *Either G is inseparable, or all cut-vertices are isolated vertices with self-loops.*

Reversible directed graphs

In [127], Montanaro tackled both directions of the relation between graphs and unitary matrices. On the one hand, he has shown that any coined quantum walk is performed over a *reversible* graph. On the other, he has given constructive proof that, provided all vertices be equipped with self-loops, a coined quantum walk may be defined over any

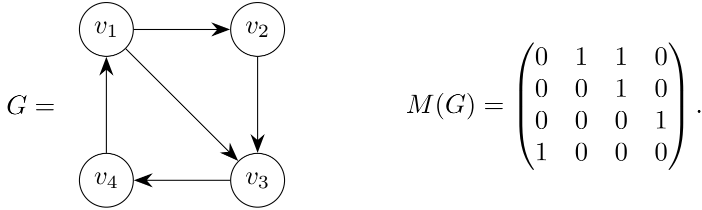


Figure 4.5: Reversible graph that is not the graph of a unitary matrix.

reversible graph. Before diving into the study of this result, let us carefully assess what it says and what it does not.

The statement declares reversibility to be both a sufficient and necessary condition for a coined quantum walk to be defined on a graph (with self-loops). By sufficiency, all reversible graphs with self-loops allow coined quantum walks. However, not all graphs allowing coined quantum walks are graphs of unitary matrices (see the infinite line from Example 4.7). Proof of sufficiency thus does *not* speak about graphs of unitary matrices. Instead, it speaks about how reversible graphs with self-loops may be mapped onto graphs of unitary matrices.

As for this section, the proof of necessity is reviewed, providing a comparison with previously illustrated conditions. Let us introduce the main character of this section.

Definition 4.13. Let $G = (V, E)$ be a graph. An edge $(u, v) \in E$ is said to be reversible if and only if there exists a path from v to u.

Initially, one could be misled to think that reversible edges are all and only those edges that *are not* directed bridges. However, whereas *any directed bridge is an irreversible edge*, the opposite is not true.

Remark 4.1. An edge is reversible if and only it does not belong to any disconnecting set of directed bridges.

Defining reversible graphs is now straightforward.

Definition 4.14. A graph $G = (V, E)$ is said to be *reversible* if and only if all its edges are reversible.

From Definition 4.14, it immediately follows that a connected component is *strongly* connected if and only if it is reversible.

In light of the observations made above, the condition of reversibility does not appear too distant from that asked by point (1) from Theorem 9. This similarity is briefly elaborated at the end of this chapter. In contrast, the comparison between reversibility and conditions (2),(3) immediately to state the following remark.

Remark 4.2. There are reversible graphs that are not those of unitary matrices. Figure 4.2 provides an example. Because G is not quadrangular it cannot be the graph of a unitary matrix.

At last begins the path towards the main result of this section, which may be formally stated as follows.

Theorem 10. *[127] Any coined quantum walk is performed on a reversible graph.*

Let us start off on our path from a well known result from Quantum Mechanics.

Theorem 11 (Quantum Recurrence Theorem [37])**.** *Let W be any unitary operator over $\mathbb{C}^n$, then for any $\epsilon > 0$ and any $|\psi\rangle \in \mathbb{C}^n$, there exists $k \geq 1$ such that $\langle\psi| W^k |\psi\rangle > 1-\epsilon$.*

In other words, given an initial state $|\psi\rangle$, merely reapplying unitary operator W must, sooner or later, lead back to a state arbitrarily close to $|\psi\rangle$.

Lemma 4.6. *Let $|\psi\rangle, |\varphi\rangle \in \mathbb{C}^n$ be two states such that $\langle\varphi| W |\psi\rangle \neq 0$, for some unitary W. Then, there exists $m \geq 0$ such that $\langle\psi| W^m |\varphi\rangle \neq 0$.*

Lemma 4.6 appears to characterize unitary operators through a sort of quantum analog of graph reversibility as given in Definition 4.13, hinting a connection to the notion of reversibility given in discussing of Postulate two.

Let us proceed on our path. The next step applies knowledge of the newly proven findings over coined quantum walk operators.

Lemma 4.7. *Let $G = (V, E)$ be a graph with $V = \{v_1, v_2, \ldots, v_n\}$ and let W be a quantum walk on G with a coin C operating on $\mathbb{C}^k$ with $k \geq 1$. For any $|c_p\rangle |v_i\rangle, |c_q\rangle |v_j\rangle$, if there exists $m \geq 0$ such that $\langle c_q| \langle v_j| W^m |c_p\rangle |v_i\rangle \neq 0$, then there exists a path from v_i to v_j in G.*

By Lemma 4.7, states encountered by a coined quantum walk describe paths on the underlying graph. The proof carefully considers m-step walks regardless of the coin. This allows to bypass any effects of destructive interference that could, potentially, cancel out amplitudes for a given path.

Finally, Theorem 10 immediately follows from Lemmata 4.6 and 4.7. Furthermore, because all graphs of unitary matrices allow for coined quantum walks, the following corollary is implied.

Corollary 4.2. *All graphs of unitary matrices are reversible.*

Theorem 8 serves as a clear and solid result about what *can* and what *cannot* be directly translated into a unitary matrix when dealing with graph. However, despite its fundamental contributions in [159], Severini never truly provided a fully characterized algorithm to convert an eulerian graph into a unitary. Moreover, he never expressed on the potential techniques to jump from a non-eulerian to an eulerian graph. Both of these issues have been tackled in [58, 57] where authors provided

1. a procedure to embed any graph into an eulerian one by adding new edges

2. two different solutions to deal with the added edges while performing a correct Quantum Random Walk.

together with a general and flexible procedure to obtain a unitary out of an eulerian graph in polynomial time.

4.1.2 An (Eulerian) Graph-to-Unitary Encoding algorithm

In [58, 57], an algorithmic solution aimed at encoding graphs into unitary matrices was proposed.

Most of the results obtained are based upon Theorem 8 which we restate for the sake of clarity:

Theorem 12. *Let G be a (connected) multigraph and $\overrightarrow{G}$ be its line graph. Then $\overrightarrow{G}$ is the graph of a unitary matrix iff G is eulerian.*

In this section we introduce a procedure which allows us to transform any eulerian multigraph G into a unitary matrix, encoding the adjacency properties of G. The procedure passes through the construction of the line graph of G, then Theorem 12 from [159] lies at the heart of the transformation. However, the focus in [159] is on the proof of the result, more than on the algorithmic construction of a unitary matrix. Instead, in this section we are going to actually build a unitary matrix starting from a line graph adjacency matrix. The reason behind the procedure we will introduce is twofold. On the one hand, as already anticipated, we provide a fully working procedure based on the theoretical results presented in [159]. On the other hand, the peculiarity of our proposal is that it can be applied to any kind of graph it is provided in input with. Other techniques like the one presented in [2] can be applied just to a small subset of graphs—regular ones. Such unitary matrix is not unique, and our construction is parametric with respect to a family of unitary matrices which are required as input. As for the computational complexity of the technique, it is linear in the size of the resulting matrix—we take into account the complexity of *producing* the resulting matrix, and we show that the algorithm takes the same time. Notice that it is common usage to build unitary matrices by columns. In our approach, since we edit the adjacency matrix, the resulting unitary will be by rows. In this way we keep a closer relationship to the initial graph. Therefore, the reader should be aware that in the circuit and in the examples we will use the conjugate transpose of the matrix. Moreover, with a slight abuse of notation, whenever we refer to the product of two rows of a matrix, we refer to the *dot product* (scalar product) between the former and the conjugate transpose of the latter, i.e., $M_i M_j = \sum_k M_{i,k} M^*_{j,k}$, where $M^*_{j,k}$ is the complex conjugate of $M_{j,k}$.

From a Graph to its Line

Let $G = (V, E)$ be an eulerian multigraph. The function LINEARIZE in Algorithm 5 returns the adjacency matrix $\widetilde{M}$ of the line graph $\overrightarrow{G}$. It can be easily checked that its time complexity is $\Theta(|E|^2)$.
Notice that, from this point, we will always refer to the set of nodes with V, to the set of edges with E, and to the input multigraph with G. Moreover, all the variables declared inside pseudocode may be used inside statements or proofs.

Algorithm 5 Construct the line graph of a given multigraph.

1: **function** LINEARIZE(V, E)
2:     $\widetilde{M} \leftarrow$ SQUAREMATRIX($|E|$)   ▷ Creating an all zero square matrix
3:     **for all** $i \in E$ **do**
4:         $v \leftarrow$ TARGET(i)   ▷ i is of the form (u, v)
5:         **for all** $k \in \delta^+(v)$ **do**   ▷ k is of the form (v, w)
6:             $\widetilde{M}_{i,k} \leftarrow 1$
7:         **end for**
8:     **end for**
9:     **return** $\widetilde{M}$
10: **end function**

Intuitively, the algorithm works as follows. For every $i = (u, v) \in E$ we find all its consecutive edges, identified by k, and we set the entry i, k to 1.

We now point out some structural properties of the matrix $\widetilde{M}$ returned by Algorithm 5 that follow from the fact that G is eulerian.

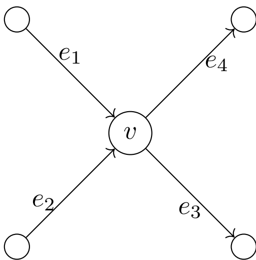
Lemma 4.8. *Let $i = (u, v)$ and $j = (u', v')$ be two edges of G. It holds that:*

$$\widetilde{M}_i = \widetilde{M}_j \quad \textit{iff} \quad v = v'$$

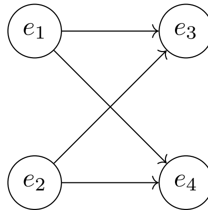
Proof. Let i, j be two edges of G. Consider some edge $k \in E$. By definition of $\overrightarrow{G}$, $\widetilde{M}_{i,k} = 1$ iff i and k are consecutive edges. The same is true for j. So, if $v = v'$ it is immediate to conclude that $\widetilde{M}_i = \widetilde{M}_j$. Otherwise, if $\widetilde{M}_i = \widetilde{M}_j$, then, since G is eulerian, it cannot be the case that $\widetilde{M}_i$ and $\widetilde{M}_j$ have only 0 elements. This means that $\exists k$ such that $\widetilde{M}_{i,k} \neq 0$. Hence, by hypothesis also $\widetilde{M}_{j,k} \neq 0$. Let $k = (v'', w)$, since $\widetilde{M}_{i,k} \neq 0$ it has to be $v = v''$. Moreover, from $\widetilde{M}_{j,k} \neq 0$ we get $v' = v''$. So, $v = v'$. □

As a consequence of the above lemma we immediately get that each node v induces as many equal rows in $\widetilde{M}$ as its in-degree.

Lemma 4.9. *For every edge $i = (u, v)$ of G, there are exactly $d^+(v)$ edges j such that $\widetilde{M}_{i,j} \neq 0$. Moreover, there are exactly $d^-(v)$ rows equal to $\widetilde{M}_i$.*



(a) A balanced node v inside a graph G.



(b) The portion of $\overrightarrow{G}$ induced by v.

$$\begin{array}{c} \\ e_1 \\ e_2 \\ \vdots \end{array} \begin{array}{c} \begin{array}{cccccc} e_1 & e_2 & e_3 & e_4 & \cdots & \end{array} \\ \begin{pmatrix} 0 & 0 & 1 & 1 & \cdots \\ 0 & 0 & 1 & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{pmatrix} \end{array}$$

(c) Effect of v on $\widetilde{M}$.

Figure 4.6: How the neighbourhood of a node inside $G = (V, E)$ is reflected in the matrix $\widetilde{M}$.

Example 4.2. We give an example of how Lemma 4 and 5 work. Let $G = (V, E)$ be an input graph and let $v \in V$ be a node of G. In Figure 4.6a we depicted v with 2 incoming edges and two outgoing edges—$d^+(v) = d^-(v) = 2$. Figure 4.6b shows how edges e_1, e_2, e_3, e_4 are connected in $\overrightarrow{G}$. By Lemmata 4.9 we know that there are exactly $d^+(v) = 2$ edges j such that $\widetilde{M}_{e_1,j} \neq 0$. Since $\widetilde{M}$ is the adjacency matrix of $\overrightarrow{G}$, then we can conclude that: $\widetilde{M}_{e_1,e_3} = 1$ and $\widetilde{M}_{e_1,e_4} = 1$. By Lemmata 4.8, it holds that $\widetilde{M}_{e_1} = \widetilde{M}_{e_2}$ since e_1 and e_2 have the same target v.

The *submatrix* of $\widetilde{M}$ induced by v is depicted in Figure 4.6c.

In Lemmata 4.8 and 4.9 we introduce a relationship between rows describing edges that have a common target. We can generalize the result considering both the case of edges incident to the same node and edges incident to different nodes.

Lemma 4.10. *Let i, j be two edges of G. Let $NZ_i = \{k : \widetilde{M}_{i,k} \neq 0\}$ and similarly $NZ_j = \{k : \widetilde{M}_{j,k} \neq 0\}$. Then, either $NZ_i = NZ_j$ or $NZ_i \cap NZ_j = \varnothing$.*

Proof. If $NZ_i = NZ_j$ there is nothing to prove. Suppose $NZ_i \neq NZ_j$ and assume, for the sake of contradiction, that $k = (v, w) \in NZ_i \cap NZ_j$. By definition of NZ_i and NZ_j, it has to be $\widetilde{M}_{i,k} = 1 = \widetilde{M}_{j,k}$. Hence, i and j share v as common target. By Lemma 4.8, $\widetilde{M}_i = \widetilde{M}_j$ and $NZ_i = NZ_i$. This is a contradiction.

□

In what follows we say that two rows $\widetilde{M}_i$ and $\widetilde{M}_j$ of $\widetilde{M}$—corresponding to the edges i and j of G—are *disjoint* if $NZ_i \cap NZ_j = \varnothing$, where NZ_i, NZ_j are defined as in the above lemma. In other terms, two rows are disjoint if and only if they correspond to edges having different targets.

Construction of the Unitary Matrix

We now describe the procedure UNITARIZE which transforms $\widetilde{M}$ (the matrix resulting from Algorithm 5) into a unitary matrix $\widehat{M}$ whose support is $\widetilde{M}$. The goal of UNITARIZE in Algorithm 6 is to edit $\widetilde{M}$ in order to obtain a unitary matrix which encodes the adjacency properties of $G = (V, E)$. The input of the function are the adjacency matrix $\widetilde{M}$ and a family of unitary matrices $\mathcal{U}$. The family has to satisfy:

$$\forall v \in V \; \exists U \in \mathcal{U} \;\; \text{of size } d^-(v) \times d^-(v) \tag{4.7}$$

Roughly speaking, let V be the set of nodes of the input graph. We require that in $\mathcal{U}$ there is at least one matrix per each possible node in-degree. This because Algorithm 6 will use such matrices to turn $\widetilde{M}$ into a unitary matrix.

At each iteration of the while loop, a node v is extracted from Q. For each of its incoming edges $i \in \delta^-(v)$ row $\widetilde{M}_i$ is edited through the procedure SPARSESUB exploiting a unitary matrix U of suitable dimension. In SPARSESUB(r, r') the notation r_k, r'_h refers to the k-th element of r and the h-th element of r', respectively. In particular, SPARSESUB(r, r') replaces the k-th element of r with r'_h.

Let t be a vector and o be a set of indexes of t, we use the notation $t(o)$ to denote the subvector of t obtained by considering only the indexes in o.

Algorithm 6 Compute a unitary matrix from the line graph.

```
1: function UNITARIZE(M̃, 𝒰)
2:     M̂ ← M̃
3:     Q ← V                                   ▷ V is the set of nodes of the input graph
4:     while Q ≠ ∅ do
5:         v ← POP(Q)
6:         U ← 𝒰(d⁻(v))                        ▷ Choose some d⁻(v) × d⁻(v) matrix from 𝒰
7:         c ← 1
8:         for all i ∈ δ⁻(v) do
9:             SPARSESUB(M̂_i, U_c)
10:            c ← c + 1
11:        end for
12:    end while
13:    return M̂
14: end function
15:
16: procedure SPARSESUB(r, r′)
17:     h ← 1
18:     for all k ∈ E do
19:         if r_k ≠ 0 then
20:             r_k ← r′_h
21:             h ← h + 1
22:         end if
23:     end for
24: end procedure
```

Lemma 4.11. *At each iteration of the for loop at line 8 of Algorithm 6 it holds that after applying* SPARSESUB*($\widehat{M}_i(\delta^+(v))$, U_c), the subvector $\widehat{M}_i(\delta^+(v))$ is equal to U_c.*

Proof. From Lemmata 4.8 and 4.9 it follows that $\widetilde{M}_{i,k} = r_k \neq 0$ (line 19) holds iff $k \in \delta^+(v)$. By construction, U_c has size $n = d^-(v) = d^+(v) = |\delta^+(v)|$. The result follows immediately. □

Theorem 13. *$\widehat{M}$ is a unitary matrix.*

Proof. It is sufficient to show that the rows of $\widehat{M}$ form an orthonormal basis. By Lemma 4.11, each row of $\widehat{M}$ is a row of some unitary matrix interleaved by zeros. Hence, the product of each row with itself is 1. Let $i, j \in E$ be distinct edges. If $\widetilde{M}_i$ and $\widetilde{M}_j$ are disjoint, also $\widehat{M}_i$ and $\widehat{M}_j$ are. Therefore, $\widehat{M}_i\widehat{M}_j = 0$. Otherwise, by construction of UNITARIZE, $\widehat{M}_i$ and $\widehat{M}_j$ refer to the same unitary matrix. Since their product depends only on their non-zero elements, and since they correspond to unitary rows, $\widehat{M}_i\widehat{M}_j = 0$. □

Since each line of $\widehat{M}$ is edited once and $\mathcal{U}$ can be stored efficiently, we get that UNITARIZE has time complexity $\Theta(|E|^2)$.

Example 4.3. We now provide an example to clarify the structure of $\widehat{M}$ after the procedure UNITARIZE. For the sake of readability, we omit the scalar multipliers of the matrices of $\mathcal{U}$ inside $\widehat{M}$.

The reader may notice that the choice of $\mathcal{U}$ is not unique. Any set that satisfies (4.7)

is a candidate.

$$\mathcal{U} = \left\{ \frac{1}{\sqrt{3}} \begin{pmatrix} 1 & 1 & 1 \\ 1 & \omega & \omega^2 \\ 1 & \omega^2 & \omega^4 \end{pmatrix}, \; \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \; \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right\} \qquad \omega = \exp\left\{ \frac{2\pi i}{3} \right\}$$

$$\widetilde{M} = \begin{pmatrix} 1 & 1 & & & & & \\ & & 1 & 1 & 1 & & \\ & & 1 & 1 & 1 & & \\ & & & & & 1 & 1 \\ & & 1 & 1 & 1 & & \\ & & & & & 1 & 1 \\ 1 & 1 & & & & & \end{pmatrix} \qquad \widehat{M} = \begin{pmatrix} 1 & 1 & & & & & \\ & & 1 & 1 & 1 & & \\ & & 1 & \omega & \omega^2 & & \\ & & & & & 0 & 1 \\ & & 1 & \omega^2 & \omega^4 & & \\ & & & & & 1 & 0 \\ 1 & -1 & & & & & \end{pmatrix}$$

The reason for this particular family $\mathcal{U}$ is to show that there can be more than one unitary matrix with the same size inside such family—In this case we have two different 2×2 unitaries. Such particular property results in the fact that $\widehat{M}$ is not unique. For example, any one of the following can be obtained through Algorithm 6:

$$\widehat{M} = \begin{pmatrix} 1 & 1 & & & & & \\ & & 1 & 1 & 1 & & \\ & & 1 & \omega & \omega^2 & & \\ & & & & & 1 & 1 \\ & & 1 & \omega^2 & \omega^4 & & \\ & & & & & 1 & -1 \\ 1 & -1 & & & & & \end{pmatrix} \qquad \widehat{M} = \begin{pmatrix} 0 & 1 & & & & & \\ & & 1 & 1 & 1 & & \\ & & 1 & \omega & \omega^2 & & \\ & & & & & 0 & 1 \\ & & 1 & \omega^2 & \omega^4 & & \\ & & & & & 1 & 0 \\ 1 & 0 & & & & & \end{pmatrix}$$

$$\widehat{M} = \begin{pmatrix} 0 & 1 & & & & & \\ & & 1 & 1 & 1 & & \\ & & 1 & \omega & \omega^2 & & \\ & & & & & 1 & 1 \\ & & 1 & \omega^2 & \omega^4 & & \\ & & & & & 1 & -1 \\ 1 & 0 & & & & & \end{pmatrix}$$

Dealing with Non-Eulerian Graphs

In the previous section we showed how to produce a unitary matrix from an eulerian multigraph. The aim of this section is to introduce a procedure called EULERIFY, based on Theorem 2, that takes as input the nodes set V of a connected multigraph $G = (V, E)$ and the array b of the balances of the nodes. It gives as output a set of additional edges $E_\perp$ such that the multigraph $G' = (V, E \cup E_\perp)$ is eulerian—such multigraph will be given as input to Algorithm 5. We recall that, since we are interested in quantum random walks, we take into account only connected graphs.

Algorithm 7 Edit the graph to make every node balanced.

```
1: function EULERIFY(V, b)
2:     E⊥ ← ∅
3:     B+ ← {v ∈ V : b_v > 0}
4:     B− ← {v ∈ V : b_v < 0}
5:     while B− ≠ ∅ do
6:         u ← POP(B−)
7:         while b_u < 0 do
8:             v ← CHOOSE(B+)                          ▷ Choose without extracting
9:             E⊥ ← E⊥ ∪ {(u, v)}
10:            (b_u, b_v) ← (b_u + 1, b_v − 1)
11:            if b_v = 0 then
12:                B+ ← B+ \ {v}
13:            end if
14:        end while
15:    end while
16:    return E⊥
17: end function
```

The procedure takes as input the set of nodes V together with the vector b of size $|V|$ initialized with the balance of the nodes, i.e., the v-th element of b is $b_v = b(v)$. The idea is to iteratively fix each node in deficiency of balance adding edges to nodes in surplus of balance. The choice of the deficient node u is done at line 6. The loop at lines 7–14 is responsible for adding edges from u to surplus nodes v until u is balanced. Theorem 2 both ensures that for each u there are always surplus nodes to choose at line 8 and that when we exit the loop 5–15 the sets B^+ and B^- are empty.

It is clear that the time computational complexity of EULERIFY is $\Theta(|V| + |E_\perp|) \subseteq O(|V|+|E|)$, since we never add more edges than the existing ones. This is the technical point were the use of multigraphs helps us: copies of existing edges, as well as completely new edges, may be added by the procedure.

The adjacency properties of G could be different from those of G'. In particular, such differences are witnessed by the set $E_\perp$. Our aim is to visit G through a quantum random walk. However, since G' is eulerian, the walk will be performed exploiting the unitary matrix constructed on the line graph of G'. Because of that, edges from $E_\perp$ could be traversed during the walk, while we only want edges from E to be used to visit the graph. In order to do this we will use a projector defined as follows:

$$P_G = I - \sum_{e \in E_\perp} |e\rangle\langle e|$$

Notice that even if in $E_\perp$ we add an edge between two adjacent nodes of G, the projector removes the new edge, while it does not affect the one existing in G.

Example 4.4. The following toy example shows the construction of P_G.

$$E = \{00, 01, 10, 11\} \qquad E_\perp = \{01, 11\} \qquad P_G = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

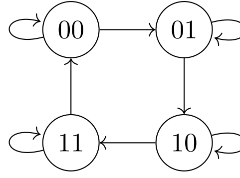


Figure 4.7: Four nodes cyclic graph.

4.1.3 The Overall Procedure

In the previous sections we introduced two key procedures, namely Unitarize and Eulerify. While the first is in charge of constructing a unitary matrix from an eulerian multigraph, the second one embeds a connected multigraph into an eulerian one. In this section we put the two procedures together and describe the resulting algorithm performing the quantum walk—such algorithm can be directly mapped to a quantum circuit.

Given any connected multigraph $G = (V, E)$, we first check whether it is balanced. If not, we apply the procedure Eulerify obtaining some G'—which is now eulerian. We also compute a projector P_G which "eliminates" the effect of the edges of $E_\perp$ along the walk. After that, we use the procedure Unitarize on G' to obtain a unitary matrix $\widehat{M}$ that would allow us to make one discrete step in the visit of G'. So, a walk on G can be implemented by using each time P_G after $\widehat{M}$.

We provided both a command-line and a graphical tool to test these algorithms. *Failure-Resilient Eulerian graph Encoding (for) Quantum tOurs*—FREEQO—is a python tool that allows to obtain the unitary encoding of any given graph. The source code with the related documentation is available at FREEQO. A web-based application has been provided as well, reachable via the following link iFREEQO. The input must be provided as a dot-formatted graph. The tool subsequently generates the matrix *before* the family of unitary is applied. This is motivated by the possibility of the user to interact with the submatrices and patch them with the unitaries he finds more suitable.

An Example

In what follows we execute step-by-step the procedure we just described using graph in Figure 4.7 as input. We will also give an example of an actual QRW performed using the resulting unitary.

Since G is eulerian, the set $E_\perp$ is empty. The adjacency matrix of $\overrightarrow{G}$ is

$$\widetilde{M} = \begin{pmatrix} \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

We now apply UNITARIZE($\widetilde{M}$,$\mathcal{U}$) to obtain $\widehat{M}$

$$\mathcal{U} = \left\{ \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \right\} \qquad \widehat{M} = \frac{1}{\sqrt{2}} \begin{pmatrix} \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & \mathbf{-1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{-1} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{-1} \\ \mathbf{1} & \mathbf{-1} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The above matrix was computed fixing the following encoding of the edges of G

$$\begin{array}{llll} |000\rangle = (00,00) & |001\rangle = (00,01) & |010\rangle = (01,01) & |011\rangle = (01,10) \\ |100\rangle = (10,10) & |101\rangle = (10,11) & |110\rangle = (11,11) & |111\rangle = (11,00) \end{array}$$

Suppose we start in state $|\psi_0\rangle = |000\rangle$. After one step, we are in state

$$|\psi_1\rangle = \widehat{M}^\dagger\,|000\rangle = \frac{1}{\sqrt{2}}\big(\,|000\rangle + |001\rangle\,\big)$$

and after another step we reach

$$|\psi_2\rangle = \frac{1}{2}\big(\,|000\rangle + |001\rangle + |010\rangle + |011\rangle\,\big)$$

Notice that since $E_\perp = \varnothing$, the associated projector P_G is the identity matrix.

Suppose that after the second step the edge $|001\rangle$ fails. Using our method we can react to this event by adding the edge $(00,01)$ to $E_\perp$. The matrix P_G is updated as introduced in the previous section, and it becomes $I - |001\rangle\,\langle 001|$.

Hence, by applying P_G to $|\psi_2\rangle$ we would delete $|001\rangle$ from the state.

A Little Comparison

We now take a little break—usually referred to in Italy as *intermezzo*—from what has been proposed in [58, 57]. We have gathered all the ingredients and finally obtained a complete procedure composed by (i) EULERIFY (ii) UNITARIZE and finally (iii) Projectors.

However, a similar approach but regarding a different class of graphs has been investigated by A. Montanaro in [127] as well. The main and key difference from what we have discussed so far regards the set of graphs taken into account. In fact, Montanaro started from the notion of *reversible* graphs. Let $G = (V, E)$ be a directed graph and let $e = (u, v) \in E$ be an edge. e is said to be reversible if there exists a path connecting v and u in G. The graph G is said to be reversible if and only if all its edges are reversible. It has been proven by Montanaro himself, that any reversible graph G equipped with a self-loop in each vertex can be straightforwardly encoded into a unitary matrix. The overall procedure is based upon a disjoint cycle decomposition. The reader may refer to [127] for the complete procedure. What we want to stress in this little paragraph is the

embedding of irreversible graphs into reversible ones as it turns out to be a procedure tightly bonded to EULERIFY since it deals with projectors.

Returning to irreversible graphs, the path towards their transformation to reversible ones begins with the identification of all those edges that are irreversible. Trivially, these edges are to be turned into reversible ones, though defining the resulting quantum walk shall require a more elaborate plan. Given a graph G, removing all such edges—irreversible ones—inevitably leads to a reversible graph G^{rev}. Indeed, no reversible edge becomes irreversible through such removal. The effect of the removal is twofold: (i) G^{rev} is structured into different reversible connected (or, equivalently, strongly connected) components $C_1, C_2, \ldots, C_D$; (ii) in G, irreversible edges always connect two distinct components C_i, C_j. Let (u, v) be the irreversible edge connecting components C_i, C_j. C_i is augmented with vertex v. To preserve reversibility of C_i, edge (v, u) is added. Finally, two distinct quantum walks W_i, W_j are constructed over reversible components C_i and C_j via the above procedure. Depending on which component the walker is currently visiting, the respecting quantum walk operator is applied.

However, with a solution comes a problem, namely that the quantum walker should be prevented from ever traversing edge (v, u), as it does not belong the original graph G. To this end, projective measurements are devised, such that, upon superposition between vertices of C_i, C_j, the walker collapses to one of the components. That is, for any connected component of G^{rev}, the following projector is defined

$$P_k = \sum_{v \in C_k} |v\rangle \langle v| .$$

After a step, the walker is measured via all projectors P_k. Measuring i or j means, respectively, finding the walker in either reversible component C_i or C_j. Thus, given measurement outcome k, the quantum walk is continued via operator W_k. Naturally, the next step could again traverse an irreversible edge. Thus, just as in the encoding procedure into Eulerian graphs, quantum walk operators and projectors need be alternated.

As the reader may have noticed, this way of tackling the embedding of a graph into a broader structure clearly resembles the one adopted in EULERIFY. Since the adoption of projectors seems to be a common *problem solver*, one question naturally arises: do their adoption have some kind of influence over the Quantum Random Walk performed? We will try to give to the reader some insights in the following sections.

4.1.4 Some Alternatives to Projectors

The algorithm proposed in [58, 57] tackled the problem of embedding graphs into eulerian ones by mean of (i) edge addiction and (ii) projectors. Such pair of operations eventually led to the introduction of the projector P_G, which is a non-unitary operation. In this section we will discuss some options to get rid of P_G.

Directed Chinese Postman Problem

By Theorem 2, in order to obtain an eulerian graph it is sufficient to *balance* each node. In Algorithm 7 we did this by adding edges from nodes with a negative balance to nodes

with a positive one. Even though the algorithm adds the minimum number of edges, it can add edges that were not in the initial graph and that must be projected during the walk. To avoid the use of projectors, a solution may be to only add copies of existing edges: this is a technique used to solve the Directed Chinese Postman Problem (DCPP) [61, 88].

In the DCPP, a postal carrier must pick up the mail at the post office, deliver them along blocks on the route and finally return to the post office. To make the job easier and more productive, every postal carrier would like to cover the route with as little travelling as possible.

Formally, we have a directed multigraph $G = (V, E)$ where each node is an intersection and each edge represents a street. A solution to the problem is a minimum-length cyclic tour that traverses each edge at least once and starts at v_0 (the postal office). If G admits an eulerian cycle, then it is a solution to the DCPP. Otherwise, some edges must be traversed more than once or, equivalently, they must be copied so that the resulting multigraph G^* is eulerian. It has been proved that DCPP admits a solution if and only if G is strongly connected [64]. This limits the set of graphs we can encode. Nevertheless, it is a reasonable assumption that enables us to remove the (costly) projectors from the circuit.

The problem is tackled via Integer Linear Programming. The optimization model (Figure 4.8) minimizes the number of copies x_e, for each edge $e \in E$, that are required to make G balanced.

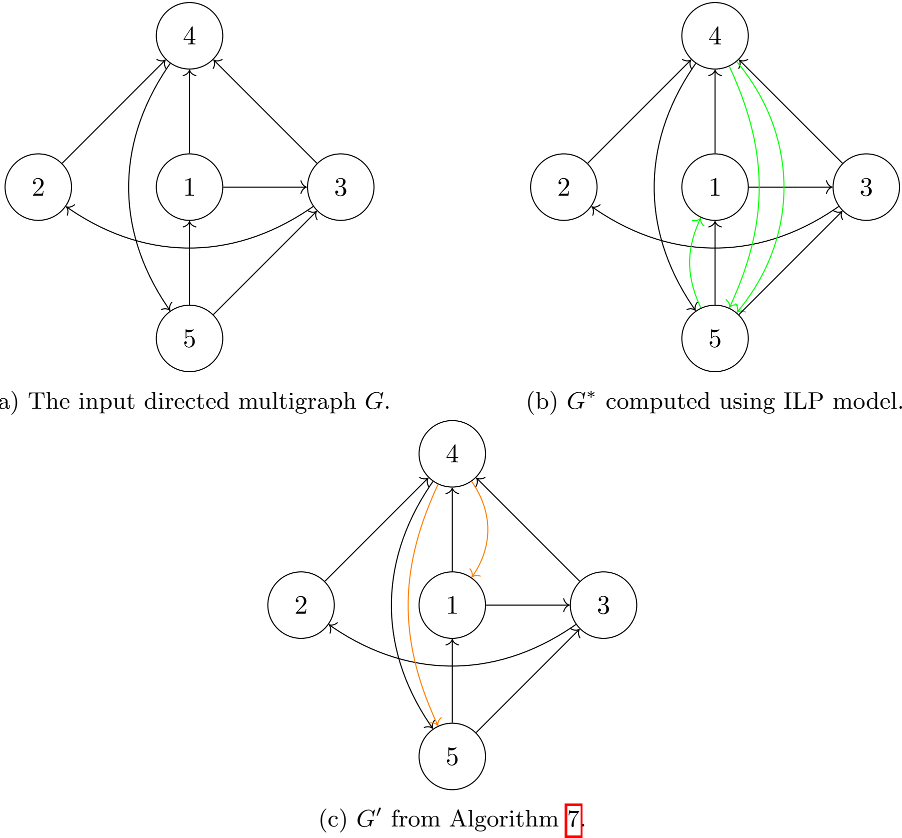
$$
\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} x_e \\
\text{subject to} \quad & \sum_{e \in \delta^-(v)} x_e - \sum_{e \in \delta^+(v)} x_e = b(v), & \forall v \in V \\
& x_e \in \mathbb{N}, & \forall e \in E
\end{aligned}
$$

Figure 4.8: The ILP model for balancing the graph.

The program models a minimum cost flow problem, which is solvable in polynomial time [62].

Given an optimal solution x^*, the graph G^* is obtained from G by adding x_e^* copies of edge e, for all $e \in E$. This new graph is balanced by construction, hence it admits an eulerian circuit.

Example 4.5. In Figure 4.9a, we depicted a graph G where nodes 1, 4, 5 are not balanced. Since the graph is strongly connected, the model admits a solution from which G^* is obtained, as shown in Figure 4.9b. On the other hand, using Algorithm 7, we would obtain a different graph G', which is depicted in Figure 4.9c. The main difference between G^* and G' is that in the former only copies of existing edges have been added, while in the latter we added the edge (4, 1) which was not present in G. Notice that in the execution of Algorithm 7 we considered nodes in B^+ and B^- sorted by label.



(a) The input directed multigraph G.

(b) G^* computed using ILP model.

(c) G' from Algorithm 7.

Figure 4.9: Multigraphs of Example 4.5.

By giving G^* as input to Algorithm 5, we obtain a matrix that can be used without projectors during the QRW.

Counters

We introduced projectors to avoid traversing edges that were not in the initial graph during the QRW. Since we use the quantum circuit formalism, the length k of the walk must be finite and fixed before compiling the circuit. Therefore, we know that all tours that will be considered during the computation have length exactly k. A tour that never traverses edges in $E_\perp$ is called *legal*, otherwise it is called *illegal*: only legal walks must contribute to a measurement.

In order to do so, we will augment the state of the walk with $m = \lceil \log(k+1) \rceil$ additional qubits to *count* the number of edges in $E_\perp$ that have been traversed by a particular tour in the multigraph. If this counter is nonzero, the walk is illegal.

Given an m-qubit register, the unitary matrix that increments its value by 1 is a

$2^m \times 2^m$ permutation that sends each element to its "successor":

$$\begin{pmatrix} 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & & \cdots & 1 & 0 \end{pmatrix} \tag{4.8}$$

The augmented state will evolve as follows:

$$\begin{cases} |\psi_0\rangle & = |e\rangle\,|0\rangle \\ |\psi_{t+1}\rangle & = (\widehat{M} \otimes I) \cdot C_m\,|\psi_t\rangle \end{cases}$$

where C_m is a matrix defined as:

$$C_m\,|e\rangle\,|c\rangle = \begin{cases} |e\rangle\,|c\rangle & \text{if } e \in E, \\ |e\rangle\,|(c+1) \mod 2^m\rangle & \text{if } e \in E_\perp \end{cases} \tag{4.9}$$

Roughly speaking, C_m increments the counter whenever an edge in $E_\perp$ is traversed and does nothing otherwise. It can be proved that C_m is unitary—blocks acting on $e \in E$ are identities and blocks acting on $e \in E_\perp$ are of the form of (4.8).

Example 4.6. Let $E = \{e_0, e_2\}$ and $E_\perp = \{e_1\}$ be the two set of edges. Consider the case of a QRW of length $k = 1$. Thus, $m = 1$ and the counter takes value $c \in \{0, 1\}$. Basis states $|e_i\rangle\,|c\rangle$ are column vectors with a single 1 in position $2 \cdot i + c$, *e.g.* $|e_0\rangle\,|1\rangle = (0\ 1\ 0\ 0\ 0\ 0)^\dagger$ and $|e_2\rangle\,|0\rangle = (0\ 0\ 0\ 0\ 1\ 0)^\dagger$.

Consider the augmented states sorted by this encoding. In our example, the matrix C_1 has the following form:

$$C_1 = \begin{pmatrix} I_2 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & X & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & I_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Notice that C_1 is a block diagonal matrix. The top-left and bottom-right 2×2 identity matrices handle the states of the form $|e_0\rangle\,|\cdot\rangle$ and $|e_2\rangle\,|\cdot\rangle$. The 2×2 center block is the permutation matrix acting on states $|e_1\rangle\,|\cdot\rangle$ as an increment modulo 2.

By generalization of Example 4.6 and by assuming the "order by encoding" of the augmented states, one can prove that C_m is a block diagonal matrix whose blocks have size $2^m \times 2^m$ and are either the identity or the increment modulo 2^m.

Notice that, with $m \geq 1$ qubits, it is possible to recognize a legal or illegal walk of length up to $2^m - 1$.

Following Equation (4.9), after applying C_m the state is evolved by applying $\widehat{M}$ on the edge-qubits and by leaving the counter-qubits unchanged. Figure 4.10 depicts the circuit implementing a single step of the visit.

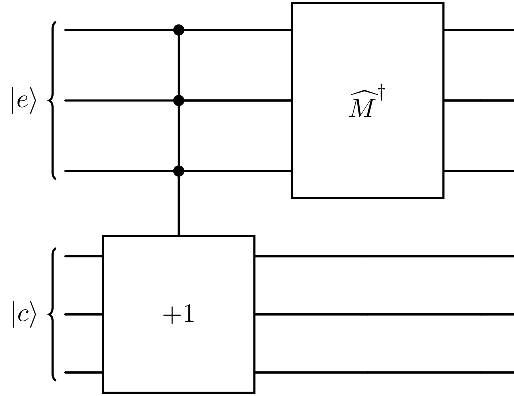


Figure 4.10: Step of the walk with counters

Let $|\psi_k\rangle$ be the last state of the walk and suppose we want to measure the probability of being in a subset $E' \subseteq E$ of a legal walk. The projector to use is the following:

$$P = \sum_{e \in E'} |e, 0\rangle \langle e, 0|$$

Consider the failure of an edge $\overline{e} \in E$. We can think of $\overline{e}$ as an illegal edge belonging to $E_{\perp}$. Therefore, the matrix C_m needs to be edited to reflect this event—identity block for $\overline{e}$ must be turned into the increment modulo 2^m. Exactly $2 \cdot 2^m \leq 4k$ bit-flips are sufficient.

4.1.5 Experimental Results

In [58, 57], we investigated experimentally the effectiveness of the encoding proposed.

The overall procedure we want to test takes a multigraph $G = (V, E)$ and a family of unitary matrices $\mathcal{U}$ as input and performs the following operations:

1. Apply Algorithm 7 on G. This step returns the set $E_{\perp}$.
2. Apply Algorithm 5 obtaining the matrix $\widetilde{M}$.
3. Use $\widetilde{M}$ and $\mathcal{U}$ as input for Algorithm 6.

The total time complexity of the procedure is $O(|E|^2)$.

We will measure the following quantities, and we will give some related statistics:

- The running time of the procedure,
- The number $|E_{\perp}|$ of edges added through the balancing procedure. This will give an insight on how distant is a random graph to become eulerian,
- The number of nonzero elements in the final matrix,

- The size of the transpiled circuit.

Random Graphs and Test Cases Generation

Test cases are generated randomly. We adopted the Erdös-Renyi-Gilbert model [77] to generate random graphs of different sizes and topologies. An Erdös-Renyi-Gilbert graph $G(n,p)$ is a directed graph with n nodes where each edge (u,v) has probability p to appear in G.

The generation of tests is done in C++ using the publicly available library BOOST [1].

We generated a sample of 120 random graphs: 10 for each pair (n,p) with $n = 10, 25, 50, 100$ and $p = 0.3, 0.5, 0.7$.

We decided to focus our attention on equiprobable QRWs—when sitting on a node u, every outgoing edge has probability $\frac{1}{d^+(u)}$ of being traversed. To ensure this, we adopted the family of unitaries $\mathcal{U} = \{\mathrm{DFT}(n) : n \in \mathbb{N}\}$, where:

$$\mathrm{DFT}(n) = \frac{1}{\sqrt{n}} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(n-1)} \\ \vdots & & & & \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \dots & \omega^{(n-1)(n-1)} \end{pmatrix}$$

and $\omega = e^{\frac{2\pi i}{n}}$.

Results

Results are presented in Table 4.1. Columns n and p are defined as in Section 4.1.5. Given n and p, the columns AVG. SIZE, AVG. NEW EDGES, AVG. NNZ, and AVG. TIME contain, respectively, the mean of the following quantities: number of edges of the graph after balancing (number of rows of the encoding matrix) , number of edges added (size of $E_\perp$), number of nonzero entries in the constructed unitary matrix, and running time for mapping a graph into its associated unitary matrix. We include the nonzero entries column since it provides a good explanation of the average time results. In fact, Algorithm 7 only edits elements that are not zero.

The obtained results are encouraging on the effectiveness of the proposed method. For instance, in graphs of size 100 and density 0.7—which have roughly 7000 edges—the average number of added edges is around 250 which is 3.5% of the initial number of edges. In the worst case the method could double the number of edges of the input graph but, the examples show that in the average we are far from that scenario, as it emerges looking at the average number of new edges in Table 4.1. As a consequence also the average number of nonzero entries of the unitary matrix and the average time are far from the worst case.

On the one hand, other *coin-based* encodings have been experimentally tested with the aim of analysing the *convergence* of the walk (*e.g.*, mixing time and hitting lime).

[1] The documentation is available at `https://www.boost.org/doc/libs/1_81_0/libs/graph/doc/erdos_renyi_generator.html`.

n	p	AVG. SIZE	AVG. NEW EDGES	AVG. NNZ	AVG. TIME (s)
10	0.3	34	8	133	< 0.01
10	0.5	55	8	311	< 0.01
10	0.7	69	7	487	< 0.02
25	0.3	215	32	1945	0.119
25	0.5	336	34	4633	0.250
25	0.7	448	31	8104	0.429
50	0.3	820	92	13831	1.186
50	0.5	1310	98	34688	3.029
50	0.7	1805	90	65457	5.908
100	0.3	3212	254	104720	17.699
100	0.5	5219	298	274053	51.520
100	0.7	7188	254	518098	96.914

Table 4.1: Experimental results: averages are over 10 random graphs per each pair (n, p).

$p = 0.3$		
n	STD-DEV NEW EDGES	STD-DEV TIME (S)
10	2	< 0.001
25	31	< 0.001
50	64	0.04
100	439	0.96

Table 4.2: Standard deviation of number of added edges and running time with $p = 0.3$.

$p = 0.5$		
n	STD-DEV NEW EDGES	STD-DEV TIME (S)
10	3	< 0.001
25	32	< 0.001
50	215	0.12
100	334	3.74

Table 4.3: Standard deviation of number of added edges and running time with $p = 0.5$.

The time required for the generation of the data structure has never been reported. In our case, since edge additions could make the size of the matrix grow critically, we were interested in testing the impact of such event.

Once the matrix has been computed for all the above-mentioned cases, it can be *transpiled* in a quantum circuit. Because of physical limitations in state of the art quantum computers, only a small set of *basic* unitaries can be physically applied to qubits. Transpilation is the act of expressing non-basic matrices in terms of basic ones.

In our experiments, we used Qiskit—a python library for creating, simulating and

$p = 0.7$		
n	STD-DEV NEW EDGES	STD-DEV TIME (S)
10	1	< 0.001
25	25	< 0.001
50	126	0.060
100	721	16.18

Table 4.4: Standard deviation of number of added edges and running time with $p = 0.7$.

p	AVG. SIZE	AVG. NNZ	AVG. TRANSP. TIME (S)	AVG. u_1	AVG. cx	AVG. u_2	AVG. u_3
0.3	64	1390	25.56	49	2956	25	3088
0.5	64	2620	30.34	54	3666	21	3834
0.7	128	4550	111.50	102	13910	24	14331

Table 4.5: Results obtained for graphs with 10 nodes.

running quantum circuits—to perform the transpilation. The basic gates we adopted are $\{u_1, u_2, u_3, cx\}$, defined as:

$$\begin{aligned} u_1(\lambda) &= U(0, 0, \lambda) \\ u_2(\phi, \lambda) &= U(\pi/2, \phi, \lambda) \\ u_3(\theta, \phi, \lambda) &= U(\theta, \phi, \lambda) \end{aligned} \qquad cx = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

where:

$$U(\theta, \phi, \lambda) = \begin{pmatrix} \cos\frac{\theta}{2} & -e^{i\lambda}\sin\frac{\theta}{2} \\ e^{i\phi}\sin\frac{\theta}{2} & e^{(\phi+\lambda)}\cos\frac{\theta}{2} \end{pmatrix}$$

The results we obtained can be found in Table 4.5 and 4.6. Tables present the following data relating the input matrix and the output (transpiled) matrix:

1. The average size (in terms of number of rows) of the input,
2. The average number of non-zero elements inside the input,
3. The average transpilation time,
4. The average number of basic gates used.

Observing the data, the transpilation time becomes really high as soon as the inputs reach the order of 512×512 elements (graphs with $n = 25$ nodes and probability $p = 0.5$). On the other hand, the number of gates to use increases already while tackling the problem with 10 nodes and $p = 0.7$.

