FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Noise-Adaptive Reinforcement Learning Strategies for Qubit Routing

Gonçalo José Cerqueira Pascoal



Mestrado em Engenharia Informática e Computação

Supervisor: Prof. João Paulo Fernandes

Second Supervisor: Prof. Rui Maranhão

October 20, 2023

© Gonçalo Pascoal, 2023

Noise-Adaptive Reinforcement Learning Strategies for Qubit Routing

Gonçalo José Cerqueira Pascoal

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. José Campos
Examiner: Prof. Shaukat Ali
Supervisor: Prof. João Paulo Fernandes

October 20, 2023

Abstract

Quantum computing is an emerging and rapidly evolving field that aims to harness quantum mechanical phenomena to address problems that lie beyond the reach of classical computation. Its relevance to a wide variety of issues, including drug discovery, financial modeling, optimization, and machine learning, has fueled research efforts and corporate investment, which have led to notable advancements on both the hardware and software fronts over the last few years.

Despite this progress, existing quantum computers face many limitations that hinder their application to real-world problems. In addition to having a small number of qubits (quantum bits) and being highly error-prone, near-term devices only support a narrow subset of operations and typically suffer from sparse qubit connectivity, preventing interactions between arbitrary qubits. Due to these constraints, quantum algorithms must be made compatible with a specific architecture through a process known as quantum compiling. Qubit routing, one of the compilation steps, ensures that multi-qubit operations are only performed between qubits that are connected in the target device. This is a challenging optimization problem that can significantly influence the success rate of a quantum program.

This dissertation proposes a method to address the qubit routing problem by leveraging deep reinforcement learning. We further augment our technique using calibration data from real quantum devices to enable routing decisions based on the fidelity of individual operations. We used Proximal Policy Optimization (PPO) to train our models and rewarded agents based on the reliability of chosen operations. When defining the observations, we improved upon a previous approach from the literature that represented quantum circuits as a numerical matrix. Moreover, we implemented other enhancements, such as BRIDGE operations and gate commutation analysis. We evaluated our strategy on the topologies of five IBM quantum computers with between 5 and 27 qubits. The source code and data are available under an open-source license.

For randomly generated circuits, the proposed method reduced the number of two-qubit gates used by up to 36.9% and improved circuit reliability by up to 26.6% when compared to the best-performing algorithm from the Qiskit quantum computing framework. When compiling real programs from a benchmark dataset, our approach achieved up to 28.4% and 18.5% improvements in number of operations and circuit fidelity, respectively. Performance gains were most noticeable for 5 and 7-qubit devices. We found no appreciable advantage from using a calibration-aware model, which merits further testing with different architectures or reinforcement learning environment specifications. Our work helps cement reinforcement learning as a valuable qubit routing strategy and serves as another stepping stone on the path to practical applications for quantum computers.

Keywords: Qubit routing, Deep reinforcement learning, Noise-adaptive compiling, Quantum compiling, Quantum computing

Resumo

A computação quântica é uma área emergente e em rápida evolução que procura tirar proveito dos fenómenos da mecânica quântica para resolver problemas que se encontram para além do alcance da computação clássica. A sua relevância para uma grande variedade de assuntos, como descoberta de medicamentos, modelação financeira, otimização e *machine learning*, tem estimulado iniciativas de investigação e investimento empresarial que, nos últimos anos, se traduziram em avanços consideráveis tanto ao nível do *hardware* como do *software*.

Apesar deste progresso, os computadores quânticos existentes atualmente enfrentam várias limitações que dificultam a sua aplicação a problemas do mundo real. Além de possuírem poucos *qubits* (*bits* quânticos) e serem altamente suscetíveis a erros, os sistemas atuais apenas conseguem executar um pequeno conjunto de operações de raiz e apresentam fraca conetividade entre *qubits*. Devido a estas restrições, é necessário tornar os algoritmos quânticos compatíveis com uma arquitetura específica através de um processo conhecido como compilação quântica (*quantum compiling*). Uma das etapas de compilação, designada como *qubit routing*, garante que todas as operações envolvendo múltiplos *qubits* são realizadas entre *qubits* que se encontram interligados no computador alvo. Este é um problema difícil de otimização que pode ter um efeito acentuado na probabilidade de sucesso de um programa.

Esta dissertação propõe um método para resolver o problema de *qubit routing* através de *deep reinforcement learning*. De modo a permitir que as decisões de *routing* tomassem em consideração a fiabilidade individual de cada operação, incorporámos dados de calibração provenientes de computadores quânticos reais. Utilizámos o algoritmo *Proximal Policy Optimization* (PPO) para treinar os modelos e recompensámos os agentes com base na fiabilidade das operações escolhidas. Para definir as observações, aprofundámos uma abordagem prévia da literatura que representa circuitos quânticos em forma de matriz numérica. Além disso, implementámos outras melhorias como operações BRIDGE e análise de comutação de portas quânticas. Avaliámos a nossa estratégia com recurso às topologias de cinco computadores quânticos da IBM, tendo entre 5 e 27 *qubits* e disponibilizámos o código-fonte e dados sob uma licença *open-source*.

Para circuitos criados aleatoriamente, o método proposto reduziu o número de portas de dois *qubits* utilizadas em até 36,9% e melhorou a fiabilidade dos circuitos em até 26,6% em comparação com o melhor algoritmo da biblioteca de computação quântica Qiskit. Ao compilar programas reais, a nossa abordagem atingiu melhorias de até 28,4% e 18,5% em termos do número de operações e fiabilidade dos circuitos, respetivamente. O aumento de desempenho foi mais notável para os computadores de 5 e 7 *qubits*. Não observámos uma vantagem apreciável por incluir dados de calibração no modelo, algo que motiva experiências adicionais usando outras arquiteturas ou definições do ambiente de *reinforcement learning*. Os nossos resultados ajudam a consolidar o *reinforcement learning* como uma estratégia eficaz para *qubit routing* e constituem mais um passo na direção de aplicações práticas para computadores quânticos.

Palavras-chave: *Qubit routing*, *Deep reinforcement learning*, Compilação adaptada ao ruído (*noise-adaptive compiling*), Compilação quântica (*quantum compiling*), Computação quântica

Acknowledgements

First, I thank my supervisors, Professor João Fernandes and Professor Rui Maranhão, for supporting and encouraging me throughout this challenging but illuminating journey. This dissertation has been an outstanding learning experience, and it has been fascinating to explore such a novel and unique paradigm of computing. I would also like to express my gratitude to LIACC's high-performance computing team for providing access to their hardware resources, without which this work would not be possible.

I thank my parents for always being my solid ground: my mother, who always believes in me even at times when I do not believe in myself, and my father, who I know I can always rely on. I dedicate this dissertation to my grandmother, Cândida.

To my second family at the Tuna de Engenharia da Universidade do Porto, this adventure would not have been as thrilling without you. Thank you for gifting me with some of my most cherished memories and deepest friendships.

I gratefully acknowledge the support of the EU COST action CA19135, titled "Connecting Education and Research Communities for an Innovative Resource Aware Society (CERCIRAS)," by means of a Short Term Scientific Mission (STSM) grant to visit the SIMULA research laboratory in Oslo, Norway, and discuss our methodological approach with other researchers working on quantum computing. I would like to thank everyone at SIMULA and, particularly, the researchers at the Complex Systems Department for kindly hosting me during that week and for the discussions that helped me steer this project towards a more solid path.

Gonçalo Pascoal

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Objectives . . . 3
- 1.4 Document Structure . . . 3

2 Background **5**
- 2.1 Fundamentals of Quantum Computing . . . 5
  - 2.1.1 Qubits . . . 5
  - 2.1.2 Quantum Gates . . . 7
- 2.2 Quantum Compiling . . . 10
  - 2.2.1 Visualizing Quantum Circuits . . . 10
  - 2.2.2 Gate Decomposition . . . 11
  - 2.2.3 Quantum Circuit Mapping . . . 12
  - 2.2.4 Optimization . . . 13
  - 2.2.5 Scheduling . . . 14
- 2.3 Variational Quantum Algorithms . . . 15
  - 2.3.1 Cost Function . . . 15
  - 2.3.2 Parametrized Quantum Circuit . . . 16
  - 2.3.3 Measurement Scheme . . . 16
  - 2.3.4 Classical Optimizer . . . 17
- 2.4 Reinforcement Learning . . . 17
  - 2.4.1 Base Concepts . . . 18
  - 2.4.2 Deep Reinforcement Learning . . . 20
  - 2.4.3 Proximal Policy Optimization . . . 21
  - 2.4.4 Action Masking . . . 22

3 State of the Art **24**

4 Problem Statement and Methodology **33**
- 4.1 Problem Formulation . . . 33
- 4.2 Methodology . . . 34
  - 4.2.1 Qubit Routing as a Reinforcement Learning Problem . . . 35
  - 4.2.2 Enhancements and Optimizations . . . 38
  - 4.2.3 Scope . . . 40

5 Implementation **45**
5.1 Technologies and Platforms . . . 45
5.1.1 Quantum Computing Framework . . . 45
5.1.2 Deep Reinforcement Learning Framework . . . 46
5.1.3 Other Tools . . . 47
5.2 Implementation Details . . . 48
5.2.1 Overview . . . 48
5.2.2 Gym Environment . . . 48
5.2.3 Training and Evaluation . . . 52
5.2.4 Data Generation . . . 53
5.2.5 Configuration Files . . . 53
5.2.6 Utility Scripts . . . 55

6 Results and Discussion **56**
6.1 Results . . . 56
6.1.1 Random Circuits . . . 56
6.1.2 Realistic Circuits . . . 64
6.1.3 Impact of Enhancements . . . 70
6.1.4 Impact of Evaluation Episodes . . . 70
6.1.5 Training and Routing Times . . . 71
6.2 Discussion . . . 72

7 Conclusion **74**

References **76**

List of Figures

2.1 Bloch sphere representation of the $|-\rangle$ state, generated using Qiskit's visualization module . . . 6
2.2 Example quantum circuit . . . 10
2.3 Depictions of several quantum gates . . . 10
2.4 Decomposition of a quantum circuit into layers . . . 11
2.5 Decomposition of a Toffoli gate . . . 12
2.6 Example gate decompositions targeting IBM quantum computers . . . 12
2.7 Different decompositions of the SWAP gate into CNOTs . . . 13
2.8 BRIDGE gate . . . 13
2.9 Gate sequence that can represent any two-qubit unitary $U \in \mathrm{SU}(4)$ . . . 14
2.10 Gate commutation rules . . . 14
2.11 Diagram illustrating the typical components of a VQA . . . 15
2.12 Relationship between an agent and the environment . . . 18
2.13 A simple Markov decision process . . . 19

3.1 Visualization of the state of the DQN routing algorithm at a given point in its execution . . . 25
3.2 Procedure for deriving a state matrix from a quantum circuit . . . 26
3.3 Architecture of the graph neural network . . . 28
3.4 Qubit coherence times and gate error rates over time for the IBM Q 16 Rueschlikon quantum computer . . . 29

4.1 A sample state of our qubit routing environment . . . 35
4.2 Constructing a circuit matrix observation . . . 37
4.3 Coupling graph of IBM Q Guadalupe . . . 38
4.4 Topology of the devices considered in our work . . . 40
4.5 Quantum circuit that implements a Fredkin gate . . . 43

