# OPTIMIZING QUANTUM CIRCUIT LAYOUTS 

## JORDI ELGUETA SERRA

Thesis supervisor: JOSEP ELGUETA MONTO (Department of Mathematics)
Degree: Bachelor Degree in Informatics Engineering (Computing)

Thesis report
Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech


#### Abstract

One of the challenges in quantum computing is the problem of optimizing quantum circuit compilation. The compilation process involves two main stages: synthesizing the circuit to be executed in terms of the quantum gates supported by the processor, and adapting the circuit to the connectivity limitations imposed by the processor. In this work, I have addressed the second of these problems, known as Quantum Circuit Layout (QCL). To tackle this problem, I have attempted to use Reiforcement Learning (RL) techniques, which require modeling the problem as a Markov Decision Process (MDP). Specifically, I describe two finite MDPs whose solution provides a solution to a part of the QCL problem. The main problem is to design a method that effectively solves these MDPs, even if it is only an approximate solution. In the thesis two approaches to the problem are discussed. The first one uses a variant of the algorithm used in AlphaZero, designed to train a machine to learn how to play Chess, Shogi, and Go. The second approach uses a more standard approximation known as Deep Q-Learning (DQL).


## Resum

Un dels problemes amb els quals s'enfronta la computació quàntica és el de l'optimització de la compilació d'un circuit quàntic. El procés de compilació inclou bàsicament dues etapes: síntesi del circuit a executar en termes de les portes quàntiques suportades pel processador, i adaptació del circuit a executar a les limitacions de connectivitat imposades pel processador. En aquest treball, he abordat el segon d'aquests problemes, conegut amb el nom de Quantum Circuit Layout (QCL). Per a la seva resolució, he intentat usar tècniques de Reinforcement Learning (RL), que requereixen modelitzar prèviament el problema en termes d'un Markov Decision Process (MDP). En concret, descric dos MDP's finits la solució dels quals proporciona una solució a una part del problema del QCL. El problema principal és dissenyar un mètode que permeti efectivament resoldre aquests MDP's, ni que sigui de manera aproximada. En el treball es discuteixen dues aproximacions al problema. La primera d'elles utilitza una variant de l'algoritme usat per AlphaZero, dissenyat amb l'objectiu d'entrenar a una màquina per tal que aprengui a jugar als jocs d'Escacs, Shogi i Go. La segona utilitza una aproximació més estàndard coneguda com a Deep $Q$-Learning (DQL).

## Resumen

Uno de los problemas con los que se enfrenta la computación cuántica es el de la optimización de la compilación de un circuito cuántico. El proceso de compilación incluye básicamente dos etapas: síntesis del circuito a ejecutar en términos de las puertas cuánticas soportadas por el procesador, y adaptación del circuito a ejecutar a las limitaciones de conectividad impuestas por el procesador. En este trabajo, he abordado el segundo de estos problemas, conocido con el nombre de Quantum Circuit Layout (QCL). Para su resolución, he intentado usar técnicas de Reinforcement Learning (RL), que requieren modelizar préviamente el problema en términos de un Markov Decision Process (MDP). En concreto, describo dos MDP's finitas cuya solución proporciona una solución a una parte del problema del QCL. El problema principal es diseñar un método que permita efectivamente resolver dichas MDP's, ni que sea de manera aproximada. En el trabajo se discuten dos aproximaciones al problema. La primera de ellas utiliza una variante del algoritmo usado para AlphaZero, diseñado con el objetivo de entrenar una máquina para que aprenda a jugar a los juegos Ajedrez, Shogi y Go. La segunda utiliza una aproximación más estándard conocida como Deep Q-Learning (DQL).

## Contents

1 Introduction ..... 3
2 Basic concepts of Quantum Computing ..... 3
2.1 Qubits ..... 4
2.2 Quantum gates ..... 5
2.2.1 Single qubit gates ..... 6
2.2.2 Multiple qubit gates ..... 8
2.3 Quantum circuits ..... 8
2.3.1 Matrix description of a quantum circuit ..... 9
2.4 Sets of universal quantum gates ..... 9
3 The problem of quantum circuit compilation ..... 10
3.1 Synthesis of a quantum circuit ..... 10
3.2 Problem to solve: Quantum circuit layout ..... 11
4 Scope ..... 12
4.1 Previous works on the subject ..... 12
4.2 Goals and approach ..... 13
5 Time planning ..... 15
5.1 Description of tasks ..... 15
5.1.1 Identified tasks ..... 15
5.1.2 Workload estimates per task ..... 16
5.2 Time planning update ..... 18
6 Basic concepts of Reinforcement Learning ..... 18
6.1 General idea. ..... 18
6.2 Mathematical description as a Markov Decision Process ..... 18
6.3 Model-based and Model-free RL algorithms ..... 20
7 Qubit mapping problem as a finite MDP ..... 20
7.1 Exact methods ..... 22
7.2 First approximate method: DNN aided with MCTS ..... 23
7.2.1 AlphaZero ..... 23
7.2.2 My algorithm ..... 25
7.2.3 Discussion of the results ..... 29
7.3 Second approximate method: Deep Q-learning (DQL) ..... 30
7.3.1 My algorithm ..... 31
7.3.2 Discussion of the results ..... 32
8 Final conclusions and future work ..... 32
9 Annex A. Budget ..... 33
9.1 Identification of costs ..... 33
9.1.1 Staff costs ..... 33
9.1.2 Generic costs ..... 33
9.1.3 Contingencies and incidentals ..... 34
9.2 Cost estimates ..... 34
9.3 Management control ..... 35
10 Annex B. Sustainability ..... 36
10.1 Economic dimension ..... 36
10.2 Environmental dimension ..... 36
10.3 Social dimension ..... 36

## 1 Introduction

In recent decades, the fields of Quantum Computation (QC) and Artificial Intelligence (AI) have gained significant attention due to their potential applications. Notably, AI has experienced rapid advancements by capitalizing on the state-of-the-art classical hardware that has been developed over several decades. This progress has led to remarkable achievements in AI systems like AlphaZero, AlphaFold, and more recently, GPT-3 and GPT-4. On the other hand, Quantum Computing has progressed at a slower pace, primarily due to the need for a fundamental shift in computing paradigms. Consequently, current quantum computers are still lacking reliable hardware with an adequate number of qubits to outperform classical computers significantly. Given the notable achievements of AI, this thesis attempts to exploit its capabilities to address a specific QC problem.

More precisely, this thesis intended to tackle one aspect of the quantum circuit compilation problem, known as Quantum Circuit Layout (QCL). In a quantum processor there are a number of qubits, called the physical qubits, with some specific connections between them. When executing a quantum circuit, consisting of a certain set of (quantum) gates acting on a certain number of qubits, called the logical qubits, it is necessary to determine the mapping of logical qubits to physical qubits. This mapping cannot be arbitrary as there may be gates acting on multiple qubits, requiring them to be connected in the execution. For instance, a gate may act on the logical qubits $q 1$ and $q 2$, and these logical qubits have to be mapped to physical qubits which are connected in the processor. Otherwise, the gate cannot be executed. In general, there may be no mapping that allows all gates in the circuit to be executed. This problem can be handled by adding SWAP gates at certain points in the circuit. However, the addition of these gates increases the circuit's execution time, which for reasons discussed in the sequel, has to be minimized as much as possible. The QCL problem basically consists of finding the optimal mapping of logical qubits to physical qubits to minimize the added overhead.

QCL has been proven to be NP-Complete, as I discuss later. Therefore the algorithms that provide exact solutions are not scalable to large processors and circuits. Instead the best we can do is to create algorithms that approximate the solution as accurately as possible. Inspired by the multiple successes of AI, in this thesis I set out to create one such algorithm by using Reinforcement Learning (RL). By formulating the QCL problem as a reinforcement learning task, this thesis seeks to train an agent, through a process of trial and error, to discover configurations that minimize overhead.

Despite the significant achievements of AI, it is essential to acknowledge the substantial cost associated with training machine learning models. This is particularly true for algorithms that use reinforcement learning. This raises the question of whether the extensive training efforts are justified in optimizing circuit execution in the quantum realm. When considering the current classical hardware, although the training process may be time-consuming, it can be managed reasonably well and given the inherent fragility of quantum systems, the pursuit of runtime improvements for quantum hardware justifies this effort.

The outline of the thesis is as follows. In Section 2 I will explain the basic notions and facts about quantum computation, in particular, the notions of qubit, quantum gate, and universal set of gates. In Section 3 the specific problem intended to be solved is explained in more detail, and a concrete example is given to illustrate it. In Section 4, after a quick review of the existing literature on the subject, the followed approach is explained and justified. Section 5 gives the initial and updated time planning of the project tasks. Section 7 is the core of this thesis. In this section two Markov Decision Processes (MDP) are described which provide solutions to the stated problem, and two strategies to solve these MDP's are presented. In Section 8 the conclusions and some ideas for future work are also explained. Finally, a discussion of the planned budget, and the sustainability concerns is included in two separate appendices.

## 2 Basic concepts of Quantum Computing

Quantum computing is a field of computing that employs the principles of quantum mechanics to process information. It has its roots in the early 1980s when the physicist Richard Feynman proposed the idea of using quantum systems to simulate physical phenomena. However, it wasn't until the 1990s that significant progress was made in developing quantum algorithms and actual quantum computers.

What makes quantum computing powerful is its ability to exploit certain quantum phenomena, such as superposition and entanglement, to perform certain calculations exponentially faster than classical computers. For example, Shor's algorithm [25], developed by mathematician Peter Shor in 1994, demonstrated that a quantum computer could factor large numbers exponentially faster than a classical computer, a task that is crucial for many encryption schemes.

In this section I will present the most relevant concepts in quantum computing that will be needed in the sequel.

### 2.1 Qubits

Just as classical computing is based on bits as the fundamental unit of information, quantum computing relies on an analogous concept called a quantum bit, or qubit for short. Unlike the classical bit, the qubit can exist not only in the states 0 or 1 but also in a superposition of both states. In this subsection, I will give the formal representation of a generic state of the qubit as well as introduce the properties of single and multiple qubit systems.

A qubit is physically implemented as a two-state quantum-mechanical system. In quantum mechanics, the states of a system are formally described by unit vectors in a two dimensional Hilbert space. Thus a state of a qubit is represented as a unit vector $|\psi\rangle$ in a two-dimensional Hilbert space, and an arbitrary state of a qubit can be expressed as

$$
\begin{equation*}
|\psi\rangle=\alpha|0\rangle+\beta|1\rangle \tag{2.1}
\end{equation*}
$$

where the coefficients $\alpha$ and $\beta$ are complex numbers such that $|\alpha|^{2}+|\beta|^{2}=1$. The special states $|0\rangle$ and $|1\rangle$ are known as the computational basis states, and they form an orthonormal basis for the Hilbert space. The coefficients $\alpha$ and $\beta$ are known as probability amplitudes because $|\alpha|^{2}$ and $|\beta|^{2}$ give the probability of finding the qubit in the state $|0\rangle$ and $|1\rangle$, respectively, when observed.

We can rewrite the state 2.1 of a qubit using the polar form of complex numbers as

$$
\begin{equation*}
|\psi\rangle=e^{i \gamma}\left(\cos \frac{\theta}{2}|0\rangle+e^{i \varphi} \sin \frac{\theta}{2}|1\rangle\right) \tag{2.2}
\end{equation*}
$$

where $\theta, \varphi$ and $\gamma$ are real numbers (the point $(|\alpha|,|\beta|)$ is in the first quadrant of the unit circle because $|\alpha|^{2}+|\beta|^{2}=1$ and hence, $|\alpha|$ and $|\beta|$ can indeed be written as the cosine and sine, respectively, of some angle $\theta / 2 \in[0, \pi / 2])$. The global phase $e^{i \gamma}$ can be ignored because it does not affect the probabilities of the qubit to be in the states $|0\rangle$ or $|1\rangle$. Thus we can rewrite the state of the qubit as

$$
\begin{equation*}
|\psi\rangle=\cos \frac{\theta}{2}|0\rangle+e^{i \varphi} \sin \frac{\theta}{2}|1\rangle \tag{2.3}
\end{equation*}
$$

The numbers $\theta \in[0, \pi]$ and $\varphi \in[0,2 \pi)$ define a point on the surface of the unit three-dimensional sphere as shown in Figure 1. This sphere is known as the Bloch sphere, and it provides an alternative geometric representation of the states of a qubit that allows for their visualization in a three-dimensional reference system. For instance, the north and south poles of the sphere respectively correspond to $\theta=0$ and $\theta=\pi$, and hence they represent the states $|0\rangle$ and $|1\rangle$, respectively. The remaining points correspond to superpositions of both states.

An example of a two-state quantum-mechanical system that is used to implement qubits are superconducting qubits. In superconducting qubits, the two states associated with the computational basis states $|0\rangle$ and $|1\rangle$ are the so called ground state, and excited state respectively. Research on this type of qubits is conducted by companies such as IBM, Google and Intel. However, research is also being conducted on other types of qubits, such as trapped ions, which try to implement qubits using the internal states of ions, and photonic qubits, which try to implement qubits using the polarization states of photons.

In multiple qubits systems, the possible states of the joint quantum system are represented by unit vectors in the so called tensor product of the state spaces of each of the qubits, instead of their cartesian product. By definition, the tensor product is the complex $2^{n}$-dimensional Hilbert space, being $n$ the


Figure 1: Bloch sphere representation of the states of a qubit
number of qubits, having as a basis the kets corresponding to the different classical states. For instance, if we have two qubits with state spaces $\mathcal{H}_{1}$ and $\mathcal{H}_{2}$, respectively, the states of the joint system are represented by a unit vector in the tensor product of $\mathcal{H}_{1}$ and $\mathcal{H}_{2}$, which is denoted by $\mathcal{H}_{1} \otimes \mathcal{H}_{2}$, and a basis of $\mathcal{H}_{1} \otimes \mathcal{H}_{2}$ consists of the kets $\{|00\rangle,|01\rangle,|10\rangle,|11\rangle\}$. These kets play the role of the computational basis states for two qubits, and an arbitrary state is given by

$$
\begin{equation*}
|\psi\rangle=\alpha_{00}|00\rangle+\alpha_{01}|01\rangle+\alpha_{10}|10\rangle+\alpha_{11}|11\rangle \tag{2.4}
\end{equation*}
$$

where, just like before, $\alpha_{i j}$ are complex numbers such that $\left|\alpha_{00}\right|^{2}+\left|\alpha_{01}\right|^{2}+\left|\alpha_{10}\right|^{2}+\left|\alpha_{11}\right|^{2}=1,\left|\alpha_{i j}\right|^{2}$ being the probability of finding the two qubit system in the state $|i j\rangle$ if it is in the state $|\psi\rangle$ before the observation. Once more, the intuitive idea is that the two-qubit system can be in any of the four classical states plus any linear combination of them, which correspond to superposed states.

### 2.2 Quantum gates

In classical computing, computation is performed through classical logic gates. These are boolean maps $f:\{0,1\}^{n} \rightarrow\{0,1\}$ where $n$ is the number of bits on which the gate acts (the output is assumed to be a single bit). As such, classical gates are defined by their truth tables, specifying the output for each possible state of the $n$ input bits, which is indeed an element in the cartesian product $\{0,1\}^{n}$.