4.1.6 How Truthful Can an Encoding Procedure Be?

Both encoding procedures here studied ([127, 58, 57]) appear to heavily rely on the use of projectors to ensure *truthfulness* with respect to the structure of the original

p	AVG. SIZE	AVG. NNZ	AVG. TRANSP. TIME (S)	AVG. u_1	AVG. cx	AVG. u_2	AVG. u_3
0.3	256	13658	504.14	187	49484	52	50414
0.5	512	31256	2063.03	359	181142	56	183314
0.7	512	58996	3083.59	462	271472	74	274395

Table 4.6: Results obtained for graphs with 25 nodes.

graph. More specifically, projectors are inevitable whenever dealing with non-strongly connected graphs: yet another hint to the restriction posed by computational reversibility and graph reversibility. The main role of this small paragraph is to introduce the reader to the idea that a *generic* algorithm encoding any directed graph into a unitary matrix may not exist without taking into account edges/nodes additions. A possible workaround which could be worth-following is a problem-dependent encoding. For example, consider the reachability problem. To solve this problem, our data structure should be able to succinctly encode the neighbourhood of a node. On the other hand, for the graph colouring problem, we would like to have a data structure to speed up the assignment of the colours. Even with just few lines, the reader may notice that the *goal* of the data structure changes with the problem. In classical computer science, we are used to exploiting the same representation for any graph problem we want to solve. In the quantum setting, due to the limitations imposed by the working of quantum mechanics, this is not possible and a change of perspective may be necessary.

Moreover, the study on truthful encoding procedures should also concern *duplicated* edges. Indeed, edge duplication also affects graph topology, albeit in a somewhat softer way. In turn, these effects appear to produce phenomena of local bias in the resulting quantum walk. Roughly speaking, to *balance*—in some sense which is usually defined by the encoding technique adopted—a graph to make it amenable for unitary encoding, we are required to add edges. A question naturally arises: do we allow adding copies of edges or not? In both cases, it is straightforward to see that some nodes may be deeply influenced by the edge addition process. For example, suppose a node u has 2 neighbours v and w. In the same context, suppose 100 copies of the edge (u, v) are added for the sake of balance. The reachability properties of the graph remains the same (u and v keeps being neighbour), but the probability of going from u to v has to be carefully set to avoid biasing the walk towards the path that goes from u to v while never visiting w.

4.1.7 Quantum walks

Before transitioning to a different paradigm of quantum computation—the Measurement-Based model—we will delve further into the concept of Quantum Random Walks. This structural choice within the section on *Graphs in Quantum Gate-Based Models* is primarily intended to maintain alignment with the treatment presented in the classical case.

Quantum walks present themselves as the extension of random walks to the principles of Quantum Mechanics. To set off the transition to the quantum realm, the particle roaming over the graph is to be understood as a quantum system. This statement alone lets the entire theory of quantum walks fall into place: one merely requires spelling out

the four postulates introduced in Section 1.6 with respect to the particle. However, the construction of quantum walks is filled with subtleties which shall be taken care of throughout this section.

Before diving into the topic, it should be noted that quantum walks as hereby defined are restricted to *discrete-time* evolution. Henceforth, when writing "quantum walk", it is assumed to speak of a discrete quantum walk.

Unitary Markov chains

A straightforward way to introduce *discrete-time* quantum walks is through the notion of *unitary Markov chains* (or *quantum Markov chains*). A unitary Markov chain is a triple $\mathcal{Q} = (\mathbb{C}^n, W, |u\rangle)$ where $\mathbb{C}^n$ is a Hilbert space with orthonormal basis $\{|i\rangle\}$, W is a unitary operator on $\mathbb{C}^n$ and $|u\rangle \in \mathbb{C}^n$ is the *initial state vector.*

Compared to their classical counterpart shown in Section 1.4.3, unitary Markov chains do not appear too different. Albeit in a new form, the memoryless property still stands: the *probability amplitude* for $\mathcal{Q}$ to lie in state $|j\rangle$ at time t solely depends on the state at time $t-1$. Replacing state space S with Hilbert space $\mathbb{C}^n$ endows $\mathcal{Q}$ with the ability to lie in a superposition of basis states: $\sum_i c_i |i\rangle$. Thus, denoting as $|\psi(t)\rangle$ the state of $\mathcal{Q}$ at time t, the state at time $t+1$ is

$$|\psi(t+1)\rangle = W|\psi(t)\rangle . \tag{4.10}$$

State vector $|\psi(t)\rangle$ is to be understood as a column vector. As a consequence, the probability amplitude for $\mathcal{Q}$ to transition from state $|i\rangle$ to $|j\rangle$ is now given by W_{ji}.

In turn, we may now say that a directed graph $G = (V, E)$ with $n = |V|$ induces a unitary Markov chain $\mathcal{Q} = (\mathbb{C}^n, W, |u\rangle)$ if it holds that

$$W_{j,i} \neq 0 \iff (i,j) \in E. \tag{4.11}$$

Notice that amplitudes need not be equally distributed: for any $(i,j) \in E$, it is merely required there be a chance to transition from state $|i\rangle$ to $|j\rangle$. On an added note, since state space $\mathbb{C}^n$ may be derived from W and the initial state $|u\rangle$ is only relevant to the time-analysis of $\mathcal{Q}$, we shall identify a unitary Markov chain $\mathcal{Q} = (\mathbb{C}^n, W, |u\rangle)$ simply as W.

Unitary Markov chains are all good and well, however, they might leave the reader feeling distant from the actual significance carried by quantum walks. These concerns are soon to be addressed, for the time being, the major take-away is that unitary Markov chains act as a certificate for quantum walks: a graph is amenable to quantum walks if it induces a unitary Markov chain. Why is the condition only sufficient? As it turns out, the restriction enforced by Equation (4.11) on unitary W is harder than the one Equation (1.32) sets on *stochastic* $\mathbf{P}$. In order not to over restrict the playground of quantum walks, coarser necessary conditions are due.

A naïve construction

Let us attempt a construction of a quantum walk starting from the random walk on the infinite line seen in Example 1.4.

To begin with, Postulate one[2] demands the particle to be described by some unit state vector. The state should represent the pertinent feature - in this case, the position of the particle on graph $G = (\mathbb{Z}, E)$. To this end, consider the canonical basis $\{|i\rangle : i \in \mathbb{Z}\}$, where $|i\rangle$ is the state for the particle lying on vertex i. The span of the basis is, then, the Hilbert space associated to the particle: $\mathbb{C}^\infty$. An immediate observation is that now the particle may lie in a superposition of different vertices.

Concerning the action of movement, it should be recalled that Postulate two enforces unitarity upon steps of the particle. Let us ingenuously test the limits implied by this condition. Assume the particle to lie on vertex 0, *i.e.,* to be described by state $|0\rangle$. To emulate the actions of the classical random walk from Example 1.4, one would ideally consider a unitary operator W such that, for any $i \in \mathbb{Z}$ and $\theta_1, \theta_2 \in \mathbb{R}$,

$$W|i\rangle = \frac{e^{i\theta_1}|i-1\rangle + e^{i\theta_2}|i+1\rangle}{\sqrt{2}}. \tag{4.12}$$

That is, an operator that produces any superposition inducing equal probabilities for the particle to collapse at either the previous or successive position.

Let us temporarily take for granted that such operator W indeed exists. Then, denoting as $|\psi(t)\rangle$ the state of the particle after t steps, we obtain

$$|\psi(1)\rangle = \frac{e^{i\theta_1}|-1\rangle + e^{i\theta_2}|1\rangle}{\sqrt{2}}. \tag{4.13}$$

So far, all seems up to code: $|\psi(1)\rangle$ is a unit vector and there is equal chance to find the particle at either vertex -1 or 1. Let us perform a second step:

$$|\psi(2)\rangle = \frac{1}{2}\Big(e^{2i\theta_1}|-2\rangle + 2e^{i(\theta_1+\theta_2)}|0\rangle + e^{2i\theta_2}|2\rangle\Big) \tag{4.14}$$

$$= \frac{1}{2}\Big(e^{2i\theta_1}|-2\rangle + e^{2i\theta_2}|2\rangle\Big) + e^{i(\theta_1+\theta_2)}|0\rangle. \tag{4.15}$$

Clearly, $|\psi(2)\rangle$ is no unit state vector and, thus, contradicts Postulate one. In turn, this allows for the conclusion that W cannot be unitary.

To which considerations do this simple example lead? Could one say that the infinite line is *not* amenable to *unbiased* quantum walks? The answer appears to be *yes*. However, we briefly postpone this discussion to appreciate how the example outlines two conditions that one would reasonably wish a quantum walk to satisfy:

- *Translation invariance.* A step of the quantum walk always exhibits the same behaviour, independently of the vertex from which it is performed;

- *One-dimensionality.* Not to be mistaken with the geometrical dimensionality of the line. With one-dimensional quantum walk one means that the state describing the particle only enjoys one *degree of freedom* - in this case, the position on the graph.

Obviously, unitarity is also part of these conditions.

[2] In this section, we refer to the Quantum Mechanics Postulate's by Postulate one, Postulate two, Postulate three and Postulate four.

What the example perhaps fails to highlight is the clash these conditions hold against each other. This peculiar behaviour was rigorously formalized by Meyer in [123] in the form of the NO-GO LEMMA.

Lemma 4.12 (NO-GO LEMMA). *In one-dimension, if a unitary quantum walk operator W satisfies the condition of translation invariance, then W always moves towards the same direction.*

The NO-GO LEMMA is a general result that applies to any undirected n-path with $n \geq 2$. With that being said, little modifications to the previous example demonstrate the result for the specific case of the infinite line. Let us give a more general definition to W. That is, for any $i \in \mathbb{Z}$,

$$W |i\rangle = \alpha |i-1\rangle + \beta |i+1\rangle . \tag{4.16}$$

The first two steps of the walk lead to state

$$|\psi(2)\rangle = \alpha^2 |-2\rangle + 2\alpha\beta |0\rangle + \beta^2 |2\rangle . \tag{4.17}$$

We then verify unitarity of $|\psi(2)\rangle$ considering its squared norm,

$$\| |\psi(2)\rangle \|^2 = |\alpha|^4 + 4|\alpha|^2|\beta|^2 + |\beta|^4 \geq \left(|\alpha|^2 + |\beta|^2\right)^2 , \tag{4.18}$$

where the right-hand inequality of Equation (4.18) turns to equality if and only if one between $|\alpha|$ and $|\beta|$ equals 0. However, because $|\alpha|^2 + |\beta|^2 = 1$, if $|\alpha|, |\beta| \neq 1$ it follows that

$$\| |\psi(2)\rangle \|^2 > \left(|\alpha|^2 + |\beta|^2\right)^2 = 1. \tag{4.19}$$

In other words, if both $|\alpha|, |\beta| \neq 1$, then $|\psi(2)\rangle$ cannot be a unit state vector. Thus, given the definition from Equation (4.16), W either always moves to the left ($|\alpha| = 1$) or to the right ($|\beta| = 1$).

Coined quantum walks

The NO-GO LEMMA clearly represents an obstacle in the construction of quantum walks on graphs. However, it also provides with hints on how to circumvent it, in the sense that it clearly asserts what a quantum walk *is not*. Having laid down a set of conditions that may not be satisfied all together, a legitimate attempt would be that of abandoning one of them. Let us overview our options.

Giving up on unitarity does not appear as a reasonable suggestion: to allow systems non-unitary evolution is to cease talking about Quantum Computing. On the other hand, translation invariance may be deemed a non-fundamental characteristic of quantum walks. Meyer himself relaxed this condition in developing Quantum Cellular Automata [123]. However, relaxing translation invariance may, in turn, alter the topology of the given graph. Because this thesis makes graph encoding its fundamental cause, it is of uttermost importance to maintain quantum walks as possibly faithful to the underlying graph. Translation invariance shall, thus, be preserved. Finally, one option is left: rejecting one-dimensionality. Enhancing the particle of a quantum walk with a second degree of freedom elegantly eludes the NO-GO LEMMA. It is arguably the

most popular solution in the literature; adopting it produces what is best known as a *coined quantum walk* (CQW) [2, 6, 131].

As previously stated, in its most basic form the particle of a quantum walk is characterized by a single degree of freedom: the position. A coined quantum walk equips the particle with a second: the *coin.* Here the nomenclature is rather unfortunate as a coin should be thought of as having an arbitrary number of faces. Intuitively, the coin should indicate the direction from where the particle came from. Let us formalize the last few statements with respect to regular directed graphs. Regular graphs are ideal to introduce CQWs as their structure is weaker than that of the infinite line, albeit strong enough to keep the explanation intuitive.

Consider the directed k-regular graph $G = (V, E)$, with $n = |V|$. Once more, to the position of the particle is associated the Hilbert space induced by the vertices of the graph, that is, $\mathbb{C}^n$ with orthonormal basis $\{|v\rangle\}_{v\in V}$. In addition, we consider the coin as a second quantum system. To the coin, Hilbert space $\mathbb{C}^k$ is associated, with orthonormal basis $\{|e\rangle\}_{e=1}^{k}$. Clearly, the idea is to have the particle described by both systems. To this end, Postulate four comes in handy, allowing to define the particle in the form of a composite quantum system. Thus, the particle shall be described by state vectors in Hilbert space $\mathbb{C}^k \otimes \mathbb{C}^n$, with orthonormal basis

$$\{|e\rangle\,|v\rangle : 1 \leq e \leq k, v \in V\}. \tag{4.20}$$

Before proceeding any further, let us intuitively justify the convenience behind working with quantum states of this form. Because G is k-regular, for any vertex u it is possible to number its outgoing edges from 1 to k. A slightly braver statement is that there exists a numbering such that, for any v, all its *incoming* edges are uniquely numbered. With that in mind, a particle in state $|e\rangle\,|v\rangle$ may be understood as a particle lying on vertex v after having traversed its e-th incoming edge.

Having reshaped the structure of our particle, its evolving behaviour also needs be rethought. As is the case for classical random walks, a step may be structured into two stages: (i) The flip of the coin; (ii) The actual step. These two phases are outlined, respectively, by Lines 2 and 3 of Algorithm 1. Analogously, let us begin our *"quantum step"* evolving the coin quantum system. As now should be well understood, this needs to be done according to some unitary operator, say C, operating on $\mathbb{C}^k$. We consider the extension $C \otimes \mathbb{I}_{\mathbb{C}^n}$, in order for C to operate on the whole composite system, albeit leaving the position unaltered.

Provided C has decreed the edge - or a superposition edges - to be traversed, all the particle must do is to reach its target. To this end, we rely on a *shift operator* T operating on $\mathbb{C}^k \otimes \mathbb{C}^n$. Given state $|e\rangle\,|u\rangle$,

$$T(|e\rangle\,|u\rangle) = |e\rangle\,|v\rangle\,, \tag{4.21}$$

where $(u, v) \in E$ is the e-th outgoing edge from u. Observe that T is a block-diagonal matrix, with each block being a permutation of vertices in V. Ideally, edge e determines the block to be applied. Moreover, because permutations are unitary matrices, T is unitary.

Finally, the CQW may be defined as the unitary operator $W = T(C \otimes \mathbb{I}_{\mathbb{C}_n})$. Since this entire construction might, at a first glance, appear obscure, let us verify whether a

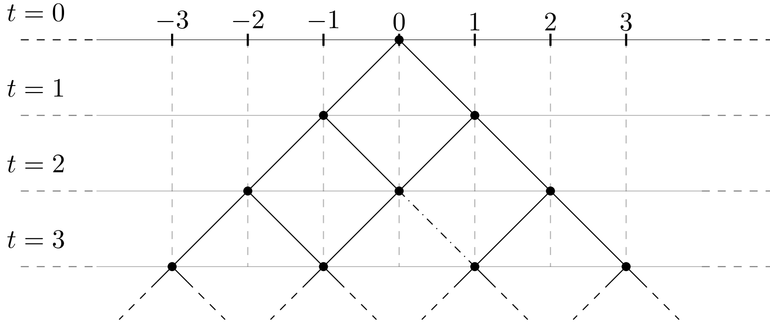


Figure 4.11: Coined quantum walk on the infinite line.

coin can help to fix our quantum walk on the infinite line.

Example 4.7 (Coined quantum walk on the infinite line)**.** To begin with, let us observe that the infinite line $G = (\mathbb{Z}, E)$ is a 2-regular graph. Accordingly, to the coin is associated to Hilbert space $\mathbb{C}^2$ with orthonormal basis $\{|0\rangle, |1\rangle\}$. As a more convenient nomenclature, let us relabel the basis as $\{|\mathtt{L}\rangle, |\mathtt{R}\rangle\}$, where the basis states represent, respectively, the edges going left and right.

The quantum walk is unbiased as long as its coin is. To this end, we define it to evolve according to *Hadamard operator* H:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad \text{where} \quad \begin{aligned} H|\mathtt{L}\rangle &= \tfrac{|\mathtt{L}\rangle+|\mathtt{R}\rangle}{\sqrt{2}}; \\ H|\mathtt{R}\rangle &= \tfrac{|\mathtt{L}\rangle-|\mathtt{R}\rangle}{\sqrt{2}}. \end{aligned} \tag{4.22}$$

Defining the shift operator T is now intuitive:

$$T(|\mathtt{L}\rangle|i\rangle) = |\mathtt{L}\rangle|i-1\rangle; \qquad T(|\mathtt{R}\rangle|i\rangle) = |\mathtt{R}\rangle|i+1\rangle; \tag{4.23}$$

All appears to be set to verify where the first two steps of $W = T(H \otimes \mathbb{I}_{\mathbb{C}^\infty})$ will lead to. Figure 4.11 provides a visualization to follow along. Each horizontal line describes the walk at a given time-step. Black dots denote possible positions of the particle at a given step.

Let us assume to start from state $|\mathtt{L}\rangle|0\rangle$. Tossing the quantum coin gives

$$H \otimes \mathbb{I}_{\mathbb{C}^\infty}|\mathtt{L}\rangle|0\rangle = \frac{|\mathtt{L}\rangle|0\rangle + |\mathtt{R}\rangle|0\rangle}{\sqrt{2}}. \tag{4.24}$$

Then, finishing the first step we obtain

$$|\psi(1)\rangle = \frac{|\mathtt{L}\rangle|-1\rangle + |\mathtt{R}\rangle|1\rangle}{\sqrt{2}}. \tag{4.25}$$

Analogously, performing the second steps leads us to

$$|\psi(2)\rangle = \frac{|\mathtt{L}\rangle\,|-2\rangle + |\mathtt{R}\rangle\,|0\rangle + |\mathtt{L}\rangle\,|0\rangle - |\mathtt{R}\rangle\,|2\rangle}{2}. \tag{4.26}$$

Differently from Equation (4.17), $|\psi(2)\rangle$ is now a unit state vector. Due to the additional degree of freedom, particle states $|\mathtt{L}\rangle\,|0\rangle$ and $|\mathtt{R}\rangle\,|0\rangle$ are distinct: their amplitudes may not be summed.

Although seemingly tedious, it is worth observing the effects of a third step of the CQW. To avoid too cumbersome equations, let us solely focus on the evolution of states $|\mathtt{R}\rangle\,|0\rangle\,, |\mathtt{L}\rangle\,|0\rangle$,

$$W\,|\mathtt{R}\rangle\,|0\rangle = \frac{|\mathtt{L}\rangle\,|-1\rangle - |\mathtt{R}\rangle\,|1\rangle}{\sqrt{2}}; \qquad W\,|\mathtt{L}\rangle\,|0\rangle = \frac{|\mathtt{L}\rangle\,|-1\rangle + |\mathtt{R}\rangle\,|1\rangle}{\sqrt{2}}. \tag{4.27}$$

State $|\psi(3)\rangle$ will have these two results summed together, leading to

$$|\psi(3)\rangle = \frac{1}{2\sqrt{2}}\Big[\cdots + \Big(|\mathtt{L}\rangle\,|-1\rangle + |\mathtt{L}\rangle\,|-1\rangle\Big) + \Big(|\mathtt{R}\rangle\,|1\rangle - |\mathtt{R}\rangle\,|1\rangle\Big) + \cdots\Big]. \tag{4.28}$$

This could be understood as the quantum walk equivalent of the well known *double-slit experiment:* whereas two paths converging to state $|\mathtt{L}\rangle\,|-1\rangle$ double the respective probability amplitude, the two paths heading towards state $|\mathtt{R}\rangle\,|1\rangle$ end up cancelling each other out. These two phenomena are known, respectively, as *constructive* and *destructive interference* and characterize the peculiar behaviour of quantum walks: *more paths leading to the same position do not necessarily imply a higher probability of reaching it.*

Measurements in Quantum Walks

Thus far, each postulate of quantum mechanics has been employed in the definition of quantum walks except for the third; where does measurement fit into the formalism? Since measurement acts as the single tool to extract classical information from quantum systems, it should eventually be used to obtain insights about the position of the particle.

To this end, one may define a projective measurement M_V on the vertices of the graph:

$$M_V = v\,|v\rangle\,\langle v|\,. \tag{4.29}$$

According to Postulate 3, such operation causes the particle to collapse on the measured vertex. With that in mind, it is possible to show that intertwining steps of a quantum walk with measurements leads to a behaviour equivalent to that of classical random walks.

Typically, measurement is linked to quantum walks with respect to their time-analysis [2, 131]. Since this connection does not quite meet the purposes of this thesis it shall not be expanded any further. On this matter, the curious reader is also referred to [6], where measurement operators have been used to define quantum walks with absorbing boundaries.

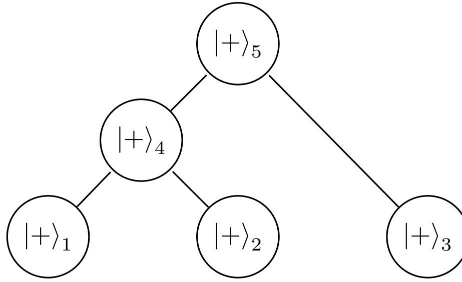


Figure 4.12: A hypothetical cluster state generated from a graph $G = (V, E)$. Each qubit is set to state $|+\rangle$. A CZ gate is applied to qubits connected by edges.

4.2 Graphs in Measurement-Based Model

We now move to the explanation of the Graph Encoding problem in the Measurement-Based Quantum Settings. In this case, the solution is pretty straightforward for undirected graphs where the notion of *Graph Cluster State* is adopted [170]. Given an undirected graph $G = (V, E)$, the connectivity of the cluster state is exactly that of G. Qubits reside on the vertices V. They are all initialized to $|+\rangle$ and CZ gates are applied to pairs of qubits connected by an edge in E.

The case of directed graph is not that easy. In fact, the CZ gate is symmetrical in the two input qubits, meaning:

$$\mathrm{CZ}\,|i\rangle\,|j\rangle = (-1)^{ij}\,|i\rangle\,|j\rangle$$
$$\mathrm{CZ}\,|j\rangle\,|i\rangle = (-1)^{ij}\,|j\rangle\,|i\rangle$$

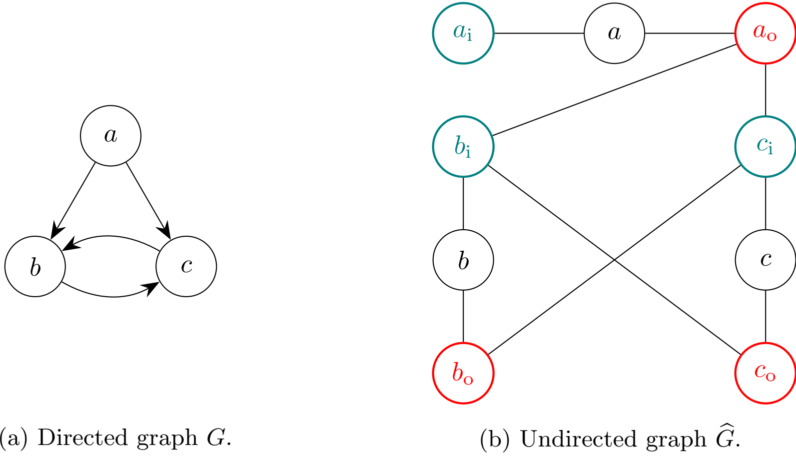
Hence, describing a directed edge using the CZ gate is not so trivial. An attempt has been made in [41], where the authors encoded Finite group into cluster states.

Instead, the problem is hereby approached by providing a procedure to encode directed graphs into undirected ones, with the condition that the encoding be uniquely reversible, *i.e.,* no two directed graphs are encoded into the same undirected graph. A directed graph $G = (V, E)$ may be reversibly encoded into an undirected graph $\widehat{G} = (\widehat{V}, \widehat{E})$, where

- $\widehat{V} = V \cup \{\, v_\mathrm{i}, v_\mathrm{o} \,:\, v \in V \,\}$;
- $\widehat{E} = \big\{\{v_\mathrm{i}, v\}, \{v_\mathrm{o}, v\} : v \in V\big\} \cup \big\{\{u_\mathrm{o}, v_\mathrm{i}\} : (u, v)\big\}$.

In other words, for any vertex $v \in V$, the undirected graph $\widehat{G}$ provides two vertices $v_\mathrm{i}, v_\mathrm{o}$ to encode, respectively, the incoming and outgoing edges to and from v. These two kinds of vertices are identified, respectively, by colours GREEN and RED. More formally, for any $v \in V$, $v_\mathrm{i}.\mathrm{color} = \mathrm{GREEN}$, $v_\mathrm{o}.\mathrm{color} = \mathrm{RED}$ and $v.\mathrm{color} = \mathrm{BLACK}$. Figure 4.13 provides a visual example of the encoding.

To show that the encoding is indeed reversible, it is demonstrated that it is injective up to isomorphism.



(a) Directed graph G.

(b) Undirected graph $\widehat{G}$.

Figure 4.13: Reversible encoding from directed to undirected graphs.

Claim 4.2 (Injectivity)**.** *Let* $G_1 = (V_1, E_1)$ *and* $G_2 = (V_2, E_2)$ *be two directed graphs and* $\widehat{G}_1 = (\widehat{V}_1, \widehat{E_1}), \widehat{G}_2 = (\widehat{V}_2, \widehat{E}_2)$ *be their respective undirected encodings. Then,* $\widehat{G}_1 \cong \widehat{G}_2$ *implies* $G_1 \cong G_2$*.*

Proof. Since $\widehat{G}_1 \cong \widehat{G}_2$, there is a mapping $\phi : \widehat{V}_1 \to \widehat{V}_2$ such that for any $v \in V_1$, v.color $= \phi(v)$.color and for any $\{u, v\} \in \widehat{E}_1$, $\{\phi(u), \phi(v)\} \in \widehat{E}_2$. To retrieve graph G_1, let $V_1 = \{v \in \widehat{V}_1 : v.\text{color} = \text{BLACK}\}$ while E_1 is reconstructed as

$$E_1 = \{(u, v) : \forall u, v \in V_1,\ \exists u', v' \text{ s.t. } u'.\text{color} = \text{RED}, v'.\text{color} = \text{GREEN}$$
$$\text{and } \{u, u'\}, \{u', v'\}, \{v', v\} \in \widehat{E}_1\}.$$

After analogously constructing G_2, the same mapping ϕ witnesses an isomorphism between G_1 and G_2. Indeed, for $(u, v) \in E_1$, one obtains $\{u, u'\}, \{u', v'\}, \{v', v\} \in \widehat{E}_1$ and $\{\phi(u), \phi(u')\}, \{\phi(u'), \phi(v')\}, \{\phi(v'), \phi(v)\} \in \widehat{E}_2$, from which one concludes $(\phi(u), \phi(v)) \in E_2$. ☐

This encoding procedure is but one of the numerous ways in which a directed graph may be reversibly transformed into an undirected one. However, it outlines an insightful parallelism with the contents in Section 4.1: whereas quantum walks required the encoded graph to be reversible, the measurement based model is fit to represent general undirected graphs.

More elaborate procedures could provide undirected graphs endowed with graph-theoretical properties that synergize with the measurement based model. For instance, the encoding provided above may be modified to avoid the use of BLACK vertices and output bipartite graphs, provided an additional vertex attribute is added.[3] It is unclear, however, whether bipartite graphs provide any advantage in either computational or descriptive terms in the measurement based model. As such, the discussion about the

[3]The encoded bipartite graph would be of the form $\widetilde{G} = (R, G, \widetilde{E})$, where R and G denote, respectively, the sets of RED and GREEN vertices.

development of more complex encoding procedures for undirected graphs is left as a potential direction for future developments.

In any case, we would like the reader to pay attention to the fact that, exactly as in gate-based quantum computation, encoding a directed graph is much harder than encoding an undirected one.

4.3 Graphs in Adiabatic Model

For what concerns the problem of graph encoding, plenty of algorithms have been developed to solve graph-related optimization problems. For example in [95], the authors solved the problem of computing the graph edit distance using adiabatic quantum computation. Another example is the graph colouring problem, that was solved with an adiabatic approach in [113]. A lot of other problems have been tackled, and Graph Isomorphism could not miss the party [89].

The reader may refer to [47] for a set of examples of QUBO encoded graph-related problems.

The encoding of a graph into a Hamiltonian is not related only to QUBO problems. In quantum computation—gate based—the same problem is faced when talking about Continuous Time Random Walks [125]. In that case, once the Hamiltonian is obtained, Schrödinger's equation takes care of making the system *walk* only on the graph's edges.

Given an undirected graph G, there are many ways to compute a Hermitian matrix that encodes G. Two of them are G's adjacency matrix and its Laplacian matrix [125]. In the case where G is directed, its adjacency matrix is not a candidate since it is not symmetric. The problem is solved by tweaking the definition of Laplacian L of a directed graph [52]. L is computed as a linear combination of two matrices M and Π. The matrix M is the adjacency matrix of G. On the other hand, Π is obtained using the Perron-Frobenius Theorem from [93].

The reader may notice that with respect to Discrete Time Quantum Random Walk, the process of encoding a graph is way easier in the continuous setting.

On top of this, to solve a problem with Adiabatic Quantum Computation, we must be sure that the ground state of H_{final} is exactly the solution to our problem. Hence, even if it could be simple to encode a graph, it may be hard to find the correct H_{final}. This is in contrast with the gate based approach, where graph encoding may be difficult, but algorithms are easy to devise.

To conclude, we must highlight how a migration to the Adiabatic Quantum Computing paradigm would lead to a consistent switch of the view on computation. In particular, once the physical system has been *fed* with the two Hamiltonians, the inner behaviour of the system remains hidden to the user/developer. On the other hand, in the classical settings we are used to having control even on the smallest part of our architecture at every moment. This change of perspective would pave the way to a *think quantum* revolution. Trading the capability of *explaining* what is going on in the system to obtain faster algorithm for optimization problems.

Conclusions on Graphs in Quantum Architectures

The primary outcome of this chapter is an end-to-end procedure for encoding any graph into a unitary matrix suitable for gate-based quantum models. This procedure has been thoroughly detailed, compared to alternative methods, and experimentally evaluated.

Our exploration of other quantum computation models—adiabatic and measurement-based—was more limited compared to the focus on the gate-based approach.

The central insight we wish to convey is this: Why do we encounter such varying levels of complexity when addressing the same problem—graph encoding—across different quantum architectures? While the gate-based model demands a detailed and intricate approach involving graph theory and matrix construction, the adiabatic model appears especially well-suited for graph encoding due to its simpler requirement of producing a Hamiltonian rather than a unitary matrix. Measurement-based quantum computing, however, faces some intrinsic challenges in graph or cluster-state encoding.

To spark further thought, we pose the following question:

What if we combine all of these quantum architectures to create an all-in-one Quantum Processor?

The gate-based model excels at implementing brief quantum routines, where approximate results with errors are acceptable. The adiabatic model, as discussed, is particularly well-suited to graph-related computations. While the role of measurement-based quantum computation remains somewhat unclear in this context, topological quantum computation may serve as a perfect third component for our Quantum Processor. Although not previously covered in depth, architectures leveraging topological structures known as anyons, termed topological quantum computers, have recently emerged. These systems exhibit both *complete* and *robust* fault tolerance.

Given the challenges in achieving a fault-tolerant gate-based quantum computer, adding a topological component to a Quantum Processor may be a viable solution for tasks requiring absolute accuracy.

Conclusions on Graphs and Their Representations

We can *informally* draw a conclusion to the part of this thesis aimed at addressing a foundational question:

How do we encode graphs?

The answer to this question is straightforward as long as we remain within the classical paradigm of computation. As thoroughly described in the first chapter of this section, data structures such as adjacency lists, adjacency matrices, and OBDDs are well-established for storing and manipulating graphs.

When we shift our focus to parallel computation, the answer remains relatively clear. This paradigm, while distinct, is still essentially *classical*, where CPUs are replaced with GPUs, allowing for parallel computations. In this context, we reviewed and briefly presented techniques for optimizing space when handling (sparse) graphs.

Finally, we approached the most complex chapter: the one involving quantum computation. Here, we demonstrated how graphs are divided into two major categories: Eulerian and non-Eulerian. While Eulerian graphs lend themselves to quantum encoding—through a unitary matrix—non-Eulerian graphs do not exhibit the same property. Consequently, to leverage quantum speed-ups for graph-related problems[4], we must first develop a method to encode *any* graph as a unitary matrix.

In the quantum computing chapter, we examined a classical procedure that achieves this objective by utilizing results from line graph theory. We compared this method with another from the literature, highlighting their similarities and differences.

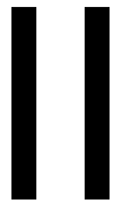
Concluding this chapter, we examined how these encodings can be applied to Quantum Random Walks, paralleling our work in classical computation. Due to their inherently probabilistic nature, Quantum Random Walks are among the most natural approaches for graph traversal in quantum computation.

Numerous open questions remain regarding this encoding. Some of these are:

- Can we develop a faster graph-to-unitary algorithm?
- Given a graph G, let U_G be its unitary encoding. How complex is it to synthesize U_G in terms of smaller quantum gates?

Answering these questions is certainly a path worth exploring.

[4] For instance, graph isomorphism.



Graphs as Semantics

Introduction to Graphs as Semantics

In this part, we will explore how graphs can serve as a semantic framework to interpret and solve various problems. The term *semantics* is somewhat broad, so narrowing it down will help clarify the aim of what lies ahead. In computational problem-solving, problems are often presented in specific encodings. A common approach in competitive programming, for instance, is to *conceal* a computational problem within an elaborate story, creating an initial layer of challenge for contestants. They must interpret what the problem is asking within a particular semantic framework. For instance, a scheduling problem involving cars might map naturally onto graphs and routes, while board game problems often draw on linear algebra, and so forth. Here, the *semantics* of a problem refers to the formal meaning or framework we assign to a given task. Some cases make this semantic assignment straightforward—such as automata, where graph-based methods are clearly relevant. Others, like 2-SAT problems[140], appear at first to belong to logic alone. However, graphs' versatility and *flexibility* often provide powerful tools for addressing such challenges. Let ϕ be a boolean formula over n variables $x_1, x_2, \cdots, x_n$ with the following form:

$$\phi = C_1 \wedge C_2 \wedge C_3 \wedge \cdots \wedge C_m$$

where each *clause* C_i is as follows:

$$C_i = \overline{x}_{i_1} \vee \overline{x}_{i_2}$$

where each $\overline{x}_{i_j}$ is either a variable or a negated variable. Notice that ϕ is a Krom formula or, equivalently, a formula in 2 Conjunctive Normal Form (2-CNF).

A 2-SAT instance is a boolean formula ϕ and the problem requires answering whether ϕ is satisfiable or not—if there exists an assignment of values to the variables such that the formula evaluates to true. How do we solve this problem?

We first start by recalling a well-known equivalence from boolean logic:

$$A \vee B \equiv \neg A \Rightarrow B$$

This result can be graphically presented as in Figure 4.14 where each variable is a node, and the implication is represented as an edge.

The graph built adopting this particular approach, starting from a 2-CNF boolean formula ϕ is called implication graph. An example is displayed in Figure 4.15. Formula ϕ from such example has 3 clauses. The first one is $\neg x_3 \vee \neg x_1$, which is equivalent to $\neg(\neg x_3) \Rightarrow \neg x_1 \equiv x_3 \Rightarrow \neg x_1$. Such clause induces the edge between nodes labelled x_3

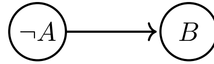


Figure 4.14: An implication as a graph

and $\neg x_1$—depicted in orange in the example. Applying the same process to the second and the third clause, we obtain edges between $x_2, \neg x_1$ and $\neg x_1, x_4$, respectively—the former is drawn in red while the latter in green. The question that naturally arises is

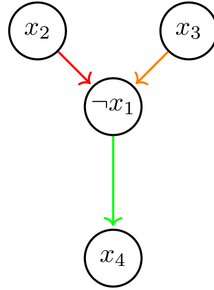


Figure 4.15: Part of the implication graph of $\phi = (\neg x_3 \vee \neg x_1) \wedge (\neg x_2 \vee \neg x_1) \wedge (x_1 \vee x_4)$

the following: how does this graph help in solving 2-SAT? The answer has been given in [19], where the authors devised an algorithm that works as follows. Given an instance of 2-SAT, dealing with some boolean formula ϕ, build its implication graph G. Proceed by computing the strongly connected components of G. Thinking of edges as carriers of *truth*—imagine as if the edges make the truth values spread across the graph—the following holds

ϕ is satisfiable if and only if there is no variable that belong to the same connected component as its negation.

This example demonstrates how a seemingly unrelated problem can be effectively solved using graph theory. By translating a problem into the context of graphs, it can become easier to understand and solve.

2-SAT is just one of many problems that can benefit from this approach. It is quite common to address a problem by first representing it in terms of graphs and then applying graph-related concepts to find a solution. This introduction serves as a proof of concept for what we will accomplish in the following chapters. We will begin by presenting a problem, translating it into a graph-based framework, and using this representation to solve the task more efficiently. The structures we will work with include automata and ASP models. The former have an inherent connection to graphs—automata are graphs—while the latter also relate to graphs, though in a slightly more abstract way.

5

Automata

In this chapter we describe and investigate the notion of automata. We decided to start from this topic since it is a clear example of how graphs can be the *landscape* to encode more complex semantics. In particular, automata can be roughly described as graphs with labels on edges, in which some states are more important than others. This partition of the nodes according to their *role* inside the graph, will allow us to characterize the idea of *languages*—a core component of theoretical computer science, spanning from computability to complexity theories.

Before starting with our discussion, we recall some basic definitions from language theory. Given a string $\mathbf{x} = x_1 x_2 \dots x_m$ we denote by $\overleftarrow{\mathbf{x}}$ its mirror image, i.e., the string $\overleftarrow{\mathbf{x}} = x_m x_{m-1} \dots x_1$. Given an index $1 \leq j \leq n$ we denote by $\mathbf{x}_j$ the prefix of $\mathbf{x}$ from x_1 to x_{j-1}, i.e., $\mathbf{x}_j = x_1 x_2 \dots x_{j-1}$. If $j = 1$, then $\mathbf{x}_j$ is the empty string. Moreover, for $h \in \mathbb{N}$ we denote by $\mathbf{x}_j^h$ the sub-string of $\mathbf{x}$ ranging from x_{j-h} to x_{j-1} if $j - h > 0$, and the prefix $\mathbf{x}_j$ otherwise. In other terms, $\mathbf{x}_j^h$ is the sub-string of $\mathbf{x}$ ranging from x_k to x_{j-1}, where k is the maximum between 1 and $j - h$. Notice that $\mathbf{x}_j^h$ has length either h or $j - 1$. A language L is a set of strings over an alphabet Σ, i.e., $L \subseteq \Sigma^*$. Given a language L, we denote by $\overleftarrow{L}$ the mirror image of L, i.e., $\overleftarrow{L} = \{\overleftarrow{\mathbf{x}} \mid \mathbf{x} \in L\}$.

5.1 Definition of Deterministic Finite Automata

A finite state automaton is a mathematical model that represents a system with discrete inputs and discrete outputs. The system we describe has the property of being in one state between a set of possible ones. A new ingredient is a (possibly) infinite tape in which a word is encoded in some alphabet. A new ingredient that distinguish automata from simple graphs is a tape where the input string is provided. We assume a moving *head* to read one symbol from such tape and then moves one step to the right. The symbol read is processed by the automaton. How an automaton reacts to a new input depends on from both current state and symbol. The way in which an automaton works for any possible pair (state, symbol) can be encoded in two ways:

- Using a *transition matrix*

	a	**b**
1	2	3
2	3	1
3	3	2

Table 5.1: A legal transition matrix for an automaton with 3 states and an input alphabet with 2 symbols.

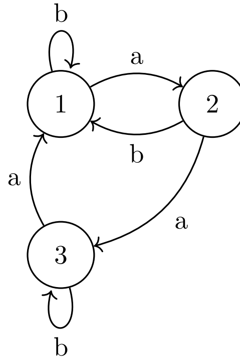


Figure 5.1: Graph representing an automaton.

- Using a graph

The transition matrix is a table labelled with states on the rows and input symbols on the columns. When a symbol σ is read as input while the automaton is currently in state s, the cell at position s, σ contains the name of the next state to move into.

Example 5.1. Suppose the set of states is $\{1, 2, 3\}$ while the input symbols can be either a or b. Then, a *legal* transition matrix is depicted in Table 5.1. We now briefly explain how the behaviour of the automaton is encoded in the transition matrix. Pick for example state 2, hence the row with a **2** as leftmost symbol. Suppose the automaton is in such state, then its behaviour is:

- When input is **a**, automaton moves to state 3.
- When input is **b**, automaton moves to state 1.

The other approach to encode the behaviour of an automaton is through graphs. The set of nodes is exactly the set of states. The set of edges is defined as follows: an edge labelled σ between a state s_1 and state s_2 exists if and only if the automaton when reading σ from state s_1, moves to state s_2.

We clarify this definition with the following example.

Example 5.2. Consider the same automaton as in Example 5.1. Its transition matrix can be turned into the graph depicted in Figure 5.1.

As for the matrix case, we focus on node—state—2. The edge with label **a** going from 2 to 3 encodes the change of state from 2 to 3 when the input symbol is **a**. Analogously

for the edge going from 2 to 1 with label **b**. Notice that graphs provide a more natural way to represent the behaviour of an automaton, as state transitions are simulated by traversing edges. Consequently, the expression "Automaton goes from state to state" mirrors the movement within a graph.

We now move to a formal description of automata, trying to bridge the technicalities with the *graphical* representations we just introduced. Clearly, we cannot explain the whole theory and results behind automata theory. Therefore, we refer the reader to [92] for further details.

Definition 5.1 (Deterministic Finite Automaton)**.** A Deterministic Finite Automaton—DFA—is a tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a *finite* set of states. We usually denote with n the number of states. Unless otherwise specified, $Q = \{q_0, q_1, q_2, \cdots, q_n\}$
- Σ is the input alphabet—finite set of symbols.
- $\delta : Q \times \Sigma \to Q$ is the transition function. Given a pair q, σ, $\delta(q, \sigma)$ is the next state in the computation.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of the final states.

Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Its encoding using a transition matrix is a one to one encoding of the function δ. Such description lacks the capability of clearly identifying the initial state and the final ones. On the other hand, the graph encoding is more complete and direct. In fact, the initial state of an automaton depicted as a graph is usually the only one with a sourceless edge labelled *start*. Moreover, the final states are depicted with a double circle node.

Example 5.3. Consider the graph-like automaton used in Example 5.2. We rename its states with the following rule: state i gets name q_{i-1}, so that state 1 is the initial one. Suppose state q_2—state 3—is the only final. The graph redrawn to encode such new information is depicted in Figure 5.2.

Formally speaking, automaton in Figure 5.2 is defined as:

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{\mathbf{a}, \mathbf{b}\}$
- $\delta(q_i, \sigma) = q_j$ if and only if an edge labelled σ has source in q_i and destination in q_j
- q_0 is the former state 1.
- $F = \{q_2\}$

We used Example 5.3 to show how a graph encodes an automaton. We now formalize how to draw a graph starting from a formally defined automaton. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We want to build, and draw, its *associated graph* $G_{\mathcal{A}} = (V, E)$

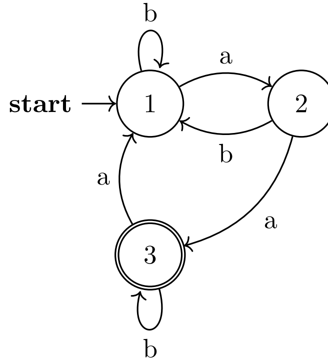


Figure 5.2: Graph with final and initial states encoded.

that *works* as $\mathcal{A}$—notice how we introduced the notion of *workings* of a graph, even if it should only serve as a representation method while now it has a semantic. The set of nodes V is exactly defined as Q. Notice that V contains the initial state q_0 and all the final states in F. The alphabet Σ is the set of possible edge labels for G. Finally, the edges E are defined as follows:

$$(q_i, \sigma, q_j) \in E \leftrightarrow \delta(q_i, \sigma) = q_j$$

Using this equivalence it is straightforward to go from an automaton to its graph and viceversa.

5.2 A Brief Recall of a DFA Semantics

We can now focus on an automaton semantics and, in parallel, of its associated graph. First, the reader may notice that input are usually words in Σ^*, but we gave a definition of σ in terms of a single symbol. Nevertheless, the transition function can be recursively extended to strings obtaining $\delta^* : Q \times \Sigma^* \to Q$:

$$\delta^*(q, s) = \begin{cases} q & \text{if } s = \epsilon \\ \delta(\delta^*(q, s'), \sigma) & \text{if } s = s'\sigma \text{ with } s' \in \Sigma^*, \sigma \in \Sigma \end{cases}$$

Now that we extended δ to its *string* version, we can introduce the notion of *Language accepted* by an automaton. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be an automaton and let $w \in \Sigma^*$ be a string. We say that $\mathcal{A}$ accepts w if and only if the following holds:

$$\delta^*(q_0, w) \in F$$

which means that spelling w symbol per symbol, starting from the initial state q_0, the automaton reaches a final state. The language $L(\mathcal{A})$ accepted by an automaton $\mathcal{A}$ is defined as:

$$L(\mathcal{A}) = \{w \in \Sigma^* \ : \ \delta^*(q_0, w) \in F\}$$

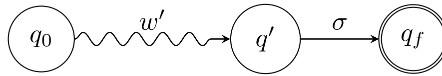


Figure 5.3: Automaton going from q_0 to q' reading w'. Reading σ from q' ends up in q_f

Roughly speaking, it is the set of all the words *accepted* by $\mathcal{A}$.

Notice that while introducing the concept of automaton accepting a word, we did not adopt the term *reaches* out of the blue. In fact, the concept of word acceptance in automata theory can be turned into a reachability concept in the graph realm. Let $w \in \Sigma^*$ be an input string and let $\mathcal{A}$ be the just defined automaton. Suppose A accepts w, which means that $\delta^*(q_0, w) \in F$. Using the definition of $\delta^*(q_0, w)$, the following holds:

$$\delta^*(q_0, w) \in F \leftrightarrow \delta(\delta^*(q, w'), \sigma) \in F$$

where w can be decomposed as $w' \in \Sigma^*$ and $\sigma \in \Sigma$. Let $q' = \delta(\delta^*(q, w')$. If we assume $\delta(\delta^*(q, w'), \sigma) \in F$, it must be true that $\delta(q', \sigma) \in F$. For the sake of readability, let $F = \{q_f\}$. Hence, we can rewrite what just stated as follows:

$$\delta(q', \sigma) \in F \leftrightarrow \delta(q', \sigma) = q_f$$

Moreover, using the relation between δ and the edges of $G_{\mathcal{A}}$ defined in 5.1, we obtain:

$$\delta(q', \sigma) \in F \leftrightarrow \delta(q', \sigma) = q_f \leftrightarrow (q', \sigma, q_f) \in E$$

Which means that w is accepted by $\mathcal{A}$ if and only if there is an edge between q' and q_f, with label σ. Lifting this approach to the *remainder* of w, namely w', it must be true that:

- q_0 reaches q' reading w'. This implies two things: (i) there must be a path connecting q_0 and q'. (ii) the word spelled reading the labels of the edges that compose such path has to be w'.