5.1 The TensorBoard web interface . . . 47

6.1 SWAP vs. BRIDGE gate usage for each device . . . 58
6.2 IBM Q Manila results (random circuits) . . . 59
6.3 IBM Q Belem results (random circuits) . . . 60
6.4 IBM Nairobi results (random circuits) . . . 61
6.5 IBM Q Guadalupe results (random circuits) . . . 62
6.6 IBM Q Mumbai results (random circuits) . . . 63
6.7 IBM Q Manila results (realistic circuits) . . . 65
6.8 IBM Q Belem results (realistic circuits) . . . 66
6.9 IBM Nairobi results (realistic circuits) . . . 67

6.10 IBM Q Guadalupe results (realistic circuits) 68
6.11 IBM Q Mumbai results (realistic circuits) 69
6.12 Model performance with different enhancements disabled 70
6.13 Routing performance relative to the number of evaluation episodes 71

List of Tables

4.1 Hyperparameters used for each device architecture 42
4.2 Learning rate schedule used for each device architecture 42

5.1 Encoding actions as discrete values 49

6.1 Relative improvements over best Qiskit algorithm (random circuits) 58
6.2 IBM Q Manila metrics (random circuits) 59
6.3 IBM Q Belem metrics (random circuits) 60
6.4 IBM Nairobi metrics (random circuits) 61
6.5 IBM Q Guadalupe metrics (random circuits) 62
6.6 IBM Q Mumbai metrics (random circuits) 63
6.7 Relative improvements over best Qiskit algorithm (realistic circuits) 64
6.8 IBM Q Manila normalized metrics (realistic circuits) 65
6.9 IBM Q Belem normalized metrics (realistic circuits) 66
6.10 IBM Nairobi normalized metrics (realistic circuits) 67
6.11 IBM Q Guadalupe normalized metrics (realistic circuits) 68
6.12 IBM Q Mumbai normalized metrics (realistic circuits) 69
6.13 Training and routing times for each device 72

List of Listings

5.1 Structure of a routing environment observation 50
5.2 Signature of the `reset` method 50
5.3 Signature of the `step` method 51
5.4 Example environment configuration 54
5.5 Example training configuration 54
5.6 Example evaluation configuration 55

Abbreviations and Symbols

DAG	Directed Acyclic Graph
DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q-Network
GPU	Graphics Processing Unit
GUI	Graphical User Interface
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process
NASSC	Not All Swaps have the Same Cost
NISQ	Noisy Intermediate-Scale Quantum
PPO	Proximal Policy Optimization
QAOA	Quantum Approximate Optimization Algorithm
QAQC	Quantum-Assisted Quantum Compiling
RL	Reinforcement Learning
SAC	Soft Actor Critic
SARSA	State-Action-Reward-State-Action
SDK	Software Development Kit
SPAM	State Preparation And Measurement
VQA	Variational Quantum Algorithm
VQE	Variational Quantum Eigensolver

Chapter 1

Introduction

This first chapter introduces the subject matter of the dissertation. We begin with a contextualization of quantum computing (Section 1.1), which discusses the current outlook and expectations for this technology, succinctly recounts the field's history, and explains some key concepts. Subsequently, we present the rationale behind our research into a topic of quantum computing known as quantum compiling and, specifically, the problem of qubit routing (Section 1.2). We outline the main goals we seek to accomplish and issues we intend to address with this work (Section 1.3) and then describe how the rest of the document is structured (Section 1.4).

1.1 Context

Quantum computing is a computational paradigm that exploits the unique properties of quantum mechanics to solve particular types of problems considered intractable for classical computers. Experiments such as the one performed by the Google AI Quantum team have already signaled the existence of this distinctive edge [1]. In this test, the Sycamore quantum chip performed a task in 200 seconds estimated to require roughly 10,000 years to complete on classical supercomputers. We have recently witnessed significant hardware developments, and enterprises such as IBM [2], Google [3], D-Wave [4], Rigetti [5] and Quantinuum [6] have been investing heavily in quantum technologies. The development of open-source frameworks [7, 8] and the ability to access quantum computers through cloud platforms [9] have contributed to the growing accessibility and openness of quantum development. Nevertheless, quantum computing is still a relatively new domain, so substantial research efforts are required before it can be employed in real-world settings.

The roots of the field can be traced back to Paul Benioff's proposal for a quantum Turing machine in 1980 [10]. Later, several proof-of-concept algorithms [11, 12, 13] demonstrated the possibility of a quantum advantage over classical processing (at times delivering an exponential speedup). However, these were designed to solve carefully crafted mathematical problems with seemingly no practical application. Building on these results, Peter Shor proposed a quantum algorithm for factoring integers in polynomial time, which could break most modern cryptographic

protocols if performed at a large enough scale [14]. In 1996, Grover presented a method for finding items in an unstructured database in $O(\sqrt{N})$ time [15], based on a technique known as amplitude amplification, which has been applied in other quantum algorithms.

The emergence of algorithms with real-world implications, such as Shor's and Grover's, sparked considerable interest in quantum computing. Regardless, implementing either of them at any useful scale would require quantum devices far more powerful and fault-tolerant than those currently available. Existing machines, often called noisy intermediate-scale quantum (NISQ) computers, possess, at most, a few hundred qubits (the basic unit of quantum information, analogous to classical bits). Errors introduced by imperfect operations and decoherence (a phenomenon where qubits lose quantum state information via interactions with the environment) further limit the scale of problems that can be tackled using NISQ devices.

In theory, quantum error correction techniques could be used to shield qubits from the effects of noise [16, 17], but the number of additional qubits required makes them inviable in the near term. Thus, developing and refining noise-resilient algorithms appears to be the best way to leverage current quantum resources [18]. Variational quantum algorithms (VQAs) are a prime example of this approach and are likely the most promising route towards achieving a quantum advantage in the near term. These algorithms operate in a hybrid scheme, employing classical optimization techniques to tune the parameters of a quantum circuit [19].

Quantum circuits are a method of expressing quantum computations or algorithms. They consist of a series of quantum gates, representing operations performed on one or more qubits, comparable to logic gates from classical computing. Real devices typically only support a small set of basis gates and have limited qubit connectivity, preventing arbitrary execution of multi-qubit gates. Due to these restrictions, the original program must be translated to an equivalent circuit that is compatible with the target architecture through a process known as quantum compiling.

Qubit routing is a stage of the compilation procedure that involves inserting auxiliary operations to ensure that all multi-qubit operations adhere to the device's connectivity constraints. This must be accomplished without changing the semantics of the algorithm and, ideally, using as few operations as possible. This work focuses on developing techniques for tackling the qubit routing problem using deep reinforcement learning.

1.2 Motivation

The remarkable potential of quantum computers becomes apparent when one considers their vast set of possible applications, spanning several influential fields. In the existing literature, we find VQAs for quantum chemistry [20], all three major branches of machine learning (supervised [21], unsupervised [22], and reinforcement learning [23]), combinatorial optimization [24], quantum simulation [25], and mathematical problems such as integer factorization [26] or solving linear systems of equations [27]. While quantum supremacy for certain deliberately constructed problems has indeed been demonstrated on NISQ devices [1], the challenge of outperforming classical supercomputers in practical applications remains.

Until we observe significant hardware advancements in terms of qubit counts and gate error rates, adapting algorithms to the limitations imposed by the NISQ era will be essential for attaining any kind of quantum advantage [18]. The *quantum volume* metric proposed by the IBM Quantum team for benchmarking quantum computers suggests that software improvements (such as better-designed algorithms or more effective compilation strategies) could deliver performance uplifts akin to hardware scaling [28]. The unreliable nature of current devices elevates quantum compiling from a simple prerequisite for executing circuits on real hardware to a process that can heavily influence the success rate of quantum programs. Thus, contributions to this topic become relevant to any algorithm designed with NISQ computers in mind.

Moreover, qubit routing is often the compilation step with the greatest effect on the reliability of the resulting circuit since it introduces more two-qubit operations, which tend to be far more susceptible to noise than single-qubit gates. A variety of methods have been developed for addressing the qubit routing problem. However, owing to its classification as NP-complete [29], exact approaches seldom scale favorably, and heuristic algorithms can sometimes yield solutions that are far from optimal [30]. Reinforcement learning's proficiency for solving similar optimization problems fuels our interest in exploring its potential as a viable qubit routing strategy and examining how it can be combined with other promising techniques from the literature.

1.3 Objectives

Under the overarching goal of contributing towards achieving quantum advantage in the NISQ era, this dissertation proposes an improved technique for qubit routing based on deep reinforcement learning. We seek to develop a noise-adaptive strategy that can leverage device calibration data, namely the error probabilities of quantum gates, to improve the fidelity of compiled circuits.

After acquiring a solid understanding of the literature on qubit routing and identifying key articles that can serve as a foundation for our work, we devise a methodology for applying reinforcement learning to the qubit routing problem and gauging its effectiveness when compared to existing strategies. In particular, we wish to evaluate our approach across a representative range of quantum device architectures and benchmarks, favoring circuit reliability as the chief performance metric. We are also interested in assessing the value of incorporating aspects of noise-adaptive routing into the reinforcement learning formulation.

To facilitate future research efforts, all source code and artifacts resulting from this dissertation are made available under a permissive license, and we provide the means to reproduce our results.

1.4 Document Structure

Following the introduction, this dissertation contains six chapters. Chapter 2 details relevant background information, including foundational quantum computing concepts, a more granular breakdown of quantum compiling and variational quantum algorithms, and some core aspects of reinforcement learning. In Chapter 3, we review the state of the art, examining different methods and

tools for quantum compiling in the available literature, with a strong emphasis on approaches that address the qubit routing problem. Chapter 4 formalizes the qubit routing problem, presents the adopted methodology, and outlines the extent of our research. Chapter 5 discusses the implementation of our strategy and lists the technologies used. In Chapter 6, we assess the effectiveness of our approach through several benchmarks and comment on the results obtained. Lastly, Chapter 7 summarizes the conclusions drawn from our work and identifies future research opportunities.

Chapter 2

Background

This chapter presents background concepts and details relevant to understanding this work. It offers foundational information regarding quantum computing (Section 2.1), including qubits, quantum states, quantum gates, and quantum phenomena such as superposition and entanglement. Then, we examine the problem of quantum compilation and its associated stages (Section 2.2) and discuss the main ideas and components behind variational quantum algorithms, a promising class of hybrid quantum-classical methods (Section 2.3). Lastly, we review the basic principles of reinforcement learning and a few specific algorithms and techniques employed in our work (Section 2.4).

2.1 Fundamentals of Quantum Computing

The following overview of the core aspects of quantum computing is chiefly based on a digital textbook included in Qiskit's documentation [31], as well as the book "Programming Quantum Computers" by Johnson, Harrigan, and Gimeno-Segovia [32].

2.1.1 Qubits

The term qubit (short for quantum bit) designates the basic unit of quantum information. The rules of quantum mechanics confer unique behavior that distinguishes them from regular bits. While classical bits can be described by one of two states, 0 or 1, qubits can exist in a superposition of both states simultaneously. Measuring a qubit collapses its state to one of the two basis states, represented as $|0\rangle$ or $|1\rangle$, thus eliminating any kind of superposition. Subsequent measurements will produce the same result.

We can represent the state of a qubit as a linear combination of the two computational basis states as seen in Equation 2.1. The coefficients α and β are complex numbers designating the probability amplitudes associated with each basis state.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad \alpha, \beta \in \mathbb{C} \tag{2.1}$$

Using polar coordinates (magnitude and phase) when discussing probability amplitudes is helpful for several reasons. First, the probability of measuring a particular basis state is given by the square of its associated magnitude. Thus, we can derive the relationship $|\alpha|^2 + |\beta|^2 = 1$. In addition, phase plays an essential role in the computational advantage conferred by quantum computers. Global phase factors have no observable influence on a computation, meaning the relative phase between α and β is sufficient to describe a quantum state. In fact, due to these constraints, we can represent quantum states as points on the surface of a unit sphere called the Bloch sphere. Figure 2.1 shows the representation of a state with an equal probability of being measured as $|0\rangle$ or $|1\rangle$. The poles of the sphere correspond to the computational basis states. In short, the nearer a state's Bloch vector is to a particular pole, the higher the likelihood of it being measured as the corresponding basis state. Equation 2.2 describes the relationship between a point's spherical coordinates and the probability amplitudes of the corresponding quantum state.