Similarly, in quantum computing computation is done by the so-called quantum gates acting on qubits. Just like classical gates, they are maps from the state space of the input qubits to the state space of the output ones. There are however a couple of important differences between classical and quantum gates. Firstly, in quantum computing every gate has the same number of input and output qubits. Thus, in general, a quantum gate $U$ is a map $U: \mathcal{H}_{1} \otimes \cdots \otimes \mathcal{H}_{n} \rightarrow \mathcal{H}_{1}^{\prime} \otimes \cdots \otimes \mathcal{H}_{n}^{\prime}$ where $\mathcal{H}_{1} \otimes \cdots \otimes \mathcal{H}_{n}$ is the Hilbert state space of the n input qubits and $\mathcal{H}_{1}^{\prime} \otimes \cdots \otimes \mathcal{H}_{n}^{\prime}$ is the Hilbert state space of the n output qubits. Secondly, in order to be a valid quantum gate, $U$ has to be a unitary linear map. Linear means that

$$
U(a|\phi\rangle+b|\psi\rangle)=a(U|\phi\rangle)+b(U|\psi\rangle)
$$

for any states $|\phi\rangle,|\psi\rangle$ and any complex numbers $a, b \in \mathbb{C}$. From this condition it follows that $U$ is completely given by the images of the computational basis states, and hence, by an $2^{n} \times 2^{n}$ complex matrix whose columns are the probability amplitudes of the respective image states. For instance, any single qubit quantum gate $U: \mathcal{H}_{1} \rightarrow \mathcal{H}_{1}^{\prime}$ is given by a $2 \times 2$ complex matrix

$$
U=\left(\begin{array}{ll}
a_{00} & a_{01} \\
a_{10} & a_{11}
\end{array}\right)
$$

meaning that $U$ is the unique linear map such that

$$
\begin{aligned}
U|0\rangle & =a_{00}|0\rangle+a_{10}|1\rangle \\
U|1\rangle & =a_{01}|0\rangle+a_{11}|1\rangle
\end{aligned}
$$

(actually, I should write $|0\rangle,|1\rangle$, and $|0\rangle^{\prime},|1\rangle^{\prime}$ to distinguish the basis states of the input and output qubits, but I will never do that). Similarly, a two qubit quantum gate $U: \mathcal{H}_{1} \otimes \mathcal{H}_{2} \rightarrow \mathcal{H}_{1}^{\prime} \otimes \mathcal{H}_{2}^{\prime}$ is given
by a $4 \times 4$ complex matrix

$$
U=\left(\begin{array}{cccc}
a_{00 ; 00} & a_{00 ; 01} & a_{00 ; 10} & a_{00 ; 11} \\
a_{01 ; 00} & a_{01 ; 01} & a_{01 ; 10} & a_{01 ; 11} \\
a_{10 ; 00} & a_{10 ; 01} & a_{10 ; 10} & a_{10 ; 11} \\
a_{11 ; 00} & a_{11 ; 01} & a_{11 ; 10} & a_{11 ; 11}
\end{array}\right)
$$

meaning that $U$ is the unique linear map such that

$$
\begin{aligned}
U|00\rangle & =a_{00 ; 00}|00\rangle+a_{01 ; 00}|01\rangle+a_{10 ; 00}|10\rangle+a_{11 ; 00}|11\rangle, \\
U|01\rangle & =a_{00 ; 01}|00\rangle+a_{01 ; 01}|01\rangle+a_{10 ; 01}|10\rangle+a_{11 ; 01}|11\rangle, \\
U|10\rangle & =a_{00 ; 10}|00\rangle+a_{01 ; 10}|01\rangle+a_{10 ; 10}|10\rangle+a_{11 ; 10}|11\rangle, \\
U|11\rangle & =a_{00 ; 11}|00\rangle+a_{01 ; 11}|01\rangle+a_{10 ; 11}|10\rangle+a_{11 ; 11}|11\rangle .
\end{aligned}
$$

As for the term unitary, it means that the matrix describing $U$ must be a unitary one, which means that it must satisfy $U^{\dagger} U=I$. Here $I$ stands for the $2^{n} \times 2^{n}$ identity matrix, and $U^{\dagger}$ is the conjugate transpose of $U$.

With this matrix representation of quantum gates, the image of a generic state $|\psi\rangle$ is computed as the matrix product of $U$ and the column vector whose entries are the probability amplitudes of $|\psi\rangle$ in the computational basis.

The linearity condition arises from the fundamental fact that the evolution of quantum systems is governed by linear operators. This basically follows from the linearity of the famous Schrödinger equation telling us how the state of a quantum system evolves with time when no measurement is made on it. A comprehensive explanation of this principle exceeds the scope of this thesis. For more details, the interested reader is referred to any quantum mechanics textbook, for instance [15]. Regarding the unitary requirement, it is needed to preserve the normalization condition on the amplitude probabilities of the output states resulting from the application of a quantum gate.

As an example, let us consider the linear transformation defined by the following action on the computational basis states:

$$
\begin{align*}
U|0\rangle & =\frac{1}{\sqrt{2}}(|0\rangle+i|1\rangle) \\
U|1\rangle & =\frac{1}{\sqrt{2}}(|0\rangle-i|1\rangle) \tag{2.5}
\end{align*}
$$

Thus its matrix representation is

$$
U=\frac{1}{\sqrt{2}}\left(\begin{array}{cc}
1 & 1  \tag{2.6}\\
i & -i
\end{array}\right)
$$

We can easily see that $U$ is a valid single qubit gate, since it is a unitary $2 \times 2$ matrix. The action of $U$ on an arbitrary state of the qubit $|\psi\rangle=\alpha|0\rangle+\beta|1\rangle$ is given by the matrix product

$$
U|\psi\rangle=\frac{1}{\sqrt{2}}\left(\begin{array}{cc}
1 & 1  \tag{2.7}\\
i & -i
\end{array}\right)\binom{\alpha}{\beta}=\frac{1}{\sqrt{2}}\binom{\alpha+\beta}{i \alpha-i \beta}=\frac{1}{\sqrt{2}}(\alpha+\beta)|0\rangle+\frac{i}{\sqrt{2}}(\alpha-\beta)|1\rangle
$$

### 2.2.1 Single qubit gates

As discussed below (see $\S(2.4)$, single qubit gates play a special role in quantum computing. In contrast to what happens in the classical setting, where there is a unique non trivial one bit gate (the NOT gate), there are infinitely many of them. In fact, as explained before, single qubit gates correspond to unitary $2 \times 2$ complex matrices, and there is a continuous infinity of such matrices (its set is denoted by $U(2)$ ). Among them we have the so called Pauli matrices, denoted by $X, Y, Z$. These are the matrices

$$
X \equiv\left(\begin{array}{cc}
0 & 1 \\
1 & 0
\end{array}\right), \quad Y \equiv\left(\begin{array}{cc}
0 & -i \\
i & 0
\end{array}\right), \quad Z \equiv\left(\begin{array}{cc}
1 & 0 \\
0 & -1
\end{array}\right)
$$

Three other important single qubit gates are the Hadamard gate (denoted H ), the phase gate (denoted S), and the T-gate (denoted T), respectively given by the unitary matrices

$$
H \equiv \frac{1}{\sqrt{2}}\left(\begin{array}{cc}
1 & 1 \\
1 & -1
\end{array}\right), \quad S \equiv\left(\begin{array}{cc}
1 & 0 \\
0 & i
\end{array}\right), \quad T \equiv\left(\begin{array}{cc}
1 & 0 \\
0 & e^{i \pi / 4}
\end{array}\right) .
$$

Each single qubit gate is represented by a square with the name of the gate inside it, and an input and an output wire representing the input and output qubits, respectively. For instance, the previous six gates are represented by the diagrams

$$
\sqrt{X}-\sqrt{Y}-\sqrt{Z}-\sqrt{H}-\sqrt{S}-\sqrt{T}
$$

It is worth to visualize the action of a unitary matrix in the Bloch sphere picture. When the states of a single qubit are identified with the points in the Bloch sphere, it turns out that, up to a global phase factor, an arbitrary unitary $2 \times 2$ matrix $U$ acts as a rotation $R_{\hat{n}}(\theta)$ by an angle $\theta$ around an axis given by a unit vector $\hat{n}=\left(n_{x}, n_{y}, n_{z}\right)$. More precisely, the unitary matrices $U$ corresponding to the rotation $R_{\hat{n}}(\theta)$ of the Bloch sphere are those given by

$$
\begin{equation*}
U=e^{i \alpha}[\cos (\theta / 2) I-i \sin (\theta / 2)(\hat{n} \cdot \vec{\sigma})] \tag{2.8}
\end{equation*}
$$

where $e^{i \alpha}$ is an arbitrary global phase, $I$ stands for the $2 \times 2$ identity matrix, and $\vec{\sigma}$ is the vector of $2 \times 2$ matrices whose components are the Pauli matrices, i.e. $\vec{\sigma}=(X, Y, Z)$ (see [18], § 4.4.2). Thus the most general unitary matrix $U$ implementing the rotation $R_{\hat{n}}(\theta)$ of the Bloch sphere is

$$
\begin{equation*}
U=e^{i \alpha}\left[\cos (\theta / 2) I-i \sin (\theta / 2)\left(n_{x} X+n_{y} Y+n_{z} Z\right)\right] . \tag{2.9}
\end{equation*}
$$

In particular, the Pauli matrices $X, Y, Z$ respectively correspond to the rotations by an angle $\pi$ around the $x, y$ and $z$ axis of the Bloch sphere, and a trivial global phase factor, respectively (this explains the notation for these matrices). Similarly, the T-gate geometrically corresponds to a rotation by $\pi / 4$ around the $z$ axis of the Bloch sphere. Indeed, by taking $\alpha=\pi / 8, \theta=\pi / 4$ and $\hat{n}=(0,0,1)$ in the previous formula we obtain the unitary matrix

$$
U=e^{i \pi / 8}\left[\cos (\pi / 8)\left(\begin{array}{ll}
1 & 0  \tag{2.10}\\
0 & 1
\end{array}\right)-i \sin (\pi / 8)\left(\begin{array}{cc}
1 & 0 \\
0 & -1
\end{array}\right)\right]=e^{i \pi / 8}\left(\begin{array}{cc}
e^{-i \pi / 8} & 0 \\
0 & e^{i \pi / 8}
\end{array}\right)=\left(\begin{array}{cc}
1 & 0 \\
0 & e^{i \pi / 4}
\end{array}\right)
$$

The fact that single qubit gates correspond geometrically to rotations of the Bloch sphere has an important consequence. It is a well known result in mathematics that every rotation around any axis can be obtained as a sequence of three rotations around the coordinates axis by suitable angles, called the Euler angles. Actually, there are many ways of doing this, giving rise to many definitions of the Euler angles. One way is to obtain the rotation by an angle $\theta$ around the $\hat{n}$ axis as the composite of a rotation by an angle $\delta$ around the $z$ axis, followed by a rotation by an angle $\beta$ around the $y$ axis, and a rotation by an angle $\gamma$ again around the $z$ axis. The three angles $\beta, \gamma, \delta$ are the Euler angles, and of course, they depend on the pair $(\theta, \hat{n})$. This leads to the following

Z-Y decomposition of a single qubit gate. Given any single qubit gate $U$ there exist real numbers $\alpha, \beta, \gamma, \delta$ such that

$$
U=e^{i \alpha} R_{z}(\gamma) R_{y}(\beta) R_{z}(\delta)
$$

where $R_{\hat{n}}(\theta)$ for any pair $(\theta, \hat{n})$ stands for the above $2 \times 2$ unitary matrix $\cos (\theta / 2) I-$ $i \sin (\theta / 2)(\hat{n} \cdot \vec{\sigma})$.

An easy computation shows that the matrix product $R_{z}(\beta) R_{y}(\gamma) R_{z}(\delta)$ is equal to

$$
R_{z}(\gamma) R_{y}(\beta) R_{z}(\delta)=e^{-i(\gamma+\delta) / 2}\left(\begin{array}{cc}
\cos (\beta / 2) & -e^{i \delta} \sin (\beta / 2)  \tag{2.11}\\
e^{i \gamma} \sin (\beta / 2) & e^{i(\gamma+\delta)} \cos (\beta / 2)
\end{array}\right)
$$

Following the notation used in Qiskit we shall denote this unitary matrix without the global phase as $U(\beta, \gamma, \delta)$. In particular, $U(\beta, 0,0)=R_{y}(\beta)$ while $U(\beta,-\pi / 2, \pi / 2)=R_{x}(\beta)$. Diagrammatically, this gate is represented by

$$
U(\beta, \gamma, \delta)
$$

### 2.2.2 Multiple qubit gates

Among the multiple qubit gates, the most important one is the CNOT gate (the C means controlledNOT). This gate has two input qubits (and two output qubits) known as the control qubit and the target qubit. The gate acts as follows. If the control qubit is set to 0 , then the target qubit is left alone. If the control qubit is set to 1 , then the target qubit is flipped. In equations this means that

$$
|00\rangle \rightarrow|00\rangle, \quad|01\rangle \rightarrow|01\rangle, \quad|10\rangle \rightarrow|11\rangle, \quad|11\rangle \rightarrow|10\rangle
$$

Thus in the computational basis $\{|00\rangle,|01\rangle,|10\rangle,|11\rangle\}$ this gate corresponds to the unitary $4 \times 4$ matrix

$$
\left(\begin{array}{llll}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0
\end{array}\right)
$$

Diagrammatically, the CNOT gate is represented by

with the dot representing the control qubit, and the circle the target one.
More generally, given an arbitrary single qubit gate $U$ the controlled- $U$ gate is a two qubit gate, again with a control and a target qubit. If the control qubit is set to 1 then $U$ is applied to the target qubit, otherwise the target qubit is left alone. Diagrammatically, this gate is represented by


### 2.3 Quantum circuits

As in the classical setting, a quantum circuit consists of several quantum gates suitably connected in order to perform a given task on a certain number of input qubits. An example of such a circuit, in this case with three input (and output) qubits, is shown in Figure 2 with the wires representing the qubits.


Figure 2: Quantum circuit example
It consists of four gates, two of them connected in parallel (gates $H$ and $X$ ), and the remaining ones (two CNOT gates) applied sequentially after the previous ones. Quantum circuits are "read" from left to right, thus the $\left\{q_{i}\right\}$ in the diagram correspond to the input qubits.

An important property of a circuit is its depth. By definition, it is the length of the longest path from the input qubits to the output ones, moving forward in time along qubit wires. In other words, the depth is the smallest amount of time-steps required to execute the circuit, if every gate is computed at some integer time-step. On every time-step at most one gate can be applied on every qubit. For instance, the depth of the previous circuit is three because the $H$ and $X$ gates can be performed at the same time-step, while the two CNOT gates have to be performed at different time-steps one another and from the time-step where the $H$ and $X$ gates are performed. A less obvious example is the circuit shown in Figure 3 whose depth can easily be checked to be 6 .


Figure 3: Quantum circuit with depth 6

### 2.3.1 Matrix description of a quantum circuit

Every circuit has an algebraic interpretation in terms of compositions and tensor products of the unitary linear maps corresponding to the gates in the circuit. More precisely, the sequential connection of gates corresponds to their composition as linear maps and hence, to the product of their respective matrices, while the parallel connection of gates corresponds to their tensor product as linear maps and hence, to the so called Kronecker product of their respective matrices. Let us recall that given a $m \times n$ matrix $A$, and a $p \times q$ matrix $B$, their Kronecker product, denoted $A \otimes B$, is the $(m p) \times(n q)$ matrix given by

$$
A \otimes B=\left(\begin{array}{ccc}
a_{11} B & \cdots & a_{1 n} B \\
\vdots & & \vdots \\
a_{m 1} B & \cdots & a_{m n} B
\end{array}\right)
$$

For instance

$$
\left(\begin{array}{cc}
1 & 2 \\
0 & -1
\end{array}\right) \otimes\left(\begin{array}{ll}
1 & 1 \\
3 & 2
\end{array}\right)=\left(\begin{array}{cc}
1\left(\begin{array}{ll}
1 & 1 \\
3 & 2
\end{array}\right) & 2\left(\begin{array}{cc}
1 & 1 \\
3 & 2
\end{array}\right) \\
0\left(\begin{array}{ll}
1 & 1 \\
3 & 2
\end{array}\right) & -1\left(\begin{array}{ll}
1 & 1 \\
3 & 2
\end{array}\right)
\end{array}\right)=\left(\begin{array}{cccc}
1 & 1 & 2 & 2 \\
3 & 2 & 6 & 4 \\
0 & 0 & -1 & -1 \\
0 & 0 & -3 & -2
\end{array}\right)
$$

This allows one to describe any circuit of $n$ qubits as simply a unitary $2^{n} \times 2^{n}$ matrix (it indeed amounts to a single quantum gate). For instance, 3 the circuit (...) corresponds to the composite of the tensor products $H \otimes X \otimes I, C N O T \otimes I$ and $I \otimes C N O T$ (tensor products are taken top to bottom). Notice that the identity gates $I$ need to be added because the three maps have to act on the same set of qubits (the three qubits in this case). Therefore the circuit is equivalent to the three qubit gate whose unitary $8 \times 8$ matrix is

$$
\left.\left.\begin{array}{rl}
{\left[\left(\begin{array}{ll}
1 & 0 \\
0 & 1
\end{array}\right) \otimes\left(\begin{array}{llll}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0
\end{array}\right)\right]} & {\left[\left(\begin{array}{cccc}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0
\end{array}\right)\right.}
\end{array}\right] \otimes\left(\begin{array}{cc}
1 & 0 \\
0 & 1
\end{array}\right)\right]\left[\begin{array}{cccccccc}
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\
0 & \left.\left(\begin{array}{cc}
1 & 1 \\
1 & -1
\end{array}\right) \otimes\left(\begin{array}{ll}
0 & 1 \\
1 & 0
\end{array}\right) \otimes\left(\begin{array}{ll}
1 & 0 \\
0 & 1
\end{array}\right)\right] \\
& =\frac{1}{\sqrt{2}}\left(\begin{array}{cccccccc} 
\\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & -1 & 0
\end{array}\right)
\end{array}\right.
$$

### 2.4 Sets of universal quantum gates

As just explained, given a family of quantum gates $\left\{U_{i}\right\}_{i \in I}$, new gates can be generated from them by taking composites and tensor products.