- q' must be connected to q_f with an edge labelled σ.

We tried to graphically explain the above concepts in Figure 5.3. We depicted with a curly edge the path that connects q_0 with q' and spells w'. On the other hand, we used a straight line labelled σ to connect q' and q_f.

We used the notion of word acceptance as a witness for the symmetry that exists between objects like graphs and automata—in some sense, as we stressed before, automata introduce a notion of semantics to a relatively *static* entity like a graph.

Another example can be made talking about the notion of regular language and, in particular, the pumping lemma.

Definition 5.2 (Regular Language)**.** Let $L \subseteq \Sigma^*$ be a language over Σ. L is said to be *regular* if there exists a DFA $\mathcal{A}$ such that

$$L(\mathcal{A}) = L$$

Given a language L, it may not be straightforward to build an automaton $\mathcal{A}$ that accepts it. On the other hand, given an automaton $\mathcal{A}$, it may not be easy to prove that

the language accepted is exactly L. However, the pumping lemma has been introduced to ease this step.

Theorem 14 (Pumping Lemma(PL)). *Let L be a regular language. Then, there exists $n \in \mathbb{N}$ such that for every $z \in L$, with $|z| \geq n$, there are three strings u, v, w for which:*

- $z = uvw$
- $|uv| \leq n$
- $|v| > 0$
- *for every $i \geq 0$, it holds that $uv^i w \in L$*

This theorem is usually adopted in its negated form to find a witness $z \in L$ to prove that L is not regular. However, what is interesting about this result is how it easily translates in the graph-theory playground. Since L is regular, there exists an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ that accepts it. Let $G_{\mathcal{A}}$ be the graph associated to the automaton $\mathcal{A}$. Let $z \in L$ with $|z| \geq n$, with $n = |Q|$:

1. there exists a path π_z in $G_{\mathcal{A}}$ that connects the node q_0 with a node $q_f \in F$,
2. following such path, the word spelled is exactly z

Since z can be decomposed as u, v, w, it means that π_z can be decomposed in three *subpaths* π_u, π_v, and π_z. Generally speaking, the three paths should create the following structure:

1. The first—π_u—starts from q_0 and goes to some state q' while spelling u,
2. The second—π_v—starts from q' and goes to some state q'' while spelling v,
3. The third—π_w—starts from q'' and goes to the final state q_f while spelling z

The above *general* situation is depicted in Figure 5.4.

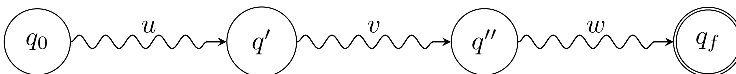


Figure 5.4: A generic path to read a word z, decomposed as three paths to read u, v, w.

What is proven using PL is that state q' and state q'' are actually equal and, as a consequence, that the path traversed while reading v is actually a loop—this result is obtained using the pigeonhole principle starting from the hypothesis that $|z| \geq |Q|$. The situation created with this new insight is shown in Figure 5.5.

Roughly speaking, if a word has more symbols than the number of states, the path that leads to its acceptance must contain a loop (traversed while spelling v). The other two words u and w are used to move *towards* the loop—u—and far from the loop up to the final state—w.

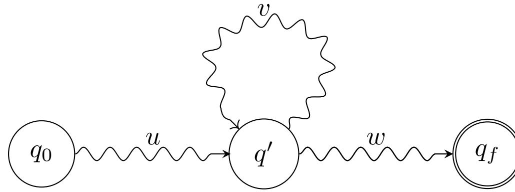


Figure 5.5: The path to read z taking into account the results of the Pumping Lemma

5.3 Some Problems Related to Automata Theory

This particular theorem, together with its graph-related interpretation, is a key ingredient to solve two important problems related to regular languages.

Definition 5.3 (Emptyness Problem)**.** Let $\mathcal{A}$ be a DFA. The emptiness problem asks to decide whether $L(\mathcal{A}) = \varnothing$.

Definition 5.4 (Infinite problem)**.** Let $\mathcal{A}$ be a DFA. The emptiness problem asks to decide whether $|L(\mathcal{A})|$ is infinite.

The answers to both these problems have been characterized by the *empty-infinite* theorem.

Theorem 15. *Let $\mathcal{A}$ be a DFA with $n = |Q|$ states. Then the language $L(\mathcal{A})$ is:*

- *non-empty if and only if it accepts a string shorter than n symbols*
- *infinite if and only if the automaton accept a string of length l, with $n \leq l < 2n$*

We focus on the emptiness problem to, again, draw a line between automata theory and graphs. The problem is clearly decidable since we can adopt Algorithm 8 to check the condition provided by Theorem 15.

Algorithm 8 CheckEmptiness

```
function CheckEmptyness(A = (Q, Σ, δ, q0, F))
    for all l ∈ {0, 1, 2, ··· n − 1} do
        W_l = Σ^l
        for all w ∈ W_l do
            if A accepts w then:
                return False
            end if
        end for
    end for
    return True
end function
```

The complexity of such algorithm is at least the complexity of generating all the possible input strings of length at most n. If we focus on one particular value for l, the number of strings of length l—the size of Σ^l—is $|\Sigma|^l$. Hence, summing over all values in $\{0, 1, \cdots, n-1\}$ we obtain:

$$\sum_{l=0}^{n-1} |\Sigma^l| = \sum_{l=0}^{n-1} |\Sigma|^l = \frac{\Sigma^n - 1}{\Sigma - 1}$$

that in the case of a binary alphabet is $2^n - 1$. Therefore, the complexity of the algorithm is bounded by $\Omega(2^n)$.

Exactly as we did up to now in this section, we ask ourselves if embedding the problem in a graph background may lead to some improvements. We already *reduced* the acceptance of a word to a path-related problem. Exploiting this idea, we can reduce the whole Algorithm 8 to a reachability problem. In fact, an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ accepts a string shorter than n symbols if and only if the state q_0 can reach one of the final states. Therefore, applying BFS algorithm with source q_0 we can decide whether q_0 reaches any state in F. This idea is described in Algorithm 9.

Algorithm 9 CheckEmptiness

```
function CheckEmptyness(A = (Q, Σ, δ, q0, F), G_A)
    d ← BFS(G_A, q0)
    for all q ∈ F do
        if d[q] < n then
            return False
        end if
    end for
    return True
end function
```

The complexity in this case is limited downwardly by the application of BFS algorithm that has a $\mathcal{O}(|V| + |E|$ running time. In the deterministic automata case, it holds that:

$$|V| = |Q| = n$$
$$|E| = \Sigma * |Q| = \text{assuming an alphabet of constant size} = \mathcal{O}(n)$$

Hence, the complexity of line 2 of Algorithm 9 is $\mathcal{O}(n)$, with n being the number of states of $\mathcal{A}$. Once BFS algorithm is completed, the only operation left is to check whether at least one of the final states is reachable in less than n steps. Such operation can be accomplished in $\mathcal{O}(n)$ time, since $|F| \leq n$. Therefore, the overall complexity of Algorithm 9 is $\mathcal{O}(n)$, which is clearly smaller than the one obtained with the *purely automaton-based* approach.

The same kind of speed up can be obtained for many classes of problems with an example being the ones concerning automata minimization.

Conclusion on Classical Automata Theory

In this chapter we introduced the theoretical minimum on automata theory required for a full comprehension of Chapter 6. We derived the notion of automata directly from the composition of a graph with *special* states and an input tape. We gave the definition of language L and in parallel provided the conditions—Pumping Lemma—required for L to be accepted from an automaton. To conclude, we moved to a very brief study of some automata-related problems. Despite the importance of the problems themselves, we wanted to stress the capability of graphs to ease the path towards their solution.

6 Quantum Automata

Quantum automata are usually introduced as the quantum counterpart of classical automata. DFAs and NFAs are recognized as ancestors of many other classes of models such as push-down automata, probabilistic automata, etc. On the other hand, in the quantum world there is no unique and surely correct answer to which model to start with when talking about *quantum finite automata* (QFA).

The very first introduction of MO-QFA can be dated back to [128]. They introduced the idea of General Quantum Automata and characterized the properties of Quantum Regular Languages. Moore and Crutchfield [128] introduced the idea of General Quantum Automata and characterized the properties of Quantum Regular Languages. The model they introduced was named *Measure-Once Quantum Finite Automata* (MO-QFAs) because the result can be observed (measured) only when the read of the input string has terminated. In the same years, Kondacs and Watrous in [112] introduced a different model of QFAs in which measurements can be used at each step of the computation. For this reason, these are called *Measure-Many Quantum Finite Automata* (MM-QFAs).

Building upon these two fundamental definitions, numerous Quantum Automata models have been proposed, including *Latvian* [7], *Quantum Automata with Control Language* [28], and *Quantum Automata with Ancilla* [142]. Despite this diversity of formalisms, this chapter will primarily focus on Measure-Once Quantum Finite Automata (MO-QFAs). We begin by providing their formal definition to establish a rigorous and comprehensive framework.

Subsequently, we turn our attention to a Quantum Automaton model that preserves certain properties of MO-QFAs while introducing a crucial distinction. Specifically, temporal dependence is no longer encoded within the state but is instead associated with the unitary operators, thereby simulating the *Heisenberg Picture* in Quantum Mechanics.

This model will be thoroughly analysed and extended by incorporating the notion of *memory*. In this revised framework, the evolution of the automaton depends on an entire input string rather than just a single symbol. We will examine cases involving both finite and infinite memory capacities, deriving expressiveness results in each scenario. These findings will be compared with those established for MO-QFAs [8, 42, 27]. The most

striking and counterintuitive result of this analysis is that MO-QFAs do not recognize all languages accepted by deterministic finite automata (DFAs), meaning they fail to accept the full class of regular languages.

The reader may refer to [29] for further readings about all the different models that have been introduced in the literature when dealing with QFAs.

Despite the variety of Quantum Automata models introduced in the literature, we will narrow our focus to the simplest one: Measure Once.

6.1 Measure Once Quantum Automata

Measure Once QFAs [128] and Measure Many QFAs [112] are the two QFAs most investigated models. Their main difference lies on the count and the timing of the measurements. While the former only measures the state at the end of the computation, the latter applies a measurement after every symbol of the input is read. One of the most peculiar consequences of this fact is that MO-QFAs read the whole input before the state collapses to a classical one. On the other hand, the latter may stop its computation before the input is terminated—which is a behaviour that is almost never encountered in classical computation.

Let $\mathbb{C}^d$ be a finite dimension Hilbert space and $Q = \{|0\rangle, |1\rangle, \dots |d-1\rangle\}$ be its canonical basis. Usually in quantum computation it holds that $d = 2^k$ for some $k \in \mathbb{N}$, where k is the number of involved qubits. However, we refer here to a generic dimension d. It is not difficult to embed all the definitions and results we present into a space of dimension $2^{k'} > d$, thus using k' qubits, whenever it is necessary in the implementations.

Definition 6.1 (MO-QFA)**.** A MO-QFA is a 5-tuple $M = (Q, \Sigma, \mathcal{U}, |\psi\rangle, F)$ where:

- Q–the set of states– is the finite canonical basis of $\mathbb{C}^d$ for some $d \in \mathbb{N}$;
- Σ is a finite alphabet;
- $\mathcal{U} = \{U_\sigma\}_{\sigma \in \Sigma}$ is a finite set of unitaries of dimension $\mathbb{C}^{d \times d}$;
- $|\psi\rangle \in \mathbb{C}^d$ is a unitary vector representing the initial superposition of M;
- $F \subseteq Q$ is the set of final states.

In the literature the standard semantics attributed to MO-QFA is based on the *Schrödinger picture* of quantum mechanics in which states evolve in time. We will come back to this in Section 6.2, when we will compare this interpretation with other possible ones. However, in the remaining of this section we will use the letter S of Schrödinger to refer to a generic QFA.

A generic configuration for a MO-QFA S is a unitary vector of $\mathbb{C}^d$, i.e., it is a vector of the form:

$$|\varphi\rangle = \sum_{|q\rangle \in Q} \alpha_q |q\rangle$$

Let $|\varphi\rangle$ be the current configuration of S and $\sigma \in \Sigma$ be the current input symbol. $|\varphi\rangle$ evolves as follows:

$$|\varphi'\rangle = U_\sigma |\varphi\rangle$$

The computation starts from $|\psi\rangle$ and evolves reading the symbols of the string x. At the end of the computation, i.e., when all the symbols of x have been read, a measurement is performed on the obtained state of S using the matrix $P_F = \sum_{|q\rangle \in F} |q\rangle\langle q|$. The probability of S accepting x is:

$$p_S(\mathbf{x}) = \|P_F U_{\mathbf{x}} |\psi\rangle \|^2 = \langle\psi| U_{\mathbf{x}}^{\dagger} P_F^{\dagger} P_F U_{\mathbf{x}} |\psi\rangle = \sum_{|q\rangle \in F} | \langle q| U_{\mathbf{x}} |\psi\rangle |^2$$

where $U_{\mathbf{x}}$,the evolution matrix accumulated along the read of $\mathbf{x}$, is defined as:

$$U_{\mathbf{x}} = U_{x_n} U_{x_{n-1}} \cdots U_{x_1}$$

We consider two different acceptance conditions. The first one is called with *cut-point*, and it recalls the acceptance condition of probabilistic automata [149].

Definition 6.2 (Cut-point MO-QFA)**.** A language $L \subseteq \Sigma^*$ is *accepted by a MO-QFA S with cut-point* λ if and only if $L = \{\mathbf{x} \in \Sigma^* \mid p_S(\mathbf{x}) > \lambda\}$.

A language $L \subseteq \Sigma^*$ is said to be *accepted by a MO-QFA with cut-point* if and only if there exist a MO-QFA S and $\lambda \geq 0$ such that $L \subseteq \Sigma^*$ is accepted by S with *cut-point* λ.

The second one is called with *certainty.* In this case we mimic the acceptance of a deterministic automata (DFA).

Definition 6.3 (Certainty MO-QFA)**.** A language $L \subseteq \Sigma^*$ is said to be accepted by a MO-QFA S with *certainty* if the following holds:

$$\mathbf{x} \in L \text{ iff } p_S(\mathbf{x}) = 1 \text{ and } \mathbf{x} \notin L \text{ iff } p_S(\mathbf{x}) = 0$$

It is straightforward to see that an acceptance with certainty implies an acceptance with cut-point $1 - \epsilon$, $\forall \epsilon \in (0, 1]$. The converse is trivially false.

The class of languages accepted by QFAs with *cut-point* was introduced and characterized in [42]. Such class is called *Unrestricted Measure-Once*, UMO. One of the main contribution to the characterization of such class is the connection with the languages accepted by Probabilistic Automata:

Theorem 16 ([42])**.** *Let L be a language accepted by a MO-QFA S with cut-point λ. There exists a Probabilistic Finite Automaton that accepts L with cut-point λ', for some λ'.*

The class UMO was further investigated in [27, 128], with the introduction of the following *pumping lemma.*

Theorem 17 ([128])**.** *Let $L \subseteq \Sigma^*$ be the language accepted by a MO-QFA S with cut-point λ. $\forall \boldsymbol{x} = \mathbf{uv} \in L$ and $\forall \mathbf{y} \in \Sigma^*$, there exists $k \in \mathbb{N}^+$ such that $\mathbf{u}\mathbf{y}^k\mathbf{v} \in L$.*

A straightforward consequence of the above theorem is that finite languages cannot be accepted by QFA.

Corollary 6.1. *QFAs can accept only languages that are either empty or infinite.*

Notice that the theorem holds for any possible split of the string x into two strings $\mathbf{u}$ and $\mathbf{v}$. So, either $\mathbf{u}$ or $\mathbf{v}$ could be empty. In particular, taking $\mathbf{v}$ empty we get that languages whose elements have a fixed suffix cannot be recognized.

Corollary 6.2. *Let $\Sigma = \{a, b\}$ and $L = \{\boldsymbol{x} \mid \boldsymbol{x}$ ends with $a\}$. L cannot be accepted by any MO-QFA S with cut-point.*

Proof. Suppose such a S exists. Let $\mathbf{x} \in L$ and $\mathbf{y} = b$, by Theorem 17 it holds that $\exists k \in \mathbb{N}^+$ such that $\mathbf{x}\mathbf{y}^k \in L$. This contradicts the definition of L. □

The above corollary also gives an example of a regular language that cannot be accepted by QFAs.

Despite being unable of accepting finite languages, QFAs can accept languages that are not regular. Let $\mathbf{x} \in \Sigma^*, \sigma \in \Sigma$. We denote by $|\mathbf{x}|_\sigma$ the number of occurrences of σ in x. It was proven in [42] that there exists a MO-QFA that accepts the language $L = \{\mathbf{x} \in \{a, b\}^* \ : \ |\mathbf{x}|_a \neq |\mathbf{x}|_b\}$ with cut-point 0.

The equivalent of UMO in the case of MM-QFAs is denoted by UMM (*Unrestricted Measure-Many*) and it was introduced in [42]. It was then characterized and further investigated in the literature (see, e.g., [8]). Results on Quantum Automata descriptive complexity can be found in [32]. Recently in [70] the expressive power of Quantum Automata over the unary alphabet under different acceptance condition has been investigated. A physical realization of Quantum Automata has been presented in [120]. Undecidability results have been proved in [24]. In [30] it has been proved that languages accepted by MO-QFAs with bounded error are not definable in Linear Time Temporal Logics, while it is definable in the case of Measure-Many. A recent survey is [31].

Even more recently, Quantum Automata minimization has been studied in [106], while succinctness has been described and implemented in [119]. Physical realizations of Quantum Computing algorithms always require to consider the noise introduced by non-perfect gates. In [34] the aim is to implement QFAs on noisy devices.

We now go in detail with the description of the proposals done in [144]. The authors presented and studied two possible ways to cope with the MO-QFAs—as defined in 6.1—limitations. The former lies on a perspective switch coming from quantum mechanics. They propose to devise quantum automata in which the state remains still and the unitaries evolve through time, recalling the Heisenberg Picture of quantum mechanics. Roughly speaking, the time dependency has been lifted from the state and moved to the unitaries The latter is a further development of an already existing model called multi-letter Quantum Automata [148]. This model tried to exploit not only the last symbol to decide which unitary to apply, but also a fixed-length suffix of the portion of the input that has already been processed. Roughly speaking, instead of using only the last symbol to decide the next step, I use a fixed-length portion of the input up to the current symbol.

We start with the automaton relying on the Heisenberg Picture of Quantum mechanics, from which they take their name.

6.2 Heisenberg Quantum Finite Automata

The most widely adopted formulation of the Copenhagen interpretation of quantum mechanics is the *Schrödinger representation*. It is based on the idea that there is a *state vector* in an Hilbert space that completely describes the configuration of the system. This state vector evolves through time according to the *Schrödinger equation*. In particular, at each time instant a unitary operator is applied to the state vector. So, in the Schrödinger picture the state vector is time-dependent, while the unitaries and the observables remain unchanged.

There exists another representation known as *Heisenberg picture* in which the state vector is time-independent and always remains fixed to its value at time 0. Therefore, the time-dependency is *shifted* on the observables.

A third representation, named *Dirac picture*, also known as *Interaction picture*, "distributes" time dependencies over both states and operators.

Even though a mathematical equivalence between Schrödinger and Heisenberg representations has been proved by Von Neumann in [167], divergences were pointed by Dirac in [60].

In terms of Quantum Finite Automata all the models described in the literature so far rely on the Schrödinger picture, where the initial state evolves through time using unitaries, while the observables never change [1].

In this section we shift to the Heisenberg picture, and we formalize a new semantics for QFAs, named *Heisenberg Quantum Finite Automata* (HQFAs). The idea is that while the string $\mathbf{x}$ is read the state is unchanged, but there is an effect on the projector. At the end of the read such modified projector is applied to the initial state to obtain the final result. The way in which the observable gets modified is in a sense arbitrarily chosen. In our definition we try to keep such choice as close as possible to that of QFAs. In particular, in quantum mechanics when one shifts from the Schrödinger picture to the Heisenberg one a transformation of the states of the form $U|\varphi\rangle$ is mapped into a transformation of the observables/projectors of the form $U^\dagger PU$, where the meaning is that $U^\dagger$ has been applied to P. As a consequence HQFAs have exactly the same definition of QFAs, while the difference is in the acceptance condition, i.e., in the semantics.

Let P be the current observable of a HQFA H and $\sigma \in \Sigma$ be the current input symbol. P evolves as follows:

$$P' = U_\sigma^\dagger P U_\sigma$$

The computation starts from the observable P_F and evolves reading the symbols of $\mathbf{x}$. At the end of the read a measurement is performed using the resulting projector and the probability of accepting $\mathbf{x}$ is:

$$\rho_H(\mathbf{x}) = \| U_{\overleftarrow{\mathbf{x}}}^\dagger P_F U_{\overleftarrow{\mathbf{x}}} |\psi\rangle \|^2 = \langle\psi| U_{\overleftarrow{\mathbf{x}}}^\dagger P_F^\dagger P_F U_{\overleftarrow{\mathbf{x}}} |\psi\rangle$$

where the evolution matrix $U_{\overleftarrow{\mathbf{x}}}$ is defined as:

$$U_{\overleftarrow{\mathbf{x}}} = U_{x_1} U_{x_2} \cdots U_{x_n}$$

The acceptance condition with cut-point for an HQFA now involves ρ_H.

[1] Some work has been done for Quantum Cellular Automata, where the equivalence between Schrödinger model and Heisenberg model has been proved (e.g., [15]).

Definition 6.4 (Cut-point HQFA)**.** A language $L \subseteq \Sigma^*$ is accepted by an HQFA H with cut-point λ if and only if $L = \{\mathbf{x} \in \Sigma^* \mid \rho_H(\mathbf{x}) > \lambda\}$.

A language $L \subseteq \Sigma^*$ is said to be *accepted by an HQFA with cut-point* if and only if there exist an HQFA H and $\lambda \geq 0$ such that $L \subseteq \Sigma^*$ is accepted by H with *cut-point* λ.

Example 6.1. Let $Q = \{|0\rangle, |1\rangle\}$ be the canonical basis of $\mathbb{C}^2$. Let $\Sigma = \{a, b\}$. Consider the two unitary matrices $U_a = X$ (the negation gate) and $U_b = H$ (the Hadamard gate), i.e.:

$$U_a = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad U_b = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Let $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle$ and $F = \{|0\rangle\}$.

If we consider M as a QFA, i.e., we endow M with the Schrödinger semantics, we get that the probability for the sting ab is:

$$p_M(ab) = \| \, |0\rangle \langle 0| \, U_b U_a \, |+\rangle \, \|^2 = \| \, |0\rangle \langle 0| \, U_b \, |+\rangle \, \|^2 = \| \, |0\rangle \langle 0|0\rangle \, \|^2 = \| \, |0\rangle \, \|^2 = 1$$

This means that no matter which is λ, the string ab is accepted.

If we consider the string abb we have to apply again U_b before projecting. Hence, we obtain $p_M(abb) = \| \, |0\rangle \langle 0| \, U_b \, |0\rangle \, \|^2 = \| \, |0\rangle \langle 0|+\rangle \, \|^2 = 1/2$.

On the other hand, if we look at M as a HQFA, i.e., we apply to M the Heisenberg semantics, the probability for the string ab is:

$$\rho_M(ab) = \|U_b^\dagger U_a^\dagger \, |0\rangle \langle 0| \, U_a U_b \, |+\rangle \, \|^2 = \|U_b^\dagger \, |1\rangle \langle 1| \, U_b \, |+\rangle \, \|^2 = \| \, |-\rangle \langle -|+\rangle \, \|^2 = 0$$

where $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. This means that no matter which is λ, the string ab is not accepted. Instead, if we consider the string ba we obtain:

$$\rho_M(ba) = \|U_a^\dagger U_b^\dagger \, |0\rangle \langle 0| \, U_b U_a \, |+\rangle \, \|^2 = \|U_a^\dagger \, |+\rangle \langle +| \, U_a \, |+\rangle \, \|^2 = \| \, |+\rangle \langle +|+\rangle \, \|^2 = 1$$

As a matter of fact, in this simple example one can notice that for all $\mathbf{x} \in \Sigma^*$ the behaviour of S on $\mathbf{x}$ is equivalent to the behaviour of H on its mirror image $\overleftarrow{\mathbf{x}}$, i.e., $p_S(\mathbf{x}) = \rho_H(\overleftarrow{\mathbf{x}})$. In the following we prove this result in the general case, for any automaton.

Theorem 18. *Let M be a QFA over an alphabet Σ. For each $\boldsymbol{x} \in \Sigma^*$ it holds that*

$$p_M(\boldsymbol{x}) = \rho_M(\overleftarrow{\boldsymbol{x}})$$

Proof. Let $\mathbf{y} = \overleftarrow{\mathbf{x}}$. We have that $\overleftarrow{\mathbf{y}} = \mathbf{x}$. So, $\rho_M(\mathbf{y}) = \langle\psi| \, U_{\overleftarrow{\mathbf{y}}}^\dagger P_F^\dagger P_F U_{\overleftarrow{\mathbf{y}}} \, |\psi\rangle = \langle\psi| \, U_{\mathbf{x}}^\dagger P_F^\dagger P_F U_{\mathbf{x}} \, |\psi\rangle = p_M(\mathbf{x})$. □

Intuitively, when we shift to the Heisenberg picture, the effect of the first character of $\mathbf{x}$ is close to the observable instead of being close to the initial state. So, the word is read in the usual way from left to right by the automaton and the effects of the read are accumulated on the observable. However, when we look to such effects on the state, it is like if the word is read from right to left. In a sense it seems that the flow of time is reverted in the Heisenberg picture.

One could argue that we could have avoided the *mirror effect* by using in the Heisenberg definition the inverse unitary operators. Since the inverse of a unitary operator is its transposed conjugate, this would have meant to define the evolution of an observable P after reading a symbol σ as $P' = U_\sigma P U_\sigma^\dagger$. In the following example we show that such choice does not help in avoiding the mirroring.

Example 6.2. Let us consider again the automaton M defined in Example 6.1. The two matrices $U_a = X$ and $U_b = H$ coincide with their transposed conjugate, i.e., $U_a^\dagger = U_a$ and $U_b^\dagger = U_b$. So, the automaton M' defined using the transposed conjugate coincides with M. Hence, $p_M(x) = \rho_{M'}(\overleftarrow{\mathbf{x}})$, for any string $\mathbf{x}$.

As a consequence of Theorem 18, the languages accepted by Heisenberg semantics are exactly the mirror images of those accepted by Schrödinger one.

Corollary 6.3. *Let $L \subseteq \Sigma^*$. L is accepted by a QFA with cut-point λ if and only if $\overleftarrow{L}$ is accepted by an HQFA with cut-point λ.*

So, now the question is whether the two formalisms have the same expressive power. As a consequence of the above corollary this is equivalent to check whether QFAs are closed under mirror images. Using example 6.1 we already know that it is not true that each language recognized by a QFA is closed under mirror images. However, it can be the case that whenever a language L is recognized by a QFA S, the language $\overleftarrow{L}$ is recognized by a QFA S'.

Invoking Von Neumann's proof of equivalence of Schrödinger and Heisenberg pictures is not satisfactory by many point of views. First, Von Neumann's result has been proved in a general setting, while here we are confined in a restricted model, where there is a single initial state, while the final states are many. Moreover, only one projective measurement can be used and only at the end of the read. Second, we are not dealing with a single quantum system, but with an infinite set of systems, one for each string $\mathbf{x}$. The input $\mathbf{x}$ does not affect the initial state, but the sequence of unitary transformations. In a sense it affects the hamiltonian of the system. Third, it would be interesting to have either a constructive proof of equivalence or a counter-example in this specific setting.

The following result shows that QFAs are closed under mirror images. We provide a constructive proof. Given a QFA for a language L, we build a QFA for the language $\overleftarrow{L}$. Intuitively, the asymmetry between a single initial state and a set of final ones is solved through an opportune increase in the state space size.

Theorem 19 (Mirror Closure of QFAs)**.** *Let $L \subseteq \Sigma^*$. L is accepted by a QFA with cut-point if and only if $\overleftarrow{L}$ is accepted by a QFA with cut-point.*

Proof. Let $M = (Q, \Sigma, \{U_\sigma\}_{\sigma\in\Sigma}, |\psi\rangle, F)$ be a QFA accepting L with cut-point λ. We recall that Q is the canonical basis of $\mathbb{C}^d$, for some $d \in \mathbb{N}$. Without loss of generality, let $F = \{q_0, q_1, \ldots q_{m-1}\}$. Let $\mathbf{x} = x_1 x_2 \ldots x_n$ be an input string. By definition, the acceptance probability of M for $\mathbf{x}$ is:

$$p_M(\mathbf{x}) = \sum_{i=0}^{m-1} |\langle q_i| U_{\mathbf{x}} |\psi\rangle|^2$$

We now define $\overleftarrow{M} = (\overleftarrow{Q}, \Sigma, \{V_\sigma\}_{\sigma\in\Sigma}, \left|\overleftarrow{\psi}\right\rangle, \overleftarrow{F})$, where $\overleftarrow{Q}$ is the canonical basis of $\mathbb{C}^d$ and:

$$V_\sigma = \sum_{i=0}^{m-1} |i\rangle\langle i| \otimes U_\sigma^\dagger, \quad \left|\overleftarrow{\psi}\right\rangle = \frac{1}{\sqrt{m}} \sum_{i=0}^{m-1} |i\rangle \otimes |q_i\rangle, \quad \overleftarrow{F} = \{|i\rangle \otimes |\psi\rangle \mid i \in [0, m-1]\}$$

We have that $V_{\mathbf{x}} = V_{x_n} V_{x_{n-1}} \cdots V_{x_1}$ and $U_{\overleftarrow{\mathbf{x}}} = U_{x_1} U_{x_2} \dots U_{x_n}$. By definition of QFA we get:

$$\begin{aligned}
p_{\overleftarrow{M}}(\mathbf{x}) &= \|P_{\overleftarrow{F}} V_{\mathbf{x}} \left|\overleftarrow{\psi}\right\rangle\|^2 \\
&= \|P_{\overleftarrow{F}} \frac{1}{\sqrt{m}} \sum_{i=0}^{m-1} |i\rangle \otimes U_{x_n}^\dagger U_{x_{n-1}}^\dagger \cdots U_{x_1}^\dagger |q_i\rangle\|^2 \\
&= \|\frac{1}{\sqrt{m}} \sum_{i=0}^{m-1} |i\rangle \otimes \Big(|\psi\rangle\langle\psi| U_{x_n}^\dagger U_{x_{n-1}}^\dagger \cdots U_{x_1}^\dagger |q_i\rangle\Big)\|^2 \\
&= \|\frac{1}{\sqrt{m}} \sum_{i=0}^{m-1} \Big(\langle\psi| U_{x_n}^\dagger U_{x_{n-1}}^\dagger \cdots U_{x_1}^\dagger |q_i\rangle\Big) |i\rangle \otimes |\psi\rangle\|^2 \\
&= \frac{1}{m} \sum_{i=0}^{m-1} |\langle\psi| U_{x_n}^\dagger U_{x_{n-1}}^\dagger \cdots U_{x_1}^\dagger |q_i\rangle|^2 \\
&= \frac{1}{m} \sum_{i=0}^{m-1} |(\langle q_i| U_{\overleftarrow{\mathbf{x}}} |\psi\rangle)^*|^2 = \frac{1}{m} \sum_{i=0}^{m-1} |\langle q_i| U_{\overleftarrow{\mathbf{x}}} |\psi\rangle|^2 = \frac{1}{m} p_M(\overleftarrow{\mathbf{x}})
\end{aligned}$$

Let $\overleftarrow{\lambda} = \frac{\lambda}{m}$. We have that:

$$\overleftarrow{\mathbf{x}} \in L \text{ iff } p_M(\overleftarrow{\mathbf{x}}) > \lambda \text{ iff } p_{\overleftarrow{M}}(\mathbf{x}) > \overleftarrow{\lambda} \text{ iff } \mathbf{x} \in \overleftarrow{L}$$

□

So, we can conclude that HQFAs do not increase QFAs expressive power.

Corollary 6.4 (Equivalence between QFAs and HQFAs)**.** *L is accepted by a QFA with cut-point if and only if L is accepted by an HQFA with cut-point.*

Proof. Let L be accepted by a QFA with cut-point. By Theorem 19 $\overleftarrow{L}$ is accepted by a QFA with cut-point. As a consequence of Corollary 6.3 L is accepted by an HQFA with cut-point.

On the other hand, let L be accepted by an HQFA with cut-point. By Corollary 6.3 $\overleftarrow{L}$ is accepted by a QFA with cut-point. By Theorem 19 L is accepted by a QFA with cut-point.

□

6.2.1 Heisenberg inspired Automata: Using Memory

The Heisenberg semantics introduced in the previous section has the same expressive power of the Schrödinger one introduced in the literature. However, we can take inspiration from Heisenberg proposal and analyse what happens if each time a character is read all the unitary matrices are transformed, i.e., instead of changing at each step the observables we modify the unitaries associated to the single characters. We do such changes by exploiting the characters that have already been read.

In particular, given an automaton $M = (Q, \Sigma, \{U_\sigma\}_{\sigma\in\Sigma}, |\psi\rangle, F)$, after reading the prefix $\mathbf{x}_j$ of the string $\mathbf{x} = x_1 x_2 \dots x_n$ the unitary matrix associated to a character σ has evolved into:

$$\mathbb{W}_\sigma^{\mathbf{x}_j} = U_{\mathbf{x}_j} U_\sigma$$

where $U_\epsilon = Id$ is the identity transformation. So, if the current configuration after reading $\mathbf{x}_j$ is $|\varphi\rangle$, and we read x_j, then the state evolves as follows:

$$|\varphi'\rangle = \mathbb{W}_{x_j}^{\mathbf{x}_j} |\varphi\rangle = U_{\mathbf{x}_j} U_{x_j} |\varphi\rangle$$

The computation starts from $|\psi\rangle$ and evolves reading the symbols of the string x. The state reached at the end of the read is:

$$\mathbb{W}_{\mathbf{x}} |\psi\rangle$$

where $\mathbb{W}_{\mathbf{x}}$–the evolution matrix accumulated along the read of $\mathbf{x}$– is defined as:

$$\mathbb{W}_{\mathbf{x}} = \mathbb{W}_{x_n}^{\mathbf{x}_n} \mathbb{W}_{x_{n-1}}^{\mathbf{x}_{n-1}} \cdots \mathbb{W}_{x_2}^{\mathbf{x}_2} \mathbb{W}_{x_1}^{\mathbf{x}_1}$$

As in the case of QFAs the projector P_F is finally applied to obtain the probability of accepting a string $\mathbf{x}$, denoted by $\omega_M(\mathbf{x})$:

$$\omega_M(\mathbf{x}) = \| P_F \mathbb{W}_{\mathbf{x}} |\psi\rangle \|^2 = \langle\psi| \mathbb{W}_{\mathbf{x}}^\dagger P_F^\dagger P_F \mathbb{W}_{\mathbf{x}} |\psi\rangle$$

Example 6.3. Let us consider again the automaton of Example 6.1. If we consider the string abb the evolution matrix that is applied to the initial state is:

$$[(U_b U_a) U_b][(U_a) U_b][(Id) U_a]$$

where we use the parenthesis to emphasize the single steps. In particular, the squared parenthesis enclose the read of a single character, while the rounded ones enclose the transformations due to the read of the prefix accumulated so far. Instantiating U_a, U_b and $|\psi\rangle$ as in Example 6.1 we obtain that the state reached at the end of the read is $-|1\rangle$. So, since $F = \{|0\rangle\}$, we get:

$$\omega_M(abb) = (-\langle 1|) P_F^\dagger P_F (-|1\rangle) = 0$$

Example 6.4. Let us now consider a simpler example in which $\Sigma = \{a\}$, $U_a = X$, the initial state is $|\psi\rangle = |0\rangle$, and $F = \{|0\rangle\}$. It is immediate to see that when a string of the

form a^k is read the evolution matrix has the form:

$$X^k X^{k-1} \cdots X^2 X = X^{\frac{(k+1)k}{2}}$$

This means that a string of length k is accepted by the automaton if and only if $(k+1)k$ is a multiple of 4.

As in the case of HQFAs, for these automata, that we call UMQFAs (*Unbounded Memory Quantum Finite Automata*), the syntactic definition is the same as for QFAs, while the accepting condition is different.

Definition 6.5 (Cut-point UMQFA)**.** A language $L \subseteq \Sigma^*$ is accepted by a UMQFA M with cut-point λ if and only if $L = \{\mathbf{x} \in \Sigma^* \mid \omega_M(\mathbf{x}) > \lambda\}$.

A language $L \subseteq \Sigma^*$ is said to be *accepted by a UMQFA with cut-point* if and only if there exist a UMQFA M and $\lambda \geq 0$ such that $L \subseteq \Sigma^*$ is accepted by M with *cut-point* λ.

Notice that we arbitrarily decided to rely on a single set of matrices. One could have considered a more general definition. The only important point is that when a character is read the unitary matrix that is applied depends also on all the characters that have been read before. However, such dependency have to be defined in a finitary way, i.e., relying on a finite initial set of matrices.

So the question now becomes: is this semantics increasing the expressive power of QFAs? In order to analyse such question we first take a step back and study what happens when, instead of using all the characters that have been read so far, we only use a bounded amount of them. On the one hand, such step back makes the situation more similar to what happen in the case of classical automata, which have a finite amount of memory. On the other hand, this naturally allows to give a more general definition, where a larger set of unitaries is used.

Bounded Memory

The most natural way to instantiate the above semantics in order to take care only of a bounded quantity of characters is to fix $h \geq 0$ and to refer to $\mathbf{x}_j^h$ instead of $\mathbf{x}_j$. The sub-string $\mathbf{x}_j^h$ takes into account at most h symbols that precede x_j in the string $\mathbf{x}$. Since there exists a finite number of strings of length at most h, the matrices $W_\sigma^{\mathbf{y}}$, with $\mathbf{y}$ of length at most h, can be directly specified in the definition of the automaton. Such automata have been already defined in the literature [148, 25], using an equivalent notation, and called *Multi-letter Quantum Finite Automata (MQFA)*. However, as we will discuss a different acceptance condition was used. Let $h \in \mathbb{N}$.

Definition 6.6 (h-MQFA)**.** An h-MQFA is a 5-tuple $M = (Q, \Sigma, \mathcal{W}, |\psi\rangle, F)$ where:

- Q–the set of states– is the finite canonical basis of $\mathbb{C}^d$ for some $d \in \mathbb{N}$;
- Σ is a finite alphabet;
- $\mathcal{W} = \{W_\sigma^{\mathbf{y}}\}_{\sigma \in \Sigma, \mathbf{y} \in \Sigma^{\leq h}}$ is a finite set of unitaries of dimension $\mathbb{C}^d \times \mathbb{C}^d$;
- $|\psi\rangle \in \mathbb{C}^d$ is a unitary vector representing the initial superposition of M;

- $F \subseteq Q$ is the set of final states.

Let $\mathbf{x} = x_1 x_2 \dots x_n$ be an input string for an h-MQFA $M = (Q, \Sigma, \mathcal{W}, |\psi\rangle, F)$. The computation starts in the state $|\psi\rangle$. Let us assume that after reading the first $j-1$ symbols of $\mathbf{x}$ a state $|\varphi\rangle$ is reached. When x_j is read the states evolves according to the following law:

$$|\varphi'\rangle = W_{x_j}^{\mathbf{x}_j^h} |\varphi\rangle$$

The computation starts from $|\psi\rangle$ and evolves reading the symbols of the string x. At the end of the computation, a measurement is performed on the state of M through the projector P_F. The probability of M accepting x is:

$$\mu_M(\mathbf{x}) = \| P_F W_{\mathbf{x}} |\psi\rangle \|^2$$

where $W_{\mathbf{x}}$ is defined as:

$$W_{\mathbf{x}} = W_{x_n}^{\mathbf{x}_n^h} W_{x_{n-1}}^{\mathbf{x}_{n-1}^h} \cdots W_{x_2}^{\mathbf{x}_2^h} W_{x_1}^{\mathbf{x}_1^h}$$

The acceptance condition with cut-point for an h-MQFA is based on μ_M.

Definition 6.7 (Cut-point h-MQFA)**.** A language $L \subseteq \Sigma^*$ is accepted by an h-MQFA M with cut-point λ if and only if $L = \{\mathbf{x} \in \Sigma^* \mid \mu_M(\mathbf{x}) > \lambda\}$.

A language $L \subseteq \Sigma^*$ is said to be *accepted by an h-MQFA with cut-point* if and only if there exist an h-MQFA M and $\lambda \geq 0$ such that $L \subseteq \Sigma^*$ is accepted by M with *cut-point* λ.

Intuitively, h-MQFAs have bounded memory h in the sense that at each point of the computation the preceding h characters are used for choosing the evolution. Notice that QFAs coincide with 0-MQFAs. Moreover, each h'-MQFA can be embedded into a h-MQFA with $h > h'$ by simply defining $\mathcal{W}$ in such a way that if $\mathbf{x}$ and $\mathbf{y}$ are two strings of length at most h that coincide on the suffix of length h', then $W_\sigma^{\mathbf{x}} = W_\sigma^{\mathbf{y}}$, for each $\sigma \in \Sigma$.

Example 6.5. Let $\Sigma = \{a, b\}$ and $L = \{a, b\}^* b$, i.e., the language of strings that end with b. The pumping lemma for QFAs ensures that this language cannot be accepted by a QFA with cut-point. Consider instead the 1-MQFA $M = (Q, \Sigma, \mathcal{W}, |\psi\rangle, F)$ where $Q = \{|0\rangle, |1\rangle\}$, $\Sigma = \{a, b\}$, $|\psi\rangle = |0\rangle$, $F = \{|1\rangle\}$. The set $\mathcal{W}$ is defined as follows:

$$W_b^\epsilon = W_b^a = W_a^b = X \qquad W_b^b = Id$$

and all the other matrices are the identity. The above matrices exactly simulate the behaviour of the following deterministic automaton, interpreting $|i\rangle$ as q_i:

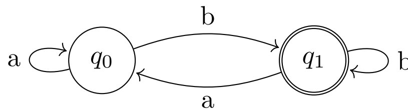


Figure 6.1: Deterministic automaton accepting $\{a, b\}^* b$.

So, M accepts L with certainty, hence also any cut-point $\lambda \geq 0$ is fine.

In [25, 148] properties of this class of automata have been studied in the case of isolated cut-point acceptance condition, also called bounded error. It was shown that the expressive power of h-MQFAs is strictly dependent on the parameter h. The set of languages recognized by h-MQFAs with bounded error coincides with those recognized by h-Group Finite Automata and are a subset of regular languages. As a consequence in [148] it has been proved that the set of languages accepted by a h'-MQFAs with bounded error is strictly included in the set of languages accepted by h-MQFAs with bounded error, for $h' < h$. This is consistent with our intuition that more memory increases the computation power. Moreover, in [148], it was proved that if the minimal DFA accepting a language L contains a particular forbidden structure, then L cannot be accepted by h-MQFAs with bounded error, for any $h \geq 0$. This is a structural characterization of languages that cannot be accepted by h-MQFAs with bounded error.

In this section we focus on cut-point acceptance condition which is less demanding than bounded error and has not been studied in the literature for h-MQFAs. We start presenting a *pumping lemma* which provides a structural characterization of the languages that are accepted from h-MQFAs with cut-point. Then we investigate on the expressive power of h-MQFAs with respect to h. Differently from QFAs, h-MQFAs can also recognize finite languages and still they constitute a proper hierarchy.

In the proof of the pumping lemma we exploit the following lemma which is also at the basis of the pumping lemma for QFAs. The norm $\|A\|$ of a matrix A is defined as:

$$\|A\| = \sup_{\langle u|u\rangle=1} \{\|A\,|u\rangle\,\|\}$$

Lemma 6.1 ([27], Quantum Recurrence Theorem). *Let $V \in \mathbb{C}^d \times \mathbb{C}^d$ be a unitary matrix let $Id \in \mathbb{C}^d \times \mathbb{C}^d$ be the identity matrix of dimension d. For any $\varepsilon > 0$ there exists $k \in N^+$ such that:*

$$\|Id - V^k\| \leq \varepsilon$$

The pumping lemma for h-MQFAs states that if we consider a sufficiently long suffix of a string which is inside the accepted language, then we can pump such suffix for an opportune number of times and fall again inside the language.

Theorem 20 (Pumping Lemma for h-MQFAs). *Let $L \subseteq \Sigma^*$ be the language accepted by an h-MQFA. Then, $\forall \boldsymbol{x} = \mathbf{uv} \in L$ with $|\mathbf{v}| \geq h$ there exists $k \in \mathbb{N}^+$ such that $\boldsymbol{x}\mathbf{v}^k \in L$.*

Proof. For the sake of readability we prove the result for $|\mathbf{v}| = h$. For $|\mathbf{v}| > h$ the idea is the same, just the notation would be much heavier.

Let $L \subseteq \Sigma^*$ be a language and let $M = (Q, \Sigma, \mathcal{W}, |\psi\rangle, F)$ be an h-MQFA that accepts L with *cut-point* λ.

Let $\mathbf{x} = \mathbf{uv} = u_1 \dots u_a v_1 \dots v_h$ be a string of L. First we write the matrix W_{xv^j}, for a generic $j \in \mathbb{N}$, in order to make explicit its relationship with $W_{\mathbf{x}}$. In particular, by applying the definition of h-QMFA we have that $W_{xv^j} = V^j W_{\mathbf{x}}$, where V is defined as:

$$V = W_{v_h}^{v_h v_1 \cdots v_{h-1}} W_{v_{h-1}}^{v_{h-1} v_h v_1 \cdots v_{h-2}} \dots W_{v_1}^{v_1 v_2 \cdots v_h}$$

Having represented a vector $|v\rangle$ in the canonical basis and being $|q\rangle$ be an element

of the canonical basis, let $(|v\rangle)_q$ be the $q-$th component of $|v\rangle$. For any $j \in \mathbb{N}$ it holds:

$$\begin{aligned} &|\mu_M(\mathbf{x}) - \mu_M(xv^j)| = \left|\sum_{q\in F}\left(|(W_{\mathbf{x}}|\psi\rangle)_q|^2 - |(W_{xv^j}|\psi\rangle)_q|^2\right)\right| \leq \\ &\leq 2\sum_{q\in F}|\,|(W_{\mathbf{x}}|\psi\rangle)_q| - |(W_{xv^j}|\psi\rangle)_q|\,| \leq 2\sum_{q\in F}|(W_{\mathbf{x}}|\psi\rangle)_q - (W_{xv^j}|\psi\rangle)_q| = \\ &= 2\sum_{q\in F}\left|\ (W_{\mathbf{x}}|\psi\rangle)_q - (V^jW_{\mathbf{x}}|\psi\rangle)_q\ \right| = 2\sum_{q\in F}\left|\ \langle q|\,(Id - V^j)W_{\mathbf{x}}|\psi\rangle\ \right| \leq \\ &\leq 2\sum_{q\in F}\|Id - V^j\| = 2|F|\|Id - V^j\| \end{aligned}$$

Since $\mathbf{x} \in L$, we have $\mu_M(x) - \lambda = \Delta > 0$. By Lemma 6.1 there exists $k \in \mathbb{N}^+$ such that:

$$\|Id - V^k\| \leq \frac{\Delta}{4|F|}$$

which yields to $\left|\mu_M(\mathbf{x}) - \mu_M(xv^k)\right| \leq \frac{\Delta}{2}$. Therefore, $xv^k \in L$, since:

$$\mu_M(xv^k) - \lambda \geq \mu_M(\mathbf{x}) - \frac{\Delta}{2} - \lambda \geq \frac{\Delta}{2} \geq 0$$

□

Notice that differently from Theorem 17, the above pumping lemma does not prevent finite languages to be accepted by h-MQFAs. As a matter of fact, if all the strings accepted by an h-MQFAs are shorter than h, then it is not possible to find a suffix that can be pumped.