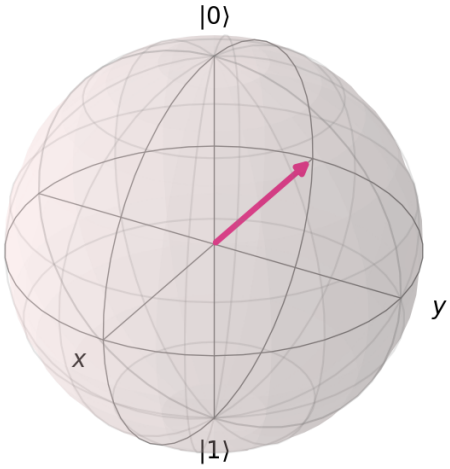


Figure 2.1: Bloch sphere representation of the $|-\rangle$ state, generated using Qiskit's visualization module [7]

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle \tag{2.2}$$

The computational basis states can be expressed as orthonormal vectors defining a two-dimensional complex vector space (see Equation 2.3). With each additional qubit, the number of dimensions of this vector space doubles (an n-qubit state corresponds to a vector in a 2^n-dimensional Hilbert space). This exponential growth explains the difficulty of classically simulating quantum computers with many qubits. Equation 2.4 presents the basis states of a two-qubit system.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.3}$$

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \; |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \; |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \; |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.4}$$

We can classify multi-qubit states as product states or entangled states. A product state can be represented as a sequence of single-qubit states. We can describe these states using only $2n$ probability amplitudes, meaning they can be efficiently simulated on a classical computer. The Kronecker product can be used to obtain the vector representation of product states, as shown in Equation 2.5.

$$|ba\rangle = |b\rangle \otimes |a\rangle = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} \otimes \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} b_0 a_0 \\ b_0 a_1 \\ b_1 a_0 \\ b_1 a_1 \end{bmatrix} \tag{2.5}$$

Equation 2.6 provides an example of an entangled state, specifically, an equal superposition of $|00\rangle$ and $|11\rangle$, meaning that there is a 50% probability of measuring either of those two states. Furthermore, by measuring just one of the qubits, we can know the value of the other qubit with certainty. It is impossible to represent entangled states as a product of individual qubit states, and their simulation require up to 2^n probability amplitudes. Entanglement is yet another core aspect of quantum mechanics that confers greater computational power to quantum devices.

$$|\Phi^+\rangle = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ 0 \\ 0 \\ \frac{1}{\sqrt{2}} \end{bmatrix} \tag{2.6}$$

2.1.2 Quantum Gates

Quantum logic gates are basic operations performed on one or more qubits that can be combined to create quantum circuits. Apart from measurement, all quantum operations are reversible, meaning the input of a gate can be recovered by applying its associated inverse. Quantum gates are represented by a unitary matrix (a matrix whose conjugate transpose is also its inverse). The product between a gate matrix and a state vector yields the resulting quantum state after applying that operation.

The most basic quantum operation is the identity gate, which has no effect on a quantum state, thus being comparable to a NOP (no operation) assembly instruction. It can be helpful when describing multi-qubit unitaries where certain qubits are not affected. Equation 2.7 shows the unitary matrix for the single-qubit version of this gate.

$$I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{2.7}$$

All single-qubit gates can be viewed as rotations around the Bloch sphere. For instance, the X, Y, and Z gates correspond to a rotation of π radians around the corresponding axes (see Equation 2.8). They are sometimes written as σ_x, σ_y, and σ_z, and, together with the identity, make up the Pauli gates. The X gate is comparable to the classical NOT gate due to its effect on the computational basis states, shown in Equation 2.9.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.8}$$

$$X|0\rangle = |1\rangle, \quad X|1\rangle = |0\rangle \tag{2.9}$$

The parametrized R_x, R_y, and R_z gates (see Equation 2.10) perform arbitrary rotations around the three conventional axes. The phase shift or P gate (shown in Equation 2.11) is another notable parametrized operation, having no impact on the $|0\rangle$ state but adding a phase of $e^{i\theta}$ to the $|1\rangle$ state. It is equivalent to the R_z gate up to a phase factor of $\frac{\theta}{2}$ ($P(\theta) = e^{i\theta/2}R_z(\theta)$). Several widely used phase shift gates with common angle values have special names, such as the $Z = P(\pi)$, $S = P(\frac{\pi}{2})$, and $T = P(\frac{\pi}{4})$ gates.

$$R_x(\theta) = \begin{bmatrix} \cos(\frac{\theta}{2}) & -i\sin(\frac{\theta}{2}) \\ -i\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{bmatrix}, \quad R_y(\theta) = \begin{bmatrix} \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{bmatrix}, \quad R_z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix} \tag{2.10}$$

$$P(\varphi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\varphi} \end{bmatrix} \tag{2.11}$$

One of the most important single-qubit gates is the Hadamard or H gate since it can be used to generate equal superpositions (where each computational basis state has the same probability of being measured). Conceptually, it can be envisioned as a rotation of π radians around the $x+z$ axis of the Bloch sphere. Equations 2.12 and 2.13 show the unitary matrix for the Hadamard gate, as well as its effect on the computational basis states.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.12}$$

$$H|0\rangle = |+\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \quad H|1\rangle = |-\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{2.13}$$

The $|+\rangle$ and $|-\rangle$ states are collectively known as the X basis. It is possible and, at times, adequate to measure quantum states in bases other than the computational (Z) basis. For instance, a qubit can be measured in the X basis by applying a Hadamard gate and then measuring in the Z

basis. Every quantum gate discussed previously can be expressed using the parametrized U gate (see Equation 2.14), which can implement any arbitrary single qubit rotation[1].

$$U(\theta,\varphi,\lambda) = \begin{bmatrix} \cos(\frac{\theta}{2}) & -e^{i\lambda}\sin(\frac{\theta}{2}) \\ e^{i\varphi}\sin(\frac{\theta}{2}) & e^{i(\varphi+\lambda)}\cos(\frac{\theta}{2}) \end{bmatrix} \tag{2.14}$$

Similarly to how product states are constructed, we can use the Kronecker product to calculate unitaries for single-qubit gates acting on multi-qubit states. An example for a two-qubit system is presented in Equation 2.15, with an X gate acting on the most significant qubit and a Z gate acting on the least significant qubit.

$$X|q_1\rangle \otimes Z|q_0\rangle = (X \otimes Z)|q_1q_0\rangle = \begin{bmatrix} 0 & Z \\ Z & 0 \end{bmatrix} |q_1q_0\rangle = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix} |q_1q_0\rangle \tag{2.15}$$

By themselves, single-qubit gates cannot produce entangled states, as this capability is exclusive to multi-qubit gates. The controlled-NOT (also referred to as CNOT or CX) gate is an example of such a gate. The two qubits involved in the operation are named the control qubit and the target qubit. In essence, if the control qubit is in the $|1\rangle$ state, an X gate will be performed on the target qubit, leaving the control qubit unchanged. Equation 2.16 shows the matrix form of the CNOT gate.

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.16}$$

The true power of entangling gates lies in applying them to superposition states. Performing a CNOT gate using a $|+\rangle$ qubit as the control and a $|0\rangle$ qubit as the target produces the $|\Phi^+\rangle$ state, seen previously in Equation 2.6. $|\Phi^+\rangle$ is one of the four maximally entangled Bell states, where the correlation between qubit states is at its highest. Equation 2.17 demonstrates how to obtain different Bell states using the CNOT gate.

$$\begin{aligned} |\Phi^+\rangle &= \text{CNOT}|+0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\ |\Phi^-\rangle &= \text{CNOT}|-0\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle) \\ |\Psi^+\rangle &= \text{CNOT}|+1\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \\ |\Psi^-\rangle &= \text{CNOT}|-1\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle) \end{aligned} \tag{2.17}$$

[1] As an example, $H = U(\frac{\pi}{2}, 0, \pi)$.

On a final note, universality is an essential concept in quantum computing and is particularly relevant to quantum compiling. A universal quantum gate set can be used to implement any quantum operation, similarly to how we can build any classical logical operation using only NAND gates. The Clifford set (H, S, and CNOT gates) coupled with the T gate is an example of a set with this property. Finding a good mapping between the theoretical version of a quantum algorithm and the physical circuit that implements it using a reduced gate alphabet is a crucial problem that quantum compiling aims to address.

2.2 Quantum Compiling

This section offers a more in-depth look at quantum circuits and provides an overview of the process of quantum compiling. The term *quantum compiling* refers to the act of converting a high-level quantum algorithm into a series of instructions that adhere to the constraints of a target architecture. Despite advancements in quantum algorithm design, the error-prone and resource-starved nature of current and near-term quantum computers justifies the need for low-level compilation tailored to specific device characteristics [33]. Such architectural limitations include the scant number of native quantum operations, sparse connectivity between qubits, high gate and measurement error rates, and circuit depth limits imposed by qubit decoherence.

2.2.1 Visualizing Quantum Circuits

Quantum algorithms are commonly represented using circuit diagrams, such as the one in Figure 2.2. Each horizontal line, or *wire*, represents an individual qubit, and quantum gates are generally depicted as labeled blocks indicating the operation being performed. For controlled gates, a black dot is used to denote the control qubit, which is connected by a vertical line to the other participating qubit(s). The meter symbol designates a measurement operation. Figure 2.3 contains several example gates, including the SWAP operation, which exchanges the state of two qubits and is quite relevant to quantum compiling.

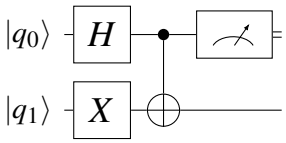
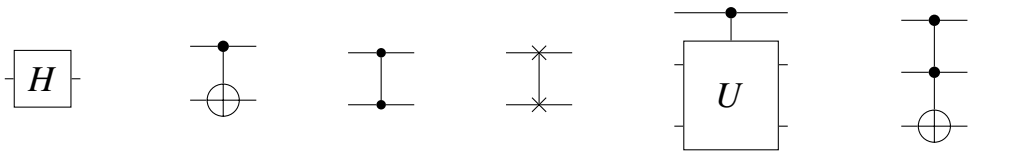


Figure 2.2: Example quantum circuit



(a) Hadamard gate (b) CNOT gate (c) CZ gate (d) SWAP gate (e) Controlled unitary (f) Toffoli gate

Figure 2.3: Depictions of several quantum gates

Operations are executed from left to right, meaning that circuit diagrams also display the dependency relationships between gates performed at different points in time. Figure 2.4 shows a circuit divided into several *layers* representing groups of gates that can be executed in parallel. The number of layers in a circuit's decomposition is referred to as its *depth*, which is one of the metrics that should be minimized during the compilation process, as qubit decoherence imposes strict limits on how deep a circuit can be while still being useful. Moreover, the term *width* is sometimes used to refer to a circuit's qubit count.