The point is that, by doing this, it is possible to obtain all quantum gates from just a subset of them.

This already happens in classical computing, where both the NOR and NAND gates alone are enough to generate the rest of the gates. However, there is an important difference between the classical, and the quantum case. While in classical computation one gate is enough to generate every other gate, in quantum computation an infinite subset of gates is needed. This is because there are a continuous infinity of quantum gates.

Any subset of quantum gates such that every other quantum gate can be obtained from it by taking composites and tensor products in any possible way is called universal. In fact, the above Z-Y decomposition of a single quantum gate shows that the rotations around the $y$ and $z$ axes are universal for the set of single qubit gates. Of course, they are not enough to generate every other quantum gate. At least some multiple qubit gate needs to be added to really have a universal set.

There are various sets of quantum gates which turn out to be universal. One of these universal sets consists of all single qubit gates (or just those corresponding to rotations around the coordinate axes) together with the CNOT gate (see [18], § 4.5.2). However, this universal set has two important drawbacks. Firstly, it consists of a continuous infinity of gates, while physical quantum processors usually are able to implement only a few number of gates. Secondly, no straightforward method is known to implement all these gates in a way which is resistant to errors. Because of this, one usually works with a "universal" finite set of gates which can be performed in an error-resistant fashion using quantum errorcorrecting codes. I will not get into the problem of quantum error correction in this work, although it is clearly an absolutely relevant aspect of quantum computation. The interested reader is referred to [18].

Obviously, a finite set of gates can not be used to implement an arbitrary gate in an exact way because the set of quantum gates is continuous, and a continuous set of gates can not be generated from a finite one. At most, it will be possible to generate an infinite but discrete set of gates (the technical term is countable set). However, it can be shown that certain finite sets of gates can be used to approximate any unitary gate with any desired precision. One such "universal" finite set is the set known as Clifford+T, which consists of the gates $H, S, C N O T, T$. ${ }^{1}$ For more details, the reader is referred again to (§ 4.5.3).

## 3 The problem of quantum circuit compilation

In the circuit model of quantum computation, quantum algorithms are formulated in terms of the quantum circuits described in the previous section. These circuits are usually known as logical quantum circuits because they disregard the hardware constraints associated with quantum processors, which are the physical devices that execute the circuits. Quantum circuit compilation is the process of adapting a given input circuit to match hardware constraints in the most optimal way for execution on the current generation of noisy quantum devices, called NISQ devices 22.

### 3.1 Synthesis of a quantum circuit

The limitations imposed by quantum processors can be divided into two types: gate set restrictions and connectivity restrictions. The first type of limitation is related to the already mentioned fact that physical quantum computers can execute only a finite number of gates. Consequently, the initial step in executing a logical circuit involves synthesizing its gates using the specific gates supported by the quantum processor. These processor-specific gates are referred to as basis gates.

For instance, in many quantum processors the basis gates consist of the universal set of Clifford+T gates. Therefore, if we want to execute the circuit in Figure 3 we need to first see how both the X-gate, and the $U(\beta, \gamma, \delta)$-gate can be approximated with the Clifford+T gates, ideally in the most efficient way possible. It turns out that it is possible to obtain good approximations to any desired gate using surprisingly short sequences of gates from the given generating set. This is the content of one of the most important results in the theory of quantum computation, the Solovay-Kitaev theorem 12 which was first announced by Robert M. Solovay in 1995, and independently proved by Alexei Kitaev in 1997. This results was later used by Christopher M. Dawson and Michael A. Nielsen to create the Solovay-Kitaev algorithm 77 which runs in $O\left(\log ^{2.71}(1 / \epsilon)\right)$ time, and produces as output a sequence of $O\left(\log ^{3.97}(1 / \epsilon)\right)$ quantum gates which is guaranteed to approximate the desired quantum gate to an accuracy within

[^0]$\epsilon>0$. Over the years, alternative algorithms with better performance have been presented, and it has also been generalised to multiple qubit gates. Some examples can be seen in $2,13,17$. However, I will not go into more details as this is out of the scope of this thesis.

### 3.2 Problem to solve: Quantum circuit layout

In this work I will address the second type of limitation, which is related to the geometry of quantum processors. The architecture of quantum processors can be represented as a graph called the connectivity graph or coupling map of the processor, where each node represents a physical qubit, and the edges represent the connections between them. These connections are crucial because executing a multiple qubit gate requires the underlying qubits to be connected in the device. Similarly, given a logical circuit, we can consider what I will call its qubit interactions graph, whose nodes represent logical qubits and whose edges represent multiple qubit gates. Two nodes are connected in the qubit interactions graph if and only if the underlying qubits in the circuit execute a multiple qubit gate together.

The logical circuit will be directly executable only if we can find an isomorphism between the qubit interactions graph and a subgraph of the connectivity graph. However, in most cases, such an isomorphism does not exist or, if it does, finding it can be excessively time-consuming due to the NP-Completeness of the subgraph isomorphism problem [6]. In such cases, we will need to find what I will call the executable version of the logical circuit for the given processor. It is any circuit that performs the same computation as the logical circuit but that can be directly executed on the quantum processor. It will just differ from the logical circuit in a number of added SWAP gates. A SWAP gate is a two-qubit quantum gate that swaps the states of the two qubits involved. In other words, it allows us to move the position of logical qubits within the physical qubits of the processor. By rearranging the qubits like this every time it is required, we can ensure that any two logical qubits involved in a two-qubit gate are placed on physically connected qubits. Therefore the wires in the executable version can be considered to represent physical qubits. However, using SWAP gates introduces additional overhead. Specifically, a SWAP gate is decomposed into three CNOT gates. Figure 4 shows the SWAP gate symbol, and its decomposition into CNOT gates.


Figure 4: SWAP gate symbol on the left and its decomposition into CNOT gates on the right

In general, even for a given mapping of the logical qubits to the physical qubits of the processor there will be multiple executable versions of the logical circuit, i.e. ways of inserting the necessary SWAP gates to be able to execute the quantum circuit with the given processor, and some solutions will be more optimal than others. Then the goal is to find the most optimal solution or, at least, an approximation to it. However, the solution can be optimal in more than one sense. For instance we can try to optimize the gate count or the circuit depth, and due to parallelization, the optimal solution in each case may not be the same.

In this work, as in most of the literature, I will focus on the optimization of circuit depth. This is because one of the most constraining limitation that quantum computers currently have is their coherence time. Broadly speaking, coherence time refers to the duration in which a computation can be performed before external interactions, primarily with the environment, render the computation useless. Given that quantum computing takes place at the atomic scale, it requires near-complete isolation from any particle that is not involved in the computation. Once external factors disrupt the computation, qubits move to a so called decoherent state. Consequently, it is crucial to complete the computation as quickly as possible, which can only be achieved with optimal circuit depth. The optimization problem consisting of minimizing the circuit depth of the executable version is often called the quantum circuit layout (QCL), and it has been shown to be an NP-Complete problem. For instance, in 30 this is proved by reduction to the subgraph isomorphism problem, in 33 by reducing it to the Hamiltonian cycle problem, and in 4 by reducing it to the SAT problem.

Let us look at an example. Suppose we have a quantum processor whose coupling map is as depicted
in Figure 5. This processor is only capable of executing Clifford+T gates. The goal is to run the logical circuit represented in Figure 6, which has already been described in terms of the basis gates, and has a depth of 4. By examining the qubit interactions graph of the circuit, shown in Figure 6, it is clear that it is isomorphic to no subgraph of the coupling map. The coupling map has only four edges while the qubit interactions graph has five. Thus we will need to add some SWAP gates to made it executable. Figure 8 shows the optimal executable version for our processor. ${ }^{2}$ As the reader may easily check, the qubit interactions graph of this new circuit is exactly the same as the coupling map and hence, it can indeed be directly executed. As shown in the rightmost part of the Figure 8, in this version each logical qubit is trivially mapped to its corresponding physical qubit, except for logical qubits 3 and 4 , which were mapped to physical qubits 4 and 3 , respectively. However, this mapping alone is not enough to execute all the gates. To be precise, the CNOT gates between the logical qubits $q 2$ and $q 3$, on the one hand, and logical qubits $q 0$ and $q 2$, on the other hand, cannot be executed because the physical qubits 2,4 and 0,2 are not connected. Two SWAP gates are needed as shown in Figure 8 . Thus the resulting depth of the circuit, when the SWAP gates are decomposed into three CNOTs, has increased from 4 to 9.


Figure 5: Coupling map of a quantum processor


Figure 6: Logical quantum circuit with depth 4


Figure 7: Qubit interactions graph of the logical circuit 6

## 4 Scope

### 4.1 Previous works on the subject

Although the quantum circuit layout problem is NP-Complete, at the current stage of quantum computing the number of qubits in the processors is still relatively small. In some cases, small enough to allow

[^1]

Figure 8: Executable version of the logical circuit 6 relative to the processor 5
for an exact solution to the problem to be found within a reasonable amount of time. As an example, out of all the quantum processors that IBM has available to the public, eight of them are of 7 qubits or less ${ }^{3}$ Moreover exact approaches are also useful as they provide the baseline for measuring the performance of the proposed non-exact solutions. Therefore, it is important to first acknowledge the efforts made in developing exact methods for the QCL problem. Various solutions have been proposed in this setting, primarily utilizing constraint programming [35, 36, 33] or dynamic programming [30]. However, these approaches have exponential running times. Hence they are only suitable for circuits of approximately 8 qubits or less. Other approaches, such as 34,5 , provide exact solutions but are restricted to specific types of physical qubits.

On the other hand, there is a separate body of research that focuses on approximate solutions to the problem. Some of these approaches employ heuristic techniques (see $[16, \sqrt[14]{2}, 30$ ), and some combine heuristics with machine learning methods, as in 20.

While the aforementioned papers provide solutions to the overall QCL problem, most of the existing literature treats the problem as two different stages: qubit mapping and qubit routing. Qubit mapping consists of finding the best possible mapping of the logical qubits to the physical qubits, i.e. the mapping for which there exists a executable version of minimal depth. The second stage, qubit routing, consists of finding this executable version of minimal depth, i.e the best way of inserting the SWAP gates to make the circuit executable using this mapping.

Regarding the first stage, the methods proposed in [30, 31, 19] use heuristics to estimate the quality of a mapping, and try to improve it with search algorithms. In 1 the mapping is instead predicted by training a neural network. However, this method is constrained by the dataset used for training, which is created using mapping algorithms that are either not scalable (if they employ exact methods) or not sufficiently accurate (if they approximate the solution). Consequently, the neural network will learn mappings no better than the approximate methods used to create the dataset, resulting in improved execution time at best. As for the qubit routing stage, heuristic methods have been proposed, such as [30] or the one employed by Qiskit [14]. Most important for this thesis is the approach followed by [11, 21] who used reinforcement learning techniques. There is also a more recent work [29] using Graph Neural Networks aided by a commonly used algorithm in reinforcement learning, the Monte Carlo Tree Search.

### 4.2 Goals and approach

I approached this thesis as a possibility to do research on two topics that I am very interested about, quantum computing and machine learning. My goal was to learn more about these topics, and hopefully to contribute to the field. As I said previously, I decided to focus my attention in the Quantum Circuit Layout problem. My goal was to explore the possibility of using reinforcement learning to obtain a program that could achieve a better performance than the one used by Qiskit [14], the most popular quantum computing framework.

[^2]To achieve this goal, and considering how the QCL problem can be broken down into two subproblems, I decided to tackle first the qubit mapping subproblem because it seemed easier to solve than the complete problem. This decision was influenced by my limited experience in developing machine learning models, as well as my hardware limitations. Consequently, it seemed prudent to pursue a more manageable problem initially, aiming to achieve favorable outcomes, and subsequently progress towards resolving the complete problem.

In the existing literature on the topic, most of the focus was either on solving the complete problem directly, which lacked scalability, or on addressing one of the two subproblems individually. However, my motivation for trying to develop a unified and scalable algorithm stemmed from the belief that it could facilitate the establishment of a dynamic "feedback" mechanism within the algorithm. This mechanism would enable the exchange of information between the qubit mapping and routing stages to be exploited for further improvements. Specifically, the mapping stage could be refined based on the insights and requirements attained from the routing stage, and conversely, the routing stage could be adjusted based on the outcomes and challenges encountered during the mapping process. Few papers attempted to provide a scalable solution for the complete problem, and to the best of my knowledge, only one publication 20 utilized machine learning techniques. However, this particular approach employed machine learning to learn a pre-invented heuristic rather than directly learn to obtain the quantum circuit layout. Consequently, its performance was limited to the effectiveness of the learned heuristic. In my research, I aimed to explore the feasibility of training a machine learning model to directly learn to find quantum circuit layouts. Despite my limited familiarity with reinforcement learning techniques, I chose this approach for three reasons. Firstly, I found that alternative machine learning methods such as supervised and unsupervised learning were not suitable for this problem. Supervised methods required creating a dataset, which would be unfeasible for medium to large circuits due to the non scalability of the exact methods. In the case of unsupervised methods, I did not believe they would provide significant assistance. Secondly, the nature of reinforcement learning algorithms themselves appealed to me. Unlike neural networks that would predict the layout directly, reinforcement learning agents can construct the layout incrementally, using learned strategies. For example, the agent could prioritize mapping qubits with the maximum number of connections to those executing the highest number of CNOT operations. Learning this type of strategies intuitively seemed more manageable than generating the entire layout in a single step. Finally, the idea of using RL seemed a good strategy because, as mentioned before, they have already been successfully applied in [11, 21] for the qubit routing subproblem.

My biggest concern was that there was a real chance that my available hardware would not be enough to successfully train a model for such a problem within a reasonable timeframe while attaining satisfactory outcomes. Particularly, this concern stemmed from the very large state space associated with the problem (I provide more details later on) as well as my intention to employ reinforcement learning techniques that are primarily CPU-bound. In contrast with other machine learning techniques such as deep learning, which can be significantly accelerated using GPUs, in reinforcement learning, the agentenvironment interaction is done in CPU (more on that in the later sections). The following are the specifications outline the computer I had at my disposal for this project:

- 11th Gen Intel® Core ${ }^{T M}$ i $7-11700 @ 2.50 \mathrm{GHz}$
- Intel Rocket Lake GT1 GPU
- 16,0 GiB RAM memory
- 1,0 TB SSD
- Ubuntu 22.04.2 LTS OS

To overcome this my only alternative would be to impose certain limitations that would effectively reduce the problem's complexity. In case this was not enough, then I would instead focus more on the theoretical aspects and provide solutions that could be shown to work, so that if in the future more powerful hardware was available, the algorithms could be tested.

## 5 Time planning

### 5.1 Description of tasks

The present thesis corresponds to 18 European Credit Transfer and Accumulation System (ECTS) credits, which translates to an estimated workload of 540 hours. The project started on February 13th and will conclude with the thesis defense in the final week of June, allowing for a total of 20 weeks for allocate tasks.

During the initial phase of this project, the problem at hand was clearly identified, yet the specific approach to tackle it had not been fully defined. The primary objective of the thesis was to conduct extensive research on the QCL problem, evaluate various potential RL solutions, and subsequently determine the most suitable approach. As a result, the planned tasks at this stage are inherently abstract, as the level of detail in the solution's implementation cannot be determined precisely.