Theorem 21 (Singleton/Finite Languages)**.** *Let $L = \{\mathbf{w}\}$ with $\mathbf{w} \in \Sigma^{h-1}$ and $h-1 > 0$. Then there exists an h-MQFA that accepts L with certainty.*

Let L be a finite language whose elements have length less than h. There exists an h-MQFA that accepts L with cut-point.

Proof. Let $M = (Q, \Sigma, \mathcal{W}, |\psi\rangle, F)$ be a h-MQFA, where $Q = \{|0\rangle, |1\rangle\}$, $|\psi\rangle = |0\rangle$, $F = \{|1\rangle\}$. The states $|0\rangle$ and $|1\rangle$ are such that $|1\rangle = X|0\rangle$ and $|0\rangle = X|1\rangle$. Since $\mathbf{w}$ has length $h-1 > 0$, $\mathbf{w} = \mathbf{u}\alpha$ with $\mathbf{u} \in \Sigma^{h-2}, \alpha \in \Sigma$. We define:

$$\begin{aligned} W_\alpha^{\mathbf{u}} &= X \\ W_\sigma^{\mathbf{w}} &= X \quad \forall\sigma \in \Sigma \end{aligned}$$

while all the other matrices inside $\mathcal{W}$ are the identity matrix. We must now prove that the language accepted by M is exactly L.

If $\mathbf{w}$ is the input for M, then the computation evolves as follows:

$$W_{\mathbf{w}} = W_m^{\mathbf{w}_m^h} W_{w_{m-1}}^{\mathbf{w}_{m-1}^h} \ldots W_{w_1}^{\mathbf{w}_1^h}$$

Since all the matrices we set to be different from the identity concern strings with length that is at least $h-1$, it holds that $W_{w_j}^{\mathbf{w}_j^{h+1}} = I, \forall j \in \{1, 2, \ldots m-1\}$ Therefore,

$$W_{\mathbf{w}} = W_{w_m}^{\mathbf{w}_m^{h+1}} = W_\alpha^{\mathbf{u}} = X$$

Since the initial state is $|0\rangle$, then $\|PW_{\mathbf{w}}|0\rangle\|^2 = \|P|1\rangle\|^2 = 1$.

Otherwise, suppose $\mathbf{x} \neq \mathbf{w}$ is the input for M. If the string $\mathbf{x}$ does not contain the sub-string $\mathbf{w}$, then clearly $W_{\mathbf{x}} = Id$, and $\mathbf{x}$ is refused. If $\mathbf{x}$ has $\mathbf{w}$ as proper prefix, then $\mathbf{x}$ is of the form $\mathbf{ws}$, with $\mathbf{s} = \sigma_1 \dots \sigma_j$, $j \geq 1$. In this case, we have that $W_{\mathbf{x}}$ is as follows:

$$\begin{aligned} W_{\mathbf{x}} &= W_{\sigma_j}^{\mathbf{x}^h_{h-j-1}} W_{\sigma_{j-1}}^{\mathbf{x}^h_{h-j-2}} \cdots W_{\sigma_1}^{\mathbf{x}^h_{h-1}} W_{\mathbf{w}} \\ &= W_{\sigma_j}^{\mathbf{x}^h_{h-j-1}} W_{\sigma_{j-1}}^{\mathbf{x}^h_{h-j-2}} \cdots W_{\sigma_1}^{\mathbf{w}} W_{\mathbf{w}} = Id \cdots XX = Id \end{aligned}$$

since all the matrices of the form $W_{\sigma}^{\mathbf{y}}$, with $\mathbf{y}$ of length h are the identity matrix. So, $\mathbf{x}$ is refused. The last case we need to consider is when $\mathbf{w}$ occurs as a proper sub-string of $\mathbf{x}$, but it is not a proper prefix of $\mathbf{x}$. This means that the input $\mathbf{x}$ is of the form $\mathbf{x} = \mathbf{vws}$, with $\mathbf{v} \neq \epsilon$ and $\mathbf{vw}$ which does not have $\mathbf{w}$ as prefix. In this case, the key point is that since $|\mathbf{w}| = h - 1$, but $\mathbf{w}$ is now preceded by at least one character the matrices $W_{\alpha}^{\mathbf{u}}$ and $W_{\sigma}^{\mathbf{w}}$ do not occur in $W_{\mathbf{x}}$. So, $W_{\mathbf{x}} = Id$ and $\mathbf{x}$ is refused. Notice that the automaton we defined accepts with certainty.

Let $L = \{\mathbf{x}_1, \dots, \mathbf{x}_\ell\}$ be a finite language whose elements have length less than h. For each element $\mathbf{x}_j$ there exists an h_j-MQFA that accepts only $\mathbf{x}_j$ with certainty. As already observed any h'-MQFA can be embedded into an h-MQFA that accepts the same language with the same cut-point, if $h \geq h'$. Let h be greater than the length of the longest string in L. We have that for each element $\mathbf{x}_j$ of L there exists an h-MQFA M_j that accepts only $\mathbf{x}_j$ with certainty. The tensor product M of the M_j's automata, whose construction is similar to that used in the proof of Theorem 19 accepts the language L. The tensor product automaton does not accept with certainty but with cut-point λ, with λ any number in the interval $(0, 1/\ell)$. □

We can exploit our pumping lemma to prove that the amount of memory we provided to the h-MQFA in the above theorem is the minimum.

Lemma 6.2. *Let $L = \{\mathbf{w}\}$ with $\mathbf{w} \in \Sigma^{h-1}, h - 1 > 0$. Then, there is no h'-MQFA, with $h' < h$ that accepts L with cut-point.*

Proof. Assume by contradiction that there exists an h'-MQFA that accepts L with $h' < h$. Since $|\mathbf{w}| = h - 1 \geq h'$, the string $\mathbf{w}$ can be written as $\mathbf{uv}$ with $\mathbf{v} \in \Sigma^{h'}$. By Theorem 20, it holds that there exists a $k \in \mathbb{N}^+$ such that $\mathbf{wv}^k \in L$. So, we have a contradiction. □

Exploiting Theorems 20 and 21, together with Lemma 6.2 we have that the set of languages accepted by h'-MQFAs is a proper subset of the set of languages accepted by h-MQFAs, with $h' < h$. The inclusion immediately follows from the definition of h-MQFAs and our results show that the inclusion is proper by exhibiting as witnesses all the singleton languages of strings of length h'.

Corollary 6.5. *The set of languages accepted by h'-MQFAs with cut-point is a proper subset of those accepted by h-MQFAs with cut-point, when $h' < h$.*

Proof. Let $h, h' \in \mathbb{N}^+$ with $h' < h$. From Lemma 6.2 we know that there exist languages accepted by $h-$MQFAs, but not by $h'-$MQFAs.
We must prove that all the languages accepted by $h'-$MQFAs are also accepted by

$h-$MQFAs.
Let $M = (Q, \Sigma, \mathcal{W}, |\psi\rangle, F)$ be a $h'-$MQFA accepting a language L. We can build an $h-$MQFA $M' = (Q, \Sigma, \mathcal{W}', |\psi\rangle, F)$ accepting the same language setting $\mathcal{W}' = \mathcal{W}$ (potentially completing with identity matrices). □

The hierarchy result proved in [148] concerns sets of languages which are all included in the set of regular languages, while our hierarchy includes already at level 0 non-regular languages.

Unbounded Memory

QFAs and also h-MQFAs fail to recognize many classical regular languages, since unitary transformations introduce a notion of memory which is quite different from the classical one.

On the one hand, it is easy to define a classical automaton for a finite language by using the finite set of states of the automaton to store the finite quantity of memory that is necessary. This cannot be achieved in QFAs and in h-MQFAs, when h is not large enough, as a consequence of the following property of unitary matrices that has been stated in Lemma 6.1:

$$\forall \varepsilon > 0\, \exists k \in \mathbb{N}^+ \ \ \|Id - V^k\| \leq \varepsilon$$

This is the key ingredient of the pumping lemmas for QFAs and h-MQFAs.

On the other hand, it is possible to define a QFA that accepts the non-regular language of strings having a different number of a and b characters. Classical automata do not have enough memory for this language, since it is necessary to count an unbounded number of characters.

We started this section introducing the Heisenberg inspired automata called UMQFAs hoping to increase the expressive power of QFA and h-MQFAs still relying on a finite set of unitaries and a single measurement at the end of the read. It is time to draw some conclusions about this. The automaton described in Example 6.4 pointed out that in UMQFAs we are not able to replicate the use of a unitary matrix V for any possible $k \in \mathbb{N}^+$, i.e., we cannot exploit Lemma 6.1. For instance in the example the matrix X can only occur with an exponent of the form $(k+1)k/2$, i.e., all the possible values assumed by a polynomial $p(k)$ when k ranges in $\mathbb{N}^+$. The proof of Lemma 6.1 in [27] is based on Cauchy sequences and cannot be easily generalized. However, there is another proof of the same result in [42] that ultimately relies on the following algebraic property:

for each $\alpha \in \mathbb{R} \setminus \mathbb{Q}$ the set of fractional parts of the multiples of α, i.e., $\{k\alpha - \lfloor k\alpha \rfloor \mid k \in \mathbb{N}\}$, is dense in $[0, 1]$.

This result generalizes to polynomials having irrational coefficients and to multiple dimensions (e.g., [26]). As a consequence we have a language that can be accepted by h-MQFAs, but not by UMQFAs.

Theorem 22. *Let $\Sigma = \{a\}$ and $L = \{\epsilon, a\}$. There is a 2-MQFA that accepts L with cut-point and there is not a UMQFAs that accepts L with cut-point.*

There are technical ingredients in the proof of the above result that are somehow interesting. We had to carefully choose the language L in order to obtain homogeneous polynomials. Otherwise, the eigenvalues related to rotations that are rational multiples of π would have given troubles. Moreover, the interplay between some eigenvalues could be favourable for constructing UMQFAs that *approximate* h-MQFAs, since the distribution of the wrong strings accepted by the UMQFA is not uniform.

Beside these technical considerations, the result shows that the unbounded memory we tried to introduce does not generalize the bounded one, and it does not seem easy to find a natural generalization with a finitary description.

Proof. By Theorem 21 there is a 2-MQFA that accepts $L = \{\epsilon, a\}$ with cut-point.

Let us assume by contradiction that there exists $M = (Q, \Sigma, \mathcal{U}, |\psi\rangle, F)$ UMQFA that accepts $L = \{\epsilon, a\}$ with cut-point λ. Since the string ϵ is in L it has to be:

$$\omega_M(\epsilon) = \| P_F |\psi\rangle \|^2 = \| P_F U_a |\psi\rangle \|^2 = \lambda + \Delta > \lambda$$

Any other string a^k, with $k > 1$ over the alphabet Σ would instead give:

$$\omega_M(a^k) = \| P_F \mathbb{W}_{a^k} |\psi\rangle \|^2 = \| P_F U_a^{\frac{k(k+1)}{2}} |\psi\rangle \|^2$$

Let us consider a generic unitary matrix V and study the sequence:

$$\{V^{\frac{k(k+1)}{2}}\}_{k>1}$$

As observed in [42], V can be diagonalized and $V^h = RD^hR^{-1}$, where R is unitary and D is the diagonal matrix of the eigenvalues of V. Let $e^{i\pi v_j}$ be the j-th eigenvalue of V.

If all the r_js are rational, then let $n = 4\Pi_j q_j$, where the q_js are the denominators of the r_js. We have that $D^{\frac{n(n+1)}{2}} = Id$, and hence $V^{\frac{n(n+1)}{2}} = Id$.

If m of the r_j are irrational, and ℓ of them are rational, we can safely assume that the first m are the irrational ones. Let again n be defined as above considering only the rational coefficients. If we consider the sub-sequence:

$$\{V^{\frac{nk(nk+1)}{2}}\}_{k>1}$$

we have that all the rational eigenvalues have always values 1 in the sub-sequence. On the other hand, the remaining eigenvalues take values of the form $e^{i\pi r_j \frac{nk(nk+1)}{2}}$ in the sub-sequence. Let $p : \mathbb{N} \to \mathbb{R}^m$ be defined as $p(k) = (r_1 4nk(4nk + 1), \ldots, r_m 4nk(4nk + 1))$. These are quadratic polynomials in the variable k with irrational coefficients. The fractional parts of each of these polynomials are dense and uniformly distributed over $[0, 1]$ (e.g., [26]). This means that each of these polynomials is infinitely many times arbitrarily close to a multiple of 4. This implies that each of the values $e^{i\pi r_j \frac{nk(nk+1)}{2}}$ is infinitely many times arbitrarily close to 1. As far as the whole polynomial function p is concerned it is uniformly distributed over $[0, 1]^m$ if the irrational r_js are independent. When some irrational r_js are linear combinations of the others the uniform distribution is no more ensured, but the density in $(0, 0, \ldots, 0)$ is preserved, since by making the fractional parts of the independent ones arbitrary small we can ensure that also the fractional parts of their linear combination are small enough.

Hence, for any unitary matrix V, and for each ε there exists $k > 1$ such that:

$$\|Id - V^{\frac{k(k-1)}{2}}\| \leq \varepsilon$$

As a consequence working as in the proof of Theorem 20 on the string ϵ which is in L and using $U_a^{\frac{k(k-1)}{2}}$ we obtain that there exist $k > 1$ such that a^k is accepted by M. This is a contradiction. □

Conclusions on Quantum Automata

In this chapter, we tackled the challenging topic of Quantum Automata. The primary difficulty encountered was the vast array of models and definitions related to this subject. Indeed, various characterizations of Automata exist depending on their:

- *nature*: whether fully quantum or a hybrid of quantum and classical;
- *movement directions*: the capability of the automata to read the input solely from left to right, or in multiple directions;
- *acceptance conditions*: several versions have been proposed, and the expressive power of automata classes naturally depends on these acceptance conditions;
- *amount of measurements*: inherent to quantum mechanics, measurements become a crucial aspect when designing automata.

A multitude of other points could be added to this list, but for brevity, we do not include them all. The key takeaway is the highly complex landscape surrounding Quantum Automata Theory.

In our study, we focused on the simplest and oldest definition: Measure Once Automata. As the name implies, this class permits only a single measurement at the end of the computation. Their overall behaviour can be summarized as the evolution of a quantum state according to a set of unitary matrices associated with the input alphabet.

The expressive power of this framework has been shown not to benefit from its quantum nature. In fact, these automata cannot even accept all regular languages, placing this model a step behind its classical counterpart.

In this chapter, we explored ways to address this limitation through two modifications. First, we incorporated the physical notion of the *Heisenberg picture* into the internals of quantum automata. This adjustment shifts the time dependency from the state to the unitaries. In essence, the state remains static while the unitaries evolve over time.

After establishing expressiveness results for this modified framework, we turned to a second possible enhancement. We introduced a form of finite (and non-finite) memory into the automaton, such that the applied unitary depends not only on the last input symbol but on an entire substring.

In both cases, however, we found no increase in the overall expressiveness of the models.

Despite these results not appearing particularly promising, the usefulness of our new models is twofold. On the one hand, they allow for simpler and more concise proofs of

the results established in [148]. On the other hand, to the best of our knowledge, this represents the first attempt to bridge quantum automata theory with the Heisenberg picture of quantum mechanics.

7

Answer Set Programming and Quantum Computation

In this chapter, we will continue the theme of this part by connecting Answer Set Programming (ASP) with graph theory.

Our primary goal is to demonstrate how graphs can facilitate solutions to problems related to Answer Set Programs. Specifically, we will show that finding a stable model can be reduced to locating nodes within a graph. We begin by introducing some preliminary results on Weighted Model Counting (WMC) and its quantum solution, as these results are instrumental in accelerating ASP through Quantum Computation.

To pursue this aim, we start with one of the most known results in quantum computation: Grover's algorithm for unstructured database search. Along with Shor's factorization algorithm, Grover's algorithm represents one of the clearest instances of *quantum speed-up*[1]. Subsequently, we will review how Grover's algorithm can be adapted not just to search for a solution but also to *count* the number of solutions.

This approach will then be applied to the classical problem of *Weighted Model Counting (WMC)*. WMC involves counting the interpretations that satisfy a given boolean formula, assigning a specific weight to each. In parallel, we will examine an algorithm for navigating the set of stable models of an ASP program. Such procedure is structured as a walk on a weighted graph, led by a routing function. In the chapter's final section, Quantum WMC and the algorithm for the stable models are brought together. Specifically, we will derive a quantum algorithm to efficiently compute a quantity that serves as a guiding function in the navigation process.

The objectives accomplished with this novel routine are twofold. On the one hand, it enables a speed-up in the computation of the routing function through quantum algorithms. On the other hand, it establishes a connection between logic programming and quantum computation—an area that remains far from being exhaustively explored.

[1] A measurable and provable acceleration in task completion compared to the best known classical algorithm. Grover's procedure allows for a quadratic speed-up. While Shor's algorithm allows to solve the factorization problem in Quantum Polynomial Time.

7.1 Grover's Quantum Search Algorithm

Let χ be an n-ary boolean function:

$$\chi : \{0,1\}^n \mapsto \{0,1\} \tag{7.1}$$

and let $\overline{X}$ be the set of all χ's inputs that correspond to an output of 1:

$$\overline{X} := \{x \in \{0,1\}^n : \chi(x) = 1\} \tag{7.2}$$

We call *Search Problem* the problem of finding (at least one) $\overline{x} \in \overline{X}$ for some χ given in input. We usually refer to $\overline{X}$ as the set of *solutions* to the problem. For convenience, define $N := |\{0,1\}^n| = 2^n$ as the size of the *search space* and $M := |\overline{X}|$ as the number of solutions.

Usually, the complexity of a Search Problem is expressed in terms of the number of queries to an *oracle*[2] that implements χ. Observe that without any further assumption, any classical algorithm solving a Search Problem requires at least $\mathcal{O}(N)$ calls to the oracle for χ.

Grover's Algorithm is a probabilistic quantum algorithm that allows to solve a Search Problem with $\mathcal{O}(\sqrt{N})$ expected queries to a quantum oracle for χ, i.e. it implements a procedure that exhibits a quadratic speed-up over the best classical algorithm.

We now try and give an intuition for Grover's algorithm. We follow two approaches. The former, more intuition-based, has the goal of giving the reader the intuition of how the algorithm works. The latter, more rigorous, is an explanation of why the algorithm works.

We start with the former. For simplicity, let us focus on the case where there is only one solution to the problem, i.e. $M = 1$. We indicate with $\overline{x}$ the unique solution. Moreover, assume to have an n-bit quantum register prepared in the state $|0\rangle^{\otimes n}$.

First, the algorithm initializes all the qubits to a uniform superposition of all the elements of the search space, by applying an Hadamard gate $\mathcal{H}$ to every qubit:

$$\mathcal{H}^{\otimes n} |0\rangle^{\otimes n} = |+\rangle^{\otimes n} \tag{7.3}$$

The reader may notice that the generalization of the Hadamard gate $\mathcal{H}$ to n qubits – which is the gate we indicated with $\mathcal{H}^{\otimes n}$ in Equation 7.3 – is often referred to as the *Walsh-Hadamard transform*, and represented by the letter $\mathcal{W}_n$.

After that, the aim of the procedure is to increase the amplitude of the element $\overline{x}$ in the superposition, while simultaneously decreasing all the other ones. Such task is carried out by repeatedly applying the *Grover Operator* $\mathcal{G}$. After some iterations, the amplitude of $\overline{x}$ is intuitively expected to be high enough so that the result of a measurement of all the qubits would result in $\overline{x}$ with high probability[3].

To be more specific, the operator $\mathcal{G}$ can be seen as the concatenation of two distinct steps:

1. first, $\mathcal{G}$ queries the quantum oracle S_χ. In particular, the oracle is defined so that the effect on the quantum state it is applied to correspond to a sign flip on the

[2] An *oracle* for a function f is basically a black-box operator that implements f.
[3] A probability greater than 1/2 is usually required.

amplitude of $\overline{x}$, while all the other amplitudes are left unchanged:

$$S_\chi |x\rangle = (-1)^{\chi(x)} |x\rangle \tag{7.4}$$

Since S_χ can also be interpreted as a reflection about the $|\overline{x}\rangle$ vector, it can be represented with a *Householder matrix*:

$$S_\chi = U_{\overline{x}} := I_n - 2|\overline{x}\rangle\langle\overline{x}| \tag{7.5}$$

where $I_n \in \mathbb{C}^{2^n \times 2^n}$ is the identity matrix. Clearly, S_χ is unitary, and thus it is a proper quantum operator;

2. after that, the operator $\mathcal{G}$ maps each of the amplitudes a_i to their reflection about the average of all the amplitudes of the state:

$$a_i \quad \rightsquigarrow \quad 2\left(\frac{\sum_{j=0}^{N-1} a_j}{N}\right) - a_i \tag{7.6}$$

Once again, the operation can be represented as a reflection[4], usually referred to as the *diffusion operator*:

$$-U_+ \quad = \quad -\mathcal{W}_n\, U_0\, \mathcal{W}_n \quad = \quad \mathcal{W}_n \left(2|0\rangle^{\otimes n}\langle 0|^{\otimes n} - I_n\right)\mathcal{W}_n \tag{7.7}$$

Combining all the pieces, the Grover Operator can be written as follows:

$$\mathcal{G} := -\mathcal{W}_n\, U_0\, \mathcal{W}_n\, S_\chi \tag{7.8}$$

Figure 7.1 should provide an intuition for why repeated applications of the gate $\mathcal{G}$ amplify the amplitude of $\overline{x}$.

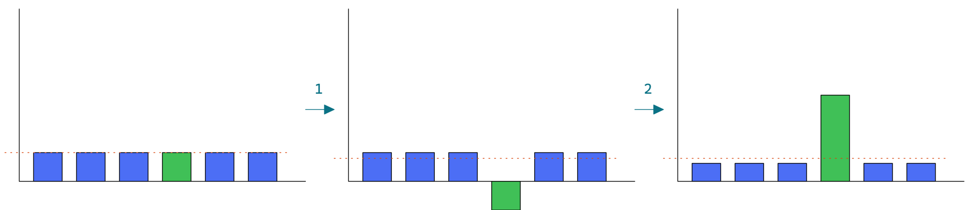


Figure 7.1: Intuition for the action performed by the Grover Operator $\mathcal{G}$. Each column represents a state of the basis, while its height is proportional to its amplitude in the superposition. The dashed red line marks the average of all the amplitudes. In particular, the figure shows the very first application of $\mathcal{G}$, when it is applied to a uniform superposition.

We can consider the *intuition* approach completed. We now move to a more rigorous and detailed study of Grover's internals.

[4]Observe that since $|+\rangle^{\otimes n} = \mathcal{W}_n |0\rangle^{\otimes n}$, the operator $-U_+$ can also be regarded as a reflection about the initial state of the system, which is defined by Equation 7.3.

As shown in [133], it turns out that the Grover Operator $\mathcal{G}$ can be regarded as a rotation in a particular basis. To observe so, consider the following two states:

$$|\alpha\rangle := \frac{1}{\sqrt{N-M}} \sum_{x \notin \overline{X}} |x\rangle \qquad |\beta\rangle := \frac{1}{\sqrt{M}} \sum_{x \in \overline{X}} |x\rangle \tag{7.9}$$

Notice that $|\alpha\rangle$ and $|\beta\rangle$ are *orthonormal*, and that the initial state belongs to the subspace spanned by these two vectors:

$$|+\rangle^{\otimes n} = \sqrt{\frac{N-M}{N}} |\alpha\rangle + \sqrt{\frac{M}{N}} |\beta\rangle \tag{7.10}$$

As shown by Figure 7.2, it is easy to observe that the Grover Operator represents a rotation in the subspace spanned by $|\alpha\rangle$ and $|\beta\rangle$:

- the gate S_χ is a reflection about the α axis, since $S_\chi(|\alpha\rangle + |\beta\rangle) = |\alpha\rangle - |\beta\rangle$;
- the diffusion operator is also by definition a reflection, about the vector $|+\rangle^{\otimes n}$.

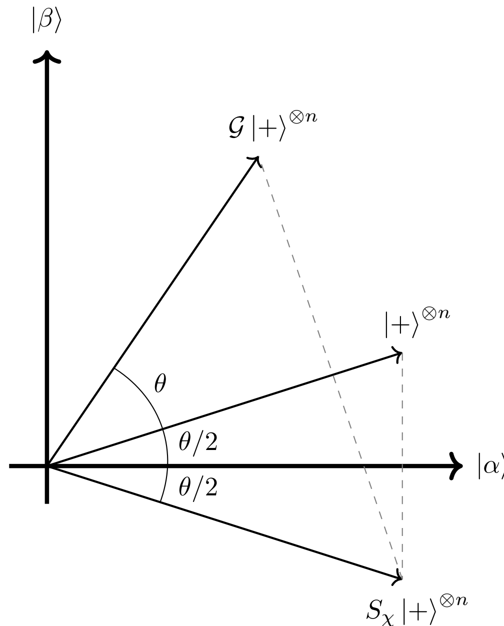


Figure 7.2: Geometric visualization of the effect of the Grover Operator.

Overall, as a consequence of Equation 7.10, the effect of the Grover operator $\mathcal{G}$ is a rotation in the plane spanned by $|\alpha\rangle$ and $|\beta\rangle$ of an angle θ such that:

$$\cos^2\frac{\theta}{2} = \frac{N-M}{N} \qquad \text{and} \qquad \sin^2\frac{\theta}{2} = \frac{M}{N} \tag{7.11}$$

The optimal number of iterations that need to be performed can be computed by elaborating on the observations made up to this point. In order to maximize the probability of measuring some $\overline{x} \in \overline{X}$, the objective is to rotate the initial vector $\mathcal{W}|0\rangle^{\otimes n}$ as close as possible to $|\beta\rangle$ using only θ-rotations. Since the angle that needs to be covered is $\pi - \theta/2 = \arccos\sqrt{M/N}$, the optimal number of iterations is:

$$R := \mathrm{CI}\left(\frac{\arccos\sqrt{M/N}}{\theta}\right) = \mathrm{CI}\left(\frac{\arccos\sqrt{M/N}}{2\arcsin\sqrt{M/N}}\right) \tag{7.12}$$

where $\mathrm{CI}(x)$ denotes the integer that is the closest to the number x. Usually, the following upper bound provides a good estimate for the optimal number of iterations:

$$R \leq \left\lceil \frac{\pi}{4}\sqrt{\frac{N}{M}} \right\rceil \in \mathcal{O}\left(\sqrt{N/M}\right) \tag{7.13}$$

Equations 7.12 and 7.13 make sense[5] only if $\theta \leq \pi/2$, i.e. if $M \leq N/2$. However, if $M > N/2$, the search space can be enlarged by simply adding a qubit to the system, so that $N' := 2N$ and $M \leq N'/2$. Refer to [133] for more details.

As a closing remark, consider the *Quantum Recurrence Theorem* shown in [37] and in [144], that we report below for the sake of completeness.

Theorem 23 (Quantum Recurrence Theorem)**.** *Let U be a unitary matrix. Then, for any $\varepsilon > 0$, there is $k \in \mathbb{N}$ such that the following condition holds:*

$$\|U^k - I\| \leq \varepsilon \tag{7.14}$$

Since the operator $\mathcal{G}$ is, after all, a unitary matrix, Theorem 23 implies that we must be careful with the number of Grover iterations: if too many are applied, the system may go back to the initial state. This is also partially the reason why we need an upper bound such as the one shown in Equation 7.13.

7.2 Counting Elements in the Search Space

Exactly as in the Quantum Search problem setting (see Section 7.1), let χ be an n-ary boolean function and let $\overline{X}$ be the set of solutions to the equation $\chi(x) = 1$. The *Quantum Counting* problem is concerned with counting how many solutions there are, i.e. computing $M := |\overline{X}|$. Considering what was discussed at the end of Section 7.1 regarding the enlargement of the state space[6], let S_χ be a quantum oracle that implements χ using $n+1$ qubits.

Now let us consider the Grover Operator $\mathcal{G}$ that uses S_χ as an oracle: since it represents a rotation of an angle θ in the space spanned by $|\alpha\rangle$ and $|\beta\rangle$, it must have two eigenvectors – $|a\rangle$ and $|b\rangle$ – that lie in the same subspace. Therefore, as a consequence of Equation 7.10, the vector $|+\rangle^{\otimes(n+1)}$ can be expressed as a linear combination of $|a\rangle$

[5] If $M > N/2$, the optimal number of Grover iterations would be $R = 0$.

[6] In this specific case, we must be as generic as possible, since the value of M is unknown and estimating it is actually the goal of the algorithm.

and $|b\rangle$:

$$|+\rangle^{\otimes(n+1)} = c_a |a\rangle + c_b |b\rangle \tag{7.15}$$

where c_a and c_b are some appropriate coefficients. Observe what follows:

- due to Equation 7.15, $|+\rangle^{\otimes(n+1)}$ can be regarded as a superposition of the states $|a\rangle$ and $|b\rangle$;

- in turn, this implies that if such a state is fed into the Phase Estimation algorithm, the circuit behaves linearly and produces thus in output digits that are either from the eigenvalue associated to $|a\rangle$ or from the one associated to $|b\rangle$, with probabilities related to c_a and c_b respectively;

- since $|a\rangle$ and $|b\rangle$ are the eigenvectors of a rotation, their associated eigenvalues can be respectively written as $e^{\imath\theta}$ and $e^{\imath(2\pi-\theta)}$. This means that estimating any of the two allows to easily retrieve the value of θ.

The value of M can be computed by reconsidering either one of the identities in Equation 7.11, while also accounting for the search space enlargement. For instance, using the second one allows us to conclude what follows:

$$M := 2N \sin^2 \frac{\theta}{2} \tag{7.16}$$

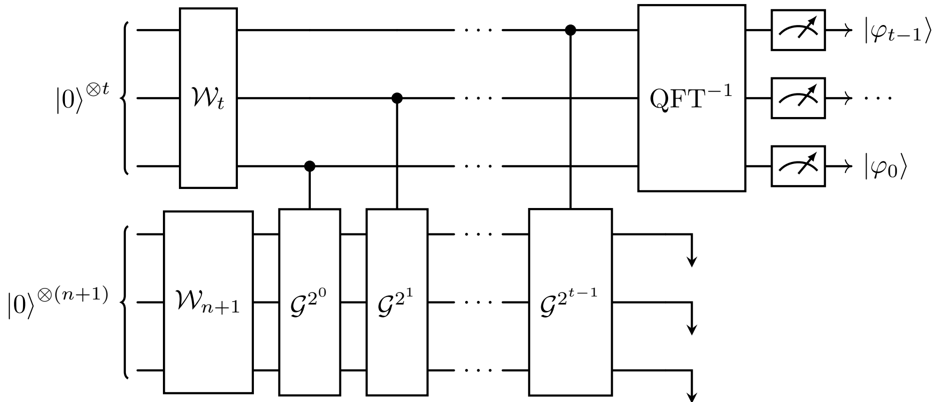


Figure 7.3: Quantum circuit implementing the Quantum Counting algorithm.

Nielsen and Chuang show in [133, Section 6.3] how accurate the estimate for M is. As it turns out, the circuit shown in Figure 7.3 produces, with probability $1 - \varepsilon$, an estimate of M with the following precision:

$$|\Delta M| < \left(\sqrt{2MN} + \frac{N}{2^{m+1}}\right) 2^{-m} \tag{7.17}$$

7.2.1 Weighted Model Counting

Let ϕ be a propositional logic formula over a set L of propositional letters, and let $n := |L|$. Observe that any $S \subseteq L$ is an interpretation for ϕ. For each $S \subseteq L$, let $en(S) := s_1 s_2 \dots s_n$ be the n-symbol bitstring such that $s_i = 1$ if and only if $l_i \in S$, where l_i is the i-th element of L (according to some order relation $\prec$ over L). We refer to $en(S)$ as the *encoding* of S. With respect to such encoding, let $\chi : \{0,1\}^n \mapsto \{0,1\}$ be the following function:

$$\chi(en(S)) := \begin{cases} 1 & \text{if } \phi(S) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{7.18}$$

Moreover, let $w : \{1, \dots, n\} \times \{0,1\} \mapsto \mathbb{R}^{\geq 0}$ be a *weight function* for χ. Roughly speaking, $w(i, b)$ represents the weight of setting the i-th input of χ to the boolean value b.

For each bitstring $x := en(S)$, let W_x be its *cumulative weight*:

$$W_x := \prod_{i=1}^{n} w(i, x_i) \tag{7.19}$$

where x_i indicates the value of the i-th bit of x.

If we consider the Search Problem associated to χ and the definition in Equation 7.19, the weight function w can be intuitively regarded as a way to formally express some kind of *preference* on the elements of the search space, i.e. over the bitstrings that encode interpretations for the formula ϕ.

We now define *Weighted Model Counting* as the problem of evaluating the following quantity:

$$\text{WMC}(\chi, w) := \sum_{x \,:\, \chi(x)=1} W_x \tag{7.20}$$

In other words, Weighted Model Counting is the problem concerned with counting the sum of the weights W_x associated to all the models x of some propositional formula ϕ, i.e. the variable assignments that make ϕ evaluate to 1.

As suggested by its name, notice that WMC can be regarded, in some sense, as a weighted variation of the problem of counting all the solutions of a Search Problem.

7.2.2 A Quantum Algorithm for WMC

In [152, Section 9], Riguzzi and Mykhailova proposed a quantum algorithm to solve WMC built upon a variation of the Quantum Counting circuit that we presented in Section 7.2. In particular, the core idea that they presented consists in replacing the Hadamard gates that initialize a uniform superposition with a gate *Rot* that uses the weight function w to define the amplitudes associated to each state in the superposition. By adapting the Grover Operator accordingly, it can be shown that the Quantum Counting circuit that uses such modified Grover Operator solves WMC, as we will discuss in the following paragraphs.

When the Weights are Normalized

Let S_χ be the quantum phase-flip oracle that implements the boolean function χ, as it was defined by Equation 7.18. In addition, for the time being, assume that the weights are normalized[7]:

$$\forall\, i = 1, \ldots, n \;\; w(i,0) + w(i,1) = 1 \tag{7.21}$$

Thanks to the normalization constraint, we can use w_i as a shorthand to indicate $w(i,1)$, and $1 - w_i$ to denote $w(i,0)$.

Since each weight w_i encodes some kind of "preference" for the i-th bit being set to 1, we would like the gate *Rot* to relate each w_i to the probability of measuring a 1 on the corresponding qubit. In other words, we want *Rot* to initialize each qubit to the following state:

$$|\psi_i\rangle := \sqrt{1 - w_i}\,|0\rangle + \sqrt{w_i}\,|1\rangle \tag{7.22}$$

where $i = 1, \ldots, n$ is the qubit index. Observe that this can be carried out by performing a rotation R_y of an angle θ_i defined as follows:

$$\theta_i := 2 \arccos \sqrt{1 - w_i} = 2 \arcsin \sqrt{w_i} \tag{7.23}$$

To check our claim, consider what follows:

$$R_y(\theta_i)\,|0\rangle = \begin{pmatrix} \cos\frac{\theta_i}{2} & -\sin\frac{\theta_i}{2} \\ \sin\frac{\theta_i}{2} & \cos\frac{\theta_i}{2} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \cos\frac{\theta_i}{2} \\ \sin\frac{\theta_i}{2} \end{pmatrix} = \begin{pmatrix} \sqrt{1 - w_i} \\ \sqrt{w_i} \end{pmatrix} = |\psi_i\rangle \tag{7.24}$$

Observe that if $w_i = 1/2$, then $R_y(\theta_i) = \mathcal{H}$ (modulo a global phase factor that we can safely ignore), which means that *Rot* would act exactly as the original Quantum Counting algorithm on the i-th qubit. Intuitively, this means that *Rot* somehow generalizes the Walsh-Hadamard transform used in Grover's algorithm.

Altogether, *Rot* can be defined as follows:

$$Rot := \bigotimes_{i=1}^{n} R_y(\theta_i) \tag{7.25}$$

As in the original Quantum Counting algorithm, we obviously do not know whether the number M of solutions to the Search Problem associated to χ is larger or smaller than half of the size $N := 2^n$ of the search space. In order to account for any possibility, we need to apply the usual trick of doubling the size of the search space by adding 1 qubit and altering the oracle S_χ to work on such new space. Therefore, a more correct definition for the gate *Rot* is the following:

$$Rot := \left(\bigotimes_{i=1}^{n} R_y(\theta_i) \right) \otimes \mathcal{H} \tag{7.26}$$

Let $\mathcal{G}_w$ be the *Weighted Grover Operator*, defined as the variant of $\mathcal{G}$ that uses *Rot*

[7] We will later show how to lift this assumption, allowing to use any positive weight function.

in place of the Walsh-Hadamard transform:

$$\mathcal{G}_w := -Rot\, U_0\, Rot^\dagger\, S_\chi \tag{7.27}$$

Now consider the quantum circuit shown in Figure 7.4, and observe that the first application of the gate *Rot* initializes the second register to the following state:

$$|\psi\rangle := Rot\,|0\rangle^{\otimes(n+1)} = \left(\bigotimes_{i=1}^{n} |\psi_i\rangle\right) \otimes |+\rangle = \sum_{x=0}^{2^{n+1}-1} \sqrt{\frac{w(1,x_1)\cdots w(n,x_n)}{2}}\,|x\rangle \tag{7.28}$$

Additionally, observe that the factor $w(1,x_1)\cdots w(n,x_n)$ that appears in Equation 7.28 corresponds to the definition of W_x (see Equation 7.19).

As a generalization of the calculations that we presented in Section 7.2 with respect to the Quantum Counting circuit, Riguzzi and Mykhailova showed in [152, Section 9] that it is indeed possible to write two orthonormal vectors $|\gamma\rangle$ and $|\delta\rangle$ such that:

- $|\psi\rangle := Rot\,|0\rangle^{\otimes(n+1)}$ can be written as a linear combination of $|\gamma\rangle$ and $|\delta\rangle$;
- $\mathcal{G}_w$ is a rotation of an angle θ in the space spanned by the same two vectors.

In particular, it can be proved that:

$$|\psi\rangle = \cos\frac{\theta}{2}\,|\gamma\rangle + \sin\frac{\theta}{2}\,|\delta\rangle \tag{7.29}$$

with:

$$\cos^2\frac{\theta}{2} := \frac{1+\sum_{x\,:\,\chi(x)=0} W_x}{2} \qquad \text{and} \qquad \sin^2\frac{\theta}{2} := \frac{\sum_{x\,:\,\chi(x)=1} W_x}{2} \tag{7.30}$$

Following the same ideas applied for Quantum Counting, the Phase Estimation circuit can be used to compute an estimate for θ. Then, exploiting Equation 7.30, we obtain:

$$\mathrm{WMC}(\chi, w) := \sum_{x\,:\,\chi(x)=1} W_x = 2\sin^2\frac{\theta}{2} \tag{7.31}$$

Lifting the Normalization Assumption

We now introduce the reader to a procedure that allows to lift the normalization assumption.

Let w be a generic, positive weight function, and let V_i be the following quantity:

$$V_i := w(i,0) + w(i,1) \qquad \forall\, i = 1,\ldots,n \tag{7.32}$$

Observe that the function $\tilde{w}(i,b) := w(i,b)/V_i$ is normalized, thus we can use the

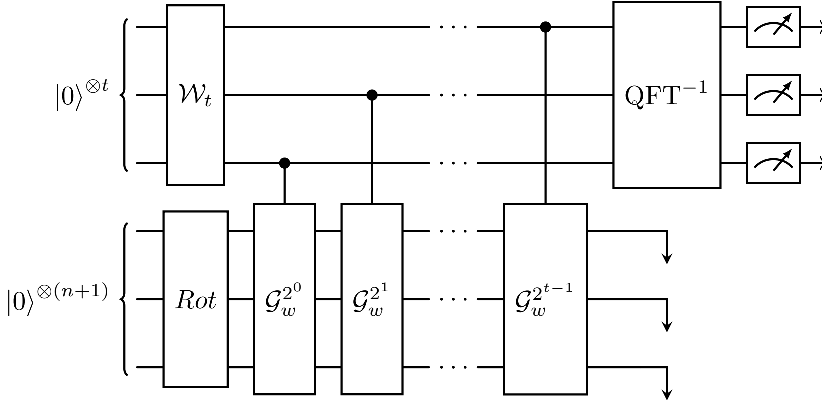


Figure 7.4: Circuit implementing the Quantum Weighted Model Counting algorithm.

circuit in Figure 7.4 in order to compute $\mathrm{WMC}(\chi, \tilde{w})$. Hence:

$$\begin{aligned}
\mathrm{WMC}(\chi, \tilde{w}) &= \sum_{x\,:\,\chi(x)=1} \left(\prod_{i=1}^{n} \tilde{w}(i, x_i) \right) = \sum_{x\,:\,\chi(x)=1} \left(\prod_{i=1}^{n} \frac{w(i, x_i)}{V_i} \right) \\
&= \sum_{x\,:\,\chi(x)=1} \left(\frac{1}{\prod_{i=1}^{n} V_i} \prod_{i=1}^{n} w(i, x_i) \right) = \frac{1}{\prod_{i=1}^{n} V_i} \sum_{x\,:\,\chi(x)=1} \left(\prod_{i=1}^{n} w(i, x_i) \right) \\
&= \frac{1}{\prod_{i=1}^{n} V_i} \mathrm{WMC}(\chi, w)
\end{aligned} \tag{7.33}$$

Therefore, $\mathrm{WMC}(\chi, w)$ can be computed as follows:

$$\mathrm{WMC}(\chi, w) = \mathrm{WMC}(\chi, \tilde{w}) \cdot \prod_{i=1}^{n} V_i \tag{7.34}$$

7.3 Faceted Navigation among Answer Sets

Fichte, Gaggl, and Rusovac presented in [67] a novel approach to the navigation of the solution space of an ASP program.

Given an ASP program Π such that $\mathcal{AS}(\Pi)$ denotes the set of all its stable models, we call:

- $\mathcal{BC}(\Pi) \coloneqq \bigcup \mathcal{AS}(\Pi)$ the set of Π's *brave consequences*;
- $\mathcal{CC}(\Pi) \coloneqq \bigcap \mathcal{AS}(\Pi)$ the set of its *cautious consequences*.

Roughly speaking, the brave consequences of Π are all the atoms that appear in some of its stable models, while the cautious consequences are those that occur in all of them.

The set of *facets* of Π is then defined as follows:

$$\mathcal{F}(\Pi) \coloneqq \mathcal{F}^{+}(\Pi) \cup \mathcal{F}^{-}(\Pi) \tag{7.35}$$

where:

$$\mathcal{F}^{+}(\Pi) \coloneqq \mathcal{BC}(\Pi) \setminus \mathcal{CC}(\Pi) \qquad \text{and} \qquad \mathcal{F}^{-}(\Pi) \coloneqq \left\{\overline{\alpha} : \alpha \in \mathcal{F}^{+}(\Pi)\right\} \tag{7.36}$$

An interpretation $S \subseteq B_{\Pi}$ for Π is said to satisfy an *inclusive facet* $\alpha \in \mathcal{F}^{+}(\Pi)$ if $\alpha \in S$. Symmetrically, S is said to satisfy an *exclusive facet* $\overline{\alpha} \in \mathcal{F}^{-}(\Pi)$ if $\alpha \notin S$. We denote by $ic(f)$ the singleton ASP program that contains the integrity constraint[8] corresponding to the facet $f \in \mathcal{F}(\Pi)$:

$$ic(f) \coloneqq \begin{cases} \{\leftarrow \neg\alpha\} & \text{if } f \text{ is an atom } \alpha \\ \{\leftarrow \alpha\} & \text{otherwise} \end{cases} \tag{7.37}$$

Let Π be an ASP program. The result of the activation of a facet $f \in \mathcal{F}(\Pi)$ is the program Π' defined as:

$$\Pi' \coloneqq \Pi \cup ic(f) \tag{7.38}$$

We say that the activation of f denotes a *navigation step* from Π to Π'. Observe that recursively applying different navigation steps to a program Π produces a graph-like structure, which we may wish to navigate. In order to do so, a finite sequence $\delta \coloneqq \langle f_1, \ldots, f_k \rangle$ of facets defines a *route*, and denotes an ordered sequence of navigation steps from an initial program Π. Notice that faceted navigation is possible as long as $\mathcal{F}(\Pi) \neq \varnothing$. As a consequence, we are usually interested in identifying a set of *safe* routes, which we denote by:

$$\Delta_s^{\Pi} \coloneqq \left\{\delta \in \Delta^{\Pi} : \mathcal{AS}(\Pi^{\delta}) \neq \varnothing\right\} \tag{7.39}$$

where Δ^{Π} is the set of all possible routes on Π, and $\Pi^{\delta} \coloneqq \Pi \cup ic(f_1) \cup \cdots \cup ic(f_k)$, with $\delta \coloneqq \langle f_1, \ldots, f_k \rangle$. In [67], the authors define two *navigation modes*, i.e. functions that prune the solution space according to some strategy that involves routes and facets. Intuitively:

- *goal-oriented* navigation aims to narrow down the solution space until a unique solution is found;

- *free* navigation does not follow any particular strategy: it allows following unsafe routes, which can be *redirected* if $\mathcal{F}(\Pi)$ becomes empty.

In addition, the authors also propose a *weighted* navigation variant: since the effect of activating a facet is basically unknown beforehand, they suggest associating a weight to each facet in order to characterize the extent to which activating said facet affects the size of the solution space. Intuitively, weight functions can be used to guide the faceted navigation process in a meaningful manner.

---

[8] An *integrity constraint* – also called *denial* in Chapter 1.3 – is a rule with an empty head.

7.3.1 Towards a Quantum Faceted Navigation

Among the different weight functions that Fichte, Gaggl, and Rusovac considered and evaluated in [67], we take as example *Absolute Weight*, defined as follows[9]:

$$w_{\#\mathcal{AS}}(f, \Pi^{\delta}) := |\mathcal{AS}(\Pi^{\delta})| - |\mathcal{AS}(\Pi^{\langle \delta, f \rangle})| \tag{7.40}$$

The authors showed that $w_{\#\mathcal{AS}}$ performs well when adopted as a function to guide navigation[10]. However, computing absolute weights is #coNP-*complete* [67]. We now show how Quantum WMC could be used to compute absolute weights with a quadratic speed-up over any classical method.

Let Π be an ASP program such that $n := |B_{\Pi}|$, and let $\delta := \langle f_1, \ldots, f_k \rangle \in \Delta^{\Pi}$ be a route for Π. Our goal is to estimate the following quantity:

$$M^{\delta} := |\mathcal{AS}(\Pi^{\delta})| \tag{7.41}$$

Being able to solve such problem would result in a simple technique to compute the weight $w_{\#\mathcal{AS}}(f, \Pi^{\delta})$ through Equation 7.40.