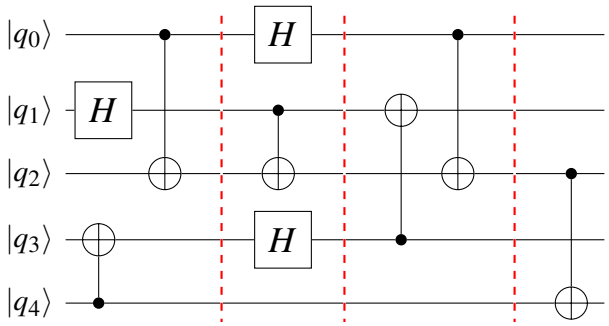


Figure 2.4: Decomposition of a quantum circuit into layers

Translating a quantum algorithm to fit a particular architecture involves multiple transformations, each with a distinct purpose. Thus, it can be helpful to envision compilation as a multiple-stage pipeline that accepts a high-level program as input and produces an equivalent circuit that can be directly executed on the target hardware. The amount and type of stages in the compilation flow can vary between compilers, and some stages can be performed multiple times or only under certain conditions. Here, we shall examine the sequence of steps employed by the Qiskit transpiler [34]. Although this work focuses primarily on the qubit routing stage, we describe each phase of the compilation procedure in the following subsections for context and completeness.

2.2.2 Gate Decomposition

Quantum algorithms can be constructed from a wide range of operations, many of which might not be directly executable on actual hardware. The purpose of the decomposition or translation stage is to rewrite each operation using only the device's native gate set, which is often quite small. As mentioned previously, a universal gate set has the ability to implement all possible unitary operations. Moreover, the Solovay-Kitaev theorem guarantees that any operation can be approximated by a finite gate sequence from a universal set, whose length increases logarithmically with the desired accuracy [35, 36].

Quantum compilers typically accomplish this task by applying a catalog of known decompositions to the operations in the circuit. Figure 2.5 contains an equivalent version of the Toffoli gate (which is essentially a NOT gate controlled on two qubits) that uses only CNOTs and single-qubit operations. The decomposition process almost always increases the depth and gate count of the circuit, meaning circuits that rely extensively on operations with costly decompositions tend to perform poorly on physical devices.

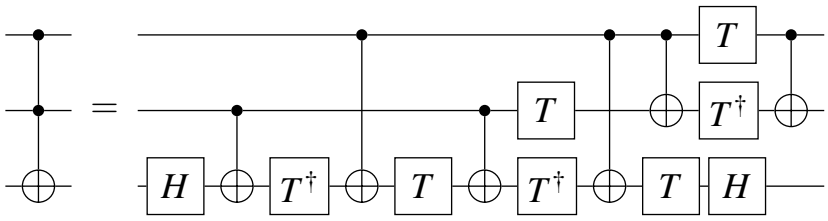
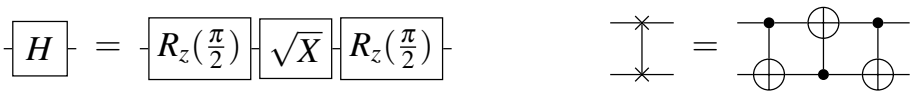


Figure 2.5: Decomposition of a Toffoli gate

The native gate alphabet for IBM systems typically includes the CNOT gate, the $\sqrt{X}$ gate (corresponding to a 90-degree rotation around the X-axis), and a parametrized Z-rotation (R_z) [9]. Figure 2.6 demonstrates how common operations can be expressed using this gate set.



(a) Hadamard gate

(b) SWAP gate

Figure 2.6: Example gate decompositions targeting IBM quantum computers

2.2.3 Quantum Circuit Mapping

Another limitation of most quantum computers is their inability to perform two-qubit operations between arbitrary qubits. Rather, each device has an associated coupling graph containing the available links between qubits. Occasionally, these links may be monodirectional, only allowing a particular orientation for two-qubit gates (although it is possible to switch the direction of a CNOT using Hadamard gates). Qubit connectivity can differ between systems, even those with the same qubit count.

Quantum circuit mapping (sometimes also referred to as layout synthesis [30] or quantum circuit placement [37]) is the act of transforming circuits to ensure they only contain two-qubit operations which are feasible under a given device topology. This task can be divided into two problems: first, picking a suitable initial mapping between virtual and physical qubits, and second, inserting additional SWAP operations to produce an equivalent circuit obeying the connectivity constraints, a procedure known as qubit routing.

2.2.3.1 Initial Mapping

In quantum compilers, the layout or placement stage aims to establish a correspondence between the logical qubits in the original program and the physical qubits of the device. Several factors can affect this decision. While it would be ideal to obtain a mapping that requires no further qubit movement via SWAP gates, this is not always possible. If these optimal layouts exist, the Qiskit compiler can identify them using an algorithm based on subgraph isomorphism (`VF2Layout`) [34].

This stage of the compilation pipeline has a major impact on the size and noise resilience of the final circuit. Finding a good initial mapping is essential, but the search space grows dramatically with the number of qubits in the circuit or the machine. As an example, there are $10! = 3{,}628{,}800$ different ways to map a ten-qubit circuit to a device of the same size.

In addition to choosing a qubit placement resulting in the fewest number of SWAPs during the qubit routing phase, one must also consider the effects of noise on the resulting circuit. Avoiding qubits with short coherence times or high gate and measurement error rates is vital for improving the success rate of compiled programs. IBM computers are usually calibrated daily, and the resulting data is available online or through an API call [9].

2.2.3.2 Qubit Routing

During the qubit routing stage, auxiliary operations are introduced so that each two-qubit gate in the original circuit is performed between neighboring qubits. SWAP gates exchange the state of two qubits, which can be used to reduce the distance between qubits involved in an operation. There are two ways to decompose SWAP gates into three CNOT gates, as shown in Figure 2.7. Since two-qubit gates are considerably less reliable on most quantum computers, the qubit routing stage often has the most impact on the circuit's performance.

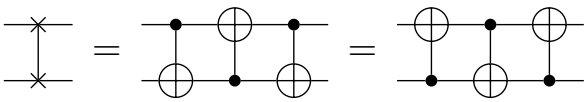


Figure 2.7: Different decompositions of the SWAP gate into CNOTs

It is also possible to use a BRIDGE gate (see Figure 2.8) to perform a CNOT between qubits that share a common neighbor. This operation uses the same number of two-qubit gates as a SWAP gate followed by a CNOT but does not exchange any qubits. Therefore, any benefits of this approach will depend on future optimizations or routing decisions. Relying on the shortest paths in the coupling graph for routing is not always the optimal choice, as some links can have disproportionately high error rates. Furthermore, the compiler should avoid conflicting paths when routing multiple qubits in parallel.

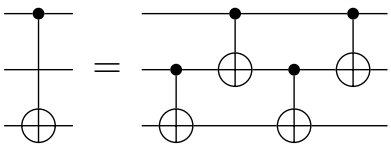


Figure 2.8: BRIDGE gate

2.2.4 Optimization

Although not strictly necessary for obtaining a circuit that satisfies device constraints, the optimization stage is still crucial for running quantum algorithms in a noisy environment. Reducing

a circuit's depth and gate count can mitigate the impact of noise on the outcome of the computation. Compilers can run optimization passes both before and after decomposition and routing, and certain procedures may be repeated until a termination criterion is met.

Several techniques rely on identifying patterns in the circuit structure and replacing them with more efficient but still logically equivalent alternatives, a process dubbed *peephole optimization*. For example, unitary operations such as the Hadamard, Pauli, and CNOT gates are self-inverse (they are their own inverse), meaning that consecutive pairs of gates can be canceled. Furthermore, it is possible to consolidate any sequence of single-qubit gates into a single operation since each gate effectively represents a rotation around the Bloch sphere. One can also replace any successive group of operations acting on the same two qubits with a sequence containing at most three CNOTs [38] (depicted in Figure 2.9).

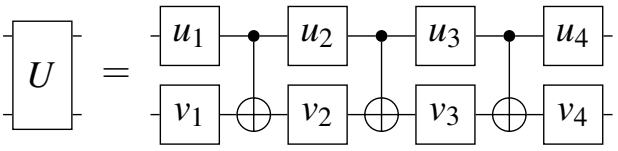


Figure 2.9: Gate sequence that can represent any two-qubit unitary $U \in \mathrm{SU}(4)$ [38]

Certain gates also have the property of commuting with each other, meaning they can be freely rearranged without altering the resulting unitary transformation. Exploiting these commutation rules can enable further optimization opportunities via gate cancellation and block re-synthesis. A concrete example of this kind of optimization is the `CommutativeCancellation` pass found in the Qiskit transpiler. Figure 2.10 presents a few standard commutation relationships.

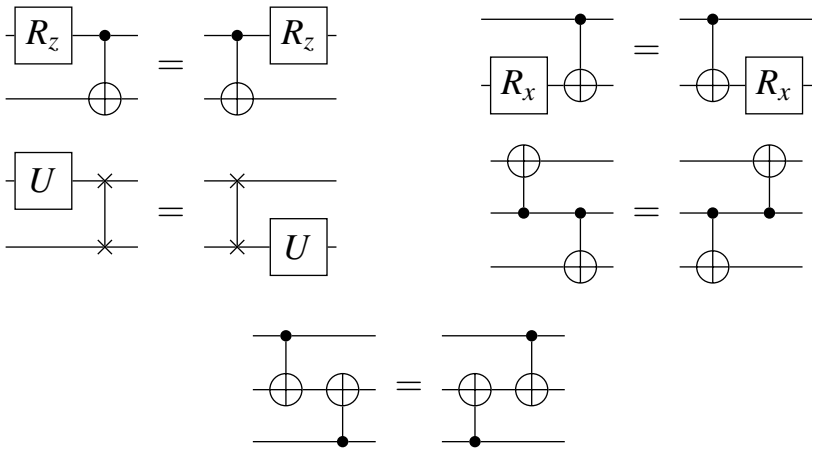


Figure 2.10: Gate commutation rules

2.2.5 Scheduling

Because the duration of operations in a quantum computer can vary depending on which qubits they are applied to, the Qiskit compiler also includes a scheduling stage. This final phase in the

compilation flow consists of inserting delay instructions to synchronize operations and account for periods where qubits are idle. We shall not cover this step in more detail, as it is not particularly relevant to our work; nevertheless, more information is available in Qiskit's documentation [34].

2.3 Variational Quantum Algorithms

Variational quantum algorithms are hybrid quantum-classical algorithms that appear to be the most promising approach for harnessing the capabilities of NISQ devices [19]. Succinctly, VQAs involve continuously adjusting the parameters of a quantum circuit to minimize an objective function related to the problem at hand. The challenging portion of the computation is offloaded to the quantum computer, while the parameter optimization is performed using a classical algorithm. As depicted in Figure 2.11, these algorithms share a set of components that can be individually modified or replaced, namely a cost function, a parametrized quantum circuit, a measurement scheme, and a classical optimizer [39].