### 5.1.1 Identified tasks

In the following list I depict all the identified tasks structured in four blocks:

- Project management. This set of tasks will be undertaken to ensure the smooth execution of the research project. The tasks are:
- Context and scope (T1). This task involves defining the context, scope and objectives of the projects as well as the methodology and resources that will be used. It doesn't depend on other tasks, but it is a prerequisite for the successful completion of all subsequent tasks.
- Time planning (T2). This task involves identifying all the tasks, allocating them the appropriate amount of time and scheduling them along the duration of the project. It depends on the completion of the previous Context and scope task.
- Budget and sustainability (T3). This task involves identifying the financial resources required for the project and assessing the environmental impact of the research. It depends on the completion of the previous Time planning task
- Final document (T4). The completion of this task requires the integration of the three tasks mentioned previously, thereby establishing a dependency on their successful completion.
- Meetings (T5). Regular meetings with the thesis advisor will be scheduled throughout the project to ensure that progress is being made and to receive feedback. Although the Meetings task does not have an explicit dependency, progress in the project is necessary in order to have substantive material for discussion in the meetings.
- Documentation (T6). This task involves recording the progress of the research project and generating the final report. It doesn't have dependencies.
- Learning. This set of tasks involves acquiring the necessary knowledge and skills to undertake the research project. They encompass the study of relevant programming languages, algorithms, and theoretical concepts that underpin the thesis. Specifically, the learning tasks include:
- Learn Quantum Computing (T7). This task involves developing an understanding of the foundational concepts and principles of quantum computing, including quantum circuits, quantum circuits compilation, quantum processors and more. It depends on the completion of the Final document task.
- Learn Reinforcement Learning (T8). The objective of this task is to acquire a comprehensive understanding of reinforcement learning algorithms and techniques, with a focus on developing and implementing a model that can optimize the compilation of quantum circuits. It depends on the completion of the Final document task.
- Learn Qiskit (T9). Qiskit is a widely-used open-source quantum computing software development kit. This task involves learning how to use Qiskit to implement quantum circuits and perform simulations, which will be essential for testing the reinforcement learning model and evaluating its performance. It depends on the completion of the Learning Quantum Computing task.
- Learn TensorFlow (T10). This task involves developing a proficient level in TensorFlow, a widely-used open-source machine learning framework. TensorFlow will be used to build and train the reinforcement learning model. It depends on the completion of the Learn Reinforcement Learning task.
- Model development. The Model Development tasks form the core of the research project and involve the creation and refinement of a reinforcement learning model for quantum circuit compilation. Although the Learning tasks are expected to be carried out throughout the project, partial completion of these tasks is necessary to establish a sufficient basis for tackling Model development tasks. However there is no explicit dependency between them as additional learning during the model development phase is anticipated. This section contains a first task that will only be done once, and then, following the DevOps methodology, the remaining tasks will be repeated in three sprints. In each sprint an update of the model will be performed. This is the full list of tasks:
- Establish baseline performance metrics (T11). This task involves defining the metrics to be used to evaluate the performance of the model. A baseline performance level will be established to provide a benchmark for comparison throughout the development process. This baseline will be based on the current best algorithm's performance metrics. Having a partial dependency with the Learning tasks, it also inherits all of their dependencies.
- Plan the model (T12). In this task, the architecture and design of the model will be planned. It depends on the completion of the Establish baseline performance metrics task.
- Implement the model (T13). This task involves writing the code to create the model in Python, using the planned architecture and design. It depends on the completion of the Implement model task.
- Train the model (T14). This task involves tuning the hyperparameters to optimize performance. The training process will be iterative, with the model being adjusted and refined based on the results of each training run. It depends on the Train the model task.
- Test and compare performance (T15). In this task, the trained model will be tested in various scenarios (different from those used during training) to assess how good the model generalizes. Then these results will be compared to the established baseline metrics to determine the performance of the model. If it is not satisfactory then I will go back to planning the model, and assess the adjustments that are needed in the model to improve performance. After that I will implement the changes and train and test again. This loop will continue until a satisfactory performance is achieved. This task depends on the completion of the Train the model task.
- Thesis defense. This section represents the final stage of the research project, where the results and findings of the study will be presented and defended. The tasks are:
- Speech preparation (T16). In this task, the content of the thesis defense presentation will be prepared. It depends on the completion of all the previous tasks, namely the Project management, Learning and Model development sets of tasks.
- Slides (T17). This task involves creating the visual aid that will be used during the thesis defense. The slides will be designed to clearly and effectively convey the key points of the speech, using appropriate visuals and graphics to enhance understanding and engagement. It depends on the completion of the Speech preparation task.


### 5.1.2 Workload estimates per task

In this section I will give the estimated workloads per task in number of hours, which in total add up to 540 hours. The estimations are given in Table 1. I have also incorporated the dependencies mentioned in the text to make them easier to visualize.

|  | Task ID | Tasks | Estimated hours needed | Dependencies | Resources used |
| :---: | :---: | :---: | :---: | :---: | :---: |
|  | T1 | Context and scope | 25 | - | PC, overleaf |
|  | T2 | Time planning | 20 | T1 | PC, overleaf spreadsheet |
|  | T3 | Budget and sustainability | 15 | T2 | PC, overleaf |
|  | T4 | Final document | 15 | T3 | PC, overleaf, spreadsheet |
|  | T5 | Meetings | 10 | - | - |
|  | T6 | Documentation | 70 | - | PC , overleaf |
|  | T7 | Learn Quantum Computing | 45 | T4 | PC |
|  | T8 | Learn Reinforcement Learning | 50 | T4 | PC |
|  | T9 | Learn Qiskit | 50 | T7 | PC |
| $\stackrel{\rightharpoonup}{V}$ | T10 | Learn TensorFlow | 50 | T8 | PC |
|  | T11 | Establish baseline performance metrics | 2 | T4 | PC |
|  | T12 | Plan the model | 45 | T11 | PC |
|  | T13 | Implement the model | 53 | T12 | PC |
|  | T14 | Train the model | 35 | T13 | PC |
|  | T15 | Test and compare performance | 30 | T14 | PC |
|  | T16 | Speech preparation | 15 | T6, T15 | PC |
|  | T17 | Slides | 10 | T16 | PC |
|  | - | Total | 540 | - | - |

Table 1: Workload estimates in hours and task dependencies

| Task ID | Tasks | Estimated hours needed |
| :---: | :---: | :---: |
| T7 | Learn Quantum Computing | 30 |
| T8 | Learn Reinforcement Learning | 65 |
| T9 | Learn Qiskit | 35 |
| T10 | Learn TensorFlow | 65 |

Table 2: Workload deviations from the initial plan

### 5.2 Time planning update

There were no major deviations in the planned tasks and their allocated hours. However, there was a slight change that concerned tasks T7, T8, T9, and T10. The tasks T7 and T9 turned out to be less challenging than than originally expected while T8 and T10 took longer than initially anticipated. Consequently, the hours needed for theses tasks changed, but despite these deviations, the total time spent on all four tasks remained unchanged from the original estimation. Table 2 shows the updated workload for these tasks. The rest of the estimates had no deviations.

## 6 Basic concepts of Reinforcement Learning

### 6.1 General idea

Reinforcement learning is a subfield of machine learning that focuses on learning through interaction in order to accomplish a specific goal. The learner is called the agent and the thing it interacts with, the environment. Figure 9 illustrates the interaction between an agent and the environment. Through a series of discrete time steps $t=0,1,2,3, \ldots$ the agent obtains a representation of the environment's state $s_{t}$ and chooses an action $a_{t}$. At the next time step, the agent receives a numerical reward $r_{t+1}$, as a result of its chosen action and the new state $s_{t+1}$ resulting from applying the action. The environment's response concludes the time step.


Figure 9: The interaction between the agent and the environment in a MDP.
In general, neither the reward $r_{t+1}$ nor the new state $s_{t+1}$ are completely determined by the current state $s_{t}$ and the chosen action $a_{t}$. Instead we have probability distributions for both things. Technically speaking, they are random variables.

Although the agent-environment interaction does not need to have an end, when it does, the sequence of all the traversed states and actions performed from the initial state to the terminal state (i.e. the state the agent has reached when the interaction concludes) is called an episode.

### 6.2 Mathematical description as a Markov Decision Process

The problem of learning trough interaction is mathematically described as a finite Markov Decision Process (MDP). Formally, a finite MDP is a tuple $\langle\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma\rangle$ where

- $\mathcal{S}$ is a finite set of states (the possible states of the environment);
- $\mathcal{A}$ is a finite set of actions (the possible actions of the agent);
- $\mathcal{P}=\left(p_{a}\left(s, s^{\prime}\right)\right)$ is the state transition probability matrix, where $p_{a}\left(s, s^{\prime}\right)=\operatorname{Pr}\left(s_{t+1}=s^{\prime} \mid s_{t}=\right.$ $\left.s, a_{t}=a\right)$ denotes the probability that action $a$ in state $s$ at time $t$ will lead to state $s^{\prime}$ at time $t+1$;
- $\mathcal{R}=\left(r_{a}\left(s, s^{\prime}\right)\right)$ is the so called reward function, where $r_{a}\left(s, s^{\prime}\right)$ is the expected immediate reward received after transitioning from state $s$ to state $s^{\prime}$ due to action $a$;
- $\gamma \in[0,1]$ is a real parameter called the discount factor.

A policy $\pi$ for a given MDP is a probability distribution over actions for each possible state, and it is assumed to only depend on the current state, i.e. it is a function $\pi: \mathcal{A} \times \mathcal{S} \rightarrow[0,1]$ such that

$$
\begin{equation*}
\pi(a \mid s)=\operatorname{Pr}\left(a_{t}=a \mid s_{t}=s\right) \tag{6.1}
\end{equation*}
$$

Solving an MDP, and the agent's goal, is to find the best policy, defined as the policy that maximizes the cumulative reward in the long run. More formally, if the sequence of rewards received after time step $t$ is denoted as $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$, then what the agent wants to maximize is the so called discounted expected return $G_{t}$ defined by

$$
\begin{equation*}
G_{t}:=r_{t+1}+\gamma r_{t+2}+\gamma^{2} r_{t+3}+\ldots=\sum_{k=0}^{\infty} \gamma^{k} r_{t+k+1} \tag{6.2}
\end{equation*}
$$

The discount factor is introduced to avoid infinite values in case $t$ goes to infinity, and because immediate rewards may be more relevant than future rewards. Nevertheless, in cases were all sequences terminate we can choose to set $\gamma=1$ to get an undiscounted expected return.

How is the best policy found? Given a policy $\pi$, we can define the state-value function $v_{\pi}: \mathcal{S} \rightarrow \mathbb{R}$ and the action-value function $q_{\pi}: \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. The former estimates how good it is for the agent to be in a given state, while the latter estimates how good it is for the agent to perform a given action in a given state. In both case goodness is measured by the expected value of $G_{t}$ for a given state $s$ or a given pair $(s, a)$ at time $t$, respectively. More formally

$$
\begin{gather*}
v_{\pi}(s):=\mathbb{E}_{\pi}\left[G_{t} \mid s_{t}=s\right]  \tag{6.3}\\
q_{\pi}(s, a):=\mathbb{E}_{\pi}\left[G_{t} \mid s_{t}=s, a_{t}=a\right] \tag{6.4}
\end{gather*}
$$

where $\mathbb{E}_{\pi}$ denotes the expected value with respect to policy $\pi$. Both functions satisfy a recursive relationship obtained by thinking of $G_{t}$ as the sum of the two terms $r_{t+1}$ and $\gamma r_{t+2}+\gamma^{2} r_{t+3}+\cdots$, and replacing the second term by its expected value. When this is done, we get

$$
\begin{gather*}
v_{\pi}(s)=\mathbb{E}_{\pi}\left[r_{t+1}+\gamma v_{\pi}\left(s_{t+1}\right) \mid s_{t}=s\right]  \tag{6.5}\\
q_{\pi}(s, a)=\mathbb{E}_{\pi}\left[r_{t+1}+\gamma q_{\pi}\left(s_{t+1}, a_{t+1}\right) \mid s_{t}=s, a_{t}=a\right] \tag{6.6}
\end{gather*}
$$

Equations (6.5) and 6.6) are known as the Bellman equations.
It follows from the definition (6.3) of $v_{\pi}$ that the best policy is found as the policy for which $v_{\pi}$ is maximum in each state. More precisely, an ordering over policies can be defined as

$$
\begin{equation*}
\pi \geq \pi^{\prime} \Longleftrightarrow v_{\pi}(s) \geq v_{\pi^{\prime}}(s) \quad \forall s \in \mathcal{S} \tag{6.7}
\end{equation*}
$$

Although this is only a partial ordering (not any two policies $\pi, \pi^{\prime}$ are comparable), it can be shown that there will always exists a greatest element, i.e. a policy $\pi_{*}$ such that $\pi_{*} \geq \pi$ for every policy $\pi$. Solving the MDP means finding this optimal policy $\pi_{*}$. The optimal state-value and action-value functions, $v_{*}$ and $q_{*}$ respectively, are then defined as

$$
\begin{align*}
v_{*}(s) & :=\max _{\pi} v_{\pi}(s),  \tag{6.8}\\
q_{*}(s, a) & :=\max _{\pi} q_{\pi}(s, a) \tag{6.9}
\end{align*}
$$

There can be multiple optimal policies but all of them will share these optimal state-value and actionvalue functions. It can be shown that these $v_{*}$ and $q_{*}$ satisfy the following Bellman optimality equations

$$
\begin{align*}
v_{*}(s) & =\max _{a}\left(r_{a}(s)+\gamma \sum_{s^{\prime}} p_{a}\left(s, s^{\prime}\right) v_{*}\left(s^{\prime}\right)\right)  \tag{6.10}\\
q_{*}(s, a) & =r_{a}(s)+\gamma \sum_{s^{\prime}} p_{a}\left(s, s^{\prime}\right) \max _{a^{\prime}} q_{*}\left(s^{\prime}, a^{\prime}\right) \tag{6.11}
\end{align*}
$$

Notice that 6.10 is a system of equations, one for each possible state, so if there are $n$ states, then it is a system of $n$ nonlinear equations with $n$ unknowns variables, and it can be shown to have a
unique solution. If the transition probabilities matrix $\mathcal{P}$ is known, then in principle one can solve this system using the methods for solving systems of nonlinear equations. The same is true for the system of equations (6.11). Once $v_{*}$ is known, it is relatively easy to determine an optimal policy. For each state $s$, there will be one or more actions at which the maximum is obtained in the Bellman optimality equation. Then any policy that assigns nonzero probability only to these actions is an optimal policy. For more details, the reader is referred to 32], Chapter 3.

### 6.3 Model-based and Model-free RL algorithms

There are two main types of reinforcement learning algorithms, that is, algorithms that solve an MDP. On one hand there are the model based algorithms, which require a complete description of the environment (i.e. of the MDP), that they will use to plan ahead the best course of action. A very common use case for these algorithms is the game of chess, for which we have that:

- The set of all the possible position on the board forms the state space
- The set of all moves that the pieces can make forms the action space. Although not all the moves will be valid in a given position, the rules can easily be coded into the environment.
- Given a position on the board (state) and a move (action) it is perfectly defined what the resulting state will be.
- The outcome of the game can only be a draw or a win of one of the players.

Therefore, all the elements of the MDP can be defined, or in other words, the agent has complete knowledge of the environment.

On the other hand there are the model free algorithms, which do not require a complete description of the MDP, and they will instead learn from experience, without any planning. For instance, let us consider a grid that represents a maze. This grid has one starting cell and one exit cell. Moreover, the rest of the cells can either be empty or contain a wall. Begining at the starting cell, the agent's goal is to reach the exit cell. The set of possible actions is to move one cell up, down, left or right. If the action would place the agent on a cell that contains a wall or outside the grid, then it will remain in the same cell, otherwise the agent moves one cell in the direction given by the action. Considering that each grid cell is a different state, it is not possible to determine the resulting state after performing the action since the agent does not know whether it will bump into a wall or not. Therefore if we frame the problem as an MDP the state transition probability matrix $\mathcal{P}$ would be unknown and hence planning ahead is not possible. Instead the agent will have to play the game repeatedly and use the results to learn from them.