Authors tackled this particular problem by mean of quantum computation in [154].

Let $\chi : \{0,1\}^n \mapsto \{0,1\}$ be a boolean function that outputs a 1 if and only if its input is a bitstring that encodes a stable model of Π, and let S_{χ} be a quantum phase-flip oracle for χ. Now consider the weight function w defined as follows:

$$w(i,1) := \begin{cases} 1 & \text{if } \exists\, j = 1, \ldots, k \;\; f_j = \alpha_i \\ 0 & \text{if } \exists\, j = 1, \ldots, k \;\; f_j = \overline{\alpha_i} \\ 1/2 & \text{otherwise} \end{cases} \qquad \forall\, i = 1, \ldots, n \tag{7.42}$$

and set each $w(i,0)$ so that w is normalized. Let $S \subseteq B_{\Pi}$ be an interpretation for Π and $en(S)$ be its binary encoding. The following observations can be done:

- if $S \notin \mathcal{AS}(\Pi^{\delta})$, then either one of the following two cases applies:
  - δ contains an inclusive facet $f_j = \alpha_i$ such that $\alpha_i \notin S$. This means that $w(i,1) = 1$ and $w(i,0) = 0$. But since $\alpha_i \notin S$ by hypothesis, then the i-th bit of $en(S)$ is 0 and thus Equation 7.19 tells us that $W_{en(S)} = 0$;
  - otherwise, δ contains an exclusive facet $f_j = \overline{\alpha_i}$ such that $\alpha_i \in S$. In this case $w(i,1) = 0$, and since the i-th bit of $en(S)$ is 1 by hypothesis then again $W_{en(S)} = 0$.
- if, instead, $S \in \mathcal{AS}(\Pi^{\delta})$, then $W_{en(S)} = 1^k \cdot (1/2)^{n-k}$, where k is the number of facets of the route δ. Notice that k does not depend on S.

Therefore, applying the quantum circuit for Weighted Model Counting produces the

[9] The actual definition of $w_{\#\mathcal{AS}}$ that the authors show in their article is a bit more complex, as it needs to account for "fallback" routes in the case $\langle \delta, f \rangle$ is not a safe route. For simplicity, we ignore here the added complexity that derives from this fact.

[10] Fichte, Gaggl, and Rusovac proved in [67] that $w_{\#\mathcal{AS}}$ actually satisfies a list of criteria and properties that they define in order to compare several weight functions.

following result:

$$\mathrm{WMC}(\chi, w) := \sum_{S \in \mathcal{AS}(\Pi)} W_{en(S)} = |\mathcal{AS}(\Pi^\delta)| \cdot \left(1^k \cdot (1/2)^{n-k}\right) \tag{7.43}$$

Thus, M^δ can be easily retrieved:

$$M^\delta := |\mathcal{AS}(\Pi^\delta)| = \frac{\mathrm{WMC}(\chi, w)}{(1^k \cdot (1/2)^{n-k})} = 2^{n-k} \cdot \mathrm{WMC}(\chi, w) \tag{7.44}$$

Example 7.1. In order to make the definition of w in Equation 7.42 a bit clearer, reconsider the program Π from Example 1.1. In addition, let $\delta := \langle p \rangle$, and observe that $\mathcal{AS}(\Pi^\delta) = \{\{p, r\}\}$. According to the definition and assuming $p \prec q \prec r$ – i.e. p corresponds to the first bit, q to the second one, and r to the third one – the weight function w is defined as follows:

$$\begin{matrix} w(1,0) := 0 & w(2,0) := 1/2 & w(3,0) := 1/2 \\ w(1,1) := 1 & w(2,1) := 1/2 & w(3,1) := 1/2 \end{matrix} \tag{7.45}$$

The introduction of a quantum routine—based on Grover's algorithm—to compute M^δ allows us to obtain a quadratic speed-up over any classical algorithm for the same problem.

Clearly, the Grover's dependency we introduced, forces us to be able to provide an oracle that clearly depends on the ASP model. It remains an open problem whether such oracle can be computed efficiently starting from the ASP encoding of a given task.

Conclusions of Answer Set Programming and Quantum Computation

This chapter focused on exploring the effectiveness of a potential overlap between graphs, quantum computation, and Answer Set Programming.

We began by introducing a foundational quantum algorithm: Grover's algorithm, which enables an unstructured search over a set of N elements with a time complexity $\mathcal{O}(\sqrt{N})$.

This result is instrumental to the novel contributions of the chapter. The steps leading to these contributions are as follows. First, we investigated the concept of Weighted Model Counting (WMC) and then presented a quantum solution to this problem.

The next step was to transform the search for stable models of an ASP program into a graph traversal problem. To accomplish this *reduction*—in the sense of translating one problem into another—we employed the concept of *facets*. We then introduced a *faceted navigation* algorithm, which uses facets to construct a graph that must be traversed using a guiding or routing function w.

This routing function w underlies the innovations presented. The optimal choice of w offers two significant advantages:

- It enhances the efficiency of the overall procedure;

- It is computationally intensive to determine.

To address this, we designed a QWMC-based algorithm to compute w, achieving a quadratic speed-up over any classical approach.

As discussed prior to this conclusion, incorporating Grover's algorithm into a quantum procedure necessitates constructing a corresponding Grover oracle. In our context, this oracle is directly influenced by the ASP model under investigation. Whether this oracle can be built efficiently remains an open question.

Conclusions about Graphs as Semantics

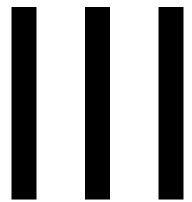
We summarize the results obtained throughout the second part of this thesis, focusing on the key concept of graphs as a semantic framework.

We began by introducing foundational results in automata theory, which are essential for understanding Quantum Finite Automata (QFA). This exploration of QFA commenced with the complex task of familiarizing the reader with the various types of quantum automata, often referred to as the "Quantum Automata zoo". Examples include Measure Once, Measure Many, Latvian, With Control, 1-Way, 1.5-Way, and 2-Way models. The introduction of this diverse array of models stems from a significant issue: Measure Once QFA (the simplest model) lacks the expressiveness of their classical counterpart, Deterministic Finite Automata (DFA). Consequently, researchers have pursued numerous avenues to address this limitation.

We contributed to this effort by presenting two distinct models of quantum automata, both derived from Measure Once QFA. The first model sought to combine MO-QFA with the concept of the *Heisenberg Picture* from quantum mechanics. The second model took a more classical approach by incorporating memory. We aimed to enhance the expressiveness of MO-QFA by introducing a finite amount of memory, which allows for a different semantic interpretation of the individual steps executed by the automaton.

Additionally, we integrated graphs into the search for stable models in Answer Set Programming (ASP). By translating the classical problem of Weighted Model Counting into the realm of quantum computing, we developed an algorithm that bridges the domains of graphs, ASP, and quantum mechanics.

The outcome is a quantum algorithm that achieves a quadratic speed-up compared to any classical procedure addressing the same task. The problem we tackled with this quantum approach is crucial for efficiently searching for stable models in ASP.



Graphs as Reduction Structures

Introduction to Graphs as Reduction Structures

In this concluding part, we address a fundamental issue often analysed through graph theory: reduction. Let us revisit the concept of Markov Chains, defined as a pair (S, M), where:

- S is the set of states,
- M is either the transition matrix or the infinitesimal generator, depending on whether we are considering discrete or continuous time.

When working with these structures, a key quantity of interest is the steady state distribution π, as defined in 1.4.4. Computing π typically involves solving a system of linear equations derived from its definition. However, this computation faces significant challenges. On the one hand, it may be computationally intractable. On the other hand, solutions may be obtained but only by means of numerical methods.

Thus, it is reasonable to seek solutions to these problems. Among various approaches, *state reduction* has proven particularly effective. Since the number of components in π equals the cardinality of S, reducing the size of S can lead to exact methods for computing π. Given that the overall Markov chain can be viewed as a graph, we must consider appropriate notions of *similarity* between nodes to facilitate this reduction. However, not all definitions of similarity are suitable; we need formalizations that (i) reduce the state space from S to some S', while (ii) maintaining a clear and computable relationship between the chain defined on S and that on S'.

In the first chapter of this part, we will specifically address this topic by examining various notions of node similarity in the study of Markov Chains.

Following this, we will shift our focus to Neural Networks (NNs). Much of the existing literature emphasizes reducing the size of the data provided to NNs—addressing the curse of dimensionality—while often overlooking the model size, or the number of neurons used for the task, especially in environments with no resource constraints. However, in scenarios like the Internet of Things (IoT), resources are limited, making the reduction of the number of neurons critical for the success of Deep Learning projects.

Numerous heuristic methods have been proposed to tackle this challenge, utilizing greedy techniques to eliminate nodes without significant regard for model accuracy. We will demonstrate that exact methods from graph theory can also be applied. In particular, by extending the concept of lumpability to Neural Networks, we will derive an exact algorithm that (i) reduces the number of neurons while (ii) preserving accuracy.

To conclude, we will return to the quantum domain to address an important issue: circuit synthesis. The physical constraints of Noisy Intermediate Scale Quantum (NISQ) architectures require quantum algorithms to be decomposed into smaller gates—either one or two-qubit gates—for execution on actual quantum devices. The process of computing a decomposition of a given unitary into smaller components is known as *synthesis*. This problem can be approached in two distinct ways:

- As a monolithic challenge, where the final algorithm takes a unitary as input and produces another unitary based on specific constraints,
- As a collection of smaller problems, where synthesis algorithms also aim to minimize certain classes of gates. Consequently, it is natural to regard these minimization/reduction problems as foundational components of a comprehensive synthesis algorithm.

We will pursue the latter approach, focusing on the problem of CNOT minimization using Answer Set Programming. Graphs naturally emerge as a framework for modelling and solving this problem.

8

Bisimulations and Lumpabilities

As it should be clear by now, graphs are elegant and ductile structures. First, they are very interesting from a pure computer science point of view—a lot of research is carried out to design efficient algorithms for graphs. Secondly, they are really flexible and easy to use when adopted as a *way to encode problems*. As we saw in the previous chapters and sections, a lot of problems reformulated in graph terms can be solved more efficiently. In Figure 8.1 we depict the general idea of this approach—this is the very procedure we explained to solve 2SAT.

We now focus on the phase in which a problem instance $\mathcal{P}$ is translated into some graph $G_{\mathcal{P}}$. In the example we provided with the 2SAT encoding at the beginning of Part II, every variable and/or negation of a variable of the boolean formula is turned into a node. Therefore, let ϕ be a 2CNF boolean formula with n variables and m clauses. The *size* of the resulting graph $G_{\mathcal{P}} = (V, E)$ depends on two factors:

- the number of distinct variables in the formula. Hence, $|V| \in \mathcal{O}(n)$.
- the number of clauses defines the number of edge in $G_{\mathcal{P}}$. In particular, $|E| = 2m$.

Hence, the space complexity analysis of the graph approach reveals no reduction in the size of the problem—a formula with $\theta(n)$ variables and $\theta(m)$ clauses has been turned into an n nodes m edges graph. We clearly obtain an advantage in tackling the problem—that becomes a reachability instance—but we have no luck in shrinking the numbers in play.

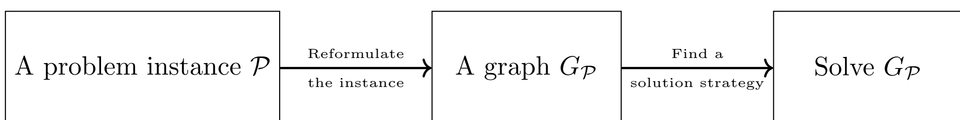


Figure 8.1: Generic procedure to start from a problem P, encoding it as a graph-related problem and then solving it.

Nevertheless, graphs do offer techniques and approaches to reduce their sizes and obtain some kind of reduction. These techniques are exact and return graphs that are equivalent to the initial ones.

Let us take a step back from our theoretical framework. We now introduce a *common sense*-based example, to better grasp and understand what may be a possible reduction strategy for graphs.

Example 8.1. Exactly as Euler did when he *encoded* the Königsberg bridges, we translate a road configuration with a graph. The scenario under consideration is illustrated

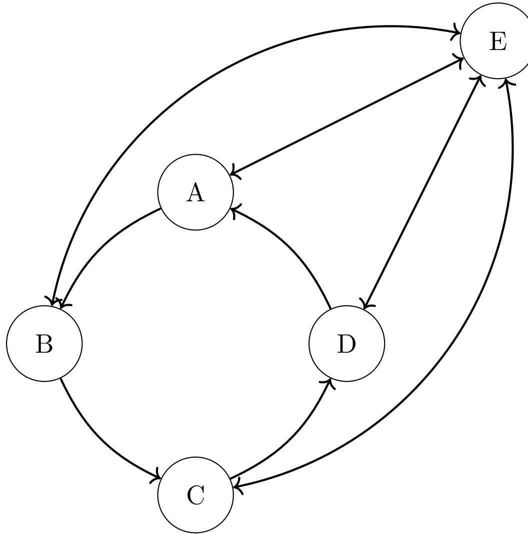


Figure 8.2: A completely misplaced roundabout configuration

in Figure 8.2. Imagine an engineer has constructed this unusual roundabout, where each of the four points follows these rules:

- From any point within the roundabout, you can access every other point in it.
- Movement within the roundabout is restricted to a counterclockwise direction.
- To exit the roundabout, one can only proceed to point E.
- If entering the roundabout from an *external* location, the entry point must be E.

It seems reasonable to conclude that this roundabout design is, in fact, *inefficient*. Setting aside the *common sense* interpretation of A, B, C, and D as (for instance) four distinct neighbourhoods, we can eliminate the roundabout, resulting in the configuration shown in Figure 8.3. This simplification is valid because, for each of the four points A, B, C, and D, they share the same reachable points and are accessible from the same set of points.

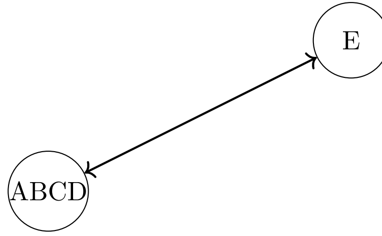


Figure 8.3: The configuration with no redundant roundabout.

In this example, we have provided a glimpse into potential strategies for reducing the size of a graph. Toward the end of the example, you may have noticed our use of the term *reach*. This concept of *removing* nodes that exhibit similar behaviour—based on reachability properties—can actually be formalized within graph theory as well.[1]

8.1 Bisimulation

Formally speaking, to define a graph reduction technique we must define a relation $\mathcal{R}$ between nodes. Two nodes can be merged whenever they are in relation according to $\mathcal{R}$. In Example 8.1, the $\mathcal{R}$ we adopted is *Bisimulation*.

Definition 8.1 (Bisimulation)**.** Let $G_1 = (V_1, E_1)$ and let $G_2 = (V_2, E_2)$ be two graphs. A relation $\mathcal{R} \subseteq V_1 \times V_2$ is a bisimulation between G_1 and G_2 if $\forall (v_1, v_2) \in \mathcal{R}$ the following holds:

- if $(v_1, u_1) \in E_1$, then there exists $u_2 \in V_2$ such that $(v_2, u_2) \in E_2$ and $(u_1, u_2) \in \mathcal{R}$
- if $(v_2, u_2) \in E_2$, then there exists $u_1 \in V_1$ such that $(v_1, u_1) \in E_1$ and $(u_1, u_2) \in \mathcal{R}$

In the case that $G_1 = G_2$ then we are computing a graph reduction as intended in Example 8.1. This relation can be roughly explained as a behavioural equivalence between nodes. Two nodes are bisimilar—behave in the same way—if and only if they reach only bisimilar states.

Bisimulation presents some particular properties. First, if $\mathcal{R}$ is the identity—every node u is in relation only with itself—it is a legal bisimulation. Moreover, let $\mathcal{R}_1$ and $\mathcal{R}_2$ be two bisimulation over the same graph $G = (V, E)$. Then the bisimulation $\mathcal{R}_{\bigcup}$ defined as:

$$\mathcal{R}_{\bigcup} = \mathcal{R}_1 \bigcup \mathcal{R}_2$$

is still a bisimulation. Using the above equation, we can obtain an interesting result. Let $R = \{\mathcal{R} : \mathcal{R} \text{ is a bisimulation over } G\}$ be the set of all the possible bisimulations over G. Then the relation $\sim$ defined as :

$$\sim = \bigcup_{\mathcal{R} \in R} \mathcal{R}$$

[1] It is worth noting that, according to the definition we will now introduce, nodes ABCD and E should also be merged, but for illustrative purposes, we stopped just before that step.

is a bisimulation. Moreover, (i) it is maximal and (ii) it is an equivalence.

Given a bisimulation $\mathcal{R}$, we can compute the reduced graph according to $\mathcal{R}$:

$$G/\mathcal{R} = (V/\mathcal{R}, E/\mathcal{R})$$

in which both nodes and edges are modified according to $\mathcal{R}$. Specifically, nodes are *partitioned* according to the classes induced by $\mathcal{R}$. Edges, on the other hand, now connect classes instead of elements.

Example 8.2. We use this example to better explain the notion of bisimulation.

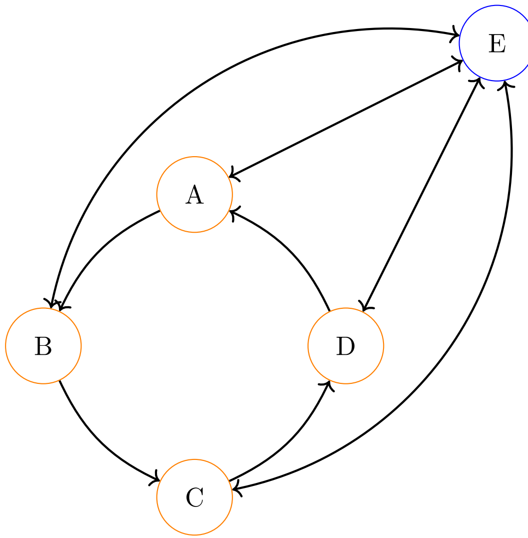


Figure 8.4: A possible bisimulation $\mathcal{R}$

In Figure 8.4 we depicted the same graph adopted in Example 8.1 but with the nodes coloured. The color coding we adopted is based on the bisimulation we took into account. Explicitly speaking, nodes A, B, C, and D are all in relation with each other. On the other hand, E is only in relation with itself.

Even if this is not the maximum bisimulation $\sim$, this relation is transitive. Therefore, we can think of it as a partition of the states of the graph. Hence, the graph in Figure 8.3 is the reduced graph.

What we just introduced is the simplest possible notion of bisimulation on directed graphs.

For example, how can Definition 8.1 be affected by the introduction of labels on both nodes and edges? Let $G_l = (V, E, l_V, l_E)$ be a *labelled* graph where:

- V and E are the set of nodes and edges, respectively,
- $l_V : V \to \mathrm{LABEL}_V$ is the nodes labelling function. It takes in input a node, and it returns a label among the set LABEL_V,

- analogously, $l_E : E \to \text{LABEL}_E$ is the edges labelling function. It takes in input an edge, and it returns a label among the set LABEL_E.

This being the case, an edge $e \in E$ is a tuple (u, v, σ) where:

- u, v are the edge's source and destination node, respectively
- $\sigma = l_E(e = (u, v))$ is the label associated to e.

This kind of graphs are adopted for example when dealing with Labelled Transition Systems.

The nodes are the states of the system, the edges are the connection between the states. The labels on the node may be the set of predicates satisfied by the state, while the labels on the edges the action performed to go from a state to another.

The notion of bisimulation must be tweaked as follows in order to deal with the two labelling functions:

Definition 8.2 (Bisimulation on labelled graphs)**.** Let $G_l = (V, E, l_V, l_E)$ be a labelled graph. A relation $\mathcal{R} \subseteq V \times V$ is a bisimulation for G if $\forall (v, u) \in \mathcal{R}$ the following holds:

- *Label equivalence:* $l_V(u) = l_V(v)$
- if $(v, v', \sigma) \in E$, then there exists $u' \in V$ such that $(u, u', \sigma) \in E$ and $(v', u') \in \mathcal{R}$
- if $(u, u', \sigma) \in E$, then there exists $v' \in V$ such that $(v, v', \sigma) \in E$ and $(v', u') \in \mathcal{R}$

Roughly speaking, the modification we introduced are:

1. We had to deal with the node labelling. Two nodes, in order to be bisimilar, must at least agree on their label,
2. We had to take into account the label on the edges. The notion of *reaching nodes with the same behaviour* remains unchanged, but we must achieve this goal with the same label on the edge we traverse.

8.2 Bisimulation and Automata Minimization

One of the most well-known contexts in which graphs bisimilarity comes into play is automata minimization.

The minimization of an automaton is the process of constructing a new (language-equivalent) automaton which is minimal in the number of states.

Hence, the problem of automata minimization requires finding states that *behave* the same and compress them in a single one. How is behavioural equivalence defined when we take into account the language acceptance notion, typical of automata theory? Two states are said to be *the same state* if they accept the same language.

Definition 8.3. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $q \in Q$ be a state. The *language accepted by* q, denoted with $L(q)$ defined as:

$$L(q) = \{w \in \Sigma^* \, : \, \delta^*(q, w) \in F\}$$

Remark 8.1. Using this definition, let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA. The language accepted by $\mathcal{A}$ is:

$$L(\mathcal{A}) = L(q_0) = L$$

We refer to two equivalent state p, q with $p \sim q$. Formalizing the notion of equivalence we obtain:

Definition 8.4. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $p, q \in Q$ be two states.

$$p \sim q \leftrightarrow L(p) = L(q)$$

We say that p and q are *equivalent* (resp. *distinguishable*) whenever $p \sim q$ (resp. $p \not\sim q$). In the special case of $p \in F$ and $q \notin F$ (or viceversa) we say that pair (p, q) is *trivially distinguishable.*

This condition has also been restated in a recursive fashion as a corollary of Myhill-Nerode theorem—which exists only for deterministic automata:

Definition 8.5 (Myhill-Nerode equivalence)**.** Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ be a deterministic automaton and let p, q be two states.

$$\begin{aligned} p \sim q \leftrightarrow & L(p) = L(q) \\ & \forall w \in \Sigma^* \big(\delta(p, w) \in F \leftrightarrow \delta(q, w) \in F\big) \\ & p \in F \leftrightarrow q \in F \ \wedge \ \forall w \in \Sigma^+ \big(\delta(p, w) \in F \leftrightarrow \delta(q, w) \in F\big) \\ & p \in F \leftrightarrow q \in F \ \wedge \ \forall x \in \Sigma \ \ \big(\delta(p, x) \sim \delta(q, x)\big) \end{aligned}$$

In the deterministic case, computing the finest relation induced by Myhill-Nerode equivalence leads to the computation of the minimal automaton—up to isomorphism—accepting a language. We now ask ourselves two questions:

- How do we efficiently compute the Myhill-Nerode equivalence?
- What about the non-deterministic case?

The answer to both these questions is closely related to the topics of this section. It is, in fact, bisimulation. Since automata introduce the notion of *final states* that is not usually present when dealing with graph, we must first tweak the notion of bisimulation to take it into account.

Definition 8.6. Given a NFA, a *bisimulation* is a binary relation B over its states such that, for every pair $(p, q) \in B$:

B1. $p \in F \iff q \in F$,

B2. $\forall x \in \Sigma \, \forall p' \in \delta(p, x) \, \exists q' \in \delta(q, x) \ \wedge \ (p', q') \in B$,

B3. $\forall x \in \Sigma \, \forall q' \in \delta(q, x) \, \exists p' \in \delta(p, x) \ \wedge \ (p', q') \in B$.

Two states p and q are said *bisimilar* if there exists a bisimulation which contains (p, q). The set of all bisimulations over the states of $\mathcal{N}$ is denoted by $\mathfrak{B}_{\mathcal{N}}$.

If B_1 and B_2 are two bisimulations over some automaton $\mathcal{N}$, it is not difficult to prove that also $B_1 \cup B_2$ is a bisimulation over $\mathcal{N}$—essentially, the union does not invalidate the three conditions of Definition 8.6 which were previously verified by pairs of B_1 and pairs of B_2. The aforementioned properties, whose proof can be found in the literature (c.f. [97, Chapter 1], and [103, Proposition 3]), are summarized in Lemma 8.1.

Lemma 8.1. *Let $\mathcal{N}$ be a NFA. Then, the following hold:*

1. *$\mathfrak{B}_{\mathcal{N}}$ is closed under union, and admits a* unique *largest (with respect to set-union) bisimulation $\mathcal{B}_{\mathcal{N}}$,*
2. *$\mathcal{B}_{\mathcal{N}}$ is an equivalence relation that relates all and only bisimilar states, and*
3. *$\mathcal{B}_{\mathcal{N}} \subseteq \sim$, and if $\mathcal{N}$ is deterministic, then $\mathcal{B} = \sim$.*

In particular, point (3) of Lemma 8.1 justifies the use of $\mathcal{B}$ as an approximation of $\sim$ for nondeterministic automata.

The aforementioned definitions served as a formal introduction to the relation $\sim$ we want to compute when minimizing automata. This allows us to switch our focus back to the two questions we anticipated earlier.

We start from the latter. The lack of a non-deterministic counterpart of Myhill-Nerode theorem makes the computation of the minimal non-deterministic automata a PSPACE-complete problem [122]. Hence, bisimilarity seems a natural choice as a trade-off between computational cost and quality results. This choice is also supported by the answer to the former question. It has been proved that Myhill-Nerode equivalence coincides with bisimilarity between states. Therefore, being able to compute bisimilarity over deterministic and non-deterministic automata would lead to minimal automata in the former case and *acceptably minimal* ones in the latter.

Thus, the problem of minimizing automata reduces to the problem of computing bisimilarity between states, which in turn is equivalent to determining the coarsest partition of a set stable with respect to some binary relation. The two main paradigms to compute the aforementioned partition are *top down* and *bottom up*.

Top down algorithms start with the partition that separates states between final and non-final and subsequently *refine* the partition until it is stable. By a careful choice of which block of the partition to split at each refining step, Paige and Tarjan [153] devised an algorithm that computes the maximum bisimulation equivalence in time $\mathcal{O}(m \log n)$, where n is the number of states and m is the number of transitions. The iconic Hopcroft's Algorithm [90] (which Paige and Tarjan's solution is based on) deals with the special case of deterministic automata. Furthermore, it has recently been proved that the bisimilarity computation requires $\Omega(m \log n)$ time assuming top down algorithms [87].

On the contrary, bottom up solutions start with the finest partition—the one where each state constitutes a singleton—and proceed by subsequently *merging* two blocks found to be equivalent. For this reason, the technique is also known in the literature as *partition-aggregation*. The main advantage of this paradigm is that the algorithm is *incremental*, *i.e.* it proceeds in subsequent stages where at the end of each merging step the resulting automaton is language-equivalent to the input one. In this way, the minimization process can be stopped at any time and can be resumed later. The first

algorithm of this kind is due to Watson [168]. After a series of improvements Watson and Daciuk [169] reduced the running time to $\mathcal{O}(n^2|\Sigma|\alpha(n))$ for deterministic automata with n states, alphabet Σ and where $\alpha(n)$ is related to the inverse of Ackermann's function [163] which can be treated as a constant for any practical value of n[2]. The main idea is to propagate the definition of bisimilarity: if states p and q are equivalent, then also their transitions by the same character must lead to equivalent states. This is done by a recursive function EQUIV which resembles an equivalence algorithm by Hopcroft and Karp [91].

Above algorithms are focused on the minimization of deterministic automata. The nondeterministic case was tackled by Björklund and Cleophas [35] adapting ideas from Watson and Daciuk. They devised an incremental algorithm for computing bisimilarity in time $\mathcal{O}(n^2r^2|\Sigma|)$, where r is the degree of nondeterminism.

8.2.1 An algorithm for Incremental DFA Minimization

Most of the aforementioned algorithms address the problem of automata minimization by traversing the graph induced by the automaton, whether deterministic or not. In [33], the authors proposed transforming the computation of the minimal automaton into a graph colouring problem. They developed a bottom-up, incremental algorithm that augments the automaton with an *associated graph*. This graph is subsequently coloured, and the outcome of this process computes the bisimilarity relation on the initial automaton. Due to its incremental nature, this algorithm can be halted after any merge step, and the resulting automaton will still be language-equivalent to the input one—a property not shared by top-down algorithms. Furthermore, the authors achieved the following results regarding time complexity:

- In the deterministic case, their algorithm matches the best-known one.
- In the non-deterministic case, they were able to improve upon the fastest-known algorithm.

We start by investigating the deterministic case. First, we define the associated graph for this setting.

Definition 8.7. Given a DFA $\mathcal{D}$ its *associated graph* $\mathcal{G} = (V, A)$ is defined as:

$$V = Q \times Q,$$
$$A = \{\langle p, q\rangle \to \langle \delta(p, x), \delta(q, x)\rangle\} p, q \in Q, x \in \Sigma.$$

$\langle p, q\rangle$ is *distinguishable* if p and q are distinguishable and *equivalent* otherwise.

Where if $p \not\sim q$ we say that p, q are *distinguishable* and if exactly one of the two is final we say they are *trivially distinguishable*.

Colouring $\mathcal{G}$ with distinguishable vertices black and equivalent vertices white, the problem of computing $\sim$ can be seen as the problem of correctly colouring the associated graph.

The procedure we propose to obtain such result is presented in Algorithm 10.

[2]It holds $\alpha(n) \leq 5$ for $n \leq 2^{2^{16}}$.

Algorithm 10 Proposed algorithm for deterministic automata.

```
 1: function MinimizeDfa(Q, Σ, δ, F)
 2:     for all ⟨p, q⟩ ∈ Q × Q do
 3:         if p, q are triv. distinguishable then
 4:             Color(⟨p, q⟩) ← Black
 5:         else if p = q then
 6:             Color(⟨p, q⟩) ← White
 7:         else
 8:             Color(⟨p, q⟩) ← Grey
 9:         end if
10:     end for
11:     for all ⟨p, q⟩ ∈ Q × Q do
12:         if Color(⟨p, q⟩) = Grey then
13:             H ← EmptyGraph
14:             eq ← Equiv(⟨p, q⟩)
15:             if ¬eq then
16:                 H ← Reverse(H)
17:                 Visit(H, h)
18:             end if
19:             for all ⟨p′, q′⟩ ∈ WhiteV(H) do
20:                 Union(p′, q′)
21:             end for
22:         end if
23:     end for
24: end function

19: function Equiv(⟨p, q⟩)                 ▷ H and h global
20:     if ⟨p, q⟩ ∈ H then
21:         return ⊤
22:     else if Color(⟨p, q⟩) = Black then
23:         h ← ⟨p, q⟩
24:         return ⊥
25:     else if Color(⟨p, q⟩) = White then
26:         return ⊤
27:     else                     ▷ here ⟨p, q⟩ is Grey and fresh
28:         Color(⟨p, q⟩) ← White
29:
30:         for all x ∈ Σ do                ▷ in lex. order
31:             ⟨p_x, q_x⟩ ← ⟨δ(p, x), δ(q, x)⟩
32:             AddArc(H, ⟨p, q⟩ , ⟨p_x, q_x⟩)
33:             eq ← Equiv(⟨p_x, q_x⟩)
34:             if ¬eq then return ⊥
35:             end if
36:         end for
37:
38:         return ⊤
39:     end if
40: end function
```

We assume that primitive Color has access to a data-structure to get/set the color of a vertex in constant time (*e.g*, an array or a succinct bitvector). Furthermore, we assume that such primitive gets/sets symmetrically a pair, *i.e.*, when Color(p, q) is used both $\langle p, q\rangle$ and $\langle q, p\rangle$ are considered.

The overall idea of the algorithm strongly lies upon $\mathcal{H}$, which represents the visited portion of $\mathcal{G}$.

The idea is that when Equiv$(\langle p, q\rangle)$ returns to the main loop, after line 18, vertices in $\mathcal{H}$ will be correctly coloured, either in White or Black. Helper procedures Reverse and Visit perform, respectively, arc-reverse of a graph and the Black-colouring of $\mathcal{H}$ starting from the source vertex h. After the loop, we gather all White pairs in $\mathcal{W}$, and we return the quotient $Q/\mathcal{W}^e$.

Let us analyze the version of Equiv$(\langle p, q\rangle)$ in Algorithm 10. At lines 20–27 some base cases are checked. In particular, if $\langle p, q\rangle$ is Black, then it is stored in the global variable h and $\bot$ is returned. Otherwise, if $\langle p, q\rangle$ is White we return $\top$ to continue the downstream inspection. Finally, in case $\langle p, q\rangle$ is Grey, it is coloured White and at lines 30–36 the for loop tries to continue the recursive visit by reading each symbol *in lexicographic* order. Before each recursive call $\mathcal{H}$ is updated by adding arc $\langle p, q\rangle \rightarrow \langle p_x, q_x\rangle$—we assume vertex $\langle p_x, q_x\rangle$ is added, if not already present.

Below we outline the main arguments for complexity and correctness.

Complexity Analysis

First, notice that $|\mathcal{G}| = |V| + |A| = n^2 + nm$. It is clear that, summing over all the iterations of the main loop, line 11, the associated graph is visited at most thrice: during the "forward" recursion pass and, optionally, during Reverse and Visit. In fact, every vertex starts Grey and becomes White during the forward pass of an Equiv-call.

Possibly, if the call returns $\perp$, some WHITE vertices become BLACK.

Finally, the connected components of graph $G_{Q,\mathcal{W}}$ can be computed in time $\mathcal{O}(n^2)$.

In total, we have the following:

Theorem 24. *Given a complete DFA with n states and $m = n|\Sigma|$ transitions,* MINIMIZEDFA *terminates in time $\mathcal{O}(nm)$.*

Correctness

To prove correctness we will show the following invariant at line 23: all vertices in $\mathcal{H}$ are correctly coloured either BLACK or WHITE—Lemma 8.2 below.

We start by showing some properties of the colouring performed by EQUIV and VISIT. First, notice that, since we are in the deterministic case, for every $u = \langle p, q\rangle \in V$ and $w \in \Sigma^*$ there is a unique path in $\mathcal{G}$ starting from u and spelling w: denote by $\delta(u, w) = \langle \delta(p, w), \delta(q, w)\rangle$ the last vertex of this path. In the following, we recall that a path in a graph is said to be *simple* when it has no repeating vertices.

Definition 8.8. Let $\mathcal{D}$ be a DFA, and $\mathcal{G} = (V, A)$ its associated graph. Let $u \in V$, and $w \in \Sigma^*$. We say that w for u is:

1. *simple* if the path $u \rightsquigarrow \delta(u, w)$ in $\mathcal{G}$ is simple, and
2. *avalanche* if it is simple and vertex $\delta(u, w)$ is BLACK.

If there exists w avalanche for u, denote by $\mathrm{av}(u)$ the lexicographically smallest such w and name $u \rightsquigarrow \delta(u, \mathrm{av}(u))$ the *avalanche path of* u.

If $\text{EQUIV}(u_0)$ is called in the main loop, line 14, the visit checks all *simple* words for u_0 in lexicographic order, either until all words are tested or—if it exists—until $\mathrm{av}(u_0)$ is found. Any vertex that can reach the avalanche path should be coloured in BLACK.

Proposition 8.1. Consider $\mathcal{H}$ upon return of $\text{EQUIV}(u_0)$ at line 14. If there exists $u \in \mathcal{H}$ distinguishable, then:

1. $\mathrm{av}(u_0)$ exists, and
2. If all WHITE vertices in $\mathcal{G} \setminus \mathcal{H}$ are equivalent, then:

$$(\forall u \in \mathcal{H})\left(u \text{ distinguishable} \implies u \overset{\mathcal{H}}{\rightsquigarrow} \delta(u_0, \mathrm{av}(u_0))\right).$$

Proof. Suppose $u \in \mathcal{H}$ is distinguishable:

1. First, we prove that there exists $w \in \Sigma^*$ such that $\delta(u_0, w)$ is BLACK. Since $u \in \mathcal{H}$, there exists w_0 such that $u = \delta(u_0, w_0)$. Since u is distinguishable, there exists w_1 such that $\delta(u, w_1)$ is *trivially* distinguishable (*i.e.* a final/non-final pair, which is BLACK from the start). Thus, $w = w_0w_1$ and $\delta(u_0, w_0w_1)$ is BLACK. Since $\mathrm{av}(u_0)$ is the lexicographically smallest such w, we proved point (1).
2. We want to prove that, under suitable conditions, every distinguishable node $u \in \mathcal{H}$ can reach the avalanche path of u_0 remaining inside $\mathcal{H}$.

Let $h = \delta(u_0, \mathrm{av}(u_0))$, $u \in \mathcal{H}$ distinguishable, and π be the $\mathcal{G}$-path leading u to some *trivially* distinguishable v. We claim π must cross the avalanche path of u_0 (call it α). Suppose not, for the sake of contradiction. Since h is the unique BLACK vertex in $\mathcal{H}$, $v \notin \mathcal{H}$. Hence, π must traverse some arc $u' \to u''$ with $u' \in \mathcal{H}$ and $u'' \notin \mathcal{H}$. By assumption on π and construction of EQUIV it follows that $u' \notin \alpha$ and u'' is WHITE. Since π leads u'' to v it follows that u'' is distinguishable contradicting the hypothesis on WHITE vertices in $\mathcal{G} \setminus \mathcal{H}$.

□

Now we prove that the colouring performed in the main loop is correct, *i.e.*, if it colours a vertex in WHITE (resp. in BLACK) the corresponding states are equivalent (resp. distinguishable).

Lemma 8.2. *The following hold at the end of each iteration of loop 11–23:*

D1. $\{\, \langle p, q\rangle \,\}\, \mathrm{COLOR}(\langle p, q\rangle) = \mathrm{BLACK} \cap \sim \, = \varnothing$,

D2. $\{\, \langle p, q\rangle \,\}\, \mathrm{COLOR}(\langle p, q\rangle) = \mathrm{WHITE} \subseteq \sim$.

Proof. Before entering the loop both properties hold by initialization.

D1. Assume (**D1**) and (**D2**) hold before EQUIV(u_0). It is sufficient to prove that at the end of the iteration for every $\langle p, q\rangle \in \mathcal{H}$ we have that $\langle p, q\rangle$ is BLACK if and only if $\langle p, q\rangle$ is distinguishable.

($\to$) If $u = \langle p, q\rangle \in \mathcal{H}$ is BLACK, then it must have been coloured by VISIT. Therefore, $eq = \bot$ and before REVERSE there was $u \rightsquigarrow h$ in $\mathcal{H}$. Since h was BLACK before the EQUIV-call, by (**D1**) it follows h distinguishable. Thus, $\langle p, q\rangle$ is distinguishable.

($\leftarrow$) If $u = \langle p, q\rangle \in \mathcal{H}$ is distinguishable, then by Prop. 8.1 it follows that after REVERSE and VISIT pair $\langle p, q\rangle$ has been coloured in BLACK.

D2. It follows from (**D1**) and the fact that all vertices in $\mathcal{H}$ are either BLACK or WHITE.

□

We summarize the results of this section as follows:

Theorem 25. *Let $\mathcal{D} = \langle Q, \Sigma, \delta, q_0, F\rangle$ be a DFA, and $\mathcal{W}$ be the set of* WHITE *pairs after any iteration of loop 11–23 of* MINIMIZEDFA(Q, Σ, δ, F). *Then, the following hold:*

1. *Incrementality: $\mathcal{D}/\mathcal{W}^e$ is a partially minimized automaton equivalent to $\mathcal{D}$, and*
2. *Correctness: if $\mathcal{W}$ is from the last iteration, then $\mathcal{D}/\mathcal{W}^e$ is the minimum automaton accepting $L(\mathcal{D})$.*

Proof. Incrementality follows from Lemma 8.2. Correctness follows from both Lemma 8.2 and the fact that, after all iterations, every vertex of $\mathcal{G}$ has been (correctly) coloured either in BLACK or WHITE. □

8.2.2 An Incremental Algorithm for NFA Minimization

Algorithm 10 is not directly applicable to the nondeterministic case, the reason being that reaching a pair of non-bisimilar states—i.e. sufficient condition to color in BLACK a node by Algorithm 10—is not a sufficient condition now to declare a pair of states distinguishable.

To tackle this issue we first turn the associated graph into a bipartite graph. In the definition below, for each state p we introduce the *shadow* state $\overline{p}$ as a distinct copy of the *real* p.

Definition 8.9. Let $\mathcal{N}$ be a complete NFA. The *associated graph* $\mathcal{G}(\mathcal{N})$ is a bipartite directed graph with vertices $V_0 \cup V_1$ and arcs $A_0 \cup A_1$, defined as:

$$\begin{aligned}
V_0 &= Q \times Q,\\
V_1 &= \{\, \langle p, \overline{q}, x\rangle \,,\, \langle \overline{p}, q, x\rangle \,\}\, p, q \in Q,\ x \in \Sigma,\\
A_0 &= \{\, \langle p, q\rangle \to \langle p', \overline{q}, x\rangle \,,\, \langle p, q\rangle \to \langle \overline{p}, q', x\rangle \,\}\, p' \in \delta(p, x),\ q' \in \delta(q, x),\\
A_1 &= \{\, \langle p', \overline{q}, x\rangle \to \langle p', q'\rangle \,,\, \langle \overline{p}, q', x\rangle \to \langle p', q'\rangle \,\}\, p' \in \delta(p, x),\ q' \in \delta(q, x).
\end{aligned}$$

Vertex $\langle p, q\rangle$ in the "left" V_0 is called *equivalent* (resp. *distinguishable*) whenever states p and q are bisimilar (resp. non-bisimilar). Furthermore, it is called *trivially distinguishable* when exactly one of the two is final.

The bisimilarity between p and q (Definition 8.6) can be checked in two steps: 0) choose $x \in \Sigma$ and $p' \in \delta(p, x)$, and 1) respond with suitable $q' \in \delta(q, x)$. The idea is to mimic step 0) traversing arcs of A_0 and step 1) traversing arcs of A_1. The triplet $\langle p', \overline{q}, x\rangle \in V_1$ indicates that we have chosen symbol x, state $p' \in \delta(p, x)$, and we are expecting to respond with some $q' \in \delta(q, x)$ ($\overline{q}$ provides the information on the state that must respond). A_1 arcs do something similar.

Then, we have to adjust the BLACK colouring of vertices in $\mathcal{G}$ to conform to nondeterminism. Observe that $u \in V_0$ needs only one BLACK child to be coloured in BLACK, while it needs all children WHITE to be coloured in WHITE. Dually, $u \in V_1$ behaves the same but with reversed colours. A check will be performed using the variable DOUBTS(u) which, roughly speaking, counts how many BLACK neighbours we need to find to mark u as BLACK.

We present Algorithms 11 and 12 for the nondeterministic case, whose essential ingredients are those of Algorithm 10.

First, notice that we are actually dealing with *four* colours: $\perp$ (never been explored), GREY (in visit), BLACK (distinguishable) and WHITE (equivalent). The procedure MINIMIZENFA is structurally the same as MINIMIZEDFA, the difference being the usage and maintenance of $\mathcal{H}$.

Function EQUIV of Algorithm 10 is now split into two separate (and mutually recursive) functions, EQUIVLEFT and EQUIVRIGHT, which test the equivalence of pairs of states respectively in V_0 and V_1.

Function EQUIVLEFT takes as input the current vertex $u \in V_0$. If u has already been encountered we return its color. Otherwise, it is coloured in GREY with zero DOUBTS. At lines 9–18 each successor v of u is recursively visited. Since $u \in V_0$, if v is recursively found BLACK, then u can be safely marked BLACK. Otherwise, there is not enough

Algorithm 11 Proposed algorithm for nondeterministic automata, adapted from DFA case.

```
 1: function MinimizeNfa(Q, Σ, δ, F)
 2:     for all ⟨p, q⟩ ∈ Q × Q do
 3:         if p, q are triv. distinguishable then
 4:             Color(⟨p, q⟩) ← Black
 5:         else if p = q then
 6:             Color(⟨p, q⟩) ← White
 7:         else
 8:             Color(⟨p, q⟩) ← ⊥
 9:         end if
10:     end for
11:     for all u ∈ Q × Q do
12:         H ← EmptyGraph
13:         EquivLeft(u)
14:         for all v ∈ V_0 ∩ H do
15:             if Color(v) ≠ Black then
16:                         ▷ v is either Grey or White
17:                 Color(v) ← White
18:             end if
19:         end for
20:     end for
21:     W ← { u ∈ V_0 } Color(u) = White
22:     return Q/W^e
23: end function

19: procedure Relax(v)            ▷ Used in Algorithm 12
20:     for u ∈ Adj(H, v) do                 ▷ H is global
21:         Doubts(u) ← Doubts(u) − 1
22:         if Doubts(u) = 0 then
23:             Color(u) ← Black
24:             Relax(u)
25:         end if
26:     end for
27: end procedure
```

information to safely assign a Black/White color to u. In particular, if v is Grey we add arc $v \to u$ to $\mathcal{H}$ (notice that it is reversed *w.r.t.* the transition) and we set Doubts(u) to 1—Blackness of u depends on the (possible) future Blackness of one of its neighbours v.

After the loop, at lines 20–24, if u is still Grey we consider two cases: if Doubts$(u) = 0$, then each of its successors has the same color (in this case White) which can be safely assigned to u.

In case u was coloured Black (lines 26–29) this information is propagated (Relaxed) to its neighbours in $\mathcal{H}$. Notice that in this case we explicitly define the procedure Relax (pseudocode of Algorithm 11): in Algorithm 10 the corresponding procedure Visit's purpose, was to color in Black all vertices reachable from some distinguishable vertex. In Algorithm 11 we must consider the doubts of each vertex v by colouring in Black only non-doubtful vertices.

EquivRight is similar to EquivLeft, the key difference being the update of Doubts(u) (line 40): vertices from V_1 need *only one* White neighbour to prove their Whiteness, while *each* of their neighbours must be Black to prove their Blackness.

As an extra (implementation) detail, notice that EquivLeft and EquivRight can be combined into a single Equiv(u, s) function whose second parameter, $s \in \{0, 1\}$, manages the *side* of $\mathcal{G}$ we are visiting (either left or right).

Complexity Analysis

First, we bound the size of the associated graph.