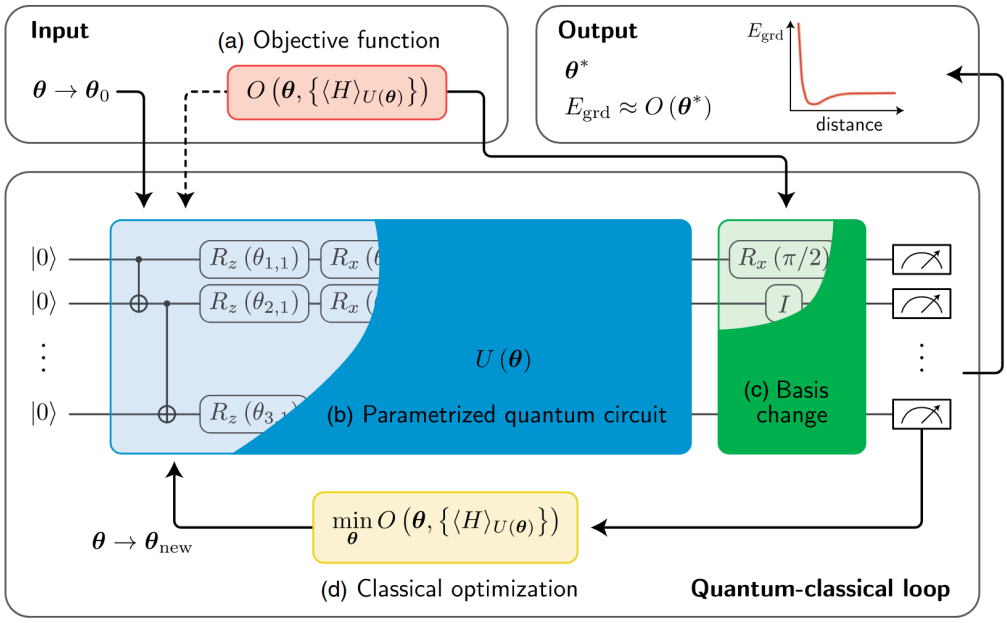


Figure 2.11: Diagram illustrating the typical components of a VQA [39]

2.3.1 Cost Function

The cost (or objective) function (see Figure 2.11 (a)) maps each set of circuit parameters to a numeric value, thus serving as a measurement of error with respect to the specific problem being tackled. The optimization procedure consists of traversing the cost landscape defined by this function in pursuit of the global minimum. An effective cost function should possess a particular set of properties, namely the following:

- **Faithfulness**, meaning the global minimum corresponds to the optimal problem solution;

- **Difficult to evaluate on a classical computer** (otherwise, there is no incentive for employing a quantum computer);
- **Operational meaning**, signifying that smaller values of the cost function correspond to better solutions;
- **Trainability**, meaning there should be a way of efficiently optimizing the cost function parameters.

Designing efficient, short-depth quantum circuits for cost function evaluation is critical for the success of VQAs [19].

2.3.2 Parametrized Quantum Circuit

Another vital component of a VQA is the parametrized quantum circuit (PQC; see Figure 2.11 (b)), containing quantum gates with associated parameters that influence the gate's behavior. The concept of an *ansatz*, denoting an initial guess for a problem that one expects to improve further, is central when discussing parametrized circuits. In this context, an *ansatz* defines the structure of the PQC and can depend on factors specific to the problem being solved or the hardware that will execute the circuit. As such, most are typically categorized as either problem-inspired or hardware efficient *ansätze*.

A problem-inspired *ansatz* relies on information stemming from the underlying physics of the problem. The unitary coupled cluster, used in quantum chemistry, is an example of this kind of *ansatz*. While this strategy can yield circuits that will quickly produce good solutions, these can be difficult to implement on NISQ devices due to considerable depth and connectivity requirements.

Hardware efficient *ansätze* have been presented as an alternative approach that considers the limitations of current quantum devices, such as short native gate alphabets, limited qubit connectivity, and errors caused by imperfect operations and decoherence. These circuits typically adopt a layered structure, with each layer consisting of single-qubit rotation gates or two-qubit entangling gates from the device's native set.

Developing an effective *ansatz* is vital due to its influence on convergence speed and the quality of solutions that can be attained. The complexity of this task is exacerbated by the numerous factors that must be considered, namely problem-specific knowledge, expressivity, trainability, device constraints, and the impact of noise [19, 39].

2.3.3 Measurement Scheme

In VQAs, the unique computational advantage granted by quantum computers provides an efficient way to estimate the cost function associated with a given problem. To perform this evaluation, one usually conducts a series of measurements to determine the probability of measuring certain basis states of an observable (see Figure 2.11 (c)). This process involves a change of basis, which, for most observables, can be achieved by applying single-qubit rotations. Experimentally, expectation

values are estimated from a finite number of measurements (shots), which introduces an amount of error that depends on the sample count [39].

2.3.4 Classical Optimizer

The choice of classical optimization strategy (see Figure 2.11 (d)) plays a prominent role in the effectiveness of a VQA. These algorithms generally feature cost landscapes with many local minima, and optimizing VQA parameters has been proven to be an NP-hard problem [40]. Additionally, one must consider challenges such as circuit depth limits imposed by decoherence, noisy environments, minimizing sampling costs, and the negative effects of barren plateaus.

When studying VQAs, it is customary to classify optimization procedures as either gradient-based or gradient-free, depending on whether or not they rely on the gradient of the cost function. The gradient of a function corresponds to the direction of greatest change, defined as a vector of partial derivatives with respect to each parameter (see Equation 2.18). Gradient descent is a technique that involves iteratively updating the parameters in accordance with the gradient. In contrast, evolutionary algorithms and reinforcement learning are examples of methods that do not depend on gradient estimation.

$$\nabla f(x_1, x_2, \ldots, x_n) = \begin{bmatrix} \frac{\delta f}{\delta x_1} \\ \frac{\delta f}{\delta x_2} \\ \vdots \\ \frac{\delta f}{\delta x_n} \end{bmatrix} \tag{2.18}$$

Recently, we have also witnessed the development of optimization algorithms that attempt to regulate the use of quantum resources. For instance, one may adjust the number of samples for each estimate based on the level of precision needed. These resource-aware strategies strive to balance the effectiveness and cost of the optimization process. [19, 39].

2.4 Reinforcement Learning

This segment succinctly describes foundational reinforcement learning concepts, explains how reinforcement learning can be augmented using deep learning and covers other relevant aspects specific to our work. We also provide a concise summary of the proximal policy optimization algorithm used in our solution. The following subject matter draws largely from Sutton and Barto's influential book [41], which offers a detailed account of the key ideas behind RL, as well as Lapan's practical overview of deep reinforcement learning algorithms and techniques [42].

2.4.1 Base Concepts

Reinforcement learning is one of the three central paradigms of machine learning, alongside supervised and unsupervised learning. Unlike these approaches, which effectively involve identifying patterns within existing data to carry out classification or aggregation tasks, reinforcement learning seeks to determine how an agent should act in a given environment to maximize its cumulative reward. Since the agent has no prior knowledge regarding the optimal behavior to adopt, it must interact with the environment to develop its understanding of which actions are expected to yield the greatest reward. Through its behavior, the agent can affect the environment and, consequently, influence the actions and gains available in the future. In many problems, making successful decisions requires taking future rewards into account, supporting the notion of cumulative reward.

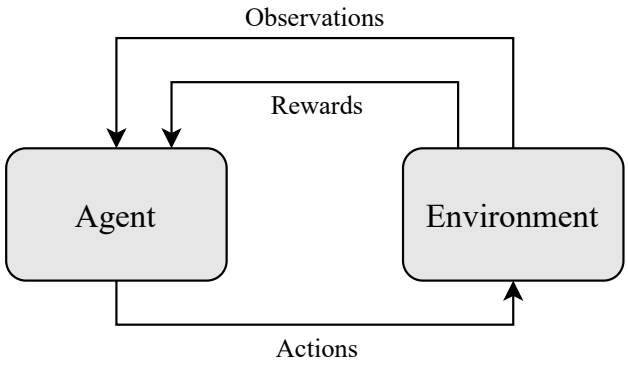


Figure 2.12: Relationship between an agent and the environment

The diagram in Figure 2.12 shows what information is exchanged when the agent interacts with the environment. Actions are classified as discrete if they involve selecting an option from a finite set (e.g., picking the next move in a chess game). Continuous actions, on the other hand, have associated numerical values that cannot be fully enumerated (for instance, controlling the rotation of a mechanical arm). The reward signal informs the agent regarding the success of its previous action. Rewards are issued whenever the agent accomplishes the desired goal or follows some desired behavior. One can also use negative rewards to discourage the agent from performing certain actions or to compel it to seek out more valuable states. The agent bases its decisions on the observations it receives from the environment and knowledge from past experiences. It is important to note the distinction between the environment's state, referring to the information required to describe the environment at a particular moment, and the observations, which are typically a partial or modified description of the environment's state. For example, a poker player is unaware of their opponent's cards, despite that information being part of the state.

A common way to formally model reinforcement learning problems is through Markov decision processes (MDP). Figure 2.13 shows an instance of a simple MDP with three states and two possible actions. MDPs consist of:

- A state space S;
- An action space A;

- A function $p(s'|s,a)$ that expresses the probability that the process will shift to state s' when taking action a in state s;
- A function $r(s'|s,a)$ conveying the reward associated with transitioning from state s to state s' through action a.

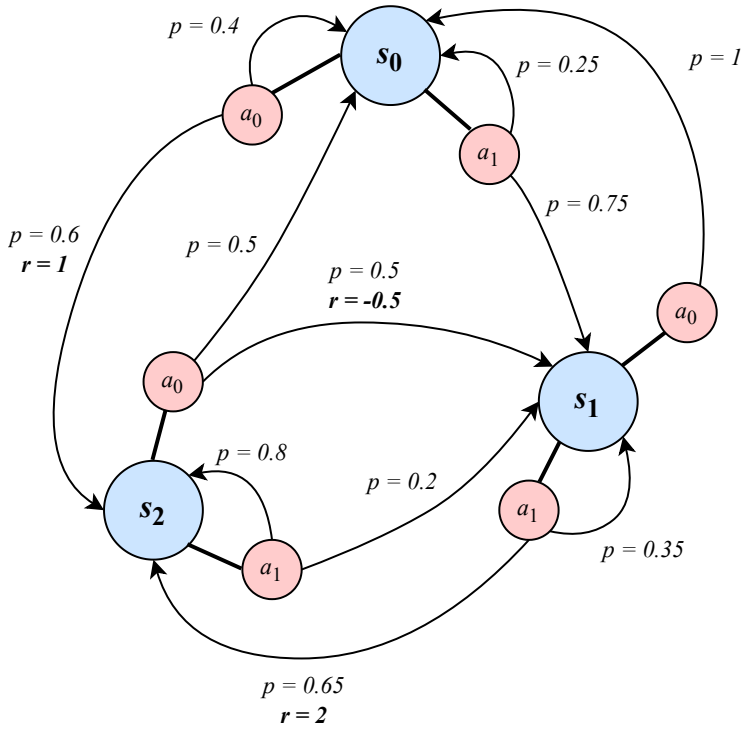


Figure 2.13: A simple Markov decision process

An MDP must satisfy the Markov property, which states that the system's future evolution must depend solely on the current state and not on past interactions, ensuring that each state holds the information necessary to fully characterize the environment. Solving an MDP means finding a policy π mapping each state to an action that maximizes the sum of expected rewards. The policy governs the agent's behavior and is what RL algorithms strive to improve through interaction with the environment. The cumulative return calculation uses a discount factor γ to weigh the relative importance of short and long-term rewards. When the task in question has a clear end point (such as playing a game of chess or solving an optimization problem), each attempt is referred to as an episode. Episodes end when the agent reaches a terminal state or when some other condition is met (for instance, reaching a maximum number of time steps).

$$\sum_{t=0}^{\infty} \gamma^t r(s_{t+1}|s_t, a_t) \tag{2.19}$$

Another relevant aspect of reinforcement learning is the distinction between model-based and model-free RL. In the first scenario, the agent is presented with or constructs a model that simulates the environment's dynamics. In contrast, the latter relies solely on the experiences accrued during

training. Reinforcement learning algorithms can also be broken down into on-policy and off-policy methods. While on-policy algorithms optimize the same policy the agent uses to collect experiences from the environment, off-policy algorithms use different policies to select actions (behavior policy) and learn the ideal behavior (target policy). Off-policy approaches are often more sample efficient since data from past interactions can be reused to train the target policy, whereas on-policy methods typically exhibit greater training stability.