## 7 Qubit mapping problem as a finite MDP

To solve the qubit mapping problem by applying reinforcement learning it is first necessary to frame it as a finite MDP. In general, there might be more than one MDP that can solve a given problem, such as finding the optimal mapping in my case. In this work, I will describe two distinct MDPs that I have used which differ only in the rewards. States, actions, probability transitions, rewards, and the discount factor in these MDP's are as follows.

- A state is a tuple $s=(q c, p, m)$ where $q c$ is a logical quantum circuit, $p$ the processor where the quantum circuit must be executed, and $m$ a mapping of the logical qubits to the physical qubits.
- There are three types of actions: permutation, assign or do nothing. The permutation action consists of permuting the physical qubits assigned to two logical qubits while the assign action consists of reassigning a logical qubit to a new physical qubit that was previously unassigned. Note that in both actions the effect is to simply change the "content" of two physical qubits.
- In this problem the new state $s^{\prime}$ is completely determined by the pair $(s, a)$. Hence we have a perfectly defined function $s^{\prime}(s, a)$, where $s^{\prime}(s, a)$ denotes the state resulting from applying action $a$ to state $s$ and differs from $s$ only in the third component. Then the state transition probabilities are given by

$$
p_{a}\left(s, s^{\prime}\right)= \begin{cases}1, & \text { when } s^{\prime}=s^{\prime}(s, a) \\ 0, & \text { otherwise }\end{cases}
$$

- By the depth of a state $s=(q c, p, m)$ (not to be confused with the depth of the logical circuit $q c$ ) I will mean the minimal depth of the executable versions of $q c$ when the mapping of the logical qubits to the qubits of the processor $p$ is $m$. Then in order to distribute rewards I used the depths of the states. However, computing the depth of a state is known to be an NP-Hard problem. ${ }^{4}$ Therefore, I instead worked with state depths computed by a heuristic algorithm (the algorithm is described in $[14)$. In fact, as pointed out before, I used two different definitions of the rewards in terms of these heuristic depths. In the first case, the reward of going from state $s$ to state $s^{\prime}$ as a consequence of doing action $a$ will be zero except when $s^{\prime}$ is the terminal state, in which case it is given by

$$
\begin{equation*}
r_{a}^{(1)}\left(s, s^{\prime}\right)=d(q c)-h d\left(s^{\prime}\right) \tag{7.1}
\end{equation*}
$$

where $d(q c)$ denotes the depth of the logical circuit $q c$ (constant over all the episode), and $h d(s)$ the heuristic depth of the state $s$ for any state $s$. In the second case, rewards are basically given by the difference between the previous state's heuristic depth and the current state's heuristic depth. Additionally, to incentivize the agent to take the fastest path to the solution, a constant reward of -0.5 will be added after every step the agent takes, except when the chosen action is the do-nothing action. Thus in the second case we have

$$
\begin{equation*}
r_{a}^{(2)}\left(s, s^{\prime}\right)=h d(s)-h d\left(s^{\prime}\right)-0.5 . \tag{7.2}
\end{equation*}
$$

- The discount factor $\gamma$ in this problem is set to one.

Let me make some comments to this description. Firstly, the possible processors $p$ are assumed to have a given set of basis gates, and only logical quantum circuits $q c$ with these basis gates are considered (as said in $\S 3.1$, I will not get into the synthesis problem). In my case I have used the rotations around $\mathrm{X}, \mathrm{Y}$ and Z axes plus the CNOT gate as basis gates. Secondly, in general not all logical qubits need to be involved in every time-step of the execution. For instance, it may happen that no gate acts on a given logical qubit until time-step $t=4$. In this sense, the mapping $m$ need not be injective. However, considering non-injective mappings would imply taking into account time, making the description of the mapping $m$ more complex. Moreover, the library I have used to manipulate quantum circuits in Python, Qiskit, does not admit non-injective mappings. Therefore $m$ will be assumed to be injective. Consequently, the number of logical qubits will always be at most equal to the number of physical qubits. Moreover, this assumption has the advantage that the number of possible states is much lower. Finally, since we want the MDP to be finite, a maximum number of qubits in the processor $p$ has to be set, as well as a maximum depth for the quantum circuit $q c$. Let us suppose that the maximum number of qubits in the processor is $K$ (consequently, the maximum of qubits in the quantum circuit is also $K$ ). Then the set $\mathcal{S}$ will be the set of all possible tuples consisting of any logical quantum circuit with up to $K$ qubits, any type of processor with up to $K$ qubits, and any possible mapping $m$, and the set $\mathcal{A}$ of all possible actions will have a size of at most $K \cdot(K-1) / 2+1$ actions. Notice that the possible actions from a given state depend on the actual number of qubits of the processor. Moreover, in the worst case the optimal mapping will take at most $K-1$ actions to be reached. 5 Therefore we can impose a maximum number of allowed actions to $K-1$. This way we ensure that the expected return will not go to infinity even if $\gamma$ is set to 1 .

Both MDP's indeed provide a solution to the qubit mapping problem. The key point is that, with the defined reward functions, the cumulative reward is in the first case

$$
\begin{equation*}
r_{1}^{(1)}+r_{2}^{(1)}+\cdots+r_{t}^{(1)}=d(q c)-h d\left(s_{t}\right) \tag{7.3}
\end{equation*}
$$

and in the second

$$
\begin{align*}
r_{1}+r_{2}+\cdots+r_{t} & =\left(\left[d\left(s_{0}\right)-d\left(s_{1}\right)\right]+\left[d\left(s_{1}\right)-d\left(s_{2}\right)\right]+\cdots+\left[d\left(s_{t-1}\right)-d\left(s_{t}\right)\right]\right)-0.5 t  \tag{7.4}\\
& =\left(d\left(s_{0}\right)-d\left(s_{t}\right)\right)-0.5 t \tag{7.5}
\end{align*}
$$

In the first case it is clear that the cumulative reward is maximum when the heuristic depth of the terminal state is minumum, as desired. In the second case, it readily follows from 7.5 that the cumulative reward

[^3]for a given initial state $s_{0}$ will be maximum when $d\left(s_{t}\right)+0.5 t$ is minimum. Since both terms are positive, this happens when $s_{t}$ is optimal (minimal depth), and the number $t$ of time-steps is minimum. Thus in this second case we have proved not only that the optimal policy leads to the optimal mapping but also that it is reached in the minimum number of time-steps. Nevertheless, the solution will be only approximate because the MDP works with heuristic depths, not the real depths of the states.

### 7.1 Exact methods

Given that the qubit mapping problem can be completely framed as an MDP (provided in the previous subsection), it seems reasonable to apply a model based algorithm to solve it. Initially, exact methods were explored, aiming to find the optimal policy. These methods use dynamic programming and they consist of two stages that are repeated until convergence (i.e. until the optimal policy is found), namely policy evaluation and policy improvement. In the policy evaluation stage, the algorithm iteratively calculates the state-value function for an arbitrary policy on each state by applying equation (6.5), whereas in the policy improvement stage a better policy $\pi^{\prime}$ is obtained by acting greedily with respect to the value function. This new policy $\pi^{\prime}$ will be given by ${ }^{6}$

$$
\pi^{\prime}(a \mid s)= \begin{cases}1, & \text { if } a=\underset{a}{\operatorname{argmax}} q_{\pi}(s, a),  \tag{7.6}\\ 0, & \text { otherwise }\end{cases}
$$

where $\pi$ is the current policy. $\pi^{\prime}$ is guaranteed to be such that $\pi^{\prime} \geq \pi,{ }^{7}$ and it will only be $\pi^{\prime}=\pi$ when they are both equal to $\pi_{*}$.

Although these exact methods guarantee a convergence to an optimal policy (see [8], Chapter 2), they become unfeasible for problems with large state spaces, which is the case of the MDPs described before for the qubit mapping problem. Indeed, let be $N P\left(n_{p}\right)$ the number of possible processors with $n_{p}$ qubits, $N Q C\left(n_{q c}, d\right)$ the number of possible logical quantum circuits with $n_{q c}$ qubits and depth $d$ (in the given basis gates), and $M\left(n_{q c}, n_{p}\right)$ the number of possible mappings from the set of logical qubits to the set of physical qubits. Clearly, the total number $\operatorname{TNS}\left(n_{q c}, n_{p}, d\right)$ of possibles states ( $q c, p, m$ ) for a number $n_{p}$ of logical qubits, a number $n_{q c} \leq n_{p}$ of physical qubits, and a depth $d$ for the logical quantum circuits is given by the product

$$
\begin{equation*}
T N S\left(n_{q c}, n_{p}, d\right)=N P\left(n_{p}\right) \cdot N Q C\left(n_{q c}, d\right) \cdot N M\left(n_{q c}, n_{p}\right), \tag{7.7}
\end{equation*}
$$

and for a maximun number $K$ of physical qubits, and a maximum depth $D$ for the logical circuits the size $|\mathcal{S}|$ of the state space for the MDP described before is given by

$$
\begin{equation*}
|\mathcal{S}|=\sum_{d=1}^{D} \sum_{n_{p}=1}^{K} \sum_{n_{q c}=1}^{n_{p}} T N S\left(n_{q c}, n_{p}, d\right) \tag{7.8}
\end{equation*}
$$

Now, restricting to processors whose connectivity graph is connected, Cayley's formula for the number of trees on a given number of vertices implies that

$$
\begin{equation*}
N P\left(n_{p}\right)>n_{p}^{n_{p}-2} . \tag{7.9}
\end{equation*}
$$

As to the number $N M\left(n_{q c}, n_{p}\right)$ of mappings, it is equal to the number of injections of a set of $n_{q c} \leq n_{p}$ elements into a set of $n_{p}$ elements, which is known to be

$$
\begin{equation*}
M\left(n_{q c}, n_{p}\right)=\frac{n_{p}!}{\left(n_{p}-n_{q c}\right)!} . \tag{7.10}
\end{equation*}
$$

Finally, we have $N Q C\left(n_{q c}, d\right)=N Q C\left(n_{q c}, 1\right)^{d}$, where $N Q C\left(n_{q c}, 1\right)$ is the number of quantum circuits of depth 1. Although in my case the set of basis gates is infinite (it includes all rotations around the coordinate axes), the concrete rotation that has to be executed is completely irrelevant for the optimization problem. Hence I can asume that there are only two basis gates: an U-gate (a rotation),

[^4]and the CNOT gate. Then the number of possible depth 1 circuits with no single gate and $r$ CNOT gates is given by
\[

$$
\begin{equation*}
\binom{n_{q c}}{2} \cdot\binom{n_{q c}-2}{2} \cdots\binom{n_{q c}-2(r-1)}{2}=\frac{n_{q c}!}{(2!)^{r} \cdot\left(n_{q c}-2 r\right)!} . \tag{7.11}
\end{equation*}
$$

\]

The binomial coefficient $\left(\begin{array}{c}n_{q c}-2(i-1)\end{array}\right)$ for each $i=1, \ldots, r$ is the numbers of ways of chosing on which qubits the $i^{t h}$ CNOT gate acts. In case $2 r<n_{q c}$ we should add to 7.11 the number of ways the remaining $n_{q c}-2 r$ qubits can be filled with U-gates. To simplify I will not take into account these possibilities in the computation. To obtain the total number of possible circuits of depth 1 we further need to sum over all possible values of $r$. Thus we have

$$
\begin{equation*}
N Q C\left(n_{q c}, 1\right)>\sum_{r=0}^{n_{q c}^{\prime}} \frac{n_{q c}!}{(2!)^{r} \cdot\left(n_{q c}-2 r\right)!}>\frac{n_{q c}!}{(2!)^{n_{q c}^{\prime}}}, \tag{7.12}
\end{equation*}
$$

where $n_{q c}^{\prime}=\left\lfloor n_{q c} / 2\right\rfloor$ is the greatest integer less than or equal to $n_{q c} / 2$. Therefore the size $|\mathcal{S}|$ of the state space for the MDP described before is

$$
\begin{equation*}
|\mathcal{S}|>\sum_{d=1}^{D} \sum_{n_{p}=1}^{K} \sum_{n_{q c}=1}^{n_{p}} n_{p}^{n_{p}-2} \frac{n_{p}!}{\left(n_{p}-n_{q c}\right)!}\left(\frac{n_{q c}!}{(2!)^{n_{q c}^{\prime}}}\right)^{d}>K^{K-2} \cdot K!\cdot\left(\frac{K!}{(2!)^{K^{\prime}}}\right)^{D} \tag{7.13}
\end{equation*}
$$

with $K^{\prime}=\lfloor K / 2\rfloor$ is the greatest integer less than or equal to $K / 2$. It follows that $|\mathcal{S}|$ increases very fast with $K$ and $D$. For instance, if $K=4$ and $D=3$ the number of possible states is already greater than 330.000 .

In this cases, the best we can do is to approximate the solution, and here is where deep reinforcement learning comes into play. These techniques parameterize either the state-value function, the policy or both using a weight vector, thus enabling the application of gradient-based optimization methods.

### 7.2 First approximate method: DNN aided with MCTS

The first approach I decided to implement was the Monte Carlo Tree Search (MCTS) algorithm combined with a deep neural network DNN. It is a model-based method that uses MCTS for planning and the DNN to approximate the value function and the policy. This method has its origins in AlphaGo [27, an AI that was trained to play the game of Go. AlphaGo used two neural networks, one to approximate the policy and another one to approximate the value function.

Building upon AlphaGo's achievements, AlphaGo Zero was introduced a year later 28. It was also trained to play Go, and defeated its predecessor, AlphaGo, with a result of 100-0 to AlphaGo Zero. This new AI introduced two significant changes compared to AlphaGo. Firstly, it merged the two neural networks into a single network capable of simultaneously providing the policy and value function estimations. Secondly, it adopted a tabula rasa approach, meaning that it did not rely on any human-generated data, guidance, or domain knowledge beyond the rules of the game.

In the same year, the AlphaZero framework was developed [26], further expanding the algorithm's capabilities to games such as Chess and Shogi, in addition to Go. AlphaZero successfully surpassed the performance of the best computer engines available at the time in all three games. The remarkable achivements of these AI engines made me believe even more that, once properly adapted, this method can provide good results for my problem.

In this subsection, I provide a general overview of the algorithm as presented in the AlphaZero paper 26], and then I discuss the adaptations I made in order to apply it to my problem.

### 7.2.1 AlphaZero

As previously mentioned, this method uses a deep neural network $f_{\theta}$ with parameters $\theta$ that takes a state $s$ as input and has two outputs $(\boldsymbol{p}, v)$. The vector $\boldsymbol{p}$ is a probability distribution vector over all possible actions, corresponding to the policy for state $s$. The value $v$ is a scalar evaluation of the state $s$ corresponding to the state-value function for state $s$. In AlphaZero a residual network architecture [10] was used. Nevertheless, the exact architecture is not inherent to the algorithm and is supposed to
be adapted for each use case. The self-play process utilizes the MCTS algorithm. The search tree is constructed such that each node corresponds to a state, and directed edges connect the nodes $s$ and $s^{\prime}$ if an action $a$ can be taken in $s$ that will take the agent to state $s^{\prime}$. Each edge $(s, a)$ in the tree contains three important pieces of information:

- $N(s, a)$ : the number of times action $a$ has been selected from state $s$, i.e. the visit count.
- $Q(s, a)$ : the mean action value, i.e. the mean of the values obtained from all the searches that traversed edged $(s, a)$.
- $P(s, a)$ : the prior probability of selecting action $a$ from state $s$ as given by the neural network $f_{\theta}$.

Starting with just a root node that represents the initial state $s_{0}$, the tree is expanded by repeatedly applying the three steps depicted in Figure 10 .