Lemma 8.3. *Given a complete NFA with n states and $m \geq n|\Sigma|$ transitions, its associated graph has size $|\mathcal{G}| \leq 7nm$.*

Algorithm 12 Procedures to color the associated graph

```
 1: function EquivLeft(u)                          ▷ H is global
 2:     if Color(u) ≠ ⊥ then
 3:         return Color(u)
 4:     end if
 5:
 6:     Color(u) ← Grey
 7:     Doubts(u) ← 0
 8:
 9:     for v ∈ Adj(G, u) ∧ Color(u) ≠ Black do
10:         col ← EquivRight(v)
11:         if col = Black then
12:             Color(u) ← Black
13:             Doubts(u) ← 0
14:         else if col = Grey then
15:             AddArc(H, v, u)
16:             Doubts(u) ← 1
17:         end if
18:     end for
19:
20:     if Color(u) = Grey then
21:         if Doubts(u) = 0 then
22:             Color(u) ← White
23:         end if
24:     end if
25:
26:     if Color(u) = Black then
27:         Relax(u)
28:     end if
29:     return Color(u)
30: end function

25: function EquivRight(u)                         ▷ H is global
26:     if Color(u) ≠ ⊥ then
27:         return Color(u)
28:     end if
29:
30:     Color(u) ← Grey
31:     Doubts(u) ← 0
32:
33:     for v ∈ Adj(G, u) ∧ Color(u) ≠ White do
34:         col ← EquivLeft(v)
35:         if col = White then
36:             Color(u) ← White
37:             Doubts(u) ← 0
38:         else if col = Grey then
39:             AddArc(H, v, u)
40:             Doubts(u) ← Doubts(u) + 1
41:         end if
42:     end for
43:
44:     if Color(u) = Grey then
45:         if Doubts(u) = 0 then
46:             Color(u) ← Black
47:         end if
48:     end if
49:
50:     if Color(u) = Black then
51:         Relax(u)
52:     end if
53:     return Color(u)
54: end function
```

Proof. The associated graph has size $|\mathcal{G}| = |V_0| + |V_1| + |A_0| + |A_1|$. Clearly, $|V_0| = n^2$ and $|V_1| = 2n^2|\Sigma|$. Arcs are more delicate:

$$\begin{aligned}
|A_0| &= \sum_{p\in Q}\sum_{q\in Q}\sum_{x\in\Sigma}\Big(|\delta(p,x)| + |\delta(q,x)|\Big)\\
&= n\sum_{p\in Q}\sum_{x\in\Sigma}|\delta(p,x)| + n\sum_{q\in Q}\sum_{x\in\Sigma}|\delta(q,x)|\\
&= 2nm
\end{aligned}$$

It is not difficult to see that $|A_1| = |A_0|$. The claim follows since $n|\Sigma| \leq m$ (the automaton is complete). □

It is evident that EquivLeft and EquivRight combined perform the equivalent of a visit of $\mathcal{G}$, and Relax visits every arc of $\mathcal{H}$ at most once. Hence, their cost over all the execution of Algorithm 11 is bounded by the size of $\mathcal{G}$.

Finally, the computation of $Q/\mathcal{W}^e$ has time complexity $\mathcal{O}(n^2)$ as in the deterministic case.

Theorem 26. *Given a complete NFA with n states and $m \geq n|\Sigma|$ transitions,* MinimizeNfa *terminates in time $\mathcal{O}(nm) \subseteq \mathcal{O}(n^2|\Sigma|r)$.*

Correctness

First, we prove that from a WHITE vertex on the left we reach in two hops, again, a WHITE vertex.

Proposition 8.2. Let $p \neq q$, and $u = \langle p, q \rangle \in V_0$ be WHITE. Then, for every arc $u \to v \in A_0$, vertex $v \in V_1$ is either GREY or WHITE.

Proof. Since $p \neq q$, there are only two places at which $u = \langle p, q \rangle$ changed from GREY to WHITE:

Line 17 of Algorithm 11.
In this case, upon termination of EQUIVLEFT(u), u is GREY and DOUBTS(u) > 0 (line 20 of Algorithm 12). Inside the loop of its EQUIVLEFT-call every neighbour was recursively found to be either GREY or WHITE—or u would have been coloured in BLACK. Finally, it cannot be the case that some GREY neighbour v became BLACK afterward, since RELAX(v) would have coloured u in BLACK by setting DOUBTS(u) $= 0$.

Line 22 of Algorithm 12.
In this case, inside the loop of EQUIVLEFT(u) none of u's neighbours were found to be either BLACK or GREY. Thus, they must all be WHITE.

Therefore, every neighbour of u is either GREY or WHITE. □

Proposition 8.3. At line 20 of Algorithm 11 (end of the main loop), for every $v \in V_1$, if v is either GREY or WHITE, then there exists $v \to u' \in A_1$ such that $u' \in V_0$ is WHITE.

Proof. If v is WHITE, then it must have been coloured at line 36 of Algorithm 12 after finding a WHITE neighbour. If v is GREY, upon termination of EQUIVLEFT at line 13 (Algorithm 12) some of its neighbours must have been found to be GREY. Since every GREY left node is coloured in WHITE before the end of the iteration, it follows again that v has some WHITE neighbour. □

As in the DFA case, the colouring performed by loop 11–20 is correct.

Lemma 8.4. *The following hold at the end of each iteration of loop 11–20 of Algorithm 11:*

N1. $\mathcal{R}_W = \{ \langle p, q \rangle \} \text{COLOR}(\langle p, q \rangle) = \text{WHITE} \subseteq \mathcal{B}$,

N2. $\mathcal{R}_B = \{ \langle p, q \rangle \} \text{COLOR}(\langle p, q \rangle) = \text{BLACK} \cap \mathcal{B} = \varnothing$.

Proof.

N1. By Lemma 8.1 it is sufficient to prove that $\mathcal{R}_W$ is a bisimulation.

First, notice that pairs violating (**B1**) are coloured in BLACK from the start. Consider $\langle p, q \rangle \in \mathcal{R}_W$. If $p = q$, (**B2**) and (**B3**) trivially hold. Otherwise, let $x \in \Sigma$ and $p' \in \delta(p, x)$. From Prop. 8.2 it follows that $v = \langle p', \overline{q}, x \rangle$ is either GREY or WHITE. From Prop. 8.3 it follows that some neighbour u' of v is in $\mathcal{R}_W$. By Definition 8.9 we have $u' = \langle p', q' \rangle$ for some $q' \in \delta(q, x)$. Thus, (**B2**) holds for $\langle p, q \rangle$. The very same argument can be used to prove that (**B3**) holds for $\langle p, q \rangle$. Hence, $\mathcal{R}_W$ is a bisimulation.

N2. The result follows from (**N1**) and the fact that vertices in $V_0 \cap \mathcal{H}$ are either BLACK or WHITE.

□

At the exit of the loop, all vertices have been coloured. As for the deterministic case, we link the behaviour of MINIMIZENFA to the correctness of $\mathcal{R}_W$:

Theorem 27. *Let $\mathcal{N} = \langle Q, \Sigma, \delta, I, F \rangle$ be a NFA, $\mathcal{G} = (V_0 \cup V_1, A_0 \cup A_1)$ be its associated graph, and $\mathcal{R}_W \subseteq V_0$ be the set of* WHITE *pairs after any iteration of loop 11–20 of* MINIMIZENFA(Q, Σ, δ, F). *Then, the following hold:*

1. *Incrementality: $\mathcal{N}/\mathcal{R}_W^e$ is a partially minimized automaton equivalent to $\mathcal{N}$, and*
2. *Correctness: if $\mathcal{R}_W$ is from the last iteration, then $\mathcal{N}/\mathcal{R}_W^e$ is the bisimulation-minimum automaton accepting $L(\mathcal{N})$—equivalently, $\mathcal{R}_W = \mathcal{B}$.*

Proof. Let $\mathcal{R}_B$ (resp. $\mathcal{R}_W$) be the set of pairs from V_0 which have been coloured in BLACK (resp. WHITE). Incrementality follows from (**N1**) of Lemma 8.4 ($\mathcal{R}_W \subseteq \mathcal{B}$ is a bisimulation).

Correctness follows from both Lemma 8.4 and the fact that, upon termination, $\mathcal{R}_B, \mathcal{R}_W$ is a partition of V_0. Therefore:

$$\begin{aligned}
\mathcal{B} &= V_0 \cap \mathcal{B} \\
&= (\mathcal{R}_B \cup \mathcal{R}_W) \cap \mathcal{B} \\
&= (\mathcal{R}_B \cap \mathcal{B}) \cup (\mathcal{R}_W \cap \mathcal{B}) \\
&= \varnothing \cup (\mathcal{R}_W \cap \mathcal{B}) \\
&\subseteq \mathcal{R}_W,
\end{aligned}$$

and we conclude $\mathcal{R}_W = \mathcal{B}$. □

To conclude this section, and recapping the results obtained in [33]. We first started with a brief introduction on the well-know concept of bisimulation. We described it as a notion of equivalence between nodes that, roughly speaking, behave in the same way. Furthermore, we then lifted this definition to the labelled graph case. This very formalization led us to the description of classical algorithm for automata minimization. The peculiarity of this algorithm is that of being *incremental*: the set of automata obtained during the minimization process are all language equivalent to the input one. The procedure was presented for both the deterministic and the non-deterministic case. All these two algorithms are built upon the idea of turning the minimization problem into a graph colouring one.

The complexity analysis showed that:

- the deterministic version time complexity matches the best known procedure in the literature
- the approach for non-deterministic cases is faster than any other approach

It remains an open problem the reason why bottom-up and top-down approaches for automata minimization are split by some kind of *complexity gap*.

8.3 Lumpability

The reader may notice that the labels we took into account up to now are supposed to be some kind of string-like structures. Nevertheless, Markov Chains—which we introduced in Section 1.4—do exploit real numbers as labels for their edges (they have no labels on the nodes). In particular, these numbers are interpreted as *rates* of traversing from one end of an edge to the other end in the case of Continuous Time Markov Chains. It becomes interesting to answer the following question:

How do we define behavioural equivalence between two nodes when rates come into play?

Answering this question prompted the introduction of the concept of *lumpability*.

The notion of lumpability provides a model simplification technique that can be used for generating an aggregated Markov chain that is smaller than the original one but allows one to determine exact results for the original process. The concept of lumpability can be formalized in terms of equivalence relations over the state space of the Markov chain. Any such equivalence induces a partition on the state space of the Markov chain and aggregation is achieved by clustering equivalent states into macro-states, thus reducing the overall state space

Before formally investigating this idea, we want to stress a key idea behind graphs reductions: the *properties* of the graph must be preserved after the aggregation. This is not a naive requirement, since there may exist approximate techniques that aim at reaching the biggest decrease in node/edge count while accounting for some behavioural loss. In the case of bisimulation—Def 8.1—we wanted to maintain the reachability properties unaffected by the reduction. In particular, an external observer should observe no change when dealing with the graph before and after the computation of the bisimulation relation. With the introduction of labels—Def 8.2, it is natural to think about automata. When applying reduction techniques to automata, we want to be sure that its expressibility properties remains the same.

Hence, getting back to the lumpability case: what kind of quantity/property do we want to preserve whenever we define a reduction over Markov chains?

The quantity we are usually interested in is the probability distribution $\mathcal{P}$ over the nodes of the chains. Such $\mathcal{P}$ describes, at each moment in time, what is the probability for the chain of being in one of its states. The limit of this distribution when time goes to infinity is called *stationary distribution*. It describes how the network will behave whenever the time goes on and on up to infinity, giving us a glimpse of the *overall* behaviour of the chain.

This is exactly the quantity we want to keep in check: lumpability have been introduced in order to reduce the size of Markov chains while keeping their stationary distributions unchanged. Therefore, we want a technique that

- Aggregates—lumps—states that behave in the same way
- Updates the weights between lumped states so that the stationary distribution remains the same.

Is has been proved in [105, 20] that if the computed lumpability satisfies the so-called *ordinary* condition, then the stationary distribution of the aggregated process can be used to derive an exact solution to the original one.

Ordinary, or Strong, lumpability has been introduced in [105].

Definition 8.10 (Strong lumpability)**.** Let $\mathcal{X}(t)$ be a CTMC with state space $\mathcal{S}$ and let $\sim$ be an equivalence relation over $\mathcal{S}$. We say that $\mathcal{X}(t)$ is *strongly lumpable* with respect to $\sim$—in the same way, $\sim$ is a *strong lumping* for $\mathcal{X}(t)$—if $\sim$ induces a partition on the state space $\mathcal{S}$ such that for any equivalence classes $S_i, S_j \in \mathcal{S}/\sim$, with $i \neq j$, and for each pair $s, s' \in S_i$, the following holds:

$$\sum_{\overline{s} \in S_j} q_{s,\overline{s}} = \sum_{\overline{s} \in S_j} q_{s',\overline{s}}$$

Roughly speaking, two states are said to be strongly lumpable if their cumulative *outgoing* rates to the other classes coincide. The aggregated markov chain is defined by using the classes induced by $\sim$ as states and the rates are changed with their cumulative versions.

Definition 8.11 (Strongly Lumped Markov Chain)**.** Let $\mathcal{X}(t)$ be a CTMC with state space $\mathcal{S}$ and let $\sim$ be a strong lumpability for $\mathcal{X}(t)$. The *Lumped* CTMC $\tilde{\mathcal{X}}(t)$ is defined as the Markov process which has $S/\sim$ as set of states. For such sets it must hold that $\tilde{\mathcal{X}}(t) = S \in \mathcal{S}/\sim$ if and only if $\mathcal{X}(t) = s \in S$. The transition rates between classes are defined as follows:

$$\tilde{q}_{S,S'} = \sum_{s' \in S'} q_{s,s'}$$

where $S, S' \in \mathcal{S}/\sim$ and $s \in S$.

The strong lumpability preserves the Markov property, and it is thus possible to exactly compute the steady state distribution of the original chain starting from the steady state distribution of the lumped chain. In particular, the following relation between the stationary distributions of the original and aggregated process has been proven:

Theorem 28. *Let $\mathcal{X}(t)$ be a CTMC with state space $\mathcal{S}$, stationary distribution π, and let $\sim$ be a strong(ordinary) lumpability for $\mathcal{X}(t)$. Moreover, let $\tilde{\mathcal{X}}(t)$ be the Strongly lumped Markov Chain with state space $\mathcal{S}/\sim$. Then, the stationary distribution $\tilde{\pi}$ for $\tilde{\mathcal{X}}(t)$ is such that for any equivalence class $S \in \mathcal{S}/\sim$:*

$$\tilde{\pi}(S) = \sum_{s \in S} \pi(s).$$

Remark 8.2. Every Markov chain is strongly lumpable with respect to the identity relation and to the trivial relation with only one equivalence class.

As stated before, Strong—Ordinary—Lumpability takes into account outgoing rates to define equivalence between to states. It is natural to think what happens when we try the opposite and define two states as lumpable if their *incoming* rates coincide for every other equivalence class. This very idea has been used to introduce Exact lumpability.

Definition 8.12 (Exact lumpability)**.** Let $\mathcal{X}(t)$ be a CTMC with state space $\mathcal{S}$ and let $\sim$ be an equivalence relation over $\mathcal{S}$. We say that $\mathcal{X}(t)$ is *exactly lumpable* with respect

to $\sim$—in the same way, $\sim$ is an *exact lumping* for $\mathcal{X}(t)$—if $\sim$ induces a partition on the state space $\mathcal{S}$ such that for any equivalence classes $S_i, S_j \in \mathcal{S}/\sim$ and for each pair $s, s' \in S_i$, the following holds:

$$\sum_{\overline{s} \in S_j} q_{\overline{s},s} = \sum_{\overline{s} \in S_j} q_{\overline{s},s'}$$

While Strong Lumpability takes into account the outgoing rates, its Exact version deals with the incoming ones. As we did before, we can define the aggregated—lumped—Markov chain:

Definition 8.13 (Exactly Lumped Markov Chain)**.** Let $\mathcal{X}(t)$ be a CTMC with state space $\mathcal{S}$ and let $\sim$ be a strong lumpability for $\mathcal{X}(t)$. The *Lumped* CTMC $\tilde{\mathcal{X}}(t)$ is defined as the Markov process which has $S/\sim$ as set of states. For such sets it must hold that $\tilde{\mathcal{X}}(t) = S \in \mathcal{S}/\sim$ if and only if $\mathcal{X}(t) = s \in S$. The transition rates between classes are defined as follows:

$$\tilde{q}_{S,S'} = \sum_{s \in S} q_{s,s'}$$

where $S, S' \in \mathcal{S}/\sim$ and $s' \in S'$.

The notions of strong and exact lumpability can be combined to obtain a stronger condition, called *strict lumpability.*

Definition 8.14 (Strict Lumpability)**.** Let $\mathcal{X}(t)$ be a CTMC with state space $\mathcal{S}$ and let $\sim$ be an equivalence relation over $\mathcal{S}$. We say that $\mathcal{X}(t)$ is *strictly lumpable* with respect to $\sim$—in the same way, $\sim$ is a *strict lumping* for $\mathcal{X}(t)$—if $\mathcal{X}(t)$ is both strongly and exactly lumpable with respect to $\sim$.

All the methods we introduced up to now are known to be *exact.* Two states are lumpable if and only if their incoming-outgoing rates enjoy some particular properties. Nevertheless, there may be Markov processes where no trivial lumpability $\sim$ can be computed because of the strict conditions. Therefore, a notion of *quasi-lumpability* has been introduced in [75]. Roughly speaking, quasi-lumpability allows lumping—aggregate—states whose aggregated rates are *almost the same* up to a small perturbation.

Definition 8.15 (Quasi-lumpability)**.** Let $\mathcal{X}(t)$ be a CTMC with state space $\mathcal{S}$ and let $\sim$ be an equivalence relation over $\mathcal{S}$. We say that $\mathcal{X}(t)$ is *quasi-lumpable* with respect to $\sim$ with bound ϵ—in the same way, $\sim$ is a *quasi-lumping* for $\mathcal{X}(t)$ with respect to ϵ—if $\sim$ induces a partition on the state space $\mathcal{S}$ such that for any equivalence classes $S_i, S_j \in \mathcal{S}/\sim$, with $i \neq j$, and for each pair $s, s' \in S_i$, the following holds:

$$\left| \sum_{\overline{s} \in S_j} q_{s,\overline{s}} - \sum_{\overline{s}} q_{s',\overline{s}} \right| \leq \epsilon$$

The very same notion was introduced as *near-lumpability* in [45]. Due to these small perturbations introduced while aggregating the states, the investigation of the steady distribution after the reduction does not easily follow from the definition—as in the

Strong and Exact cases. The topic has been extensively studied in [156, 69] and many more.

Quasi-lumpability allows major reductions in the number of states thanks to its looser aggregation condition. Despite this advantage, the relation between steady state pre and post lumping is not that useful.

To cope with this problem, the notion of *proportional lumpability* was introduced in [115] and further investigated in [116]. As the name suggests, this lumpability introduces the notion of proportionality as condition aggregation. This ingredient extends the original definition of Strong—Ordinary—Lumpability in the way quasi-lumpability does, but on the other hand, it also allows to exactly solve the original chain by working on the aggregated one.

Definition 8.16 (Proportional Lumpability)**.** Let $\mathcal{X}(t)$ be a CTMC with state space $\mathcal{S}$ and let $\sim$ be an equivalence relation over $\mathcal{S}$. We say that $\mathcal{X}(t)$ is *proportionally lumpable* with respect to $\sim$—in the same way, $\sim$ is a *proportional lumping* for $\mathcal{X}(t)$—if there exists a function $\kappa : \mathcal{S} \to \mathbb{R}$ such that $\sim$ induces a partition on the state space $\mathcal{S}$ such that for any equivalence classes $S_i, S_j \in \mathcal{S}/\sim$, with $i \neq j$, and for each pair $s, s' \in S_i$, the following holds:

$$\frac{\sum_{\overline{s} \in S_j} q_{s,\overline{s}}}{\kappa(s)} = \frac{\sum_{\overline{s}} q_{s',\overline{s}}}{\kappa(s')}$$

Exactly as we did for the other lumpabilities, we now describe how the proportionally lumped Markov Process $\tilde{\mathcal{X}}(t)$ is obtained starting from its original version $\mathcal{X}(t)$, the equivalence $\sim$, and the function κ.

Definition 8.17 (Proportionally Lumped Markov Chain)**.** Let $\mathcal{X}(t)$ be a CTMC with state space $\mathcal{S}$. Let $\kappa : \mathcal{S} \to \mathbb{R}^+$ be a function. We say that $\tilde{\mathcal{X}}(t)$ is a proportionally lumped version of $\mathcal{X}(t)$ with respect to κ if it is obtained from $\mathcal{X}(t)$ by perturbing its rates such that for all $s, s' \in \mathcal{S}$, with $s \neq s'$, it holds that:

$$\tilde{q}(s, s') = \frac{q_{s,s'}}{\kappa(s)}$$

In [116], authors also provided a constructive relation between $\pi(s)$ and $\tilde{\pi}(s)$ the steady state distributions of $\mathcal{X}(t)$ and $\tilde{\mathcal{X}}(t)$, respectively. In particular, they derived a property satisfied by the former with respect to the latter.

Theorem 29. *Let $\mathcal{X}(t)$ be a CTMC with state space $\mathcal{S}$ and steady state distribution π. Moreover, let $\kappa : \mathcal{S} \to \mathbb{R}^+$ be a function and let $\tilde{\mathcal{X}}(t)$ be a proportionally lumped version of $\mathcal{X}(t)$ with respect to κ. Assuming $\tilde{\pi}$ is the steady state distribution of $\tilde{\mathcal{X}}(t)$, then π satisfies the following property for all $s \in \mathcal{S}$:*

$$\pi(s) = \frac{\tilde{\pi}(s)}{K\kappa(s)}$$

where $K = \sum_{s \in \mathcal{S}} \tilde{\pi}(s)/\kappa(s)$

For the sake of completeness, we refer the reader to [116] for more details on Proportional lumpability, in particular for the characterization of proportionally lumpable CTMCs.

Conclusions on Bisimulations and Lumpabilities

In this chapter, we reviewed some key families of reduction relations employed in formal methods and system evaluation: bisimulations and lumpabilities. Both concepts arise when we attempt to formally define the essential properties that a *proper* reduction relation must possess to be effective in contexts such as Markov Chains and Neural Networks.

On the one hand, bisimulation is a relation that connects two nodes based on their neighbourhoods. While it can be viewed as a *local version* of graph isomorphism, it is computationally efficient to determine and surprisingly effective in practice. Bisimulation proves particularly useful for both labelled and unlabelled graphs. We have proven bisimulation capabilities by presenting a graph-based algorithm to compute the minimum automaton both in the deterministic and in the non-deterministic case.

On the other hand, lumpabilities can be considered the *weighted counterpart* of bisimulations. They assess the relationship between two nodes not only by examining their neighbourhoods but also by considering the weights of the edges connecting them. Like bisimulations, lumpabilities have demonstrated significant effectiveness in various applications. For example, a substantial line of work deals with the generalization of lumpability concepts to differential equations [48], boolean networks [49] and, most recently, quantum circuits [101].

9

Reducing Neural Networks

This chapter's goal is to introduce the reader to the concept of Neural Network reduction. Clearly, a major focus will be put in compression techniques adopting graph as key ingredients. Nonetheless, we will also give a brief introduction to other approaches that may be applied.

Nowadays, Neural Networks (NNs) are experiencing an apparently never-ending spike of interest from both the academia and the industry. Especially thanks to their great performances in approximating problems that may seem intractable, they have been adopted as first source of solution for many real life tasks. Some examples may be object recognitions, image/video generation, text analysis and so on and so forth.

When approaching Neural Networks with a task to solve, one must provide tons of data and come up with a model that performs well in generalizing solutions to the given problem. Clearly, not all the models are a good fit for every task and not all data are good for training[1].

While the data cleaning and augmenting techniques are out of the scope of this thesis, we will focus on the model choice. In particular, NN models must cope with the rapid increase of data they are provided. To do so, AI architects must use larger and larger models. The size of NNs are not an issue when dealing with supercomputers: the same does not hold when IOT manufacturers want to fit Machine Learning components into devices with a limited amount of memory and computational power. To fix this particular issue, authors are investigating techniques to reduce the size of Neural Networks. While techniques to remove nodes are usually taken into account while training—dropout—network reductions involve edge deletion.

9.1 Weight Sharing

The first and most simple technique for NN reduction is known as *Weight sharing* [134]. With this approach, adopted before the training, some network layers are forced to have

[1] *Notice that all the terms like model, training, Neural Network and similar have been introduced in Section 1.5*

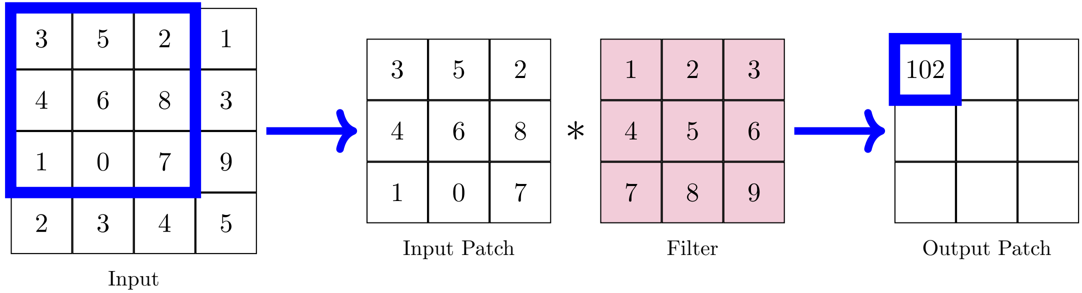


Figure 9.1: Applying a 3x3 filter—in pink—to a 4x4 image.

the same set of weights. For example, in a Neural network with 4 hidden layers, we can force the first and the third in sharing the same weights between neurons. In this way, it is clear that the amount of memory required to store the entire model is smaller. However, expressiveness may be effected since some part of the function landscape may not be accessible during the gradient descent phase.

This kind of approach is often used when the tackled task concerns images. To solve this kind of problems, Convolutional Neural Networks (CNNs) are adopted. The main difference between this kind of networks and standard NNs is the introduction of filters. Usually described as tensors, their *application* to images is used to extract features. We depicted the application of a depth 1 filter in Figure 9.1.

While the forward pass of a Neural Network can be represented as a matrix-vector product, the application of a filter is a slightly more complicated process. Despite these theoretical details, filters, exactly as weights, can be learned and tuned during the network training phase. Hence, they count in the overall weight of a Network. Since two filters may be extracting similar features but in different layers, it is natural to think of applying weight sharing. Instead of training—and storing—two different filters, I just keep one of the two.

This approach seems really promising, but its application is not that easy. In fact, one cannot know in principle if two filters may tend to some similar values, especially due to the lack of explainability. Hence, a first training phase must be done without any sharing. Subsequently, a statistical study must be performed to check if this technique can be applied.

9.2 Pruning

Weight sharing in some sense *hides* the problem of overparametrization—too many parameters involved to solve a single task—by making weights coincide between layers. A technique that actually removes weights to reduce the dimension of the network is pruning [134].

While weight sharing is an internal feature of the model—*hardcoded*—, pruning algorithms are applied after the network is trained. The type of algorithms that can be applied lie in two categories—See Table 9.1:

- **Exact**: the algorithm prunes weights if and only if some strict conditions hold.

The accuracy of the network remains the same before and after the reduction. They may be more expensive in terms of computational power and could possibly prune less weights, but they *theoretically* ensure no loss in terms of problem-solving capacity.

- **Approximate**: weights are removed using some heuristics—less strict conditions to be met—that are in many case tweaked to best suit the specific use case. The accuracy of the network may drastically drop after the algorithm is applied. They are usually computationally cheaper but may require a retraining phase after their application.

A brief aside is needed to discuss the computational complexity of pruning. Exact techniques are grounded in strong theoretical foundations that ensure the network's accuracy remains unaffected by changes. As a result, it is reasonable to expect that the conditions required for removing a weight are computationally intensive. However, once a network is pruned using an exact algorithm, no further modifications are needed. In contrast, approximate methods may offer faster pruning but can become more computationally demanding if retraining is necessary. In summary, choosing between exact and approximate approaches to network pruning involves a trade-off between time and accuracy.

	Exact	**Approximate**
Accuracy	Unaffected	Affected
Computational Cost	Hypothetically high	Hypothetically low
Retraining	Not Required	Often Required

Table 9.1: Comparison between Exact and Approximate Pruning Methods

We begin with a brief overview of approximate pruning techniques.

The most straightforward pruning strategy involves setting a threshold, θ, which determines which weights are removed. The value of θ can vary for each layer, possibly based on the mean of the weights or another statistical measure. Alternatively, θ can be set globally for the entire network, without distinguishing between layers. This technique can be refined by eliminating θ altogether and pruning weights with small absolute values, often those with vanishing gradients. However, this approach may not always lead to significant pruning, especially if (i) the threshold is set incorrectly or (ii) most weights have a high magnitude.

To address this issue, a target percentage for deletion can be introduced. For each layer l, a specific percentage p_l of connections is targeted for removal. A strategy is devised to organize the connections so that the specified percentage can be effectively pruned. For example, all weights can be sorted, and the smaller p_l values removed. To introduce some randomness, a portion of the percentage can be pruned deterministically, while the remainder is pruned randomly. In any case, this approach ensures that a percentage p_l is removed, leading to a reduction in network size.

Similar to the threshold approach, a global percentage p_L can be set instead of individual percentages for each layer.

A crossover between the aforementioned methods can be found in [56]. They devise an algorithm that scores connections—weights—according to what they call an *information theoretic* approach. Roughly speaking, they compute how much information every edge carries taking into account the training set too. For each layer, they find which weights are contributing less than some threshold to the overall computation, and they remove them. No particular information theoretic techniques are adopted, but just means and statistical measures are used instead. Hence, the reference to information theory is more related to the fact that they try to estimate how much information every connection is carrying through the evaluation step.

What is interesting about this proposal, is that it looks like an instantiation of a more general pruning framework. In particular, Algorithm 13 can be extracted.

Algorithm 13 Pruning-Framework (Net, f, θ)

```
1:  for every layer l in Net do
2:      for every neuron v in l do
3:          for every edge e that enters in v do      ▷ The weight associated to connection e is w_e
4:              f_e ← f(w_e)
5:              if f_e < θ then
6:                  Remove e
7:              end if
8:          end for
9:      end for
10: end for
11: return The pruned network
```

We have to run through all the possible connections—in this case, we interpret them as ingoing links—which we achieve using the three for loops in the algorithm. Then, a score has to be computed for every weight. The score function does not need to follow any specific rule as long as it aligns with the pruning requirement we want to achieve. It can be designed to be either hard or easy to compute, it can produce large or small values, and it might lead to many collisions. The only requirements are that it must be computable and suitable for the specific use case we are addressing. Once this score is computed, we check if it is above or below a certain threshold θ. With this simple check, we can decide whether a connection remains in the network or not.

In the literature, the value that the score function assigns to the weight is referred to as *magnitude*. All techniques that calculate some form of magnitude and use it to prune the network are categorized under *Magnitude-Based Pruning* (MBP). However, these techniques make local decisions—considering individual layers, *greedily*—while neglecting the overall network structure and accuracy. This can potentially lead to a noticeable drop in accuracy.

To mitigate this issue, it may be beneficial to consider the loss function when deciding whether to keep or remove a weight. Some strategies involve the following steps:

1. Traverse a set of weights W.
2. For each weight $w \in W$, compute the loss function as if w were removed.
3. Remove the weight that causes the smallest change in loss.

This method is computationally expensive, as it requires a forward pass for each weight considered. However, it minimizes the potential loss in accuracy. We now turn

our attention to exact solutions for the pruning problem, beginning with the algorithm introduced in [50]. A key distinction between the algorithm presented in [50] and methods such as MBP approaches is that while the latter removes connections—i.e., edges or weights—the former operates on neurons, or nodes.

Deleting a neuron can be conceptualized as removing all its incoming and outgoing connections—see Figure 9.2. Thus, pruning a node v in a neural network can be viewed as pruning a set of edges that share the same destination—node v—and a set of edges that share the same source—again, node v. These two sets are central to the algorithm.

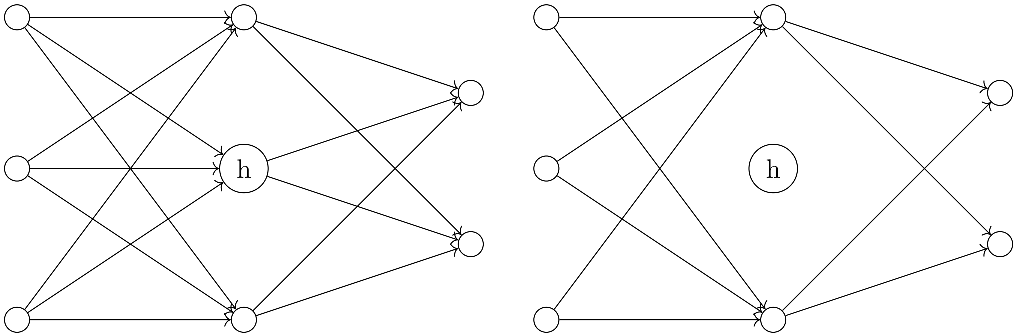


Figure 9.2: Left: Node h is selected for deletion. Right: removing all h's incoming and outgoing connections.

Specifically, let $u_1, u_2, \ldots, u_{p_v}$ represent the NN units such that there exists an edge with source v and destination u_i for all $i \in \{1, 2, \ldots, p_v\}$. The set $P_v = \{u_1, u_2, \ldots, u_{p_v}\}$ is termed the *projective field* of v. Roughly speaking, this is the set of nodes *that are fed by* v.

Similarly, let $u_1, u_2, \ldots, u_{r_v}$ represent the NN units such that there exists an edge with source u_i and destination v for all $i \in \{1, 2, \ldots, r_v\}$. The set $R_v = \{u_1, u_2, \ldots, u_{r_v}\}$ is referred to as the *rejective field* of v. Essentially, this is the set of nodes *that feed* v.

Assume you have established some criteria to decide whether a unit should be removed—this could be random, guided by an external procedure, or based on a percentage goal. Suppose unit h has been selected as the next candidate for removal. As previously mentioned, removing a unit is equivalent to removing all its incoming and outgoing edges.

Given that this method is *exact*, it is not sufficient to simply remove h, set all associated weights to zero, and proceed. Instead, we must adjust other weights to compensate for the removal of h-related edges, thereby ensuring that accuracy remains unaffected. The authors propose addressing this by modifying all the weights of the edges entering nodes in P_h—the projective field of h.

Consider one specific node $u \in P_h$. Let w be the weight removed by eliminating the edge that connects h and u. Since we need to maintain the input value to u unchanged, all other weights must be adjusted by certain factors to achieve this goal. Therefore, for node u, we define a system of linear equations to determine a set of factors which, when added to the current weights, allow u's input to remain constant.

Applying this procedure to all nodes within P_h constitutes the core of the algorithm described in [50].

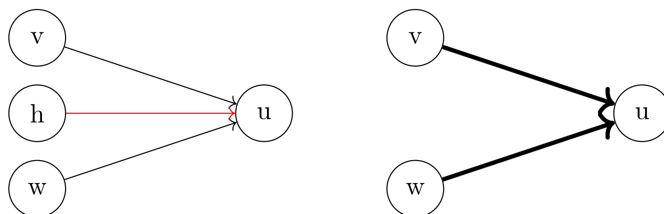


Figure 9.3: Left: a network where node h has only u in its projective field. Right: the edges between v, w and u are thicker since they have to compensate for the edge (h, u) removal.

An attentive reader may notice that this method is far from being *universally* exact. The reason is twofold:

1. First, to find the coefficient adopted to adjust the weights, one must solve linear systems. Since the network is overparametrized, it is natural to think that the system may be oversized. Therefore, to find a *good* solution one must use approximation methods like least squared. Hence, obtaining coefficients that minimize the difference between the input given to u before removing h and after removing it.

2. Even if that was not the case—we are lucky enough that the system can be solved exactly—we have to deal with the very core of NNs: data. When we refer to the input given to a neuron, such value does not depend solely on the weights but also from the sample that is being fed to the network. Thus, when we adopt this technique we try to minimize the change in values with respect to one particular data set—may it be the training, test, or validation one—but we cannot assume any drop in performance when dealing with extra data.

Instead, a method that *ensures* that the accuracy is actually unaffected from pruning has been presented in [151]. We investigated such proposal in Section 9.3.

It is important to notice that all the aforementioned approaches fall under the *unstructured pruning* technique. With this term we mean all the techniques that after being applied, may generate networks that do not have a symmetric structure. This is a major issue especially for GPU operations: the NNs forward pass is computed as a set of matrix-vector products. The pruning of a weight is represented in the weight matrix by zeroing some entries. This operation generates sparse matrices that are not well suited for the parallel computation performed by GPUs.

To cope with this problem, *structured pruning* has been introduced. The techniques that belong to this category are really effective especially when dealing with filters in CNNs. For example, when *some parts* of a filter must be removed, one may suppose to remove the same part from all the other filters to keep some kind of regularity in the network.

Readers that are interested in delving into the details of Neural Network Pruning, they can refer to [134, 36] for further details.

9.3 Lumping

Neural Network can naturally be interpreted as weighted graphs where:

- The set of nodes is the set of neurons
- The set of edges is made by all the connections in the network
- Weights matrix is the graph's weight function

By the way, interpreting NNs as graphs may sound like an overkill for two main reasons:

- on the one hand, a lot of graphs-theoretical aspects can be neglected. Clearly, all nodes in a Neural Network are reachable, we are not interested in the shortest paths, a fully connected NN is always a tree and so on and so forth.
- on the other hand, properties like the *layering* of the nodes in Fully Connected NNs or the presence of biases are completely lost. To introduce them, one must tweak the definition of weighted graph. In most of the cases this procedure leads to a re-definition of NNs.

Hence, we ask ourselves if this may be the case in which graphs should not be tampered. Unfortunately for our readers, this is wrong. In particular, we will now investigate how a concept like Bisimulation can be adopted to devise a NN pruning technique.

When dealing with bisimulations in Chapter 8, we roughly defined them as behavioural equivalences. This holds for both bisimulations with and without weights on edges. Therefore, it seems like an *easy task* to adapt bisimulation to Neural Networks since they can be interpreted as weighted graphs. *Spoiler*: it is not.

NNs, and Machine Learning models in general, deal with data. We are used to describing graphs as structures that do not receive any kind of input from the outside environment, usually they are the input. In the case of a network, the graph is flooded with data that passes through edges, is given as input to non-linear functions and the repetition of these operations generate the *predictions*. Thus, defining equivalence between two neurons cannot remain aseptic from the concepts of input data. Whenever two units are defined as equivalent, it must be true that they behave in the same way for any input sample they are fed with. Hence, since we do not have access to the whole universe of samples for a single task, we must adopt clever techniques to ensure the equivalence. Solving this task would lead to a pruning algorithm that can substitute any equivalence class with a single neuron.

The very first appearance of bisimulation-like approaches linked to neural network pruning can be found in [146]. The authors adopt the notion of Exact Lumpability (see Def 8.12) as method to aggregate neurons in a network. In particular, they prove that the equivalence between what they call *Presums* are enough to define two states equivalent while preserving the network semantics—accuracy. With the term *Presum* they refer to the sum of the incoming weights of a neuron.

Formally speaking:

Definition 9.1. Let $\mathcal{S}$ be a set of states in layer $l-1$ and let s be a state in layer l. Then, $Presum(S, s)$ is defined as follows:

$$Presum(S, s) = \sum_{\overline{s} \in S} W^l_{\overline{s},s}$$

The models they prune with this approach are Fully Connected. Suppose layer $l-1$ has already been partitioned in three classes $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$. Two nodes s, s' in layer l are defined equivalent if and only if all the following conditions hold:

1. They have the same activation function
2. They have the same bias
3. $Presum(\mathcal{P}_i, s) = Presum(\mathcal{P}_i, s')$ holds for all $i = \{1, 2, 3\}$

Using this equivalence notion they were able to both devise a pruning algorithm and to prove that it maintains the accuracy unaffected. Nevertheless, the condition seems a bit too strict. Forcing the exact equivalence on biases is a strong condition since we deal with real number that can vary over all $\mathbb{R}$. Even if the check of the biases is overcome, the same problem holds also for the sums: it is difficult that two sums of real numbers that do not lie in any particular range end up being equal. Therefore, even if the algorithm is theoretically strong and sound, the range of real cases where it can be applied is limited.

For this reason, a slightly different definition of bisimulation—lumpability—between neurons in NNs has been proposed in [151]. The idea developed in such paper is to introduce proportionality between both the *Presums* and the biases. In this way the conditions are easier to be met, and the network may benefit a larger degree of pruning. The authors started from introducing the notion of proportional lumpability that we stated in Definition 8.16. However, since we are going to investigate this proposal, we recall the definition given by the authors to align with their notation.

Definition 9.2 (Proportional Lumpability)**.** Let $\mathcal{X}(t)$ be a CTMC and $\sim$ be an equivalence relation over $\mathcal{S}$. $\sim$ is a *proportional exact lumpability* if there exists a function $\rho : \mathcal{S} \to \mathbb{R}_{>0}$ such that for all $S, S' \in \mathcal{S}/\mathcal{R}$, for all $x, y \in S$ it holds that:

$$\rho(x) \sum_{z \in S'} q_{z,x} = \rho(y) \sum_{z \in S'} q_{z,y}. \tag{9.1}$$

Notice that this definition is the same as Def 8.16 where $\frac{1}{\kappa} = \rho$.

After introducing this relation, we embed it into the NNs background, obtaining the fresh new definition of Proportional Exact Lumpability over a NN.

Definition 9.3 (Proportional Exact Lumpability over a NN)**.** Let $\mathcal{N}$ be a NN. Let $\mathcal{R} = \cup_{\ell \in [k)} \mathcal{R}_\ell$ be such that $\mathcal{R}_\ell$ is an equivalence relation over $\mathcal{S}_\ell$, for all $\ell \in (k)$ and $\mathcal{R}_0$ is the identity relation over $\mathcal{S}_0$. We say that $\mathcal{R}$ is a *proportional exact lumpability* over $\mathcal{N}$ if for each $\ell \in (k)$ there exists $\rho_\ell : \mathcal{S}_\ell \to \mathbb{R}_{>0}$ such that for all $S \in \mathcal{S}_\ell/\mathcal{R}_\ell$, for all $S' \in \mathcal{S}_{\ell-1}/\mathcal{R}_{\ell-1}$, for all $s_1, s_2 \in S$ the following Equations hold:

$$\rho_\ell(s_1) b^\ell_{s_1} = \rho_\ell(s_2) b^\ell_{s_2}, \tag{9.2}$$

$$\rho_\ell(s_1) \sum_{r \in S'} W^\ell_{r,s_1} = \rho_\ell(s_2) \sum_{r \in S'} W^\ell_{r,s_2} \,. \tag{9.3}$$

As the reader may see, there are some differences with respect to the definition of proportional lumpability over CTMCs.

- it must be true that, in order to be equivalent, two neurons have to belong to the same layer
- Input and output nodes must not be aggregated. For the input layer the reason is the lack of incoming edges. There would be no *previous level partition* to start from when computing the presums. For the output one, the reason is more *semantic*. While dealing with a multiclass classification task, we must have one output per class. Lumping the last layer would cause this condition to fail.

If the relation adopted to lump is the one defined in 9.3, the topology and the weights of the NN must be modified as follows:

Definition 9.4 (Proportional Reduced NN)**.** Let $\mathcal{N} = (k, \{\mathcal{S}_\ell\}_{\ell\in[k]}, \{W_\ell\}_{\ell\in(k]}, \{b_\ell\}_{\ell\in(k]}, \{A_\ell\}_{\ell\in(k]})$ be a NN. Let $\mathcal{R}$ be a proportional exact lumpability over $\mathcal{N}$.
The NN $\mathcal{N}/\mathcal{R} = (k, \{\mathcal{S}'_\ell\}_{\ell\in[k]}, \{W'_\ell\}_{\ell\in(k]}, \{b'_\ell\}_{\ell\in(k]},$
$\{A'_\ell\}_{\ell\in(k]})$ is defined by:

- $\mathcal{S}'_\ell = \{[s] \,|\, [s] \in \mathcal{S}_\ell/\mathcal{R}\}$, where s is an arbitrarily chosen representative for the class;
- $(W')^\ell_{[s_1],[s_2]} = \rho_{\ell-1}(s_1) \sum_{r\in[s_1]} \frac{W^\ell_{r,s_2}}{\rho_{\ell-1}(r)}$;
- $(b')^\ell_{[s]} = b^\ell_s$;
- $A'_\ell([s]) = A_\ell(s)$.

An example of the application of proportional lumping to a NN is depicted in Figure 9.4. One node is removed from the network and the other weights are updated in order to cope with the deletion.

We prove that computing a Proportionally Lumped NN as defined in 9.4, the accuracy does not change. To obtain such result, we start by defining the relation between the semantic of a layer and its pruned version.

Lemma 9.1. *Let $v \in Val(\mathcal{S}_{\ell-1})$ be a valuation for layer $\ell-1$. Let $v'' = (\llbracket\mathcal{N}\rrbracket_{\ell+1} \circ \llbracket\mathcal{N}\rrbracket_\ell)(v)$ and $u'' = (\llbracket\mathcal{N}/\mathcal{R}_\ell\rrbracket_{\ell+1} \circ \llbracket\mathcal{N}/\mathcal{R}_\ell\rrbracket_\ell)(v)$ be the valuation for layer $\ell+1$ in $\mathcal{N}$ and $\mathcal{N}/\mathcal{R}_\ell$, respectively. The following Equation (9.4) hold:*

$$v'' = u'' \,. \tag{9.4}$$

Proof. Let $t \in \mathcal{S}_{\ell+1}$. For the sake of readability, we make some assumptions on layer ℓ. In any case, the theorem is easily extendable to the generic case. Let $s_1, s_2, s_3, s_4, s_5 \in \mathcal{S}_\ell$ be five neurons from layer ℓ such that $s_2, s_3 \in [s_1]_{\mathcal{R}_\ell}$ and $s_5 \in [s_4]_{\mathcal{R}_\ell}$. Let $v' = \llbracket\mathcal{N}\rrbracket_\ell(v)$

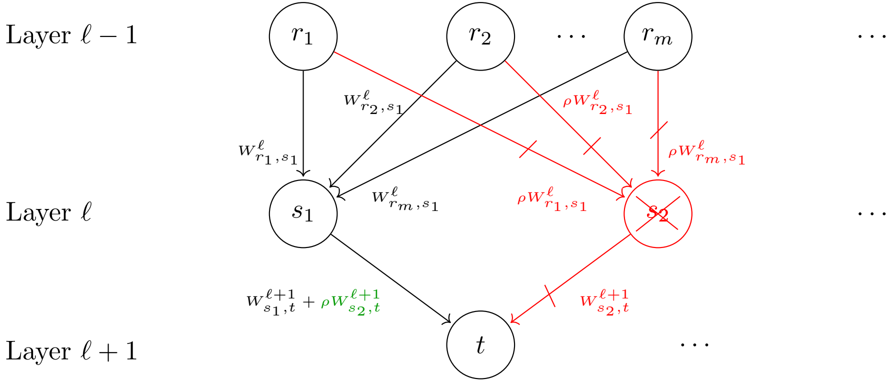


Figure 9.4: Pruning one node and updating the network.

be the valuation at layer ℓ. The following relations between $v'(s_1), v'(s_2), v'(s_3), v'(s_4)$, and $v'(s_5)$, represented in Equations (9.5), (9.6), and (9.7), hold:

$$\rho_\ell(s_1)v'(s_1) = \rho_\ell(s_2)v'(s_2) \tag{9.5}$$
$$\rho_\ell(s_1)v'(s_1) = \rho_\ell(s_3)v'(s_3) \tag{9.6}$$
$$\rho_\ell(s_4)v'(s_4) = \rho_\ell(s_5)v'(s_5) \tag{9.7}$$

The reason for such a result is twofold: first, we are not lumping layer $\ell - 1$. Hence, there is no change in the valuation for layer $\ell - 1$. Second, s_1, s_2, and s_3 are equivalent according to $\mathcal{R}_\ell$ (the same holds for s_4 and s_5). By definition, the value of $v''(t)$ is as in (9.8)

$$\begin{aligned} v''(t) = A_{\ell+1}(t)\big(&W^{\ell+1}_{s_1,t}v'(s_1)+ \\ &+ W_{\ell+1}(s_2,t)v'(s_2) + W^{\ell+1}_{s_3,t}v'(s_3) + W^{\ell+1}_{s_4,t}v'(s_4) + W^{\ell+1}_{s_5,t}v'(s_5) + b^{\ell+1}_t\big) \end{aligned} \tag{9.8}$$

Using the relation between the values of v' represented in Equations (9.5), (9.6), and (9.7), we can rewrite Equation (9.8) as in the following Equation (9.9):

$$\begin{aligned} &= A_{\ell+1}(t)\left(\sum_{i=1}^{3}\left[W^{\ell+1}_{s_i,t}\frac{\rho_\ell(s_1)}{\rho_\ell(s_i)}v'(s_1)\right] + \sum_{i=4}^{5}\left[W^{\ell+1}_{s_i,t}\frac{\rho_\ell(s_4)}{\rho_\ell(s_i)}v'(s_4)\right]\right) \\ &\qquad = A_{\ell+1}(t)\Bigg(v'(s_1)\left[W^{\ell+1}_{s_1,t} + W_{\ell+1}(s_2,t)\frac{\rho_\ell(s_1)}{\rho_\ell(s_2)} + W^{\ell+1}_{s_3,t}\frac{\rho_\ell(s_1)}{\rho_\ell(s_3)}\right] + \\ &\qquad\qquad + v'(s_4)\left[W^{\ell+1}_{s_4,t} + W^{\ell+1}_{s_5,t}\frac{\rho_\ell(s_4)}{\rho_\ell(s_5)}\right]\Bigg) \end{aligned} \tag{9.9}$$

which is equal to $u''(t)$. □

Secondly, we focus on the consequences of pruning a layer j on the remainder of the network. What we obtain is that the semantics of the other before and after the j-th remains unaffected.