A value function is an essential component of many reinforcement learning algorithms and relates to the intent of maximizing total rewards in the long term rather than greedily focusing on actions that provide immediate incentives. A state's value is determined by the potential future rewards obtainable from it, meaning that an unrewarding state can still have a high value if it helps the agent attain larger benefits in subsequent interactions.

Furthermore, we briefly discuss two of reinforcement learning's main challenges: the exploration-exploitation trade-off and dealing with sparse rewards. The former alludes to the balance that agents must strike between performing actions that have resulted in greater rewards in the past (exploitation) and trying different actions in pursuit of more effective strategies (exploration). An excessive degree of exploration makes it difficult to achieve high rewards; conversely, exploring too infrequently may prevent the agent from discovering more successful policies. One simple way to control exploration is with an ε-greedy strategy, which selects a random action with probability ε instead of the one dictated by the policy.

Sparse rewards are a recurring problem whenever the desired goals require substantial planning effort. When dealing with a complex task, an agent is unlikely to be able to complete it successfully (and thus receive a positive reward) solely through trial and error. Several techniques have been proposed to mitigate this issue, including one known as reward shaping. Shaping the reward function equates to providing feedback to the agent through smaller incentives as it approaches the intended outcome or dividing the problem into a sequence of more easily achievable steps with intermediate rewards. Nonetheless, this strategy can sometimes lead to unwanted behavior as the agent learns to exploit a flaw in the reward function rather than accomplishing the true objective. Moreover, in some problems, it can be inherently difficult to judge whether an action will bring the agent closer to the goal.

2.4.2 Deep Reinforcement Learning

The incorporation of deep learning has been a decisive factor in the success experienced by reinforcement learning methods in real-world use cases. The field of deep learning centers on using multi-layer artificial neural networks to extract patterns from data through abstraction and has achieved remarkable results across various domains, matching or even surpassing human-level performance [43].

Traditional tabular reinforcement learning techniques operate by continuously keeping track of value function estimates for every possible input. As an example, Q-learning maintains a table of the total expected rewards associated with each state-action pair, referred to as a Q-table or Q-function. In practice, the usefulness of such approaches is limited by the fact that many real-world

problems suffer from prohibitively large observation spaces (e.g., autonomous driving, robotics, or natural language processing tasks). In deep reinforcement learning algorithms, neural networks generally assume the role of the policy or value function, owing to their exceptional ability to learn and approximate functions. Using neural networks as estimators enables the use of RL agents for tasks that involve high-dimensional inputs, such as proficiently playing Atari games using only raw pixel data [44].

2.4.3 Proximal Policy Optimization

Proximal policy optimization (PPO) refers to a group of on-policy reinforcement learning algorithms introduced by OpenAI in 2017 [45], under the goal of balancing ease of implementation, training efficiency, and robustness to hyperparameter tuning. PPO falls under the category of policy gradient (PG) methods, which directly optimize the policy by computing its gradient with respect to a set of parameters. Instead of randomly sampling from a large buffer of past experiences, such as in deep Q-learning, PG algorithms update the policy parameters using batches of experiences obtained under the current policy.

Performing multiple iterations of gradient descent using the same batch of experiences would be ideal for accelerating the training process. However, this practice can result in overly large policy updates, nullifying previous training efforts. Trust region policy optimization (TRPO) was proposed in a 2015 paper by Schulman et al. [46] as a means to address this problem. The main idea behind TRPO is to limit the extent to which the updated policy can deviate from the old policy. The conventional way to estimate the policy gradient uses the expression in Equation 2.20.

$$\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{2.20}$$

The term $\log \pi_\theta(a_t|s_t)$ refers to the log probability of selecting action a_t in state a_t under policy π_θ. $\hat{A}_t$ represents the advantage function, corresponding to the difference between the discounted sum of rewards encountered in the current batch of experiences and an estimate typically computed using a value network. A positive advantage means that the chosen action was more rewarding than initially expected, while a negative advantage indicates that the value of the action was overestimated. The advantage estimate helps determine if the selected actions should be prioritized or avoided in the future. Lastly, $\hat{\mathbb{E}}_t$ denotes that we are calculating an expectation value based on an average of the samples in the current batch. TRPO instead uses the ratio between action probabilities in the new and old policies and introduces an additional constraint based on the KL divergence [47] between policies, as shown in Equations 2.21 and 2.22.

$$\max_\theta \quad \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t\right] \tag{2.21}$$

$$\text{subject to} \quad \hat{\mathbb{E}}_t\left[\text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]\right] \leq \delta \tag{2.22}$$

PPO follows the same principle of imposing a limit on how much the policy can vary but instead does so directly through its objective function. By clipping the probability ratio to within a set interval (as seen in Equation 2.23, where $r_t(\theta)$ represents this ratio), the algorithm prevents excessive updates to the policy without the need for extra constraints. The hyperparameter ε controls the clipping interval and, thus, how much the policy is allowed to deviate. This enhancement makes PPO simpler to implement compared to TRPO and often results in improved training performance. The complete loss function for PPO (presented in Equation 2.24 includes two additional terms, the value function loss (as it is common for the policy and value networks to share parameters), and an entropy term S that can be used to encourage exploration. The c_1 and c_2 coefficients are hyperparameters that dictate the importance of these extra elements.

$$L_t^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t) \right] \tag{2.23}$$

$$L_t^{\text{CLIP + VF + S}}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{\text{CLIP}} - c_1 L_t^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t) \right] \tag{2.24}$$

2.4.4 Action Masking

One of the obstacles that can arise when designing a reinforcement learning environment to address a particular problem is the existence of invalid actions. In many tasks, the set of feasible actions depends on the current state of the environment, and neglecting this factor can hinder an algorithm's performance and trainability. In [48], Huang and Ontañón investigate this problem in the context of policy gradient algorithms (such as PPO) and study the effectiveness of the technique known as invalid action masking.

One possible strategy when dealing with invalid actions is to preserve the current environment state and optionally issue a negative reward to deter the agent from attempting that action in the future. Nevertheless, this approach suffers from a glaring weakness in that it does not prevent the agent from selecting invalid actions; the agent could still choose to accept the penalty depending on its severity compared to the other rewards. In addition, any time steps in which a forbidden action is taken are effectively wasted, resulting in slower training, especially when a greater portion of actions are invalid.

Policy gradient methods typically use a neural network to model the policy being refined. In discrete action spaces, the values output by the final layer of the policy network (referred to as logits) are passed to the softmax function, which produces a probability distribution over the set of actions. The idea behind action masking is to replace the logits corresponding to invalid actions with a large negative number so that the probability of selecting those actions will be negligible after applying softmax.

The article mentioned above argues for the theoretical soundness of this approach by observing that the masked values still yield a valid policy gradient. The authors also measured the efficacy of penalty and masking strategies using a test environment consisting of a simplified strategy game. While the practice of penalizing the agent proved successful for small-scale environments, it failed in larger game maps due to the growing number of invalid actions and the sparseness of rewards.

In contrast, action masking was effective across all environment dimensions tested and also helped the agent obtain its first reward faster during training [48].

Chapter 3

State of the Art

In this chapter, we review the existing literature on quantum circuit mapping, primarily focusing on approaches that utilize reinforcement learning or consider the effects of noise more thoroughly during the routing process. We cover multiple algorithms and frameworks, examining the employed methodology, limiting factors, results obtained, and future research opportunities.

In [49], Pozzi et al. propose a strategy for qubit routing that combines Deep Q-learning (DQN) and simulated annealing. Instead of using the number of additional SWAP operations (or two-qubit gates) as the optimization objective, the procedure seeks to minimize the depth of the resulting circuit. The authors argue that deep circuits, regardless of whether or not they contain fewer overall gates, have limited usefulness in the NISQ era, as decoherence can occur even when qubits are idle. This decision presents additional challenges since envisioning actions as layers of SWAP operations (as opposed to the more straightforward approach of actions corresponding to a single SWAP) results in a combinatorial action space that grows exponentially with the number of links in the target architecture.

In modeling qubit routing as a reinforcement learning problem, the article defines states as being comprised of the current mapping between physical and logical qubits and, for each logical qubit, the next qubit it interacts with in the circuit and the number of interactions (two-qubit gates) it has performed thus far. Figure 3.1, taken from the paper, depicts an example state of the algorithm, overlaying a graph representing the topology of the underlying device. The numbers at the top correspond to the logical qubits associated with each physical node. The target qubits for the next interaction are presented at the bottom (a value of -1 means that the qubit is not involved in any further two-qubit gates). Edges in red denote SWAP operations, while the green edge shows a two-qubit interaction that can be executed under the current mapping.

Due to the sheer number of possible actions, standard Q-learning based on state-action pairs would be infeasible. To address this problem, the algorithm employs a modified Q-function that expresses the value of transitioning between states and uses simulated annealing, a well-known strategy for combinatorial optimization, to select an action resulting in the highest-value state transition. The agent is rewarded for scheduling two-qubit gates from the original circuit and for reducing the distance (as determined by the device topology) between mutually targeting qubits.

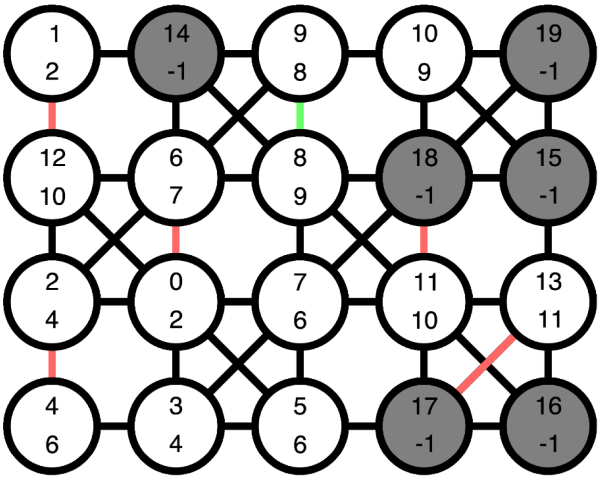


Figure 3.1: Visualization of the state of the DQN routing algorithm [49] at a given point in its execution

For improved learning, two common extensions of DQN are used, specifically Double Q-learning and Prioritized Experience Replay.

The authors compare their approach with several prominent quantum compilers, namely Qiskit from IBM [7], Cirq from Google [8], and t|ket⟩ from Cambridge Quantum Computing [50]. The initial mappings were chosen randomly to focus exclusively on the routing performance. The main metric used for evaluation was the ratio between the depth of the routed circuit and the original circuit. When tested on random circuits, the DQN algorithm generally matched or outperformed t|ket⟩ and managed to beat Qiskit's `StochasticSwap` when routing sparser circuits (with fewer CNOTs per layer). When assessed on a set of realistic circuits, DQN achieved lower circuit depth ratios than Qiskit and t|ket⟩ across three different architectures.

The fundamental limitations of this RL strategy revolve around the lack of a penalty associated with the additional gate count incurred by the routing procedure and the assumption that SWAP operations require a single time step (or layer) to execute. These factors can result in the inclusion of redundant SWAPs, leading to higher gate counts despite the reductions in depth. Moreover, when the textbook decomposition of SWAP gates into three CNOTs is used, DQN performs worse than its alternatives when routing realistic circuits (although it maintains its lead for random circuits). Therefore, tackling these drawbacks remains the most pressing avenue for future improvements. Nonetheless, other valuable contributions, such as introducing support for BRIDGE gates, incorporating information about individual gate error rates, and developing a more elaborate state representation, are also suggested.