1. Selection: starting from the root node, the tree is traversed until a node $s_{T}$ is reached at timestep $T$ that was not previously visited. At each time-step $t<T$ an action is selected according to $a_{t}=\operatorname{argmax}(Q(s, a)+U(s, a))$ where $U(s, a)$ is an upper confidence bound (UCB) defined by

$$
\begin{equation*}
U(s, a)=c_{p u c t} P(s, a) \frac{\sqrt{\sum_{b} N(s, b)}}{1+N(s, a)} \tag{7.14}
\end{equation*}
$$

where $c_{p u c t}$ is a constant that controls the level of exploration.
2. Expansion: each new node is added to the tree as a leaf node and it is expanded by adding all the edges that come out of it, one for each valid action $a$ that can be applied to the state it represents (represented by the new leaf node). The newly added node's state $s$ is evaluated by the neural network to obtain the tuple $(\boldsymbol{p}, v)$. Each newly added edge will be initialized with $N(s, a)=0$, $Q(s, a)=0$ and $P(s, a)=p[a]$.
3. Backup: the value $v$ obtained from the neural network is backpropagated up the tree. For each edge ( $s, a$ ) traversed during the selection step the action value $Q(s, a)$ is updated as follows:

$$
\begin{equation*}
Q(s, a)=\frac{N(s, a) Q(s, a)+v}{N(s, a)+1} \tag{7.15}
\end{equation*}
$$

and the visit count $N(s, a)$ is incremented by 1 .


Figure 10: MCTS algorithm in AlphaZero 28.
This three-step process is repeated for a certain number of simulations, and once it is finished, an action is chosen from the root state $s_{0}$ based on a policy computed using the visit counts:

$$
\begin{equation*}
\pi(a \mid s)=\frac{N(s, a)^{\frac{1}{\tau}}}{\sum_{b} N(s, b)^{\frac{1}{\tau}}} \tag{7.16}
\end{equation*}
$$

where $\tau$ is a so called "temperature parameter" that controls the level of exploration. If we set $\tau \rightarrow 0$ the action will be deterministically selected to the edge with maximum visit count whereas if we set $\tau=1$ edges will be selected proportionally to their visit count.

The entire AlphaZero algorithm is depicted in Figure 11. At first, the neural network is initialized with random weights $\theta_{0}$. Iterations are performed, and in each iteration, several games are played through self-play. Each game will go on for a series of time-steps $t$ and at each time-step an MCTS is executed to determine the policy used to select the next action. The tuple $\left(s_{t}, \pi_{t}\right)$ is stored for the training later on. The chosen action becomes the root node, and the subtree below it is retained, while the remaining part of the tree is discarded. The self-play process continues until the end of the game, at time-step $T$. A reward $r_{T} \in\{+1,0,-1\}$ (win, draw, loss respectively) is assigned based on the game outcome. This reward is then added to the generated tuples such that the final tuple is ( $S_{t}, \pi_{t}, z_{t}$ ), where $z_{t}= \pm r_{T}$ is the game outcome from the perspective of the player to move at time-step $t$. Once all the


Figure 11: AlphaZero algorithm 28.
data is collected from self-play games, it is used to train the neural network. The parameters $\theta$ are updated by gradient descent and the loss function is defined as the sum of the cross-entropy losses between the predicted policy $\boldsymbol{p}$ and the "true" policy $\boldsymbol{\pi}$, and the mean squared error between the predicted value $v$ and the "true" value $z$ :

$$
\begin{equation*}
l=(z-v)^{2}-\boldsymbol{\pi}^{T} \cdot \log \boldsymbol{p}+c\|\theta\|^{2} \tag{7.17}
\end{equation*}
$$

where $c$ is a parameter that adjusts the L2 weight regularization.

### 7.2.2 My algorithm

Since the AlphaZero method is designed in the context of board games, I translated the problem to a game. In this game, the board will be defined by the quantum processor's coupling map, with one cell per node and the edges defining the neighboring cells and the pieces are the logical qubits of the logical quantum circuit.

To simplify the problem I have imposed two additional constraints on the state. Firstly, the processor is assumed to be the same throughout the execution of the program. Secondly, I have assumed that $n_{q c}=n_{p}$, i.e. the number of qubits in the quantum circuit and in the processor are the same. With these assumptions, the processor $p$ is no longer needed in the state (it would be constant in all states) and its connections can be inferred by the neural network from the reward given to each mapping. Moreover,
the mapping $m$ will be bijective and hence the set of actions will now contain only two types of actions, the do-nothing action and the permute action. In the initial states, $m$ could theoretically be any random bijective function, but for clarity purposes we will use as initial states those that have a trivial mapping, meaning $m(q)=q$. For the terminal states, ideally they would only be the states that contain the optimal $m$ for the given circuit and processor, but since this states are unknown, the game will either finish once the player chooses the do nothing action or after a number of moves equal to the number of qubits minus one.

The implications of the previous paragraph to the game are that each cell of the board can only contain one piece. The initial position contains all the pieces trivially placed on the board, i.e. piece one goes to cell one, piece two to cell two and so on and the possible moves are to permute the positions of two pieces on the board or do-nothing.

As for the algorithm itself there are three important differences between AlphaZero and my problem that need to be addressed. First of all, AlphaZero was designed for two-player games, whereas my game is single-player. This actually simplifies the algorithms since there is no need to handle different rewards based on players. Secondly, the issue of repeated states arises. The AlphaZero paper 26 does not explicitly discuss how repeated states are managed (if at all) in the Monte Carlo Tree Search (MCTS). Nonetheless, due to the nature of the games they explored, encountering the exact same position multiple times from different paths is unlikely. However, in my problem, repeated states can occur more frequently. For example, performing the same permutation action twice will always bring you back to the same state, because one permutation cancels out the other. To address this, I implemented the algorithm in a way that, if a state already exists in the tree, instead of creating a duplicate node, the search continues from the existing node. Consequently, and as the third difference, I do not only keep the subtree hanging from the next state, instead I keep all the tree throughout the entire game. Note that on average in the games of Chess, Shogi and Go, each game takes around 40, 80 and more over 200 moves per player respectively, whereas each game in my problem will at most take the number of qubits minus one moves. Since the number of qubits in most quantum processors nowadays is very low (maximum 7 qubits for the freely available QPU from IBM) and there are a lot of repeated states, the size of the tree is much lower and therefore it should be feasible to keep it in memory.

The pseudocode for my version of the MCTS algorithm is shown in Algorithm 1 . All the information of the tree is stored in four dictionaries: $P s, N s, Q s a$ and $N s a$. The dictionaries $P s$ and $N s$ have states as keys. Ps stores an array of length the number of actions as values such that in each position $P[s][a]$ indicates the probability of taking action $a$ from state $s$. On the other hand, $N s$ stores an integer such that $N[s]$ indicates the total number of times the outgoing edges from state $s$ have been visited through the simulations. The $Q s a$ and the $N s a$ dictionaries have pairs state-action $(s, a)$ as keys. In each element $Q s a[(s, a)]$ the $Q(s, a)$ values needed for the formula of the select step are stored, whereas $N s a[(s, a)]$ stores an integer that represents the amount of times action $a$ has been taken from state $s$.

The first step in the search function, which executes the MCTS, is to check if the state is already in the tree. If it is not, it means we have to do the expansion step, which means adding an entry to the $P s$ and $N s$ dictionaries for this state. The probabilities are given by the neural network as previously explained. Then we return the value, also predicted by the neural network, so that it can be backed up. If the state was already in the tree then we check whether the maximum number of allowed actions inside an episode is reached. Recall that an action is represented as an edge in the tree, thus by tracking the depth of the current state's node we can assess if the maximum number of actions has been reached. In case it has, we increment the visit count for state $s$ and we return the reward for having ended the episode in that state. If the episode has not ended yet, we take the best action using the policy from the select step. Note that in this game, all the defined actions are always allowed from any given state, therefore we do not need to check the validity of each action. Once the action is chosen two things can happen. Either the action is the do-nothing action or not. In the former case, we will either create a new entry in the $Q s a$ and $N s a$ dictionaries for this state-action pair in case it did not already exists, or we will simply increment the counter of the $(s, a)$ entry in the $N s a$ dictionary in case it already existed. Note that in the latter case we do not need to update the $Q s a$ value because being all the nodes that are reached from the do-nothing action are leaf nodes, thus there will never be a backpropagated value that affects $Q s a$. Finally we will increment the visit count for that state in the dictionary $N s$ and return the terminal value. In case that the chosen action is not the do-nothing action, we will transition to the new
state and recursively call the search action from the new state to continue the selection process down to a leaf. The function returns the value to be backed up. Once we have that value we will again create or update $\sqrt{7.15}$ the entry $(s, a)$ for both the $Q s a$ and the $N s a$ dictionaries as well as incrementing the visit count for the $s$ on the $N s$ dictionary. The value $v$ is returned in order to continue with the backup.

```
Algorithm 1 Monte Carlo Tree Search
    Initialize:
    Ps \(=\{ \}\) \#Stores probability \(p[a]\) of all edges
    Ns \(=\{ \}\) \#Stores probability visit count of all the nodes
    Qsa \(=\{ \} \#\) Stores Q value of all edges
    Nsa \(=\{ \}\) \#Stores visit count of all the edges
    procedure \(\operatorname{SEARCH}(\mathrm{s}, \mathrm{nn}\), game, depth)
        if \(s\) not in Ps then
            \(\mathrm{Ps}[\mathrm{s}], \mathrm{v} \leftarrow\) nn.evaluate(child.s)
            \(\mathrm{Ns}[\mathrm{s}] \leftarrow 0\)
            return v
        end if
        if depth \(==\) game.maxDepth then
            \(\mathrm{Ns}[\mathrm{s}]+=1\)
            return game.getReward \((s)\)
        end if
        action \(\leftarrow\) getBestAction(s)
        if game.isDoNothing(action) then
            \# Check if the action 'do-nothing' from state s had been taken before
            if \((\mathrm{s}, \mathrm{a})\) in Qsa then
                \(\mathrm{Nsa}[(\mathrm{s}, \mathrm{a})]+=1\)
            else
                \(\operatorname{Qsa}[(\mathrm{s}, \mathrm{a})] \leftarrow\) game.getReward \((s)\)
                \(\mathrm{Nsa}[(\mathrm{s}, \mathrm{a})] \leftarrow 1\)
            end if
            \(\mathrm{Ns}[\mathrm{s}]+=1\)
            return \(\mathrm{Qsa}[(\mathrm{s}, \mathrm{a})]\)
        end if
        \(\mathrm{s}^{\prime} \leftarrow\) game.getNextState \((s, a)\)
        \(\mathrm{v} \leftarrow \operatorname{search}\left(s^{\prime}, n n\right.\), game, depth +1\()\)
        if ( \(\mathrm{s}, \mathrm{a}\) ) in Qsa then
            \(\operatorname{Qsa}[(\mathrm{s}, \mathrm{a})] \leftarrow(\mathrm{Nsa}[(\mathrm{s}, \mathrm{a})] * \operatorname{Qsa}[(\mathrm{~s}, \mathrm{a})]+\mathrm{v}) /(\mathrm{Nsa}[(\mathrm{s}, \mathrm{a})]+1)\)
            \(\operatorname{Nsa}[(\mathrm{s}, \mathrm{a})]+=1\)
        else
            \(\operatorname{Qsa}[(\mathrm{s}, \mathrm{a})] \leftarrow \mathrm{v}\)
            \(\mathrm{Nsa}[(\mathrm{s}, \mathrm{a})] \leftarrow 1\)
        end if
        \(\mathrm{Ns}[\mathrm{s}]+=1\)
        return v
    end procedure
```

In Algorithm 2 we can see how the main algorithm works. The first step is to initialize the neural network model and initializing the data structure that will contain all the training data to an empty list. Then the loop over all the iterations starts. In each iteration we run all the episodes, and each episode consists of a self-play game. Over the course of the game, each move or action is selected by running the mcts algorithm presented above for a certain number of simulations. After the execution of the MCTS we collect the state and the policy, obtained from the MCTS, that is used to determine the next move. Once the game is over we calculate the reward and we add it to all the data generated from that game, such that we will append to the training data list, as many elements as moves in the game and each element contains a tuple with the state, the policy and the final reward of the game. Once all the episodes have been executed we will train the neural network using the collected data and in the next iteration, this updated model will be used to play the games. After finishing with all the iterations, we returned the learned model.

```
Algorithm 2 DNN aided with MCTS
    Input: game
    model \(\leftarrow\) InitializeModel ()
    training_data \(\leftarrow\) []
    for iteration \(\leftarrow 1\) to total_iterations do
        for episode \(\leftarrow 1\) to total_episodes do
            \(\mathrm{s} \leftarrow\) game.InitialState ()
            game_data \(\leftarrow\) [ ]
            mcts \(\leftarrow M C T S()\)
            while not game.isTerminalState(s) do
                for simulation \(\leftarrow 1\) to total_simulations do
                    mcts.search (s,nn, game)
                    end for
                    policy \(\leftarrow\) mcts.getPolicy ()
                action \(\leftarrow\) selectActionBasedOnPolicy(policy)
                new_state \(\leftarrow\) game.ExecuteAction(state, action)
                game_data.append ([s, policy])
                state \(\leftarrow\) new_state
            end while
            reward \(\leftarrow\) game.getReward \((s)\)
            for element in game_data do
                training_data.append \(((x[0], x[1]\), reward \())\)
            end for
        end for
        trainNetwork(training_data)
    end for
    return model
```

To encode the state we need to make sure it always has the same shape once encoded as we cannot input data of different shapes to the neural network. As previously explained, the processor $p$ does not need to be encoded in the state as it is assumed to be always the same. Therefore the state depends only of the quantum circuit $q c$ and the mapping $m$. The encoded state will be condensed into a matrix of shape $(n, d)$ where $n$ is the number of qubits and $d$ is the depth of the circuit (both of them are constant as previously mentioned). Then the element $(i, j)$ of the matrix corresponds to the gate that physical qubit $i$ will execute at depth $j$, assuming that at each depth level the maximum possible number of gates are executed. Note that the function $m$ has already been applied to the logical qubits in $q c$ which is why each row in the matrix represents a physical qubit. In each element of the matrix there can be a CNOT gate, a U-gate or nothing. The latter will be encoded with the number -1 . The integer numbers in the interval $[0, n-1]$ are used to denote CNOTs, so that if in qubit $i$ a CNOT is executed with qubit $k$ at depth level $j$, the element $(i, j)$ of the matrix will contain the number $k$, and the element $(k, j)$ of the matrix will contain the number $i$. Finally, the U-gates will be encoded in the matrix using the number $n$. The logical qubits involved in each CNOT gate need to be mapped to their respective physical qubits. We have already established that the initial state in each episode will be the trivial mapping, whose matrix representation can be easily obtained. For instance, Figure 12 b shows the matrix representation of the quantum circuit from Figure 12a. Since the quantum circuit has five qubits and a depth of four, the matrix representing it has a shape $(5,4)$. Since the mapping is the trivial one, the first wire goes to the first row, the second wire to the second row and so on. Each gate is translated into a number as previously explained. The matrix representation for the initial state, which I will call init_state, is stored in memory during the entire episode and used to obtain the matrix for any given $m$. To be precise the matrix state for a generic mapping $m$ is computed from init_state as follows:

$$
\text { state }[m[i]][j]= \begin{cases}m[\text { init_state }[i][j]], & \text { if init_state }[i][j] \in[0, n)  \tag{7.18}\\ \text { init_state }[i][j], & \text { otherwise }\end{cases}
$$

Indeed, the element $(i, j)$ in the matrix state is equal to $k \in[0, n)$ when at depth $j$ a CNOT between logical qubits $m^{-1}[i]$ and $m^{-1}[k]$ has to be executed and hence, when the entry ( $m^{-1}[i], j$ ) of init_state is equal to $m^{-1}[k] \in[0, n)$. For instance, let us consider the mapping $m$ given by $m=[3,2,4,0,1]$ in which
logical qubit $i$ is mapped to physical qubit $m[i]$. Then the matrix representation is given in Figure 12c

(a) Logical quantum circuit with depth 4

$$
\left(\begin{array}{cccc}
5 & 1 & 2 & -1 \\
-1 & 0 & 4 & 5 \\
3 & -1 & 0 & -1 \\
2 & 5 & -1 & 4 \\
5 & -1 & 1 & 3
\end{array}\right)
$$

(b) Matrix representation of circuit (a) and trivial mapping

$$
\left(\begin{array}{cccc}
4 & 5 & -1 & 1 \\
5 & -1 & 2 & 0 \\
-1 & 3 & 1 & 5 \\
5 & 2 & 4 & -1 \\
0 & -1 & 3 & -1
\end{array}\right)
$$

(c) Matrix representation of circuit (a) an mapping $[3,2,4,0,1]$

Figure 12: Quantum circuit with its matrix representation for two mappings

Regarding the neural network architecture, instead of using Convolutional Networks like in AlphaZero, I decided to first try a simpler network consisting of just a flattening layer at the beginning plus some fully connected layers, without convolutions. In order to find the exact number of hidden layers as well as the units per layer I needed to run the algorithm with multiple configurations to assess which one yielded the best results, but as I discuss in the next section this was not possible.