Lemma 9.2. *Let i, j be two layers index. If $j < i-1$ or $j > i$, then $[\![\mathcal{N}]\!]_i = [\![\mathcal{N}/\mathcal{R}_j]\!]_i$*

Proof. Let i, j be two layers. In the first case, $j > i$. Since the reduced layer is after the i-th, the semantic at layer i is unchanged. In the second case, $j < i-1$. We are reducing a layer that is *before* the i-th. By Lemma 9.1, we know that the semantic of layer j remains the same. Hence, layer i will not be affected by the reduction □

Composing the results obtained from Lemmata 9.1, 9.2, we are able to prove the semantic equivalence between a neural network and its lumped version.

Theorem 30. *Let $\mathcal{N}$ be a NN and $\mathcal{R}$ be a proportional exact lumpability over $\mathcal{N}$. It holds that*

$$[\![\mathcal{N}/\mathcal{R}]\!] = [\![\mathcal{N}]\!]\,. \tag{9.10}$$

Proof. By definition of $[\![\mathcal{N}]\!]$, we have:

$$[\![\mathcal{N}]\!] = [\![\mathcal{N}]\!]_k \circ [\![\mathcal{N}]\!]_{k-1} \circ \cdots \circ [\![\mathcal{N}]\!]_1 \circ [\![\mathcal{N}]\!]_0 \tag{9.11}$$

The application of Lemma 9.1 to the two rightmost terms of Equation (9.11) yields to:

$$[\![\mathcal{N}]\!]_k \circ [\![\mathcal{N}]\!]_{k-1} \circ \cdots \circ [\![\mathcal{N}/\mathcal{R}_0]\!]_1 \circ [\![\mathcal{N}/\mathcal{R}_0]\!]_0 \tag{9.12}$$

Using Lemma 9.2 on all the other terms of (9.12) we get:

$$[\![\mathcal{N}/\mathcal{R}_0]\!]_k \circ [\![\mathcal{N}/\mathcal{R}_0]\!]_{k-1} \circ \cdots \circ [\![\mathcal{N}/\mathcal{R}_0]\!]_2 \circ [\![\mathcal{N}/\mathcal{R}_0]\!]_1 \circ [\![\mathcal{N}/\mathcal{R}_0]\!]_0 \tag{9.13}$$

We can interpret $[\![\mathcal{N}/\mathcal{R}_0]\!]$ as our *new* network. Hence, we proceed by applying Lemma 9.1 to $[\![\mathcal{N}/\mathcal{R}_0]\!]_2 \circ [\![\mathcal{N}/\mathcal{R}_0]\!]_1$ obtaining $[\![\mathcal{N}/(\mathcal{R}_0 \cup \mathcal{R}_1)]\!]_2 \circ [\![\mathcal{N}/(\mathcal{R}_0 \cup \mathcal{R}_1)]\!]_1$. Consequently, the application of Lemma 9.2 to the remaining terms of (9.13) yields to:

$$[\![\mathcal{N}/(\mathcal{R}_0 \cup \mathcal{R}_1)]\!]_k \circ [\![\mathcal{N}/(\mathcal{R}_0 \cup \mathcal{R}_1)]\!]_{k-1} \circ \cdots \circ [\![\mathcal{N}/(\mathcal{R}_0 \cup \mathcal{R}_1)]\!]_2 \circ [\![\mathcal{N}/(\mathcal{R}_0 \cup \mathcal{R}_1)]\!]_1 \circ [\![\mathcal{N}/(\mathcal{R}_0 \cup \mathcal{R}_1)]\!]_0 \tag{9.14}$$

Repeating such process starting from (9.14) and recalling that $\mathcal{R} = \bigcup_{i \in [k]} \mathcal{R}_i$, we end up with (9.15):

$$[\![\mathcal{N}/\mathcal{R}]\!]_k \circ [\![\mathcal{N}/\mathcal{R}]\!]_{k-1} \circ \cdots \circ [\![\mathcal{N}/\mathcal{R}]\!]_2 \circ [\![\mathcal{N}/\mathcal{R}]\!]_1 \circ [\![\mathcal{N}/\mathcal{R}]\!]_0 \tag{9.15}$$

which is exactly $[\![\mathcal{N}/\mathcal{R}]\!]$, hence proving (9.10). □

After obtaining this strong equivalence result, we shift our focus to the creation of an algorithm to efficiently compute the lumped version of a Neural Network. We produce a *greedy* Algorithm based on the following result.

Lemma 9.3. *Let $\ell, \ell+1 \in (k)$ be two layers, and $\mathcal{R}$ be a proportional exact lumpability such that $t_1, t_2 \in \mathcal{S}_{\ell+1}$ and $(t_1, t_2) \in \mathcal{R}$. Choose $s_1, s_2 \in \mathcal{S}_\ell$ such that $(s_1, s_2) \notin \mathcal{R}$. Finally, define $\mathcal{R}'$ as in the following Equation (9.16)*

$$\mathcal{R}' = (\mathcal{R} \cup \{(s_1, s_2)\})^{tr} \tag{9.16}$$

where we denote the transitive closure of a relation with $(\cdot)^{tr}$*. It holds that:*

- $\rho_{\ell+1}(t_1)b_{t_1}^{\ell+1} = \rho_{\ell+1}(t_2)b_{t_2}^{\ell+1}$
- $\forall S \in \mathcal{S}_\ell/\mathcal{R}'_\ell : \rho_{\ell+1}(t_1)\sum_{s\in S} W_{s,t_1}^{\ell+1} = \rho_{\ell+1}(t_2)\sum_{s\in S} W_{s,t_2}^{\ell+1}$

Proof. Let s_1, s_2, t_1, t_2 be defined as in the statement of the Lemma. Since $(t_1, t_2) \in \mathcal{R}$ and $(s_1, s_2) \notin \mathcal{R}$, we have:

- $\rho_{\ell+1}(t_1)b_{t_1}^{\ell+1} = \rho_{\ell+1}(t_2)b_{t_2}^{\ell+1}$
- $\rho_{\ell+1}(t_1)\sum_{s\in[s_1]} W_{s,t_1}^{\ell+1} = \rho_{\ell+1}(t_2)\sum_{s\in[s_1]} W_{s,t_2}^{\ell+1}$
- $\rho_{\ell+1}(t_1)\sum_{s\in[s_2]} W_{s,t_1}^{\ell+1} = \rho_{\ell+1}(t_2)\sum_{s\in[s_2]} W_{s,t_2}^{\ell+1}$

Let us now consider $\mathcal{R}'$ instead of $\mathcal{R}$, where s_1 and s_2 have been put in the same equivalence class. Since the equivalence classes of $\mathcal{R}'$ are union of classes of $\mathcal{R}$, we get that:

- $\rho_{\ell+1}(t_1)b_{t_1}^{\ell+1} = \rho_{\ell+1}(t_2)b_{t_2}^{\ell+1}$
- $\forall S \in \mathcal{S}_\ell/\mathcal{R}'_\ell : \rho_{\ell+1}(t_1)\sum_{s\in S} W_{s,t_1}^{\ell+1} = \rho_{\ell+1}(t_2)\sum_{s\in S} W_{s,t_2}^{\ell+1}$

□

Roughly speaking, this Lemma proves that whenever two nodes are declared equivalent with respect to $\mathcal{R}$, the lumping of other layers does not affect their relation.

Exploiting this very result, we devise Algorithm 14 that given as input a Neural Network, produces its maximum proportionally lumped version.

Theorem 31. *Let* $\mathcal{N}$ *be a NN. There exists a unique maximum proportional exact lumpability* $\mathcal{R}$ *over* $\mathcal{N}$*. Moreover,* $\mathcal{R}$ *and* $\mathcal{N}/\mathcal{R}$ *can be computed in linear time with respect to the size of* $\mathcal{N}$*, i.e., in time* $\Theta\left(\sum_{\ell\in(k]} |\mathcal{S}_{\ell-1} \times \mathcal{S}_\ell|\right)$.

Algorithm 14 Computes the maximal proportional exact lumpability on $\mathcal{N}$

1: **function** LumpNeuralNetwork($\mathcal{N} = (k, \{\mathcal{S}_\ell\}_{\ell\in[k]}, \{W^\ell\}_{\ell\in(k]}, \{b^\ell\}_{\ell\in(k]}, \{A_\ell\}_{\ell\in(k]})$)
2: $\quad \mathcal{S}'_0 \leftarrow \{S_i = [s_i] \ : \ s_i \in \mathcal{S}_0\}$
3: $\quad$ **for all** $S_i = [s_i] \in \mathcal{S}'_0$ **do**
4: $\quad\quad$ **for all** $\overline{s}_j \in \mathcal{S}_1$ **do**
5: $\quad\quad\quad O(S_i, \overline{s}_j) \leftarrow W^1_{s_i,\overline{s}_j}$
6: $\quad\quad\quad \tilde{W}^1_{S_i,\{\overline{s}_j\}} \leftarrow W^1_{s_i,\overline{s}_j}$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: $\quad$ **for all** $l \in (k]$ **do**
10: $\quad\quad \mathcal{S}'_l \leftarrow \varnothing$
11: $\quad\quad$ **while** $\mathcal{S}_l \neq \varnothing$ **do**
12: $\quad\quad\quad s \leftarrow$ Pick($\mathcal{S}_l$)
13: $\quad\quad\quad C \leftarrow \{s\}$
14: $\quad\quad\quad (b')^l_C \leftarrow b_l(s), A'_l(C) \leftarrow A_l(s)$
15: $\quad\quad\quad$ **for all** $S' \in \mathcal{S}'_{l-1}$ **do**
16: $\quad\quad\quad\quad W'_l\,(S', C) \leftarrow \tilde{W}^l_{S',\{s\}}$
17: $\quad\quad\quad$ **end for**
18: $\quad\quad\quad$ **for all** $s' \in \mathcal{S}_l \setminus \{s\}$ **do**
19: $\quad\quad\quad\quad \rho_{s'} \leftarrow b^l_s / b^l_{s'}$
20: $\quad\quad\quad\quad$ **for all** $S' \in \mathcal{S}'_{l-1}$ **do**
21: $\quad\quad\quad\quad\quad$ **if** $\rho_{s'} * O(S', s') \neq O(S', s)$ **then**
22: $\quad\quad\quad\quad\quad\quad$ GO TO Line 18
23: $\quad\quad\quad\quad\quad$ **end if**
24: $\quad\quad\quad\quad$ **end for**
25: $\quad\quad\quad\quad C \leftarrow C \cup \{s'\}$
26: $\quad\quad\quad\quad \mathcal{S}_l \leftarrow \mathcal{S}_l \setminus \{s'\}$
27: $\quad\quad\quad\quad R(s') \leftarrow \rho_{s'}$
28: $\quad\quad\quad$ **end for**
29: $\quad\quad\quad \mathcal{S}'_l = \mathcal{S}'_l \cup \{C\}$
30: $\quad\quad$ **end while**
31: $\quad\quad$ **for all** $C \in \mathcal{S}'_l, s' \in \mathcal{S}_{l+1}$ **do**
32: $\quad\quad\quad O(C, s') \leftarrow \sum_{r\in C} W^{l+1}_{r,s'}$
33: $\quad\quad\quad \tilde{W}_l\,(C, \{s'\}) \leftarrow \sum_{r\in C} R(r) W^{l+1}_{r,s'}$
34: $\quad\quad$ **end for**
35: $\quad$ **end for**
36: $\quad$ **return** $\mathcal{N} \setminus \mathcal{R} = \left(k, \{\mathcal{S}'_\ell\}_{\ell\in[k]}, \{(W')^\ell\}_{\ell\in(k]}, \{(b')^\ell\}_{\ell\in(k]}, \{A'_\ell\}_{\ell\in(k]}\right)$
37: **end function**

Let $\mathcal{N} = (k, \{\mathcal{S}_\ell\}_{\ell\in[k]}, \{W^\ell\}_{\ell\in(k]}, \{b^\ell\}_{\ell\in(k]}, \{A_\ell\}_{\ell\in(k]})$ be a Neural Network. `LumpNeuralNetwork`($\mathcal{N}$) defined in Algorithm 14 computes the maximum proportional exact lumpability $\mathcal{R}$ over $\mathcal{N}$.

Before unravelling the technical details, we introduce a small example to describe the idea behind the algorithm internals.

Suppose we want to find the equivalence classes in layer l, with $1 \leq l \leq k$. For the sake of readability, assume $\mathcal{S}'_{l-1} = \{C_1, C_2\}$—layer $l-1$ has been previously split in two equivalence classes. Pick one element s from $\mathcal{S}_l$. Roughly speaking, s is the representative of the *next* equivalence class we will create. We want to find all the elements $s' \in \mathcal{S}_l \setminus \{s\}$ such that $s\mathcal{R}s'$. Using Definition 9.4, s and s' are equivalent according to $\mathcal{R}$ if and only if:

- $b_l(s)$ is proportional to $b_l(s')$
- for both C_1 and C_2—the two equivalence classes identified in $\mathcal{S}'_{l-1}$—the sum of the weights from $C_1(C_2)$ to s is proportional to the sum of the weights from $C_1(C_2)$ to s'.

To identify all these s', we start by noticing that $\rho_l(s)$ can be set to 1. Exploiting this fact, we can compute $\rho_l(s')$—from Definition 9.4—as $b^l_s/b^l_{s'}$. Using such value, what is left to check is whether the weights from C_i to s are proportional to the weights from C_i to s'. If this condition holds, then s and s' are equivalent. Otherwise, they are not. Once s is checked against all the elements inside $\mathcal{S}_l \setminus \{s\}$, the equivalence class $[s]$ is complete. We iterate the procedure starting with the choice of s until all elements of $\mathcal{S}_l$ have been put in some equivalence class.

This very idea has been exploited to devise Algorithm 14. The for loop at lines 3-8 handles the first layer of the Neural Network. Each element is treated as a singleton equivalence class.

For loop 9-35 takes care of splitting all the other layers, one per time. Let $\mathcal{S}_l$ be the current layer. It is split completely inside for loop 11-30. We denote with C the current equivalence class, which at the beginning contains just s—notice that any strategy to pick s from $\mathcal{S}_l$ at line 12 can be adopted.

The for loop 18-28 is devoted to find all the elements left in $\mathcal{S}_l$ that are equivalent to s. This goal is accomplished by first iterating through all elements $s' \in \mathcal{S}_l$. For each one of them, we scan all the equivalence classes S' from the layer $l-1$—For loop 20-24. At line 21 we check if the sum of the weights from S' to s—stored in $O(S', s)$—is proportional to the sum of the weights from S' to s'—stored in $O(S', s')$. If there exists one class S' such that this condition does not hold, then s and s' are not equivalent, and we move to the next s'. If no such S' is found, then s' is added to C, and we remove it from $\mathcal{S}_l$.

Variable O is updated in lines 31-34. On the other hand, the variable $\tilde{W}$ is used to store the weighted sum of the rates, so that we can correctly populate W'_l at each iteration. The function returns the lumped neural network $\mathcal{N} \setminus \mathcal{R}$.

We now investigate the complexity of the proposed algorithm.

To prove such We focus on one specific iteration of loop 9-35.

The complexity of while loop 11-30 is composed of: the number of iterations performed and the complexity of each one of them. The while loop is performed $\mathcal{O}(|\mathcal{S}_l|)$ times, with the worst case reached when all the equivalence classes are singleton. For what concerns the complexity of each iteration, there are two for loops. The former, lines 15-17, has a $\mathcal{O}(|\mathcal{S}'_{l-1}|)$ running time. The latter, lines 18 - 28, has complexity $\mathcal{O}(|\mathcal{S}_l|) * \mathcal{O}(|\mathcal{S}'_{l-1}|) = \mathcal{O}(|\mathcal{S}_l| * |\mathcal{S}'_{l-1}|)$. Hence, the overall complexity of one iteration of while loop 11-30 is $\mathcal{O}(|\mathcal{S}'_{l-1}|) + \mathcal{O}(|\mathcal{S}_l| * |\mathcal{S}'_{l-1}|) = \mathcal{O}(|\mathcal{S}_l| * |\mathcal{S}'_{l-1}|)$.

Recalling that the while loop has to be performed $\mathcal{O}(|\mathcal{S}_l|)$ times, we obtain $\mathcal{O}(|\mathcal{S}_l|^2 * |\mathcal{S}'_{l-1}|)$.

In [151], we tested our proposal on artificially synthesized Neural Networks with different architectures—Fully Connected and Convolutional NNs. Results showed that even introducing a proportionality relation is not enough to completely remove the problems due to the lack of control over the values weights can assume. In particular, as previously stated, our procedure performs better when weights share the same sign. This constraint is not realistic in a real life scenario. This being the case, we adopted to potential turnarounds. On the hand, introducing a factor of *approximation* when computing the proportionality between nodes. On the other hand, adopting a *pipeline* to reduce neural network. This approach will be investigated in this thesis when dealing with quantization.

The former solution we proposed lead to the introduction of a quasi-proportionality relation between nodes. Two neurons are defined quasi-proportionally lumpable if their proportional sums are close enough. Formally speaking, we substituted equation 9.2 with:

$$\rho_\ell(s_1)b^\ell_{s_1} = \rho_\ell(s_2)b^\ell_{s_2} + \epsilon_{S'}$$

and the same was done for equation 9.3:

$$\rho_\ell(s_1)b^\ell_{s_1} = \rho_\ell(s_2)b^\ell_{s_2} + \epsilon_{S'}$$

Under such new conditions, the amount of lumped neurons—in the aforementioned tests—increased. Therefore, the introduction of a fresh degree of freedom—represented by ϵ—seems to increase the algorithm effectiveness.

It remains an open question whether there exists settings in which neural networks' weights are actually proportional. However, in the end of this chapter, we will propose a pipeline of pruning techniques to achieve better reduction performances.

9.4 Quantization

All the methods we mentioned in this chapter tackled the problem of reducing the size of a neural network by efficiently reducing the number of its components—weights or neurons. Nevertheless, other approaches may be worth to be exploited. For example, weights in a network are usually left untamed and free to take any value allowed by the precision of the computer they are trained by. Therefore, when talking about *size* of a Neural Network, the space it requires to be stored is a factor that should not be neglected.

Hence, techniques that try to reduce the memory required to maintain a Neural Network fall under the name of *quantization* techniques. Roughly speaking this approach can be described as follows:

- Define a maximum precision you want to allow. It can be defined in terms of bit used to store weights;
- Select a strategy to modify each weight so that it respects the new memory constraints;
- Apply an algorithm that edits all the weights in the network.

For the first step, the most adopted approach is to limit the number of bits used for storing weights and biases. Nevertheless, two possible paths can be followed. On the one hand, one can just fix the number of bits and let the weight take any possible value. On the other hand, one can fix the number of bits and impose that the weights can only assume values coming from a fixed set—such set is usually called *alphabet*.

The former technique is in general less interesting. Given some neural network where weights have no constraints, one can obtain a quantized one just by truncating/rounding all the weights.

On the other hand, the latter, has some interesting implications.

Given an alphabet Σ, first we must define a strategy to substitute each weight with one coming from Σ. To solve this issue, many paths can be taken. The first and more

natural way is to change all the weights with the closest counterpart in the alphabet. In some sense, we locally minimize the distance between the old and the new weight. Figure 9.5 is an example of this technique. We have a three value alphabet $\Sigma = [-3, 0, 0.5]$ and each weight is replaced by its closest one from Σ. Specifically, we replace 1 with 0.5 and -1 with 0.

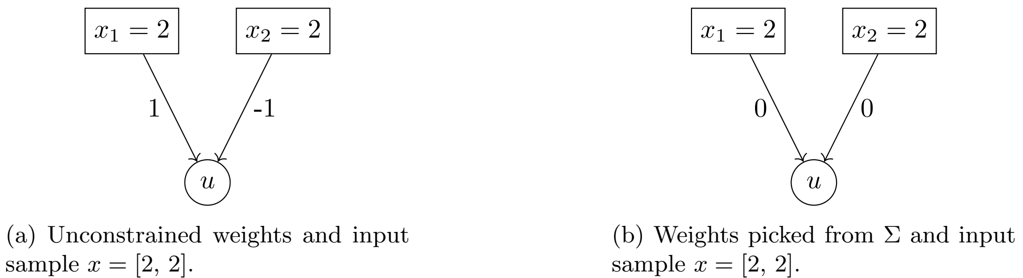
(a) Unconstrained weights

(b) Weights picked from Σ.

Figure 9.5: A neural network before (left) and after (right) the quantization with alphabet $\Sigma = [-3, 0, 0.5]$. Each weight is replaced with the closest one in Σ.

On the other hand, one can think in terms of neurons. Let u be the neuron we are analysing, let $x = [x_1, x_2, x_3]$ be an input sample and let W be the weights of u's incoming edges. The neuron u will receive in input a value $\mathbf{x}$ which depends on from x and from W. A strategy to quantize neuron u is to compute a new set of weights W'—where all of its elements come from the alphabet—in such a way that the new value that u receive with respect to x, W' is as close as possible to $\mathbf{x}$.

Example 9.1. We depicted an example in Figure 9.6.



(a) Unconstrained weights and input sample $x = [2, 2]$.

(b) Weights picked from Σ and input sample $x = [2, 2]$.

Figure 9.6: A neural network before (left) and after (right) the quantization with alphabet $\Sigma = [-3, 0, 0.5]$. Each weight is replaced so that the input to the neuron is not too perturbed.

The left network is the same as in Figure 9.5 but in this case we also take into account the input sample $x = [2, 2]$. For the leftmost network, the input to neuron u is

$$\mathbf{x} = -1 \cdot 2 + 2 \cdot 1 = 0$$

We now want to replace the weights $(-1, 1)$ with two weights from the alphabet $\Sigma = [-3, 0, 0.5]$ so that the new value of $\mathbf{x}$ is as close as possible to $-1 \cdot 2 + 2 \cdot 1 = 0$.

Hence, it is easy to see that the best choice is the pair $(0, 0)$ since the new input to the neuron would be:

$$\mathbf{x} = -1 \cdot 2 + 2 \cdot 1 = 0$$

This example is useful since it also shows that the two quantization techniques we introduced are different. If we provide input $(2, 2)$ to the network in Figure 9.5b, we obtain a new input $\mathbf{x}$ that is:

$$\mathbf{x} = 2 \cdot 0.5 + 2 \cdot 0 = 0.5$$

which is not the closest possible value we can obtain.

As for the conclusions we have drawn at the end of Example 9.1, the two quantization techniques we just introduced are different. Nevertheless, one key aspect must not be neglected. The former—Figure 9.5—approach does not depend on from the input. Hence, the substitution can be done in $\mathcal{O}(log(\Sigma))$ time using a sorted alphabet and applying a binary search looking for the weight w that has to be quantized. Thus, quantizing the whole network reduces to the application of a binary search procedure for each weight in the NN. Let w be such quantity and let $m = |\Sigma|$ be the length of the alphabet. The overall quantization algorithm has time complexity $\mathcal{O}(w \log m)$. Such running time becomes linear in the number of weights if $|m \in \mathcal{O}(1)|$.

On the other hand the latter—Figure 9.6—depends on from the input set and has to solve some minimization problem, too. Therefore, it guarantees that the quantization process will try to minimize the error introduced, but it is clearly much more computationally expensive. Exactly as in the pruning approach, a trade-off between accuracy and computation time is at stand.

However, we now want to present a possible new solution to create a pipeline that exploits both quantization and pruning. In 9.3 we described a proportional lumping adapted to the NN case. The amount of *lumping* that could be performed if we could ensure that all the weights are proportional to each other could be considerably high. In [114] authors defined a quantization technique where all the elements of the alphabet have this peculiar property of being proportional.

Hence, a pipeline composed of the following steps could be worth investigating. Let $\mathcal{N}$ be a neural network. In the first phase, we quantize $\mathcal{N}$. The result will be a network $\mathcal{N}'$ whose weights are proportional. Subsequently, use $\mathcal{N}'$ as input for the lumping-based pruning algorithm.

The proportionality property that the quantization ensures could be a favourable setting for the lumping algorithm.

Conclusions on Neural Networks Reductions

In this chapter we have introduced and briefly explained different approaches to reduce the size of neural networks. We investigated both techniques that reduce the number of nodes—pruning and lumping—and approaches that aim at making the impact of weights in terms of space lower—quantization and weight sharing. Weight sharing is the most *collateral* one, since it is applicable only in models with filters. On the other hand, we described pruning and quantization as greedy and non-exact algorithms. The

former aims at removing nodes with *local* choices to obtain a *global* reward. However, it does not ensure that the accuracy remains unchanged. The latter—quantization—tackles space issues by switching its focusing towards weights. It assumes that just a small and pre-set amount of bits are available to encode weights. The algorithm must turn every weight into a *quantized* one trying to minimize the accuracy drop.

Finally, we introduced also an exact algorithm based on lumping. Borrowing the notion of bisimilarity from graph theory, the algorithm we presented is able to aggregate nodes in a Neural Network while preserving its accuracy.

An interesting question to pose is about the relation between classical and quantum neural networks. Without delving too many into details, quantum neural networks are circuits with parametrized gates. Such parameters are usually angles to be fed to rotation operators. What is the quantum counterpart of Neural Network pruning? One answer may be the following. Assume we are provided with a fully trained quantum neural networks. This being the case, such network is nothing more than a circuit. Therefore, it can be described by one single unitary matrix. What if synthesizing such unitary is actually a pruning process? Roughly speaking, what if relations between rotation angles pop out during the training process? We may be able to obtain smaller circuits through synthesis: it may be interpreted as pruning.

10
Quantum Circuit Synthesis

Finally, we switch our focus to the investigation of quantum circuits synthesis tackled by means of graph theory. Firstly, we will introduce the reader to the concept of circuit synthesis, by distinguishing between synthesis and re-synthesis. Subsequently, motivated by the high computational complexity of the problem, we will describe small *sub-tasks* that pop out when tackling circuits decomposition procedures. The first subtask we will investigate is the T-gate minimization. The same will be done for CNOT gates, where we will also describe and Answer Set Programming (ASP) encoding to solve such problem.

When we refer to *circuit synthesis* in quantum computing, the aim is to decompose a unitary transformation using a fixed universal set—*a base*—of transformations. Therefore, a **circuit synthesis algorithm** over an universal set S is an algorithm such that:

Input: A unitary U of size $2^n \times 2^n$

Output: A circuit over S that implements U

In the Clifford+t settings, it can be proved [129] that circuit synthesis can be achieved within a polynomial time frame, specifically in $poly(N)$ time, where $N = 2^n$ denotes the input size. Frequently, our objective is to synthesize a circuit while adhering to specific conditions. For instance, we may aim to construct a circuit that minimizes a particular resource. The resources to minimize can vary; it might entail minimizing the number of qubits, reducing the total gate count, or even targeting the minimization of a specific gate that poses implementation challenges. Irrespective of the resource we seek to optimize, this problem is referred to as **resource-optimal circuit synthesis**. In this context, the computational complexity may not remain polynomial and instead hinges on the specific instance or resource constraint under consideration.

It is readily apparent that the complexity involved in circuit synthesis cannot be less than $O(2^n)$, which is the complexity required for reading the input unitary. Often, this level of complexity can be overly resource-intensive. Therefore, an alternative class of algorithms, known as **circuit re-synthesis**, is often employed. These algorithms

have the objective of synthesizing a circuit while optimizing a specific resource, much like resource-optimal algorithms. However, they take as input an existing circuit that already implements U. By employing this category of algorithms, we can aspire to achieve a complexity lower than polynomial in 2^n—most of the re-synthesis algorithms do not take into account the complexity of generating the first decomposition of U.

In spite of their higher complexity compared to re-synthesis algorithms, it is important to study optimal synthesis algorithms. They give optimal solutions and thus can be used for assessing re-synthesis algorithms, for example, how close are their output to an optimal one. From a theoretical point of view, designing synthesis algorithms throws light on the complexity of a problem, which is usually harder than their relaxed re-synthesis counterparts. They can also be useful for re-synthesis algorithms, which usually require a circuit description to begin with. These kinds of algorithms—synthesis one—can be further subdivided according to the type of solution they provide.

- **Exact synthesis algorithms** that given as input a unitary U, produce a circuit that implements U' which is equal to U up to a global phase. Focusing on the Clifford+T basis—that we will introduce soon—authors characterized the set of unitaries that can be implemented exactly using such basis in [78, 108].

- **Approximated synthesis algorithm** that given as input a unitary U, produce a circuit that implements U' which is *close* to U according to some distance notion.

Before delving into the description of the most known circuit synthesis algorithm related to graphs, we must set a target universal set. In fact, it is natural to think that when a unitary in synthesized, we must exploit the properties of the basic transformations to obtain faster algorithms. From now on, we will denote with *universal set of gates* $\mathcal{B}$ the so-called *Clifford+T* base:

$$H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad P = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \qquad T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix}$$

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{10.1}$$

This choice is especially due to the particular properties that such base shows related to topics like compilation, error correction, quantum simulation [81, 40, 138, 80, 83].

Before moving to the next section, we would like to stress the fact that *Clifford+T* is not the only universal set of gates for quantum computation. For example, the following theorem characterizes another set that enjoys the property of being universal.

Theorem 32. *The set composed by all single-qubit unitary gates and* C-NOT *is universal for quantum computations [133, Section 4.5].*

It is important to note that while *Clifford+T* is universal for approximate synthesis, the set mentioned in Theorem 32 is universal for exact approaches.

10.1 T gate optimality

It is worth noticing that circuits made only of Clifford gates—CNOT, H, P—can be efficiently simulated on classical computers[81, 1]. Adding the T-gate to the recipe is the only way to achieve universal Quantum Computation as proven in [55, 133]. However, achieving universality comes at a cost. Consider a circuit that contains k T gates: its simulation in a classical computer turns out to be exponential in t [100, 39]. Another aspect that should be taken into account is the cost of fault-tolerantly implementing the T gate. With *fault-tolerant* implementation we address the task of adding redundancy into real-world circuits so that each gate's behaviour is as close as possible to its theoretical one. For what concerns the Clifford+T basis, the cost of fault tolerantly implementing the T gate exceeds the cost of Clifford group gates by as much as a factor of hundred or more [4, 68, 11]. Therefore, it is natural to think about synthesis—or, analogously re-synthesis—algorithms that aim at minimizing the number of T gates in a circuit. The metrics that are mostly taken as a target for minimization are T-count and T-depth. We start from the latter, which is defined as the number of T gates that cannot be performed in parallel [10]. On the other hand, T-count is defined as follows.

Definition 10.1 (T-count)**.** Let $U \in \mathbb{C}^{N \times N}$ be a unitary matrix. Its T-count $\mathcal{T}(U)$ is the minimum number of T gates that are required to implement U—up to a global phase.

Theoretically speaking, $\mathcal{T}(U)$ is the minimum k such that:

$$e^{i\phi}U = C_k T_k C_{k-1} T_{k-1} \dots C_1 T_1 C_0$$

where each C_i is a set of Clifford group gates while each T_i is a T gate. It is important to clarify the difference between the T-count of a *circuit* and the T-count of a unitary. On the one hand, the T-count of a *circuit* is the number of T gate in that specific circuit. On the other hand, the T-count of a unitary is the quantity $\mathcal{T}(U)$ which is completely circuit *independent*. A circuit implementing U is called T-count-optimal—for U—if the number of T gates in the circuit is exactly $\mathcal{T}(U)$. This implies that among all the circuits that can implement U, a T-count-optimal one has the minimum number of T gates.

The **COUNT-T** problem is defined exactly as the task of computing $\mathcal{T}(U)$ given a unitary U. Notice that the *decision* version of this problem can be obtained by introducing an integer k and the problem becomes:

Definition 10.2. Given a unitary $U \in \mathbb{C}^{N \times N}$ and a number $k \in \mathbb{N}$, compute whether $\mathcal{T}(U) \leq k$

It has been shown in [79] that an algorithm capable of computing $\mathcal{T}(U)$ can be converted into an algorithm that computes a T-count-optimal circuit for U with overhead polynomial in $\mathcal{T}(U)$ and the dimension of U.

A preliminary survey of the literature pertaining to the T-count computation is presented prior to the subsequent discussion on the minimisation of CNOT gates. A theoretical foundation of the complexity of synthesizing T-count-optimal circuits has been given in [12] and further investigated in [165]. In [78], authors proposed an algorithm for exactly synthesizing unitaries over the Clifford+T base. Such work was then extended by obtaining a superexponentially faster version in [108].

T-count optimization has been tackled by mean of a *meet-in-the-middle* technique in [10]. This particular approach was extended with a nested binary search/meet-in-the-middle later on in [129]. The technique proposed in [10] had also been adopted to prove the minimality of some 3 qubit circuits. In fact, notice that since no closed formula exists to compute $\mathcal{T}(U)$, the only way to prove that a circuit is T-count-optimal is by showing that no circuit with less T gate can be obtained. Hence, when dealing with these kinds of minimization problem it is hard to retrieve the *ground-truth* values for the unitaries we are tackling. For example, the proof that the 4-qubit 1-bit full adder has optimal T-count 7 has been given in [59]. In such work authors adopted a parallel framework built upon deterministic walk to solve the T-count-optimal synthesis problem.

All the aforementioned papers tackled the T gate problem against any unitary U. However, a lot of work has been also when dealing with single qubit circuits like in [109, 157, 139].

10.2 CNOT minimization

The T-gate is not the only Clifford+T gate that has been subjected to intensive studies when minimization is concerned. CNOT has received a lot of attention, too. Such gate is the only one acting on two qubits coming from the Clifford+T basis—In Equation 10.1, its matrix is a 4×4— and its graphical counterpart is depicted in Figure 10.1.

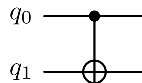


Figure 10.1: CNOT gate in the circuit settings. The top qubit (q_0) is the control, q_1 is the target.

The two qubits on which a CNOT acts are usually referred to as *control* and *target*. Roughly speaking, the effect of a CNOT is to

Flip the target qubit if and only if the control qubit is one.

Formally, the effect of a CNOT on a generic state $|\psi\rangle = |a\rangle\,|b\rangle$ is as follows:

$$\text{CNOT}\,|a\rangle\,|b\rangle = |a\rangle\,|a \oplus b\rangle$$

where with $\oplus$ we refer to the sum modulo 2. CNOT can be interpreted as a *conditional gate*. It changes the target qubit if and only if some condition is met on the value of the control. Another example of a gate that acts in the same way is TOFFOLI's, which in turn takes as input three qubits. The third one is flipped if and only if both the other two are one. Composing CNOT and TOFFOLI allows obtaining circuits that flip the target qubit if and only if the controls hold some specific truth value.

Moreover, CNOT gate also plays a pivotal role in Quantum Computation as it is a key component to generate *entanglement*.

Entanglement is a unique phenomenon in quantum mechanics that appears to challenge the principles of Einstein's general relativity. Imagine two particles that become

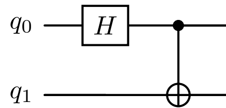


Figure 10.2: The quantum circuit to generate the Bell-pair β_{00}

entangled and are then separated, with one remaining in our galaxy and the other placed in a distant galaxy many light years away. When we measure the particle in our galaxy, it instantly affects the state of the other particle, despite the vast distance between them. This is puzzling because the "information transfer" between the particles seems to occur faster than the speed of light. In quantum mechanics and quantum computation, a state is considered entangled if it cannot be separated into components involving only individual qubits.

In the realm of quantum computation, the simplest examples of entangled states is represented by *Bell pairs* [23]. Bell introduced these four states as a counterexample to the claim stated by Einstein, Podolsky, and Rosen [63]. In [63], the authors claimed that any form of entanglement could be described in terms of something that had yet to be added to the quantum mechanical description of reality. They formalized it as an *hidden variable* that if added to quantum mechanics could explain the *spooky effect*—as Feynmann would call it—of quantum entanglement. Bell pairs was used to show that they were wrong, and that quantum mechanics is actually complete and does not need any further addiction. The four Bell pairs are formalized as follows:

$$
\begin{aligned}
|\beta_0 0\rangle &= \frac{|00\rangle + |11\rangle}{\sqrt{2}} \\
|\beta_0 1\rangle &= \frac{|00\rangle - |11\rangle}{\sqrt{2}} \\
|\beta_1 0\rangle &= \frac{|10\rangle + |01\rangle}{\sqrt{2}} \\
|\beta_1 1\rangle &= -\frac{|10\rangle + |01\rangle}{\sqrt{2}}
\end{aligned}
$$

where the first state $|\beta_{00}\rangle$ is obtained using the circuit depicted in Figure 10.2. All the other pairs are obtained by tweaking a bit the same circuit. In particular:

$$
\begin{aligned}
|\beta_0 1\rangle &= (X \otimes I)\, \beta_0 0 \\
|\beta_1 0\rangle &= (Z \otimes I)\, \beta_0 0 \\
|\beta_1 1\rangle &= (iY \otimes I)\, \beta_0 0
\end{aligned}
$$

A less case-per-case definition of the Bell pairs is the following:

$$|\beta_{x,y}\rangle := \frac{|0y\rangle + (-1)^x |0\overline{y}\rangle}{\sqrt{2}} \qquad \text{with}\, x, y \in \{0, 1\} \tag{10.2}$$

These states have the peculiar property of being not only entangled, but *maximally entangled*. Since CNOT is part of the very small circuit that generates Bell pairs, it is crucial for Quantum computation, since entanglement is at the basis of algorithms and protocols like Quantum Superdense Coding and Quantum Teleportation [133].

However, in real-world implementation of gate-based Quantum Computers apply a CNOT gate must be done carefully. This because physical implementation of two qubits gates we are provided with are intrinsically faulty and can produce mixed states even if the input state was originally pure. Therefore, it is natural to think about reducing the number of two qubits gates—CNOTs—we adopt when we rewrite a unitary in terms of smaller terms. The **CNOT-count problem** arises here. Exactly as its T gate counterpart, the CNOT-count problem asks to find the implementation of a unitary matrix U with the minimum number of CNOT gates.

This kind of problem is not important just to minimize the noise inside circuits, but also for its effect on stabilizer circuits. These particular circuits are the ones that can be produced just by using gates inside the Clifford group. As we stated before, they cannot achieve universality until the T gate is added to the equation. However, they have been intensively studied and a lot of results have been obtained. In particular, Gottesman-Knill Theorem 33 lies at the core of simulation of quantum computers.

Theorem 33. *Suppose a quantum computation is performed which involves only the following elements:*

- *state preparations in the computational basis.*
- *Hadamard gates, phase gates, controlled- gates, Pauli gates, and measurements of observables in the Pauli group (which includes measurement in the computational basis as a special case).*
- *possibility of classical control conditioned on the outcome of such measurements.*

Such a computation may be efficiently simulated on a classical computer in polynomial time.

A particular result about stabilizer circuits comes from a subsequent paper [1] in which authors provided a *normal form* for stabilizer circuits.

Theorem 34. *Let $\mathcal{C}$ be a circuit consisting of CNOT, Hadamard and phase gates—a stabilizer. There exists an equivalent circuit with the following normal form:*

$$H - C - P - C - P - C - H - P - C - P - C$$

where an H, C, and P indicates a layer of Hadamard, CNOT, and Phase gates, respectively.

Layers of Hadamard and Phase gates have length exactly one—being single qubit gates, they can all be parallelized. On the other hand, the whole length of the circuit

depends on the length of the CNOT gate layers. Therefore, it is crucial to produce algorithms that can minimize the number of such gates.

Circuits composed only of CNOT gates are usually called either *CNOT circuits* or *Linear reversible circuits*. Since the effect of a CNOT gate—looking at its matrix description—is intrinsically classical, CNOT circuits can be represented using classical tools. In particular, any linear reversible circuit on n qubits can be completely described by an $n \times n$ invertible[1] boolean matrix M. As the reader may notice, we converted a problem on n qubits—hence, with a representation that has dimension $2^n \times 2^n$—to a problem on a $n \times n$ matrix. Since this shift in dimension, the definition of CNOT we gave in 10.1 is not coherent.

Definition 10.3. Let M be an $n \times n$ matrix representing a linear reversible circuit. The effect of applying a CNOT with i-th row as a control and the j-th row as a target is to multiply M by a matrix M' defined as follows. M' is the identity matrix with an additional off-diagonal 1 element in position i, j.

This particular definition of the effect of a CNOT gate, allows us to treat it as an elementary row operation.

Lemma 10.1. *The action of a CNOT gate, whose control is on the i-th row and target is in the j-th row, is an elementary row operation on a $n \times n$, which adds the i-th row to the j-th row.*

Before introducing the CNOT minimization problem, we introduce the following theorem:

Theorem 35 ([16])**.** *Any invertible matrix can be transformed into an identity matrix using elementary row and/or column operations. Thus, any invertible matrix can be decomposed as a product of elementary matrices.*

Therefore, any invertible matrix M—linear reversible circuit—can be decomposed into the identity matrix using only CNOT gates.

Definition 10.4. Let M be an invertible binary matrix. The CNOT minimization problem is the problem of finding the shortest sequence of k matrices $M_1, M_2, \ldots, M_k$ such that:

$$MM_1M_2 \ldots M_k = \mathbb{I}$$

where each M_i is the matrix of a CNOT gate.

Dually, it can also be defined as the problem of turning the identity matrix into M applying only CNOT gates.

Before moving to the discussion about topological constraints on the application of CNOTs, we briefly discuss the notion of *the shortest* circuits of CNOTs. CNOT gate is clearly a XOR between two rows of a matrix. In literature, usually 3 XOR metrics are adopted to measure the *complexity* of a invertible boolean matrix.

The simplest and naivest one is the d-XOR metric. First introduced in [107] it relates the matrix to its hamming weight—number of 1 entries.

[1] Notice that if the matrix is not invertible, then the circuit cannot be decomposed in terms of CNOT gates.

Definition 10.5 (d-XOR)**.** Let M be a boolean $n \times n$ matrix. Its d-XOR count is defined as $h(M) - n$ where $h(M)$ is M's hamming weight.

This measure is very intuitive, and its computational cost is linear in the size of the matrix. However, the d-XOR metric corresponds to the very naive implementation of a linear matrix. This may compute an intermediate value several times, thus resulting in an overestimation of the XOR operations needed.

Some kind of advancements to this definition have been made with the introduction of the g-XOR count or *general XOR* count. The version of XOR operation we adopted—CNOT—only takes into account two rows.

$$M_i \leftarrow M_i \oplus M_j$$

The i-th row becomes itself *plus* the j-th row. In the g-XOR metric, three rows i, j, k are taken into account:

$$M_k \leftarrow M_i \oplus M_j$$

Roughly speaking, the result of the XOR between rows is stored in a third (potentially different) row. The g-XOR metric is defined upon this operation.

Definition 10.6 (g-XOR)**.** Let M be a boolean $n \times n$ matrix. Its g-XOR count is the minimum number of operations of the form $M_k \leftarrow M_i \oplus M_j$ to turn M into the identity.

This metric has been adopted in many heuristics like [137, 38]. Finding the optimal implementation of a linear matrix under g-XOR metric has been proved to be an NP-hard problem in [38].

Finally, the metric we adopted in the definition of the CNOT minimization problem (where only two rows are affected by the CNOT) is usually referred to s-XOR.

Comparing the different metrics, what we can for sure state is that d-XOR count of a matrix is always larger than or equal to its g-XOR count. Similarly, the s-XOR count is larger than or equal to the g-XOR count, as we can easily transform the in-place XOR instruction $M_i \leftarrow M_i \oplus M_j$ to an out-of-place instruction $M_k \leftarrow M_i \oplus M_j$ by introducing a new variable M_k. It was conjectured that the s-Xor count is smaller than or equal to the d-XOR count in [98]. However, Lukas disproved this conjecture by presenting a counterexample in [111]. Although, the g-XOR count is the smallest one among the three metrics theoretically, it is not always easy to find the best implementations under the g-XOR metric. Properties and features of s-XOR metric with respect to other two have been extensively studied in [172].

We focus on the s-XOR metric as well—the CNOT minimization problem. When tackling the CNOT minimization problem, algorithms can be divided into two major categories: with and without topological constraints. Quantum Computers architectures rely on physically manufactured qubits which are connected to one another according to some topology—there is no current Quantum Computer that uses an all-to-all connection. For a real-world quantum computer topology see Figure 10.3.

No particular problem arises when we apply a single qubit gate. However, when applying two qubits gates like CNOT, the two qubits involved must be *neighbours* according to the topology before physically applying the gate.

For example, assume that the underlying topology we are dealing with is the one depicted in Figure 10.3. If we want to apply a CNOT between qubit 0 and qubit 1—the

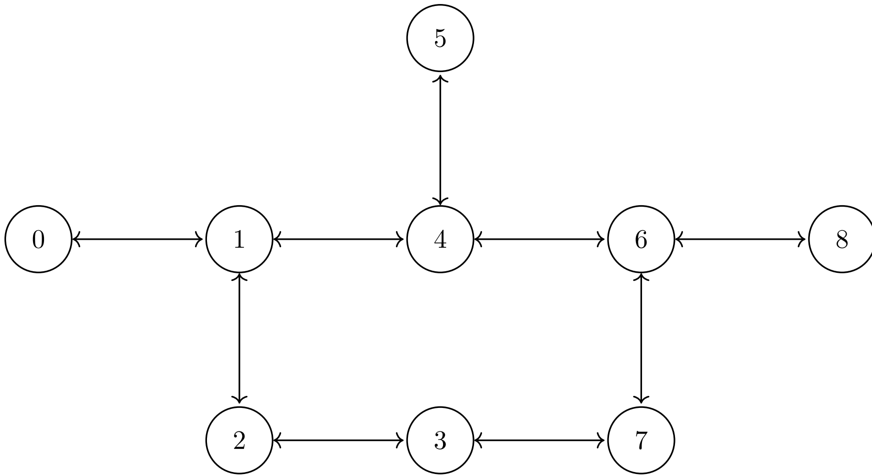


Figure 10.3: Guadalupe Quantum Computer topology reduced to 8-qubit.

number of the qubit is the label on the nodes—there is no particular attention we must pay since they are connected by an edge. On the other hand, if the control qubit of a CNOT is qubit 6 and the target is qubit 1, such gate is not directly applicable. To perform such gate we must apply a *swap gate*—that swaps the value of two qubits—between qubit 6 and qubit 4. In this way, the *logical* mapping of the qubits becomes as in Figure 10.4. We used a color coding to emphasize the swap between the two qubits.