Deep reinforcement learning was also explored in [37], this time in conjunction with a genetic algorithm. The article frames quantum circuit mapping as a two-level optimization problem: the RL agent handles the lower-level qubit routing task while an evolutionary search is used to look for a good initial mapping.

In contrast to the previously discussed approach, the environment state consists of a matrix

representation of the circuit that encodes information about the current qubit mapping and any dependencies between two-qubit gates. Figure 3.2 demonstrates the construction of a state matrix from a given quantum circuit. Each row corresponds to a physical qubit, and each column corresponds to a time step (or layer). The cell values equate to the qubit targets (or -2 if there is no two-qubit interaction at that moment). Furthermore, an action specifies the edge of the coupling graph where the next SWAP operation should be inserted, which equates to exchanging the associated rows of the state matrix.

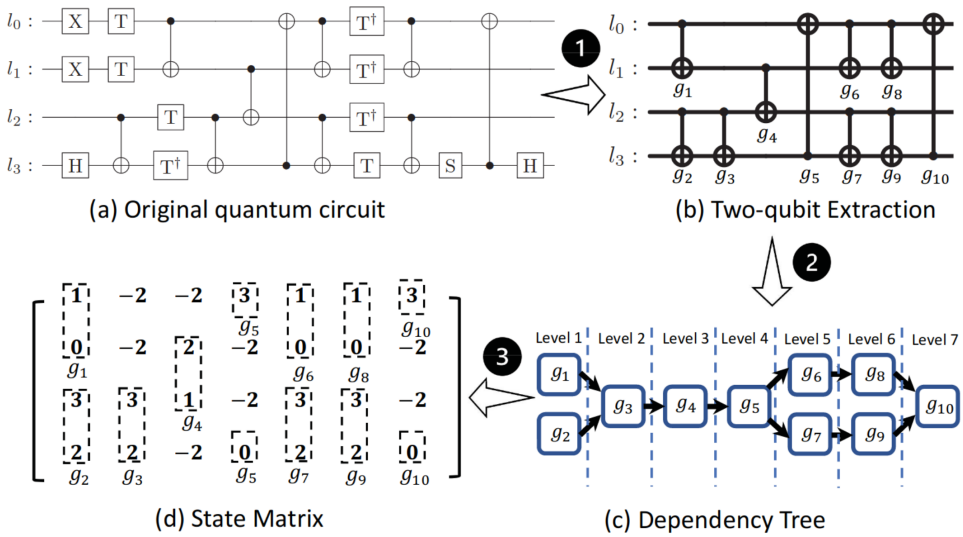


Figure 3.2: Procedure for deriving a state matrix from a quantum circuit [37]

The agent earns a positive reward for each two-qubit gate performed and for completing the routing process, whereas scheduling a SWAP operation or being in a state where no gates can be executed after a SWAP both result in a penalty. The state matrix is fed directly to the neural network's input layer, with Proximal Policy Optimization (PPO) being used for deep reinforcement learning. Regarding the evolutionary strategy used to optimize the initial mapping, the population consists of a set of mappings with fitness equal to the total reward obtained by the agent during the routing process. The next generation is produced by selecting the fittest individuals and applying crossover and mutation operators.

Across a selection of circuits with varying qubit counts, targeting three different device architectures, the proposed technique utilized fewer SWAP gates than heuristic algorithms from the Qiskit and t|ket⟩ compilers. Additionally, it produced similar results to an exact search method while significantly reducing runtime. Increasing the population size reduced the SWAP cost at the expense of runtime, though the improvements varied by circuit. Future research directions identified by the authors include extending this approach to larger circuits and performing noise-aware compilation.

Alternatively, Zhou et al. [51] have devised a technique for qubit routing based on Monte Carlo Tree Search (MCTS), a heuristic search algorithm with ties to reinforcement learning that was shown to be exceptionally effective at playing board games such as Chess and Go [52]. In

MCTS, a problem is represented as a search tree where nodes correspond to states and branches to valid actions within those states. By focusing on the most promising areas of the tree and using random sampling to extend the search space, the algorithm aims to strike a balance between exploration and exploitation.

Each state consists of an assignment of logical to physical qubits, a physical circuit containing already scheduled gates, and a logical circuit with gates that have yet to be executed. States also have associated cost equal to the number of SWAP operations (actions) performed since the initial state. In an effort to minimize this cost, the algorithm uses a reward formulation that incorporates both short and long-term components. The short-term reward equates to the number of two-qubit gates rendered possible after a SWAP, whereas the long-term value of being in a particular state is estimated through efficient random simulation. When expanding the search tree, only SWAP operations on qubits involved in two-qubit interactions in the front layer of the logical circuit are considered, reducing the branching factor.

When juxtaposed with other heuristic search techniques (including simulated annealing and subgraph isomorphism), MCTS achieved substantial reductions in the amount of additional CNOT gates. Designing a more sophisticated objective function incorporating factors such as circuit depth and gate reliability and relaxing the policy used to identify relevant SWAP operations could constitute potential enhancements to this framework.

In [53], Sinha et al. present another example of Monte Carlo Tree Search applied to the qubit routing problem, in this case utilizing graph neural networks (GNN). The adopted methodology bears some similarities to the strategy based on DQN and simulated annealing already examined in this review, notably in its goal of minimizing added circuit depth and in the way actions are defined. At each step in its execution, QRoute immediately schedules all gates made viable under the current qubit assignment and uses MCTS to obtain a group of SWAP operations that can be executed in parallel. The algorithm leverages a graph neural network to estimate the value of nodes in the tree, helping it limit the search depth.

The framework defines a state as a 4-tuple encompassing the device topology, the current qubit mapping, a list of unscheduled gates, and the set of physical nodes presently locked due to ongoing operations. This use of mutex-like variables is helpful, considering that actions are regarded as parallelizable layers of operations. In contrast to what we observed with the DQN approach, QRoute models action construction itself as a Markov Decision Process to handle the combinatorial explosion within the action space. Actions consist of a series of moves that either insert a new valid SWAP or apply the current set of exchanges.

For each node, the algorithm also keeps track of how many times it has been visited and the Q-values computed by the neural network, which reflect the estimated future discounted rewards. These metrics help regulate exploration and exploitation when expanding the search space, and the Q-values are used to select the most valuable action in the current time step after finishing the search. Following feature extraction, the neural network receives a state representation comprised of a square boolean matrix encoding information about upcoming qubit interactions, the physical links currently restricted due to ongoing operations, and the number of gates left to schedule for

each logical qubit. Figure 3.3 presents an architectural diagram of the GNN. The matrix of node targets undergoes an edge convolution layer utilizing the hardware's connectivity graph.

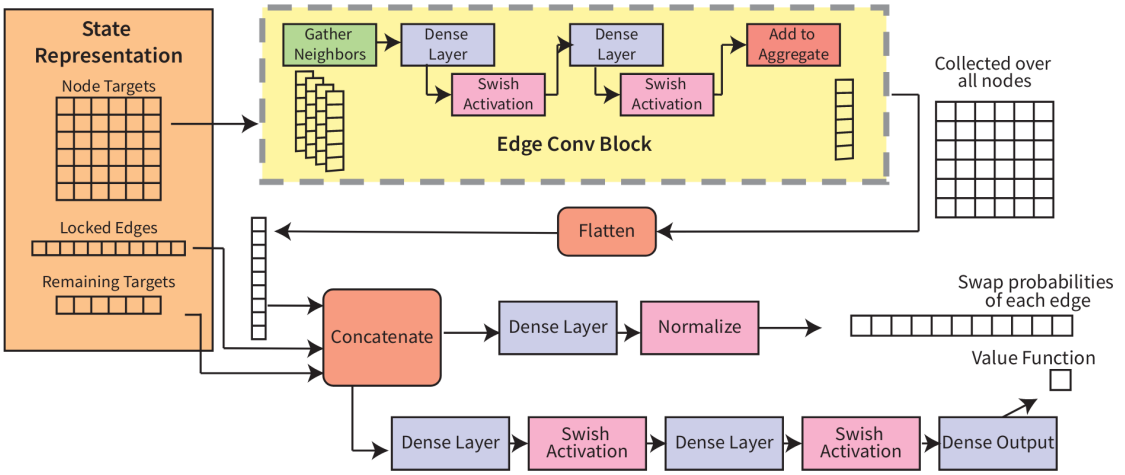
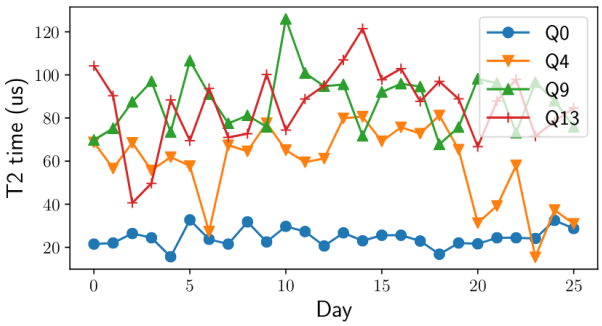


Figure 3.3: Architecture of the graph neural network [37]

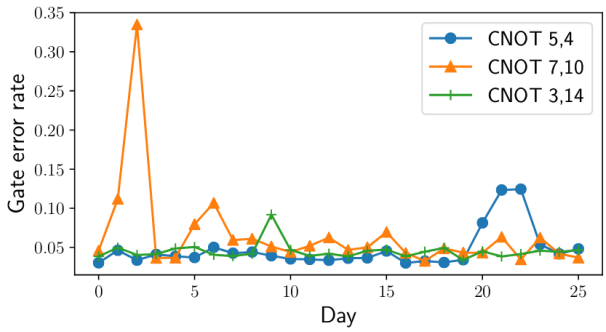
QRoute was benchmarked against existing strategies from the Qiskit, Cirq, and t|ket⟩ compilers, as well as results from the DQN article authored by Pozzi et al. [49]. On randomly generated circuits, the framework was slightly surpassed by DQN yet was still ahead of other algorithms. For small, realistic benchmarks, QRoute managed to edge out the deep reinforcement learning approach in most cases. In larger experiments (with up to 5960 gates), MCTS pulled ahead of Qiskit and t|ket⟩ (as DQN was not able to scale to such dimensions). Despite this, the most glaring weakness of QRoute is that it does not consider gates that occur after the front layer when routing. Designing a way to incorporate this data in the neural network input or considering BRIDGE operations and gate commutation during routing could boost the algorithm's performance.

In addition to metrics such as circuit depth or the number of additional two-qubit gates, one can also judge the effectiveness of routing algorithms by examining the reliability of compiled circuits. Murali et al. study this perspective in [54], presenting exact and heuristic procedures for qubit routing that leverage noise and error-related information originating from device calibration. This data includes qubit relaxation and coherence times, as well as gate and measurement error rates. As evidenced by the plots in Figure 3.4, these values are highly volatile, meaning that the most reliable qubits and links can vary day by day. These results motivate the development of techniques that strive to minimize the likelihood of errors rather than strictly aiming for more compact circuits.

Since error rates for single-qubit operations are substantially lower on most quantum computers, the compiler only considers the effects of two-qubit gates and measurements. Besides selecting reliable physical nodes and links, a noise-aware compilation must ensure that all gates execute within the coherence time of the qubits. The article experiments with different objectives, specifically minimizing execution time and maximizing program reliability, defined as the product of the reliabilities of individual gates in the compiled circuit. The authors also point out that



(a) Coherence time (T2)



(b) CNOT gate error rate

Figure 3.4: Qubit coherence times and gate error rates over time for the IBM Q 16 Rueschlikon quantum computer [54]

optimizing for reliability will often inadvertently reduce the number of costly SWAP operations.