The full implementation of the algorithm is given in 9 .

### 7.2.3 Discussion of the results

From the first execution it became apparent that my available hardware was far from being powerful enough for executing this algorithm for enough iterations so that the neural network could learn. Not even with the aforementioned simplifications (fixing the processor, same number of qubits in the processor and the circuit and considering the mapping $m$ as bijective), and for a small number of qubits was I able to obtain some results. Since the neural network was initialized to random weights, the predictions at the beginning of the program's execution were not helpful in directing the tree search to the most promising nodes. A lot of exploration had to be done at first in order to collect enough data for the network to start learning a little bit and become helpful in directing the tree search. Unfortunately I was never able to go pass that first stage.

For comparison, according to 26, during AlphaZero's training for the game of Chess, 44 million games were played, with 800 MCTS simulations per move. To play these games, 5000 first-generation TPUs were used making the execution time per move 40 ms . Additionally, the training of the neural network proceeded for 700.000 steps with a mini-batch size of 4096 . For this training, 64 second-generation TPUs were used. Meanwhile, with the hardware I have available $\S 4.2$, for a number of qubits of 5 , a depth of 7 , and a maximum number of moves set to 4 , running the 800 MCTS simulations per move took me 6 min and 13s. The reason for that was not exclusievely because of hardware limitations, but also due to the time it took to compute the reward on a terminal state. Whereas in AlphaZero's games it was nearly immediate to determine the reward of a terminal state based on who had won, in my case, computing the heuristic depth was much harder. Moreover, due to the heuristic behavior, for the same input state it would not always output the same value. Therefore, to make sure that rewards were as consistent as possible, I had to run the heuristic depth multiple times and use the minimum obtained value.

There were a couple more factors, a part from the execution time, that lead to the algorithm's failure. First of all, in contrast to AlphaZero, were the rewards could only be $r \in-1,0-1$, my problem could have a very wide range of values. Once a terminal node was assigned a value according to its reward, if the value was specially big or small, the tree would became very unbalanced. That is because the values assigned to the non-terminal nodes, which were based on the predictions of the neural network would, had values around 0 at the beginning when the neural network had still not been trained. This imbalance would significantly affect the balance between exploration and exploitation in the policy followed in the selection step, as the value of $U(s, a)$ that encourages exploration would become insignificant with the value of $Q(s, a)$ which encourages exploitation. Theoretically, this imbalance should ease through the course of the simulations, however in practice this involved many more simulations to be executed, further slowing the training process. To solve this a normalization of the rewards would have been needed. However, although the maximum value of the reward was obviously 0 (in case no SWAP gates are needed), a lower bound for the minimum was a bit more difficult to get. A possible solution would have been to calculate the worst case in which the involved qubits in each CNOT gate to be executed in the circuit were at maximum distance in the coupling map. However this lower bound was clearly very far away from the real rewards in most cases, and would have caused the reward values after normalization to be concentrated in a very small range of numbers.

Secondly, and more important, was the fact that my problem did not have true terminal states, which I realised that was causing inconsistent values. Technically speaking, and as previously mentioned, the true terminal states would be the optimal states, for which the agent, once fully trained, should choose the do-nothing action. However, in practice this optimal states would not be reached most of the time, which is why I had to introduce a maximum number of actions in one episode. In the cases where the episode was ended with due to the maximum actions being reached, I would calculate the reward of this "false" terminal state, and add it to all the tuples collected on every move. But for all the tuples that belonged to the intermediate moves it was not actually true that their values was the one assigned to them, because if the episode had started in any of these states it would have been able to continue making moves beyond the terminal state that was reached and possibly ended up in an even better state. This could have perfectly been the case for another episode, which means that the values assigned to a state depended on the time-step they were encountered inside an episode, which should not be the case.

### 7.3 Second approximate method: Deep Q-learning (DQL)

The second approach is based on a much more well known algorithm in the reinforcement learning literature. Although at first I did not consider this approach because it is a model-free algorithm, after analysing the flaws of the previous approach I realised that they could be fixed with DQL.

This algorithm is an evolution of the Q-learning algorithm. In Q-learning, the agent interacts with the environment following a certain policy that I will call the collect policy, and keeps a Q-table that stores the $q(s, a)$ values for each state-action pair in the environment. In each interaction of the agent with the environment the agent will collect a tuple ( $s_{t}, a_{t}, s_{t+1}, r_{t+1}$ ) where $s_{t}$ and $a_{t}$ respectively denote the state and the chosen action at time-step $t$, and $s_{t+1}$ and $r_{t+1}$ are the resulting state and the reward at time-step $t+1$, respectively. With these the $\left(s_{t}, a_{t}\right)$ entry of the Q-table will be updated according to the rule

$$
\begin{equation*}
q\left(s_{t}, a_{t}\right) \leftarrow q\left(s_{t}, a_{t}\right)+\alpha\left[r_{t+1}+\gamma \max _{a^{\prime}} q\left(s_{t+1}, a^{\prime}\right)-q\left(s_{t}, a_{t}\right)\right] \tag{7.19}
\end{equation*}
$$

where $\alpha$ is a step size parameter. This equation has been shown to converge under certain assumptions with probability 1 , and if it does it will necessarily converge to $q_{*}$. In fact, from the Bellmann optimality equation (6.11), and assuming a deterministic scenario like my problem, we can see that the term $r_{t+1}+\gamma \max _{a^{\prime}} q\left(s_{t+1}, a^{\prime}\right)$ of the equation is equal to $q_{*}\left(s_{t}, a_{t}\right)$. Therefore we will have finished updating when $q\left(s_{t}, a_{t}\right)=q_{*}\left(s_{t}, a_{t}\right)$. Note that the policy used to collect the data for the update rule, the collect policy, is independent from the policy that is being approximated, called the target policy.

However, when the state and action spaces are too big it becomes unfeasible to keep this Q-table in memory, and instead it is replaced by a neural network called the Deep $Q$-Network (DQN). This neural network will be given a state as input and it will output $q(s, a)$ for each possible action $a$. This version of the Q-learning algorithm is called the Deep Q-learning algorithm and it is abbreviated in the literature as DQL or DQN interchangeably.

For this new approach, I changed the MDP with respect to the previous approach, using now the reward given by 77.2 . With this new reward, calculated after each step inside an episode, there was no longer the issue of assigning the same reward to all the steps of the episode. Furthermore, all the calculated heuristic depths involved in the reward function will be used to train the neural network. In the previous approach, heuristic depths calculated during the MCTS were not directly used as we were only gathering one tuple of data from the entire tree. As previously mentioned, computing the heuristic dephts is the most expansive and time-consuming part, therefore using all the computed values can save up a lot of time. Finally we no longer have issues to manage the exploitation vs exploration trade-off, because in this case this will be determined by the collect policy and the target policy. The former is chosen so that it always explores with a certain probability. i.e. $\epsilon-$ greedy, and the latter is established as greedy. This will not change over the course of the execution keeping the trade-off balanced.

### 7.3.1 My algorithm

As for the the actual implementation of the algorithm, I used the TF-Agents library from tensorflow ${ }^{8}$, which already has an implementation of the agent for the DQL method, called DqnAgent. I defined the environment as a Python environment. ${ }^{9}$ The state was encoded just like in the first approach, therefore a good part of the code could be reused. Here is an step by step overview of how the main algorithm works:

1. Initialize the python environment previously implemented.
2. Create the neural network model. For that I used the QNetwork class from the TF-Agents library.
3. Create and initilaze the DqnAgent.
4. Create the replay buffer. This buffer will store the data collected from the agent's interaction with the environment that will be used to train the neural network. For this I used Reverb ${ }^{10}$ a framework developed by Deep Mind that is compatible with tensorflow and it is efficient, extensible, and easy to use for storing and transporting the data.
5. Run a few episodes in order to get some data into the replay buffer. In the TF-Agent library, this is done with a driver. There are different types of drivers and in my case I used the python driver that works with python environments.
6. Create a dataset from the replay buffer. The replay buffer has a method to create and return a dataset object that will contain data from the buffer and train the neural network with it.
7. Define the main driver to collect data. The reason why the previous one is not used is because when creating the driver we have to specify as a parameter the number of steps it will do to collect data every time it calls its run method. The way the algorithm works is that it first creates a driver that will be set to run a rather high number of initial collection steps to get the replay buffer considerably filled up. Then in the main algorithm loop, every time we train the neural network we will randomly select a batch of data from the replay buffer (much smaller than the initial collection steps). Additionally another driver, configured with a much smaller collection steps parameter, will be used to keep adding some new data into the buffer to ensure that we are not constantly iterating over the same data.
8. Define the main training loop. This loop will be training and collecting new data on every iteration. Like previously mentioned this method works with two policies, the collect and the target policy. The target policy is greedy and uses the neural network to select actions. To keep track of the progress we can evaluate every 50 iterations of the loop the reward obtain over a few episodes using the current target policy. If the agent is learning properly this value should increase as training progresses.

The full implementation of the algorithm is given in 24.

[^5]
### 7.3.2 Discussion of the results

To train the algorithm, like in the previous method, a progressive approach was employed, starting with a simple neural network architecture and gradually increasing its complexity until it reached a point of overfitting. Initially, the architecture consisted of a single fully connected layer with 16 units. However, after 25000 training iterations, the model only learned to consistently choose the do-nothing action as a strategy to avoid penalties. Further training did not result in any policy improvement.

This observation suggests two potential explanations. Firstly, it could indicate that the loss function had become trapped in a local optima, preventing the model from finding more effective policies. However, upon examining the behavior of the loss function, it was still fluctuating considerably, which led to the dismissal of the local optima hypothesis.

The second possibility considered was underfitting, which occurs when the complexity of the function to be approximated surpasses the learning capacity of the chosen architecture. Different architectures were explored, but I continued experiencing the same behavior, with the difference that as the model's complexity increased, the time required to reach the point of consistently selecting the do-nothing action also increased.

It is my believe that better results could be achieved with a more sophisticated network architecture, such as incorporating convolutions similar to the ResNet model used in AlphaZero. However, due to hardware limitations and time constraints, it was not feasible to train such neural network architectures in this study.

Although this model had the advantage of using all the computed heuristic depths during the training of the neural network as discussed earlier, it also had inherent disadvantages associated with model-free algorithms compared to model-based ones. Model-based algorithms incorporate a planning stage that enables more informed decisions at each step, leading to a faster learning process by quickly guiding the search towards promising states. In contrast, model-free algorithms lack this planning step, which can result in longer learning times as the agent gradually discovers the more advantageous moves and navigates towards them. There exists a trade-off between these two approaches, and it remains unclear which one is more efficient. However, considering that the computation of heuristic depth in the previous model-based approach proved to be a significant bottleneck in terms of time efficiency, it was considered valuable to utilize all the available information, even at the expense of a planning step. Nevertheless it is possible that the model-free approach may not offer substantial improvements in efficiency, if any, which would explain why I kept having issues to train the model.

## 8 Final conclusions and future work

The outcomes of this project were not the desired ones, as the proposed methods did not yield satisfactory performances. The reward issue encountered in the first method revealed its inconsistency, making it unlikely to achieve convergence in training, even with powerful hardware and infinite running time. However, the second method resolved the issues of the first one, and I firmly believe that with proper training, promising results could be obtained.

There were two factors that complicated the training process. Firstly, my PC's operating system killed the process that was executing the training in more than one occasion in order to free up resources, significantly slowing down the process. Although I had implemented checkpoints to prevent losing all progress, I had to manually resume the execution whenever such events occurred, which was beyond my control. Despite closely monitoring the PC, for obvious reasons, I could not be vigilant $24 / 7$ to resume training as soon as possible.

The second factor relates to the intrinsic nature of reinforcement learning methods. Unlike traditional supervised methods that train a neural network with a fixed dataset, reinforcement learning constantly adapts its dataset with newly collected data from the agent. Consequently, the loss function experiences significant fluctuations, which reduces the clarity of its trend and makes it challenging to ascertain whether it is consistently decreasing. Typically, the descending trend of the loss function serves as a reliable indicator of the model's effectiveness. However, in this context, a substantial number of additional
iterations were necessary to obtain a reliable assessment.
I was aware of the risk of the hardware limitations, and it appears that my anticipation was indeed well-founded. However, at the theoretical level, I think the goal has been achieved as I have provided a complete description of the problem as an MDP, argued that its solution would solve the qubit mapping problem and provided an algorithm to solve it. Overall, I firmly believe that reinforcement learning is a viable approach to solving this problem, and with proper training, good performances can be achieved.

On a personal level, my objective was also to learn and gain experience in both the fields of quantum computing and machine learning, and I am very satisfied with all the knowledge I have acquired throughout the completion of this thesis. Even so, I acknowledge that I still have a long way to go to get up to date with the most recent lines of research in both fields. This project has motivated me to continue this journey and hopefully some day be able to contribute to their progress and future breakthroughs.

In terms of future work, there are several aspects to consider. A first step would be to train the DQN algorithm presented in this thesis 24 with more powerful hardware, if available, to explore its potential to achieve good performance. Additionally, it remains to be addressed the complete QCL problem. This requires framing it as an MDP. A straightforward definition of this MDP would be to take as action set all the possible permutations in the initial mapping, along with the insertion of a SWAP gate between any pair of qubits at any depth of the circuit. While this MDP formulation, with the appropriate rewards, may hold potential for solving the QCL problem, the large number of actions introduced makes it considerably challenging for any algorithm to find an optimal solution. For instance, considering a maximum depth $D$ for quantum circuits and a maximum number of qubits $K$ for the processor (assuming the quantum circuit has fewer or equal qubits than the processor), the action space would already contain $K \cdot(K-1) / 2$ permutations and $D \cdot K \cdot(K-1) / 2$ SWAP insertions. For instance, with $K=10$ and $D=10$, there would already be 495 actions, which could result in significantly long episodes and extensive computation times when collecting training data. Therefore, finding an MDP that can actually be efficiently solved, even approximately, seems to be non obvious.

## 9 Annex A. Budget

### 9.1 Identification of costs

### 9.1.1 Staff costs

On the human side these are the roles that are needed. Firstly, the Project Manager (PM). This role is responsible of planning, organizing, and overseeing the execution of a project from start to finish. Project management involves defining the project's goals, objectives, and deliverables, and then developing a comprehensive plan to achieve them. Secondly, the Researcher (R). This role is responsible for acquiring the necessary knowledge on quantum computing and reinforcement learning to decide the best algorithm to tackle the problem. Finally, we have the Developer (D) role. This role is responsible for implementing the algorithm, training it and testing its performance. The estimation of the staff costs for each of the activities in the Gantt chart is depicted in section 9.2 along with the other estimations.

### 9.1.2 Generic costs

In this section, I will specifically consider the expenses that are directly associated with the thesis project. As a result, I will not include costs such as the workspace or internet usage, since these are already part of my daily life and will not be impacted by the completion of this project.

In relation to the cost of materials required for the research and learning phase of the thesis, I will not be including any cost as all necessary resources are freely available online. While some resources may not be available to the general public for free, my status as a student provides me with free access. Similarly, as I will be using open-source libraries for coding and an open-source operating system on my computer, the cost for software will also be zero. In terms of hardware, there will be two costs involved, namely the amortization cost of the computer that will be utilized throughout the project, and the cost of using cloud computing. The electricity cost will be incurred only while using the PC and the router for
internet access, but not for using cloud resources as the electricity required for this is typically included in the price paid for using the service.

### 9.1.3 Contingencies and incidentals

To ensure that the thesis project can proceed without any significant disruptions or unexpected expenses, contingencies and incidentals have been accounted for in the budget. A contingency fund of $10 \%$ of the total generic and staff costs has been allocated to cover any unforeseen expenses, such as hardware or software failures. Additionally, alternative plans have been put in place to address any potential obstacle that may arise during the project. These alternative plans would not add any extra generic cost, as they would require the use of the PC and electricity just as it is required by the main plan. They might however incur in extra staff costs due to a possible extra time needed to complete the project. As mentioned in section ??, an upper bound of 15 extra hours has been allocated in the case of a deviation of the main plan. For the budget I will consider the worst case scenario in which the entire 15 hours are needed.