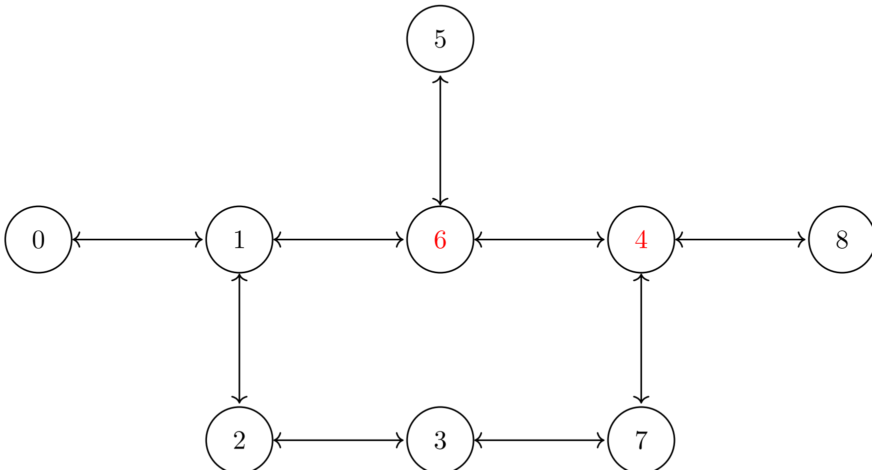


Figure 10.4: Guadalupe Quantum Computer topology reduced to 8-qubit where qubits 4 and 6 have been swapped.

From this particular settings, the two major CNOT synthesis algorithm arise. On one side, algorithms that do not take into account the topology of the quantum computer. Therefore, any CNOT can be applied in any moment without any swap. On the other

side, algorithms that do take into account the topology and require two qubits to be *close*—connected by an edge—before a CNOT between the two of them can be applied.

We start with the former category. Namely, the one where any CNOT can be applied with no restrictions. First, we want to clarify that as stated in [44]:

The computational complexity of the CNOT-optimal circuit synthesis (with no constraints on the topology) is yet to be established

One of the most important results concerning the optimal synthesis of circuits made only of CNOTs comes from [143]. They proved that each n-qubit CNOT circuit can be synthesized with $\mathcal{O}(n^2/\log n)$ CNOT gates. Moreover, this bound is asymptotically tight.

The literature is somehow split when dealing with this particular problem. A lot of theoretical work has been done to completely formalize all the properties that CNOT gates enjoy [96, 147, 160, 9, 22]. All of this effort has then been exploited to devise algorithms that actually produce the CNOT minimal versions of CNOT circuits. Due to the lack of answers on the actual complexity of this problem, these techniques may be either exact [71, 121] or approximate [173].

For what concerns the results about the topological constrained version of the CNOT-minimization problem, in [99] it has been proved to be NP-complete. Therefore, plenty of algorithms have been proposed, all of them giving different approaches to solve this problem [54, 166, 76, 84, 85, 171, 86, 162, 51, 174, 18].

Despite all these proposals, we now focus on the work presented in [145]. The algorithm devised solves the CNOT minimization problem by reducing it to a graph-related version [2] which is then tackled in terms of Answer Set Programming.

10.2.1 An ASP approach

In [145], we started from a slightly tweaked version of the CNOT minimization problem. In particular, the circuits we handle are made of CNOT and T gates (same problem tackled in [121]), and we will minimize the number of CNOT gates. To do so, we first turn a *Quantum* problem, into a classical one using the following definition from [10]:

Definition 10.7 (phase polynomial representation)**.** The action of a {CNOT, T} circuit on the initial state $|x_1, x_2, \cdots x_n\rangle$ has the form:

$$|x_1, x_2, \cdots x_n\rangle \mapsto e^{i\frac{\pi}{4}p(x_1,x_2,\cdots,x_n)}\,|g(x_1, x_2, \cdots x_n)\rangle$$

with $p(x_1, x_2, \cdots, x_n)$ defined as:

$$p(x_1, x_2, \cdots, x_n) = \sum_{i=1}^{k}(c_i \bmod 8) f_i(x_1, x_2, \cdots x_n)$$

where $g : \mathbb{B}^n \to \mathbb{B}^n$ is a linear reversible function and p is a linear combination of linear boolean functions $f_i : \mathbb{B}^n \to \mathbb{B}$.

The coefficients c_i (reduced modulo 8) measure the number of $\pi/4$ rotations applied to

[2] Which is the reason why we focused on it

each f_i.
The number k represents the number of T gates in the circuit.

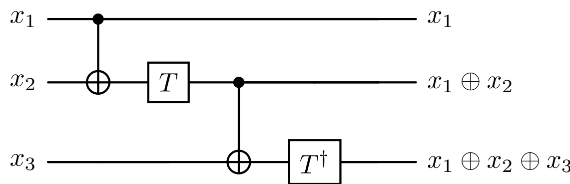
Each circuit has its phase polynomial representation, uniquely defined by

$$g, f_i, c_i \text{ for } i = 1, 2, \dots k.$$

More than one {CNOT, T} circuit can share the same phase polynomial representation.

Since g is a linear reversible function with n inputs and n outputs, it can be written as a $n \times n$ boolean matrix G. On the other hand, each f_i can be expressed as a boolean row vector F_i.

Example 10.1. Consider the following circuit:



Its phase polynomial representation is the following:

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} \qquad F_1 = \begin{pmatrix} 1 & 1 & 0 \end{pmatrix} \qquad F_2 = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$$

Since the number of T gates in the input circuit is supposed to be optimal, the problem we want to solve is the following:

Definition 10.8.

- **INPUT**: $G, S = (V_S, E_S), F_1, F_2, \cdots F_k$
- **OUTPUT**: a sequence of CNOT gates to be applied such that the final behaviour of the circuit is the one described by G.

The constraints we must fulfil are the following:

- We can only apply CNOT gates that are *legal* according to S.
- For each F_i with $i \in \{1, 2, \cdots, k\}$, there must exist a moment during the computation in which a row of G is exactly F_i.

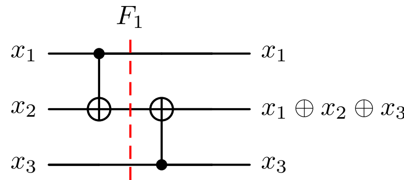
The two constraints allow us to avoid caring about the T gates. We just have to create the *correct* slots for them to be applied, but nothing more.

Example 10.2. Let G and F_1 be defined as follows:

$$G = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \qquad F_1 = (1\,1\,0)$$

We must find a circuit such that x_1 and x_3 are unchanged while x_2 becomes the XOR of all the three variables. Moreover, it must be true that at some point one of the variables must be the XOR between x_1 and x_2 to match the constraint on F_1.

The following circuit, made only of CNOT gates, solves the problem.



We pointed out with the red line the moment in which the constraint on F_1 is satisfied.

For what concerns the topological constraint, we adopted a graph based encoding. Let $S = (V_S, E_S)$ be a directed graph. The set of nodes is $V_S = \{1, 2, \cdots, n\}$, where n is the number of qubit. The set of edges E_S is such that a CNOT(x_i, x_j) can be applied if and only if $(i, j) \in E_S$.

In Figure 10.5, we show an example of a topology turned into a graph. Only two CNOT gates are allowed: with control 1 and target 2 or with control 2 and target 3.

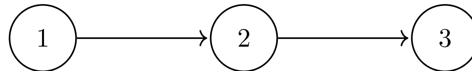


Figure 10.5: An example of a topology constraint depicted as a graph.

With this additional constraint, it is straightforward to see that the problem may become unsolvable in some cases. For example, if we need a CNOT between qubit x_i and x_j, there must exist a path in G from i to j. If such path does not exist, the problem is unsolvable.

The model we propose is based on a Directed Acyclic Graph. The circuit from Example 10.1 (without the T gates) can be interpreted as the labelled graph $\mathcal{G} = (V, E)$ depicted in Figure 10.6.

The leaves of the DAG represent the problem variables. Hence, we will have n leaves labelled $x_1, x_2, \cdots, x_n$. A CNOT with control x_j and target x_i is represented by a node x_i with two incoming edges. One edge comes from the closer node labelled x_i. The other from the closer node labelled x_j. The generic structure of a node is depicted in Figure 10.7.

The DAG structure induces a *layering* of the nodes. Leafs are at layer 0. A node at layer j, can only have incoming edges from nodes in smaller layers. One layer can contain more than a single node. Nevertheless, any layer l can contain at most one node labelled x_i, for any $i \in \{1, 2, \cdots, n\}$.

We solve the problem introduced in Definition 10.8 by computing $\mathcal{G} = (V, E)$ starting from $G \in \{0, 1\}^{n \times n}$, a set of F_i, and some topological constraint $S = (V_S, E_S)$. The set of nodes V is always $\{x_1, x_2, \cdots x_n\}$. Our aim is to build the set E of minimum size l. First, it must hold that for each layer $t \le l$, each variable can be the target of at most

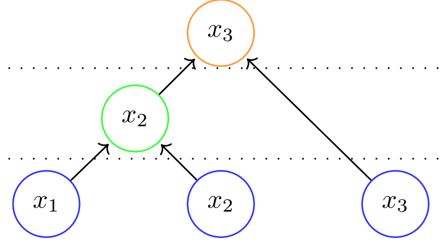


Figure 10.6: Blue nodes are in layer 0, the green node composes layer 1, and layer 2 is composed by the single orange node. The circuit induced by $\mathcal{G}$ applies CNOT(x_1, x_2) followed by CNOT(x_2, x_3).

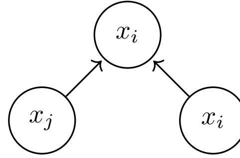


Figure 10.7: Generic DAG node describing CNOT(x_j, x_i).

one CNOT gate. After all the l steps, the following has to be true:

$$\text{VAL}_l(x_i) = G_i \qquad \forall i \in \{1, 2, \cdots, n\}$$

where VAL is defined as follows:

$$\begin{aligned} \text{VAL}_0(x_i) &= x_i && \forall i \in \{1, 2, \cdots, n\} \\ \text{VAL}_t(x_i) &= \text{VAL}_{t-1}(x_i) && \text{if } \nexists x_k \mid (x_k, x_i, t) \in E \wedge t > 0 \\ \text{VAL}_t(x_i) &= \text{VAL}_{t-1}(x_i) \oplus \text{VAL}_{t-1}(x_k) && \text{if } \exists x_k \mid (x_k, x_i, t) \in E \wedge t > 0 \end{aligned}$$

With the notation $x_j \in \text{VAL}_t(x_i)$, we mean that the XOR operation described by $\text{VAL}_t(x_i)$ contains x_j.

The above constraint was necessary to ensure that the final circuit respects the limitations imposed by G.

As far as F_i is concerned, the following must hold:

$$\forall F_i \; \exists t \leq l \; \exists x_j \mid \text{VAL}_t(x_j) = F_i$$

In the end, we must ensure that CNOT are applied only to a pair of qubits that are connected according to the input topology $S = (V_S, E_S)$. Hence, it must hold that if at layer $\hat{l}$ there is a CNOT(x_i, x_j), then $(i, j) \in E_S$.

Encoding in `clingo`

We now briefly describe how we encoded the aforementioned solution in the `clingo` solver.

The input has been encoded in a very simple way. We provide the model with n (the size of G) and k, the number of different F. Then, with a predicate of the form $\text{G}(i, j, b)$ we describe the matrix G, with the following rules:

$$G_{i,j} = 1 \leftrightarrow \text{G}(i, j, 1)$$
$$G_{i,j} = 0 \leftrightarrow \text{G}(i, j, 0)$$

where $i, j \in \{1, 2, \cdots, n\}$

The same relation holds between predicate $\text{F}(i, j, b)$ and F. The only difference is that the domain of variable i is $\{1, 2, \cdots, k\}$.

The variables we adopted to solve the problem have two types:

- NODE that describes leafs of the DAG. Given an $n \times n$ boolean matrix, we will have exactly n NODES with labels $1, 2, \cdots, n$. The mapping from nodes to the variables x_i is straightforward: node i describes the variable x_i—the variable that will have to match G_i.

- LAYER, which describes the layers of the DAG.

Inner nodes of the DAG are described by a predicate of the form $\text{XOR_NODE}(I, J, L)$ which has to be interpreted as follows: at layer L, there is a node labelled x_I which is the result of the XOR between x_I and x_J — $\text{CNOT}(x_j, x_i)$. Given a variable X and a layer L, we imposed that a single XOR_NODE with target X can exist at layer L.

For what concerns the VAL of a node, we introduced the predicate

$$\text{VALUE}(X, I, Y)$$

where X, Y are NODEs and I is a STEP.

$\text{VALUE}(X, I, Y)$ holds if and only if $x_Y \in \text{VAL}_I(x_X)$. We then encoded the update of VALUE using the recursive definition introduced above.

We used the predicate VALUE to check the final solution of the model at the l-th step. For all $i, j \in \{1, 2, \cdots, n\}$ the following must hold:

$$\text{G}(i, j, 1) \leftrightarrow \text{VALUE}(I, l, J)$$
$$\text{G}(i, j, 0) \leftrightarrow \neg\text{VALUE}(I, l, J)$$

Forcing the model to search for graphs that satisfy the total behaviour imposed by G.

Something similar has been done for the constraint on F, with the only difference that F_i can match the VAL of a NODE at any time. Hence, for all F_i there must exist a $t \leq l$ and a J such that for all $k \in \{1, 2, \cdots, n\}$:

$$\text{F}(i, k, 1) \leftrightarrow \text{VALUE}(J, t, K)$$
$$\text{F}(i, k, 0) \leftrightarrow \neg\text{VALUE}(J, t, K)$$

Handling the topological constraints induced by S is pretty straightforward. We encoded the edges of S with a binary predicate S.

$$\text{S}(x, y) \leftrightarrow (x, y) \in E_S$$

Using such predicate, we can enforce that XOR_NODE(X, Y) can hold if and only if S(x, y).

Conclusions on Quantum Circuits Synthesis

To conclude this chapter, we invite the reader to reason about a peculiar aspect of the quantum circuits compilation problem. All the papers we cited above try to solve either the T or CNOT minimization problem on any input unitary. However, another line of research tries to solve the same problems but for circuits that solve only one specific problem. For example, in [135] authors only tackled the problem of circuits compilation for the graph colouring problem. The same was done also for another graph related problem, namely the max cut [17].

In classical computation, the focus is typically on developing generic algorithms capable of optimally solving any instance of a problem with minimal computational cost. In contrast, quantum computing may necessitate a shift in perspective when addressing certain problems. It is possible that we can achieve optimal solutions for smaller subtasks, yet fail to solve the entire problem optimally. Nonetheless, such partial solutions may still be considered satisfactory. For instance, for researchers working exclusively with quantum solutions to graph colouring, having an optimal algorithm specifically for compiling graph colouring unitaries is more valuable than employing a suboptimal algorithm designed to compile any arbitrary unitary. Therefore, it may seem natural to address tasks that *intrinsically intractable* like circuit synthesis—the input alone has size $\Theta(2^n \times 2^n)$—in terms of local minima instead of searching for global optimum. One example supporting this idea is the CNOT synthesis algorithm adopted by Qiksit. It mainly relies on the procedure presented in [143], which is suboptimal, yet computationally light. However, the result it produces are on average acceptable—the circuits does not contain too many gates.

Conclusions on the role of Graphs as Reduction Structures

In the last part of this thesis, we examined how graphs can be used to reduce the complexity of various problem domains. We began by establishing a general problem-solving framework that takes an initial problem $\mathcal{P}$ and creates a graph-based representation $G_{\mathcal{P}}$. The graph $G_{\mathcal{P}}$ is then optimized to reduce the dimensions involved in the problem.

This framework was applied to address three key problems: Classical Automata Minimization, Neural Network Compression, and Minimal Quantum Circuit Synthesis.

In the context of automata minimization, we explored a graph colouring algorithm that incrementally derives the minimal representation of (Non-)Deterministic Automata. An open question remains concerning the observed complexity gap between top-down and bottom-up approaches to automata minimization.

Next, we shifted our focus to Neural Networks. We began with a brief overview of non-exact reduction techniques, such as pruning and quantization. The main focus of the Neural Network chapter is a lumpability-based algorithm that compresses the network without altering its accuracy. Although promising, this approach requires further investigation, especially in real-world applications.

Notice that these two chapters—about automata and neural networks—are focused on classical topics. However, the notions of bisimulation and lumpability may find a good playground in the quantum settings as well. For example, using the notion of Quantum Markov Chain (QMC) as a starting point, we may think about describing quantum computations in terms of QMC. To cope with the possible state explosion of these tools, bisimulations and lumpabilities may serve as fundamental notions.

Finally, we leveraged the adaptability of graphs to tackle CNOT-minimal circuit synthesis. First, we clarified the definitions of circuit synthesis, re-synthesis, and C-optimal circuits. Then, we examined an ASP-based approach for CNOT minimal synthesis, utilizing a directed acyclic graph (DAG) to achieve optimal results. Roughly speaking, with our encoding, the smaller the graph the shorter the resulting CNOT circuit. While this model has proven effective for matrix sizes ranging from 2×2 to 6×6, it still faces significant challenges with larger inputs. Enhancing this model is essential, as achieving *optimal* solutions is often challenging. For instance, our ASP approach could be used as a benchmark to evaluate the performance of heuristic methods addressing the same problem.

Conclusions

This thesis has explored the role of graphs in quantum computing, evaluating whether they can serve as effectively in quantum systems as they do in classical computing. Our investigation began by examining classical uses of graphs, where they provide intuitive representations, efficient data structures, and semantic frameworks for various computational tasks. Extending this to quantum computation required both adapting existing graph-based methods and developing new strategies compatible with quantum mechanics' unique constraints, particularly the requirement for unitary transformations.

One of the central contributions of this work is the development of encoding methods that allow classical graphs to be represented within quantum gate-based models. Through these encoding methods, we demonstrated that unitary matrices can model graph-based transitions, facilitating processes like quantum random walks and state-based quantum automata. Experimental results suggest that these quantum adaptations of classical graph structures can yield advantages in both performance and accuracy for certain types of computations, particularly where complex relational data or state transitions are involved.

In addition to encoding, the thesis addressed specific applications where graphs enhance quantum computational tasks. Quantum circuit synthesis emerged as a particularly promising area: by representing circuit dependencies with graphs, we were able to optimize gate counts and minimize the use of costly gates like CNOT, thereby reducing circuit depth and error rates. Graph-based approaches allowed us to translate optimization problems into ASP frameworks, yielding efficient solutions for circuit design and gate synthesis. Results from quantum circuit optimizations suggest that graph-based methods can indeed streamline the design of quantum circuits, providing a practical benefit in terms of computational resource savings.

Before drawing some conclusions about our findings, we would like to stress a peculiarity of this thesis. The three parts structure we adopted for can be viewed as a *role* driven organization: topics are organized according to the role that graphs play in them. Some readers may find themselves confused by this choice. In fact, another potential organization we may have investigated is a *topic* related one. We would have kept the three parts separation, with the first one untouched. On the other hand, the second part would have been devoted to automata. That being the case, we would have discussed the topics of Chapter 5, Chapter 6, Chapter 8, and Chapter 9. The link between the four of them would have been the relation between graphs, bisimulation (and its weighted version, lumpability), and neural networks. To conclude, the third part would have gathered the results shown in Chapter 7, and in Chapter 10. In some way, we would have given to this part a ASP x Quantum, Quantum x ASP structure.

Our findings support the hypothesis that graphs can play a powerful role in quan-

tum computing, offering a bridge from classical structures to quantum applications and enhancing quantum computational models' adaptability and performance. Nonetheless, certain challenges remain. For instance, encoding efficiency can degrade with highly complex graphs, indicating that further refinement of encoding algorithms could be valuable. Additionally, while quantum random walks show promise, adapting classical graph traversal algorithms to quantum systems introduces non-trivial complexities, particularly regarding measurement and probabilistic state evolution.

Future research could focus on extending these graph-based methods to other quantum computing paradigms, such as measurement-based and adiabatic quantum computing, where different operational constraints may offer new opportunities for graph applications. Exploring more efficient encoding techniques or hybrid approaches that combine classical preprocessing with quantum execution could further enhance the applicability of graph-based models. Ultimately, this thesis provides a foundation for integrating classical graph theory into quantum computation, offering insights that may inform the design of future quantum algorithms, circuit designs, and problem-solving frameworks.

Bibliography

[1] S. Aaronson and D. Gottesman. Improved simulation of stabilizer circuits. *Physical Review A*, 70(5):052328, 2004.

[2] D. Aharonov, A. Ambainis, J. Kempe, and U. Vazirani. Quantum walks on graphs. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 50–59, 2001.

[3] D. Aharonov, W. van Dam, J. Kempe, Z. Landau, S. Lloyd, and O. Regev. Adiabatic Quantum Computation is Equivalent to Standard Quantum Computation, 2005.

[4] P. Aliferis, D. Gottesman, and J. Preskill. Quantum accuracy threshold for concatenated distance-3 codes. *Quantum Info. Comput.*, 6(2):97–165, 2006.

[5] E. Alpaydin. *Introduction to machine learning.* MIT press, 2020.

[6] A. Ambainis, E. Bach, A. Nayak, A. Vishwanath, and J. Watrous. One-dimensional quantum walks. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 37–49, 2001.

[7] A. Ambainis, M. Beaudry, M. Golovkins, A. Kikusts, M. Mercer, and D. Thérien. Algebraic results on quantum automata. *Theory of Computing Systems*, 39(1):165–188, 2006.

[8] A. Ambainis and R. Freivalds. 1-way quantum finite automata: strengths, weaknesses and generalizations. In *Proceedings 39th Annual Symposium on Foundations of Computer Science (Cat. No. 98CB36280)*, pages 332–341, 1998.

[9] M. Amy, P. Azimzadeh, and M. Mosca. On the controlled-not complexity of controlled-not–phase circuits. *Quantum Science and Technology*, 4(1):015002, 2018.

[10] M. Amy, D. Maslov, M. Mosca, and M. Roetteler. A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(6):818–830, 2013.

[11] M. Amy, D. L. Maslov, and M. Mosca. Polynomial-time t-depth optimization of clifford+t circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(10):1476–1489, 2013.

[12] M. Amy and M. Mosca. T-count optimization and reed–muller codes. *IEEE Transactions on Information Theory*, 65(8):4771–4784, 2019.

[13] A. Apicella, F. Donnarumma, F. Isgrò, and R. Prevete. A survey on modern trainable activation functions. *Neural Networks*, 138:14–32, 2021.

[14] B. Apolloni, C. Carvalho, and D. de Falco. Quantum stochastic optimization. *Stochastic Processes and their Applications*, 33(2):233–244, 1989.

[15] P. Arrighi. An overview of quantum cellular automata. *Natural Computing*, 18(4):885–899, 2019.

[16] M. Artin. *Algebra*. Pearson Education, 2011.

[17] L. Arufe, M. A. González, A. Oddi, R. Rasconi, and R. Varela. Quantum circuit compilation by genetic algorithm for quantum approximate optimization algorithm applied to maxcut problem. *Swarm and Evolutionary Computation*, 69:101030, 2022.

[18] L. Arufe, R. Rasconi, A. Oddi, R. Varela, and M. A. González. Solving quantum circuit compilation problem variants through genetic algorithms. *Natural Computing*, 22(4):631–644, 2023.

[19] B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.

[20] S. Baarir, M. Beccuti, C. Dutheillet, G. Franceschinis, and S. Haddad. Lumping partially symmetrical stochastic models. *Performance Evaluation*, 68(1):21–44, 2011.

[21] M. Baioletti and F. Santini. Abstract argumentation goes quantum: An encoding to qubo problems. In *PRICAI 2022: Trends in Artificial Intelligence*, volume 13629, pages 46–60, 2022.

[22] M. Bataille. Quantum circuits of cnot gates, 2020.

[23] J. S. Bell. On the einstein podolsky rosen paradox. *Physics Physique Fizika*, 1:195–200, 1964.

[24] P. C. Bell and M. Hirvensalo. Acceptance ambiguity for quantum automata. In *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[25] A. Belovs, A. Rosmanis, and J. Smotrovs. Multi-letter reversible and quantum finite automata. In *International Conference on Developments in Language Theory*, pages 60–71. Springer, 2007.

[26] V. Bergelson and A. Leibman. Distribution of values of bounded generalized polynomials. *Acta Mathematica*, 198(2):155–230, 2007.

[27] A. Bertoni and M. Carpentieri. Analogies and differences between quantum and stochastic automata. *Theoretical Computer Science*, 262(1-2):69–81, 2001.

[28] A. Bertoni, C. Mereghetti, and B. Palano. Quantum computing: 1-way quantum automata. In *International conference on developments in language theory*, pages 1–20, 2003.

[29] A. S. Bhatia and A. Kumar. Quantum finite automata: survey, status and research directions, 2019.

[30] A. S. Bhatia and A. Kumar. On relation between linear temporal logic and quantum finite automata. *Journal of Logic, Language and Information*, 29(2):109–120, 2020.

[31] S. A. Bhatia and A. Kumar. Quantum finite automata: survey, status and research directions. *ArXiv*, abs/1901.07992, 2019.

[32] M. P. Bianchi, C. Mereghetti, and B. Palano. Quantum finite automata: Advances on bertoni's ideas. *Theoretical Computer Science*, 664:39–53, 2017.

[33] C. Bianchini, A. Policriti, B. Riccardi, and R. Romanello. Incremental nfa minimization. *Theoretical Computer Science*, 1004(C), 2024.

[34] U. Birkan, Ö. Salehi, V. Olejar, C. Nurlu, and A. Yakaryılmaz. Implementing quantum finite automata algorithms on noisy devices. In *International Conference on Computational Science*, pages 3–16, 2021.

[35] J. Björklund and L. Cleophas. Aggregation-based minimization of finite state automata. volume 58, page 177–194, 2021.

[36] D. Blalock, J. J. Gonzalez Ortiz, J. Frankle, and J. Guttag. What is the state of neural network pruning?, 2020.

[37] P. Bocchieri and A. Loinger. Quantum recurrence theorem. *Physical Review*, 107:337–338, 07 1957.

[38] J. Boyar, P. Matthews, and R. Peralta. Logic minimization techniques with applications to cryptology. *Journal of Cryptology*, 26:280–312, 2013.

[39] S. Bravyi and D. Gosset. Improved classical simulation of quantum circuits dominated by clifford gates. *Physical Review Letters*, 116(25), 2016.

[40] S. Bravyi and A. Kitaev. Universal quantum computation with ideal clifford gates and noisy ancillas. *Physical Review A*, 71(2):022316, 2005.

[41] C. G. Brell Brell. Generalized cluster states based on finite groups. *New Journal of Physics*, 17(2):023–029, 2015.

[42] A. Brodsky and N. Pippenger. Characterizations of 1-way quantum finite automata. *SIAM Journal on Computing*, 31(5):1456–1478, 2002.

[43] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, 1986.

[44] A. Bu, E. Fan, and R. S. Joo. Minimum synthesis cost of cnot circuits, 2024.

[45] P. Buchholz. Exact and ordinary lumpability in finite markov chains. *Journal of Applied Probability*, 31(1):59–75, 1994.

[46] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[47] C. S Calude, M. J Dinneen, and R. Hua. Qubo formulations for the graph isomorphism problem and related problems. *Theoretical Computer Science*, 701(C):54–69, 2017.

[48] L. Cardelli, G. Squillace, M. Tribastone, M. Tschaikowski, and A. Vandin. Formal lumping of polynomial differential equations through approximate equivalences. *Journal of Logical and Algebraic Methods in Programming*, 134:100876, 2023.

[49] L. Cardelli, M. Tribastone, M. Tschaikowski, and A. Vandin. Comparing chemical reaction networks: A categorical and algorithmic perspective. LICS '16, page 485–494. Association for Computing Machinery, 2016.

[50] G. Castellano, A. M. Fanelli, and M. Pelillo. An iterative pruning algorithm for feedforward neural networks. *IEEE transactions on Neural networks*, 8(3):519–531, 1997.

[51] X. Chen, M. Zhu, X. Cheng, P. Zhu, and Z. Guan. Nearest neighbor synthesis of cnot circuits on general quantum architectures, 2023.

[52] F. Chung. Laplacians and the cheeger inequality for directed graphs. *Annals of Combinatorics*, 9(1):1–19, 2005.

[53] F. R. K. Chung. *Spectral Graph Theory.* American Mathematical Society, 1997.

[54] M. G. Davis, E. Smith, A. Tudor, K. Sen, I. Siddiqi, and C. Iancu. Towards optimal topology aware quantum circuit synthesis. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 223–234, 2020.

[55] C. M. Dawson and M. A. Nielsen. The solovay-kitaev algorithm, 2005.

[56] A. Dekhovich, D. M. J. Tax, M. H. F. Sluiter, and M. A. Bessa. Neural network relief: a pruning algorithm based on neural activity. *Machine Learning*, 113(5):2597–2618, 2024.

[57] D. Della Giustina, C. Londero, C. Piazza, B. Riccardi, and R. Romanello. Quantum encoding of dynamic directed graphs. *Journal of Logical and Algebraic Methods in Programming*, 136:100925, 2024.

[58] D. Della Giustina, C. Piazza, B. Riccardi, and R. Romanello. Directed Graph Encoding in Quantum Computing Supporting Edge-Failures. In *Reversible Computation*, volume 13354 of *LNCS*, pages 75–92. Springer, 2022.

[59] O. Di Di Matteo and M. Mosca. Parallelizing quantum circuit synthesis. *Quantum Science and Technology*, 1(1), 2016.

[60] P. A. M. Dirac. Lectures on quantum field theory. *American Journal of Physics*, 37:233–233, 1969.

[61] J Edmonds. Chinese postmans problem. In *Operations Research*, page B73, 1965.

[62] J. Edmonds and E. L. Johnson. Matching, euler tours and the chinese postman. *Math. Program.*, 5(1):88–124, 1973.

[63] A. Einstein, B. Podolsky, and N. Rosen. Can quantum-mechanical description of physical reality be considered complete? *Physical Review*, 47(10):777–780, 1935.

[64] H. A. Eiselt, M. Gendreau, and G. Laporte. Arc routing problems, part I: the chinese postman problem. *Oper. Res.*, 43(2):231–242, 1995.

[65] L. Euler. Solutio problematis ad geometriam situs pertinentis. *Commentarii academiae scientiarum Petropolitanae*, pages 128–140, 1741.

[66] E. Farhi, J. Goldstone, S. Gutmann, and M. Sipser. Quantum computation by adiabatic evolution, 2000.

[67] J. K. Fichte, S. A. Gaggl, and D. Rusovac. Rushing and strolling among answer sets–navigation made easy. 36(5):5651–5659, 2022.

[68] A. G. Fowler, A. M. Stephens, and P. Groszkowski. High-threshold universal quantum computation on the surface code. *Physical Review A*, 80(5):052312, 2009.

[69] G. Franceschinis and R.R. Muntz. Computing bounds for the performance indices of quasi-lumpable stochastic well-formed nets. *IEEE Transactions on Software Engineering*, 20(7):516–525, 1994.

[70] A. Gainutdinova and A. Yakaryılmaz. Unary probabilistic and quantum automata on promise problems. *Quantum Information Processing*, 17(2):1–17, 2018.

[71] N. Gavrielov, A. Ivrii, and S. Garion. Linear circuit synthesis using weighted steiner trees, 2024.

[72] M. Gelfond and Y. Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach.* Cambridge University Press, 07 2014.

[73] R. Gentili. *Graph algorithms for massive data-sets.* PhD thesis, University of Udine, Italy, 2005.

[74] P. Gera, H. Kim, P. Sao, H. Kim, and D. Bader. Traversing large graphs on gpus with unified memory. *Proceedings of the VLDB Endowment*, 13(7):1119–1133, 2020.

[75] G. GFranceschinis and R. R. Muntz. Bounds for quasi-lumpable markov chains. *Performance Evaluation*, 20(1):223–243, 1994.

[76] V. Gheorghiu, J. Huang, S. M. Li, M. Mosca, and P. Mukhopadhyay. Reducing the cnot count for clifford+t circuits on nisq architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(6):1873–1884, 2023.

[77] E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141 – 1144, 1959.

[78] Brett Giles and Peter Selinger. Exact synthesis of multiqubit clifford+t circuits. *Physical Review A*, 87(3):032332, 2013.

[79] D. Gosset, V. Kliuchnikov, M. Mosca, and V. Russo. An algorithm for the t-count, 2013.

[80] D. Gottesman. *Stabilizer codes and quantum error correction.* California Institute of Technology, 1997.

[81] D. Gottesman. The heisenberg representation of quantum computers, 1998.

[82] D. Gottesman and I. L. Chuang. Demonstrating the viability of universal quantum computation using teleportation and single-qubit operations. *Nature*, 402(6760):390–393, 1999.

[83] Daniel Gottesman. Theory of fault-tolerant quantum computation. *Physical Review A*, 57(1):127, 1998.

[84] T. Goubault de Brugière, M. Baboulin, B. Valiron, S. Martiel, and C. Allouche. Quantum cnot circuits synthesis for nisq architectures using the syndrome decoding problem. In *Reversible Computation*, volume 12227, page 189–205, 2020.

[85] T. Goubault de Brugière, M. Baboulin, B. Valiron, S. Martiel, and C. Allouche. Gaussian elimination versus greedy methods for the synthesis of linear reversible circuits. *ACM Transactions on Quantum Computing*, 2(3):11, 2021.

[86] T. Goubault de Brugière, M. Baboulin, B. Valiron, S. Martiel, and C. Allouche. Decoding techniques applied to the compilation of cnot circuits for nisq architectures. *Science of Computer Programming*, 214:102726, 2022.

[87] J. F. Groote, J. Martens, and E. P. de Vink. Lowerbounds for bisimulation by partition refinement. *Logical Methods in Computer Science*, Volume 19, Issue 2, 2023.

[88] M. Guan. Graphic programming using odd and even points. *Chinese Math.*, 1:237–277, 1962.

[89] I. Hen and A. P. Young. Solving the graph-isomorphism problem with a quantum annealer. *Physical Review A*, 86(4), 2012.

[90] J. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[91] J. E. Hopcroft. *A linear algorithm for testing equivalence of finite automata*, volume 114. Defense Technical Information Center, 1971.

[92] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[93] R. A. Horn and C. R. Johnson. *Matrix Analysis*. Cambridge University Press, 1985.

[94] K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[95] M. Incudini, F. Tarocco, R. Mengoni, A. Di Pierro, and A. Mandarino. Computing graph edit distance on quantum devices. *Quantum Machine Intelligence*, 4(2), 2022.

[96] K. Iwama, Y. Kambayashi, and S. Yamashita. Transformation rules for designing cnot-based quantum circuits. In *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*, pages 419–424, 2002.

[97] B. Jacobs and J. Rutten. An introduction to (co)algebra and (co)induction. In *Advanced Topics in Bisimulation and Coinduction*, volume 52 of *Cambridge tracts in theoretical computer science*, pages 38–99. Cambridge University Press, 2012.

[98] J. Jean, T. Peyrin, S. Meng Sim, and J. Tourteaux. Optimizing implementations of lightweight building blocks. *IACR Trans. Symmetric Cryptol.*, 2017(4):130–168, 2017.

[99] J. Jiang, X. Sun, S. Teng, B. Wu, K. Wu, and J. Zhang. Optimal space-depth trade-off of cnot circuits in quantum logic synthesis, 2022.

[100] J. Jiang and X. Wang. Lower bound for the t count via unitary stabilizer nullity. *Physical Review Applied*, 19(3), 2023.

[101] A. Jiménez-Pastor, K. G. Larsen, M. Tribastone, and M. Tschaikowski. Forward and backward constrained bisimulations for quantum circuits. In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 343–362, Cham, 2024. Springer Nature Switzerland.

[102] R. Jozsa. An introduction to measurement based quantum computation, 2005.

[103] P. C. Kanellakis and S. A. Smolka. Ccs expressions, finite state processes, and three problems of equivalence. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, page 228–240. Association for Computing Machinery, 1983.

[104] T. Kato. On the adiabatic theorem of quantum mechanics. *Journal of the Physical Society of Japan*, 5(6):435–439, 1950.

[105] J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. Finite Markov Chains. Van Nostrand, 1960.

[106] K. Khadiev, A. Khadieva, M. Ziatdinov, I. Mannapov, D. Kravchenko, A. Rivosh, and R. Yamilov. Two-way and one-way quantum and classical automata with advice for online minimization problems. *Theoretical Computer Science*, 920:76–94, 2022.

[107] K. Khoo, T.s Peyrin, A. Y. Poschmann, and H. Yap. Foam: Searching for hardware-optimal spn structures and components with a fair comparison. In *Cryptographic Hardware and Embedded Systems – CHES 2014*, pages 433–450. Springer Berlin Heidelberg, 2014.

[108] V. Kliuchnikov. Synthesis of unitaries with clifford+t circuits, 2013.

[109] V. Kliuchnikov, D. Maslov, and M. Mosca. Asymptotically optimal approximation of single qubit unitaries by clifford and t circuits using a constant number of ancillary qubits. *Physical Review Letters*, 110(19):190502, 2013.

[110] G. Kochenberger, J. Hao, F. Glover, M. Lewis, Z. Lü, H. Wang, and Y. Wang. The unconstrained binary quadratic programming problem: A survey. *Journal of Combinatorial Optimization*, 28(1):58–81, 2014.

[111] L. Kölsch. Xor-counts and lightweight multiplication with fixed elements in binary finite fields. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38*, volume 11476, pages 285–312, 2019.

[112] Attila Kondacs and John Watrous. On the power of quantum finite state automata. *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 66–75, 1997.

[113] J. Kwok and K. Pudenz. Graph coloring with quantum annealing, 2020.

[114] E. Lybrand and R. Saab. A greedy algorithm for quantizing neural networks, 2021.

[115] A. Marin, C. Piazza, and S. Rossi. Proportional lumpability. In *Formal Modeling and Analysis of Timed Systems - 17th International Conference, FORMATS 2019, Amsterdam, The Netherlands, August 27-29, 2019, Proceedings*, volume 11750 of *Lecture Notes in Computer Science*, pages 265–281. Springer, 2019.

[116] A. Marin, C. Piazza, and S. Rossi. Proportional lumpability and proportional bisimilarity. *Acta Informatica*, 59(2-3):211–244, 2021.

[117] E. Mastrostefano and M. Bernaschi. Efficient breadth first search on multi-gpu systems. *Journal of Parallel and Distributed Computing*, 73(9):1292–1305, 2013.

[118] W. S McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.

[119] C. Mereghetti and B. Palano. Quantum finite automata: From theory to practice. *ACM SIGACT News*, 52(3):38–59, 2021.

[120] C. Mereghetti, B. Palano, S. Cialdi, V. Vento, M. G. A. Paris, and S. Olivares. Photonic realization of a quantum finite automaton. *Physical Review Research*, 2(1):013089, 2020.

[121] G. Meuli, M. Soeken, and G. De Micheli. Sat-based {CNOT, T} quantum circuit synthesis. In *International Workshop on Reversible Computation*, 2018.

[122] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *SWAT*, 1972.

[123] D. A. Meyer. From quantum cellular automata to quantum lattice gases. *Journal of Statistical Physics*, 85(5-6):551–574, 1996.

[124] M. Mezard, G. Parisi, and M. A. Virasoro. *Spin Glass Theory and Beyond: An Introduction to the Replica Method and its Applications.* World Scientific Singapore, 1987.

[125] G. Minello, L. Rossi, and A. Torsello. Can a quantum walk tell which is which? a study of quantum walk-based graph similarity. *Entropy*, 21(3):328, 2019.

[126] M. Minsky and S. Papert. An introduction to computational geometry. *Cambridge tiass., HIT*, 479(480):104, 1969.

[127] A. Montanaro. Quantum walks on directed graphs. *arXiv preprint quant-ph/0504116*, 2005.

[128] C. Moore and J. P. Crutchfield. Quantum automata and quantum grammars. *Theoretical Computer Science*, 237(1-2):275–306, 2000.

[129] M. Mosca and P. Mukhopadhyay. A polynomial time and space heuristic algorithm for t-count. *Quantum Science and Technology*, 7(1):015003, 2021.

[130] R. Motwani and P. Raghavan. *Randomized algorithms.* Cambridge university press, 1995.

[131] A. Nayak and A. Vishwanath. Quantum Walk on the Line. 2000.

[132] M. A. Nielsen. Quantum computation by measurement and quantum memory. *Physics Letters A*, 308(2):96–100, 2003.

[133] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, 10 edition, 12 2010.

[134] J. O' Neill. An overview of neural network compression, 2020.

[135] A. Oddi, R. Rasconi, M. Baioletti, V. G. Santucci, and H. Beck. Quantum circuit compilation for the graph coloring problem. In *International Conference of the Italian Association for Artificial Intelligence*, pages 374–386, 2022.

[136] A. Oddi, R. Rasconi, M. Baioletti, V. G. Santucci, and H. Beck. Quantum Circuit Compilation for the Graph Coloring Problem. In *AIxIA 2022 – Advances in Artificial Intelligence*, volume 14318 of *LNCS*, pages 374–386. Springer, 2023.

[137] C. Paar. Optimized arithmetic for reed-solomon encoders. In *Proceedings of IEEE International Symposium on Information Theory*, pages 250–, 1997.

[138] A. Paetznick and B. W. Reichardt. Universal fault-tolerant quantum computation with only transversal gates and error correction. *Physical review letters*, 111(9):090505, 2013.

[139] A. Paetznick and K. M. Svore. Repeat-until-success: non-deterministic decomposition of single-qubit unitaries. *Quantum Info. Comput.*, 14(15–16):1277–1301, 2014.

[140] C. H. Papadimitriou. Computational complexity. In *Encyclopedia of computer science*, pages 260–265. Addison-Wesley, 2003.

[141] P.M. Parldaos and S. Jha. Complexity of uniqueness and local search in quadratic 0–1 programming. *Operations Research Letters*, 11(2):119–123, 1992.

[142] Kathrin Paschen. Quantum finite automata using ancilla qubits. Technical report, Universität Karlsruhe (TH), 2000.

[143] K. N. Patel, I. L. Markov, and J. P. Hayes. Optimal synthesis of linear reversible circuits. *Quantum Info. Comput.*, 8(3):282–294, 2008.

[144] C. Piazza and R. Romanello. Mirrors and memory in quantum automata. In *Quantitative Evaluation of Systems*, pages 359–380. Springer International Publishing, 09 2022.

[145] C. Piazza and R. Romanello. Synthesis of cnot minimal quantum circuits with topological constraints through asp. In *Proceedings of the International Workshop on AI for Quantum and Quantum for AI (AIQxQIA 2023)*, volume 3586, page 37 – 49. CEUR-WS, 2023.

[146] P. Prabhakar. Bisimulations for neural network reduction, 2021.

[147] A. K. Prasad, V. V. Shende, I. L. Markov, J. P. Hayes, and K. N. Patel. Data structures and algorithms for simplifying reversible circuits. *J. Emerg. Technol. Comput. Syst.*, 2(4):277–293, 2006.

[148] D. Qiu and S. Yu. Hierarchy and equivalence of multi-letter quantum finite automata. *Theoretical Computer Science*, 410(30-32):3006 – 3017, 2009.

[149] M. O. Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963.

[150] R. Raussendorf, D. E. Browne, and H. J. Briegel. Measurement-based quantum computation on cluster states. *Physical Review A*, 68(2), 2003.

[151] D. Ressi, R. Romanello, S. Rossi, and C. Piazza. Compressing neural networks via formal methods. *Neural Networks*, 178(C):106411, 2024.

[152] F. Riguzzi. Quantum algorithms for weighted constrained sampling and weighted model counting. *Quantum Machine Intelligence*, 6(2), 2024.

[153] R. Robert Paige and R. E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.

[154] R. Romanello, D. Della Giustina, S. Pessotto, and C. Piazza. Speeding up answer set programming by quantum computing. In *Proceedings of the 2024 Workshop on Quantum Search and Information Retrieval*, page 1–8. Association for Computing Machinery, 2024.

[155] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[156] W. H. Sanders, P. Buchholz, and D. Daly. Bound-preserving composition for markov reward models. In *Third International Conference on the Quantitative Evaluation of Systems*, pages 243–252. IEEE Computer Society, 2006.

[157] P. Selinger. Efficient clifford+t approximation of single-qubit operators. *Quantum Info. Comput.*, 15(1–2):159–180, 2015.

[158] S. Severini. Graphs of unitary matrices, 2003.

[159] S. Severini. On the digraph of a unitary matrix. *SIAM Journal on Matrix Analysis and Applications*, 25(1):295–300, 2003.

[160] V. V. Shende and I. L. Markov. On the cnot-cost of toffoli gates. *Quantum Info. Comput.*, 9(5):461–486, 2009.

[161] P. Shor. Lecture notes in quantum computation, 2003.

[162] Y. Tang. Efficient cnot synthesis for nisq devices. 2020.

[163] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22:215–225, 1975.

[164] W. van Dam, M. Mosca, and U. Vazirani. How powerful is adiabatic quantum computation? In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. IEEE, 2001.

[165] J. van de Wetering and M. Amy. Optimising quantum circuits is generally hard, 2024.

[166] V. Vandaele, S. Martiel, and T. Goubault de Brugière. Phase polynomials synthesis algorithms for nisq architectures and beyond. *Quantum Science and Technology*, 7(4):045027, 2022.

[167] J. Von Neumann. Mathematical foundations of quantum mechanics. In *Mathematical Foundations of Quantum Mechanics*. Princeton university press, 2018.

[168] B. W. Watson. *Taxonomies and toolkits of regular language algorithms*. PhD thesis, Mathematics and Computer Science, 1995.

[169] B. W. Watson and J. Daciuk. An efficient incremental dfa minimization algorithm. *Natural Language Engineering*, 9(1):49 – 64, 2003.

[170] T. Wei. Measurement-based quantum computation. *Oxford Research Encyclopedia of Physics*, 2021.

[171] B. Wu, X. He, S. Yang, L. Shou, G. Tian, J. Zhang, and X. Sun. Optimization of cnot circuits on limited-connectivity architecture. *Phys. Rev. Res.*, 5(1):013065, 2023.

[172] Z. Xiang, X. Zeng, D. Lin, Z. Bao, and S. Zhang. Optimizing implementations of linear layers. *IACR Transactions on Symmetric Cryptology*, 2:120–145, 2020.

[173] Y. Yuan, W. Wu, T. Shi, L. Zhang, and Y. Zhang. A framework to improve the implementations of linear layers. *IACR Transactions on Symmetric Cryptology*, 2024(2):322–347, 2024.

[174] M. Zhu, X. Cheng, P. Zhu, L. Chen, and Z. Guan. Physical constraint-aware cnot quantum circuit synthesis and optimization. *Quantum Information Processing*, 22(1):10, 2022.