### 9.2 Cost estimates

For the staff costs I have obtained the gross salary per year of each role from [20]. Then I added to this amount the social security costs using the formula from [21]. This gave me the yearly cost of each role, but I only have to consider the hours worked. Thus I calculated the cost per hour. The year 2023 has 248 working days in Spain [22]. With 8 hours of work per day, it makes a total of 1984 hours of work throughout the year. Therefore, dividing the yearly cost by 1984 gave me the hourly cost for each staff member. The numbers are depicted in Table 3. Using the estimated hours per task represented on the Gantt chart, we can obtain the estimated work hours per each role and thus obtain the total staff cost. The results are shown in Table 4 .

| Role | Annual gross <br> salary (€) | Social <br> Security (€) | Total annually <br> $(€)$ | Total hourly <br> $(€)$ |
| :---: | :---: | :---: | :---: | :---: |
| Project Manager | 39474 | 13184.3 | 52658.3 | 26.54 |
| Researcher | 30521 | 10194 | 40715 | 20.52 |
| Developer | 28325 | 9460.55 | 37785.55 | 19.05 |

Table 3: Staff costs for each role

The cost of the PC, whose specifications are depicted in section ??, is $900 €$ and it is amortized in 4 years but I only consider the part that is proportional to the hours of use for this project. I assume that the amount of working days during these four years is the same as in the year 2023 which I mentioned earlier, and each working day consists of 8 working hours. From that I obtain the cost per hour. I have estimated that $90 \%$ of the project's total hours of work are spent using the PC. This amount multiplied by the cost per hour results in the cost of the PC attributable to this project.

The cloud resources can vary considerably depending on the complexity of the model, which will determine the training difficulty, but I have calculated a rough estimate with the Google Cloud pricing calculator [23]. According to the calculations, using their Vertex AI training product for one month, with a 2 hour long training per week and using a machine with 2 CPUs and 8 GB of RAM, the cost would be $1.25 €$.

Regarding the electricity, the average price per kWh is $0.1638 € / \mathrm{kWh}[24]$, the average consumption of my PC is around 200 W with a PSU efficiency of $87 \%$ and, as previously stated I will use it in $90 \%$ of the project's total hours. Table 5 shows the resulting generic costs.

Finally we have to add to the budget the costs for contingencies and incidents. As previously explained, the contingencies cost will be $10 \%$ of the total sum of staff and generic costs (without the incidentals) and the incidental costs are only from the staff. Table 6 shows the incidental costs in detail. A full summary of the budget is shown in table 7

| Task | Estimated hours needed | Worker | Cost (€) |
| :---: | :---: | :---: | :---: |
| Context and scope | 25 | PM | 663.5 |
| Time planning | 20 | PM | 530.8 |
| Budget and sustainability | 15 | PM | 398.1 |
| Final document | 15 | PM | 398.1 |
| Meetings | 10 | PM,R,D | 661.1 |
| Documentation | 70 | PM | 1857.8 |
| Learn Quantum Computing | 45 | R | 923.4 |
| Learn Reinforcement Learning | 50 | R | 1026 |
| Learn Qiskit | 50 | R | 1026 |
| Learn TensorFlow | 50 | R | 1026 |
| Establish baseline performance metrics | 2 | D | 38.1 |
| Plan the model | 45 | D | 857.25 |
| Implement the model | 53 | D | 1009.65 |
| Train the model | 35 | D | 666.75 |
| Test and compare performance | 30 | D | 571.5 |
| Speech preparation | 15 | PM | 398.1 |
| Slides | 10 | - | 12317.55 |
| Total | 540 |  | 265.4 |

Table 4: Staff costs for each task.

| Generic cost | Cost (€) |
| :---: | :---: |
| Software | 0 |
| PC | 55.12 |
| Cloud computing resources | 1.25 |
| Electricity | 36.6 |
| Total | 92.97 |

Table 5: Generic costs

### 9.3 Management control

The management control section of this thesis aims to propose mechanisms for controlling potential budget deviations and define numerical indicators to facilitate control. To achieve this, we will use a budget variance analysis methodology to compare actual expenditures against budgeted amounts and identify any deviations.

To control budget deviations, I will be implementing budget monitoring, forecasting, and risk management mechanisms. Budget monitoring involves regularly reviewing and comparing actual expenditures against budgeted amounts to identify any deviations. To achieve this I will use two numerical indicators, namely the percentage deviation, which measures the percentage difference between the actual and budgeted amounts, and the cost variance, which measures the difference between the actual and budgeted amounts in monetary terms. Forecasting involves using historical data and trends to predict future expenditures and identify potential budget deviations. This will allow to make the necessary adjustments to the budget in advance. Risk management involves identifying and managing risks that may impact the budget and implementing strategies to mitigate these risks.

| Task | Extra hours | Role | Cost (€) |
| :---: | :---: | :---: | :---: |
| Learn Supervised Methods | 2 | R | 41.04 |
| Learn TensorFlow | 2 | R | 41.04 |
| Plan the model | 3 | R | 61.56 |
| Implement the model | 4 | D | 76.2 |
| Train the model | 2 | D | 38.1 |
| Test and compare performance | 2 | D | 38.1 |
| Total | 15 | - | 296.04 |

Table 6: Incidental costs

| Cost | Amount (€) |
| :---: | :---: |
| Staff | 12317.55 |
| Generic | 92.97 |
| Contingencies | 1241.05 |
| Incidentals | 296.04 |
| Total | 13947.61 |

Table 7: Summary of estimated costs

## 10 Annex B. Sustainability

### 10.1 Economic dimension

The economic dimension of this project encompasses the estimation of costs and the potential economic improvements of the proposed potential solution. A detailed estimation of the costs involved can be found in $\S 9.2$. Regarding the existing state of the art, there are algorithms that address the problem at hand. However, from an economic standpoint, the proposed potential solution aims to enhance these existing algorithms by achieving better optimization. By doing so, it is expected to reduce the cost of executing more complex programs through lower energy and hardware requirements, thereby providing a more economically efficient approach.

### 10.2 Environmental dimension

The environmental dimension of this project includes the estimation of the environmental impact and the strategies employed to minimize this impact. The estimation of the project's environmental impact has been carefully considered. As machine learning models require significant computing resources, steps have been taken to minimize the carbon footprint. To minimize this impact, I will seek to reuse resources wherever possible. For example, I will use my personal computer for initial development and testing, rather than purchasing new hardware.

Considering the current state of the art algorithms aforementioned, from an an environmental perspective, my solution can offer a considerable improvement in two main aspects. Firstly, in machine learning algorithms, once they are trained they can be executed in constant time. Thus they use much less energy to execute. Secondly, and most notably, if I achieve the goal of improving the quality of the optimizations, programs will be able to be executed much more efficiently which ultimately results in a much lower energy cost.

### 10.3 Social dimension

The social dimension of this project explores the personal contributions and the potential benefits to society. Undertaking a project on quantum computing and machine learning has allowed me to deepen my understanding of both these fields, improve my research and analytical skills, and develop my programming abilities. It has also helped me to hone my time management and organization skills as I balanced the demands of the project with other commitments. Overall, working on this project has contributed to my personal growth and provided a valuable opportunity to apply my knowledge and skills to a cutting-edge research topic in the field of quantum computing.

The project's contribution extends beyond personal growth. In the current state of the art, existing
algorithms for the addressed problem lack efficiency. Quantum computing is a rapidly growing field with the potential to revolutionize various domains, including drug discovery, climate modeling, and cryptography. By improving the efficiency of quantum circuit compilation, we can accelerate the development of quantum algorithms and enable the realization of these potential benefits. Compared to existing solutions, the proposed approach offers several advantages, such as increased accuracy and efficiency, and the ability to handle larger and more complex circuits.

The real need for this project stems from the expanding field of quantum computing, where efficient circuit compilation techniques are in demand. By addressing this need, the project aims to contribute to the development of new applications and technologies that can improve the quality of life for people around the world.

## References

[1] Giovanni Acampora and Roberto Schiattarella. "Deep neural networks for quantum circuit mapping". In: Neural Computing and Applications 33 (Oct. 2021). DOI: 10.1007/s00521-021-060093.
[2] M. Amy et al. "A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits". In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32.6 (June 2013), pp. 818-830. DOI: $10.1109 /$ tcad.2013.2244643.
[3] Édouard Bonnet, Tillmann Miltzow, and Pawel Rzazewski. Complexity of Token Swapping and its Variants. 2018. arXiv: 1607.07676 [cs.CC].
[4] Adi Botea, Akihiro Kishimoto, and Radu Marinescu. "On the Complexity of Quantum Circuit Compilation". In: 9.1 (2018), pp. 138-142. DOI: doi.org/10.1609/socs.v9i1.18463.
[5] Sebastian Brandhofer, Ilia Polian, and Hans Peter Büchler. "Optimal Mapping for Near-Term Quantum Architectures based on Rydberg Atoms". In: 2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD). 2021, pp. 1-7. DOI: 10.1109/ICCAD51958.2021.9643490,
[6] Stephen A. Cook. "The complexity of theorem-proving procedures". In: Proceedings of the third annual ACM symposium on Theory of computing (1971), pp. 151-158. DOI: 10.1145/800157. 805047 .
[7] Christopher M. Dawson and Michael A. Nielsen. The Solovay-Kitaev algorithm. 2005. arXiv: quant-ph/0505030 [quant-ph].
[8] H. Dong, Z. Ding, and S. Zhang. Deep Reinforcement Learning: Fundamentals, Research and Applications. Springer Nature Singapore, 2020. ISBN: 9789811540943.
[9] First approach implementation. URL: https://github.com/JordiE1Se/Qubit_mapping_MCTS_ DNN.
[10] Kaiming He et al. Deep Residual Learning for Image Recognition. 2015. arXiv: 1512.03385 [cs.CV]
[11] Steven Herbert and Akash Sengupta. Using Reinforcement Learning to find Efficient Qubit Routing Policies for Deployment in Near-term Quantum Computers. 2019. arXiv: 1812.11619 [quant-ph].
[12] A. Yu Kitaev. "Quantum computations: algorithms and error correction". In: Russian Mathematical Surveys 52.6 (1997), pp. 1191-1249. DOI: https://doi.org/10. 1070 <br>%2Frm1997v052 \} $\backslash$ n06abeh002155.
[13] Vadym Kliuchnikov, Dmitri Maslov, and Michele Mosca. "Asymptotically optimal approximation of single qubit unitaries by Clifford and T circuits using a constant number of ancillary qubits". In: Physical Review Letters 110.19 (May 2013). DOI: 10.1103/physrevlett.110.190502
[14] Gushu Li, Yufei Ding, and Yuan Xie. "Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices". In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. Association for Computing Machinery, 2019, pp. 1001-1014. ISBN: 9781450362405 . DOI: $10.1145 / 3297858.3304023$
[15] F. Mandl. Quantum Mechanics. John Wiley \& Sons, 1992.
[16] Dmitri Maslov, Sean M. Falconer, and Michele Mosca. "Quantum Circuit Placement". In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 27.4 (Apr. 2008), pp. 752-763. DOI: doi.org/10.1109<br>%2Ftcad.2008.917562.
[17] Ken M. Nakanishi, Takahiko Satoh, and Synge Todo. Quantum-gate decomposer. 2021. arXiv: 2109.13223 [quant-ph].
[18] M. A. Nielsen and I. L. Chuang. Quantum computation and quantum information. Cambridge University Press, 2000.
[19] Alexandru Paler. On the Influence of Initial Qubit Placement During NISQ Circuit Compilation. 2019. arXiv: 1811.08985 [quant-ph].
[20] Alexandru Paler et al. Machine Learning Optimization of Quantum Circuit Layouts. 2022. arXiv: 2007.14608 [quant-ph].
[21] Matteo G. Pozzi et al. Using Reinforcement Learning to Perform Qubit Routing in Quantum Compilers. 2020. arXiv: 2007.15957 [quant-ph].
[22] John Preskill. "Quantum Computing in the NISQ era and beyond". In: Quantum 2 (Aug. 2018), p. 79. DOI: 10.22331/q-2018-08-06-79.
[23] QMAP GitHub repository. https://github.com/cda-tum/qmap/tree/main.
[24] Second approach implementation. URL: https://github . com/JordiE1Se / DQN_for_Qubit_ Mapping.
[25] Peter W. Shor. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". In: SIAM Journal on Computing 26.5 (Oct. 1997), pp. 1484-1509. DOI: 10.1137/s0097539795293172.
[26] David Silver et al. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. 2017. arXiv: 1712.01815 [cs.AI].
[27] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: Nature 529.7587 (Jan. 2016), pp. 484-489. ISSN: 1476-4687. DOI: 10.1038 /nature16961. URL: https://doi.org/10.1038/nature16961.
[28] David Silver et al. "Mastering the game of Go without human knowledge". In: Nature 550 (Oct. 2017), pp. 354-. URL: http://dx.doi.org/10.1038/nature24270.
[29] Animesh Sinha, Utkarsh Azad, and Harjinder Singh. Qubit Routing using Graph Neural Network aided Monte Carlo Tree Search. 2022. arXiv: 2104.01992 [quant-ph].
[30] Marcos Siraichi et al. "Qubit Allocation". In: CGO 2018 - International Symposium on Code Generation and Optimization. Vienna, Austria, Feb. 2018, pp. 1-12. Doi: 10.1145/3168822.
[31] Marcos Yukio Siraichi et al. "Qubit Allocation as a Combination of Subgraph Isomorphism and Token Swapping". In: 3.OOPSLA (2019). DOI: $10.1145 / 3360546$.
[32] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. Second. The MIT Press, 2018.
[33] Bochen Tan and Jason Cong. "Optimality Study of Existing Quantum Computing Layout Synthesis Tools". In: IEEE Transactions on Computers 70.9 (Sept. 2021), pp. 1363-1373. DOI: 10.1109/tc. 2020.3009140
[34] Bochen Tan et al. "Qubit Mapping for Reconfigurable Atom Arrays". In: Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design. 107. Association for Computing Machinery, 2022. ISBN: 9781450392174. DOI: $10.1145 / 3508352.3549331$.
[35] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. "Mapping Quantum Circuits to IBM QX Architectures Using the Minimal Number of SWAP and H Operations". In: Proceedings of the 56th Annual Design Automation Conference 2019. ACM, June 2019. DoI: doi.org/10.1145 \} \%2F3316781. 3317859 ,
[36] Robert Wille, Aaron Lye, and Rolf Drechsler. "Optimal SWAP gate insertion for nearest neighbor quantum circuits". In: Jan. 2014, pp. 489-494. ISBN: 978-1-4799-2816-3. DOI: 10.1109/ASPDAC. 2014.6742939 .


[^0]:    ${ }^{1}$ The set of gates $\{H, S, C N O T\}$ is known as the Clifford group

[^1]:    ${ }^{2}$ This circuit has been obtained using the QMAP library in python 23 which finds the optimal circuit by treating it as a MAX-SAT problem.

[^2]:    ${ }^{3}$ The list with all the available quantum processors can be found at https://quantum-computing.ibm.com/services/ resources

[^3]:    ${ }^{4}$ Determining the optimal swaps given an qubit assignment is equivalent to the Token Swapping problem which is known to be NP-Hard (see 3]).
    ${ }^{5}$ The mapping of the logical qubits to the physical qubits is injective. Thus in the worst case we can always take each individual qubit and map it to its corresponding physical qubit

[^4]:    ${ }^{6}$ Note that acting greedly with respect to the state-value function is equivalent to acting greedly with respect to the action-value function. Indeed, the action $a$ that makes $q_{\pi}(s, a)$ maximum is the same action that, applied to state $s$, leads to a state $s^{\prime}$ whose state-value function $v_{\pi}\left(s^{\prime}\right)$ is maximum.
    ${ }^{7}$ For the interested reader, the demonstration can be found in 32, Chapter 4.

[^5]:    8 https://www.tensorflow.org/agents
    9 https://www.tensorflow.org/agents/api_docs/python/tf_agents/environments/PyEnvironment.
    10 https://www.deepmind.com/open-source/reverb