The article first models noise-adaptive compiling as a constrained optimization problem and uses a satisfiability modulo theory (SMT) solver to produce optimal solutions. The constraints enforce that gates are scheduled according to their dependencies and prevent the existence of routing conflicts. To ensure linearity, the reliability objective function is instead defined as the sum of the logarithms of the gate reliabilities. The optimization problem is formulated solely for devices exhibiting a grid-like architecture, and two distinct routing policies are presented to reposition qubits while avoiding collisions.

Alternatively, the article also covers two heuristic methods with superior scalability: GreedyV and GreedyE. The first strategy seeks to place qubits that participate in many distinct interactions in locations with high readout and gate reliability. In contrast, GreedyE instead prioritizes the interactions (gates) that occur most frequently and tries to assign them to the best sections of the device. Both techniques employ Dijkstra's algorithm using the negative logarithm of the edge error rates to determine the most reliable paths to route qubits.

The algorithm variants were evaluated using common quantum algorithms and subroutines, with randomly generated circuits being used to gauge their scalability. Each circuit was tested on the IBM Q 16 Rueschlikon with a set number of shots to determine its success rate. When optimizing for reliability, the exact method (referred to as R-SMT) achieved substantial improvements in program success over the Qiskit compiler. Moreover, this strategy also proved universally

superior compared to minimizing execution time (T-SMT). The authors argue that, while it is suitable to consider qubit coherence time, high gate error rates currently impose the strictest limits on the usefulness of quantum programs. Regarding the performance of the heuristic algorithms, GreedyE generally prevailed over GreedyV, even managing results comparable to R-SMT but with markedly shorter compilation times. As a final note, it is worth mentioning that some aspects from the paper have since been incorporated into the Qiskit framework in the form of a layout pass (`NoiseAdaptiveLayout`), which can be used to obtain noise-aware initial qubit mappings [55].

Similar reasoning was adopted in [56], which used a heuristic algorithm based on beam search to boost the estimated success probability (ESP) of routed circuits, a metric equivalent to the circuit reliability defined in the previous article. Beam search is a variant of breadth-first search that only expands a fixed number of promising states (based on a scoring function). It was used to navigate the state space of the routing problem, coupled with a heuristic technique similar to GreedyE from [54] to obtain initial qubit mappings. As the output of most quantum circuits is typically not a singular value but a probability distribution of potential outcomes, the article uses Kullback-Leibler (KL) divergence [47] to compare the ideal and measured distributions. This method has the caveat of not being suitable for circuits that cannot be simulated classically.

The authors also conducted a range of experiments to evaluate the effectiveness of circuit reliability for predicting actual performance, observing that the truly optimal route or operation did not coincide with the estimate in a notable portion of cases. Proposed explanations for these results include the testing limitations (the experiments cannot detect phase-related errors), the effects of cross-talk, or fluctuations in the device's error rates.

In [57], Liu et al. investigate a unique and intriguing perspective on the qubit routing problem, titled "Not All SWAPs have the Same Cost" (NASSC). Since some compiler optimizations usually carried out after routing can reduce the true cost of added SWAP gates, the authors defend that truly optimal routing policies should be optimization-aware. In this sense, NASSC uses knowledge about future optimizations to guide routing and SWAP decomposition choices. Hence, the primary evaluation metric becomes the number of additional CNOT gates in the final circuit.

The NASSC algorithm is designed to be incorporated into the Qiskit compiler and uses information from two optimization passes, namely two-qubit block re-synthesis and commutative gate cancellation. The first of these procedures hinges on the fact that any sequence of gates acting on the same pair of qubits can be condensed down to a block consisting of three CNOTs and eight single-qubit gates. Depending on the amount of two-qubit gates in the sequence, this transformation may even fully negate the cost of a SWAP gate. The second optimization corresponds to a `CommutativeCancellation` pass, in which the compiler analyzes sets of commuting gates and cancels out self-inverse operations, such as CNOT, Hadamard, or Pauli gates. It is essential to point out that SWAP decomposition can affect the opportunities for this kind of optimization, as there are two different ways to express a SWAP as a sequence of three CNOT gates.

The data required by NASSC is obtained by running the optimization passes before the layout and routing stages in Qiskit's compilation pipeline. The algorithm uses the existing `SabreLayout` strategy to select the initial qubit mapping. When scheduling SWAP operations, NASSC examines

neighboring gates for applicable optimizations and selects the best SWAP candidate, as determined by a heuristic cost function. The article defines two cost functions: one that solely considers the reduction in CNOT gates due to optimizations and another that includes an additional lookahead component, equating to the cost of routing a fixed number of subsequent layers. The extended cost function disregards optimizations in lookahead layers since they are less significant than the front layer. Regardless, any estimates would be approximate due to the lack of knowledge about future routing decisions. The time complexity for routing each two-qubit gate is shown to be $O(N^{2.5})$, where N stands for the number of physical qubits.

The paper compares the performance of NASSC against the SABRE routing algorithm, already implemented in Qiskit, across a collection of realistic circuits of varying dimensions, targeting three backends with distinct topologies. On average, the optimization-aware technique required 21.30% fewer additional CNOTs and frequently also managed slight reductions in circuit depth. Experiments revealed that NASSC often performs better on larger circuits and when compiling for devices with greater connectivity, as these factors result in more room for optimization. The authors also experimented with noise-aware compilation by including gate error rates in the qubit distance matrix, but this actually yielded lower program success rates. These findings hint that further adjustments to the cost function or its parameters might be necessary.

The concept of enhancing qubit routing schemes by exploiting gate commutation rules had also been explored by Itoko et al. in a 2020 paper [58], which additionally gauged the benefits of augmenting the set of available routing operations with BRIDGE gates. The article provides an exact method rooted in dynamic programming as well as a heuristic lookahead algorithm that strives to reduce the shortest path lengths between interacting qubits.

Routing strategies that operate on a layer-by-layer basis tend to rely on a directed acyclic graph (DAG) representation of the original circuit that encapsulates the logical dependencies between gates in different layers. The authors extend this representation by incorporating information garnered from applying commutation identities (such as those discussed in Section 2.2.4 of the Background) to the operations in the circuit. In essence, commuting gates are regarded as belonging to the same layer (regardless of their placement in the original DAG), resulting in greater flexibility during the routing process. Benchmarks exhibited considerable savings in the number of CNOT gates compared to fixed-layer formulations. The inclusion of BRIDGE gates also improved results in general, though the benefits were most noticeable in architectures with inferior connectivity (specifically, linear nearest neighbor).

Beyond the main works already covered in this review, various other techniques have been applied to quantum circuit mapping. These range from optimal approaches, such as temporal planning [59, 60] and constraint programming [61], to approximate methods like evolutionary algorithms [62]. Another recurring tactic consists of reducing the compilation task to equivalent problems that have been previously studied. [63] optimally solved instances of quantum circuit mapping by modeling it as a maximum satisfiability (MAXSAT) problem, whereas Siraichi et al. liken it to a mixture of subgraph isomorphism and token swapping [64]. The previously mentioned SABRE (SWAP-based BidiRectional search) algorithm, available within the Qiskit framework,

uses a heuristic cost function to select SWAP operations from a reduced candidate list. Afterward, it computes a good initial mapping by routing the circuit in reverse, thus prioritizing gates near the start of the original circuit [65].

A compelling yet still developing perspective on quantum compiling is the idea of leveraging quantum computers themselves during the compilation process. The VQA introduced in 2019 by Khatri et al., known as quantum-assisted quantum compiling (QAQC) [66], is arguably the most prominent example of this approach. This hybrid algorithm seeks to maximize the fidelity between a fixed unitary U corresponding to the target circuit and a trainable unitary V representing the compiled circuit. Simulated annealing is used in small-scale problems to optimize discrete parameters (circuit structure) in addition to continuous (gate) parameters. However, due to the exponentially increasing circuit design space, larger problems use a fixed *ansatz* and only involve continuous optimization. The main drawback of the algorithm's cost evaluation circuits is that compiling an n-qubit unitary for all possible input states requires $2n$ qubits. QAQC exhibited a remarkable resilience to noise; follow-up research formally confirmed this property and conducted further empirical analysis [67].

Summarizing the key findings stemming from this literature review, we first observe that multiple exact and heuristic strategies have been proposed to address the quantum circuit mapping problem. However, as highlighted by Tan and Cong in [30], current approximate techniques still exhibit significant performance gaps when compared to known optimal solutions. Due to the problem's complexity, exact methods are not expected to scale to larger architectures and circuits, encouraging the use of machine learning approaches. In the available literature, we also find algorithms that seek to adapt compiled circuits to the error-prone nature of current quantum devices by prioritizing program success over raw circuit size. The combination of reinforcement learning with noise-resiliency measures constitutes a research gap that this work aims to fill.

Chapter 4

Problem Statement and Methodology

This chapter defines the qubit routing problem (Section 4.1) and details the methodology adopted to tackle it (Section 4.2), explaining the decisions involved in modeling qubit routing as a reinforcement learning environment, describing the individual elements of our solution, and concretely outlining the extent of our research.

4.1 Problem Formulation

The rigid constraints and unreliable nature of NISQ-era quantum computers justify the need for effective ways of compiling quantum algorithms. As observed in the previous chapter, the literature on qubit routing is still developing, and opportunities for deepening existing research and integrating aspects from different strategies are abundant. Accordingly, this dissertation proposes an improved method for qubit routing leveraging deep reinforcement learning.

Qubit routing can be described as a combinatorial optimization problem as it involves selecting operations from a finite set to obtain an optimally routed circuit that adheres to device constraints. Its input data consists of:

- A coupling graph G_C for the target device, whose edges indicate the allowed two-qubit operations between physical qubits. Since all device architectures studied in this work do not place restrictions on the direction of qubit interactions, we assume this to be an undirected graph. As was already noted, the orientation of a CNOT gate can be inverted using Hadamard gates, so it would be easy to extend our methods to support such constraints;
- The circuit to route, which can be given as a list of operations (each specifying the gate type and logical qubits involved). We assume that the circuit includes only one and two-qubit gates (that is, gate decomposition has already been performed);
- An initial mapping $P \rightarrow L$ between physical and logical qubits. The choice of this mapping can have a considerable impact on solution quality. As our work is solely focused on routing, we use the same initial layout when evaluating different methods;

- Information capturing the noisy behavior of qubits and gates on the target device, relevant only for noise-adaptive algorithms. Our method depends solely on the error probabilities of two-qubit operations.

By introducing auxiliary operations, one can obtain a functionally equivalent circuit where the physical qubits involved in each two-qubit gate are connected in the device graph. Our approach considers both SWAP and BRIDGE operations during the routing process.

In qubit routing, several metrics can be used to judge solution quality. Since we seek to develop a technique that can adapt to differences in the error probabilities of two-qubit gates, we define the optimization goal as the maximization of circuit reliability, estimated as the product of the individual reliabilities of two-qubit gates in the routed circuit. We do not consider the impact of single-qubit gates (which typically have considerably lower error rates), crosstalk, or state preparation and measurement (SPAM) errors.

Given this formalization of qubit routing, our work aims to answer the following research questions:

- **RQ1**: How does a strategy based on deep reinforcement learning stack up against other qubit routing algorithms employed by existing quantum compilers in terms of the fidelity and compactness of resulting circuits?
- **RQ2**: When targeting realistic quantum architectures, to what extent can awareness about device error rates help reinforcement learning models route circuits more reliably?
- **RQ3**: How does the reinforcement learning approach fare across different device topologies with respect to both the routing performance and the resources required to train the model?

4.2 Methodology

Our motivation for applying deep reinforcement learning to qubit routing hinges on the following aspects:

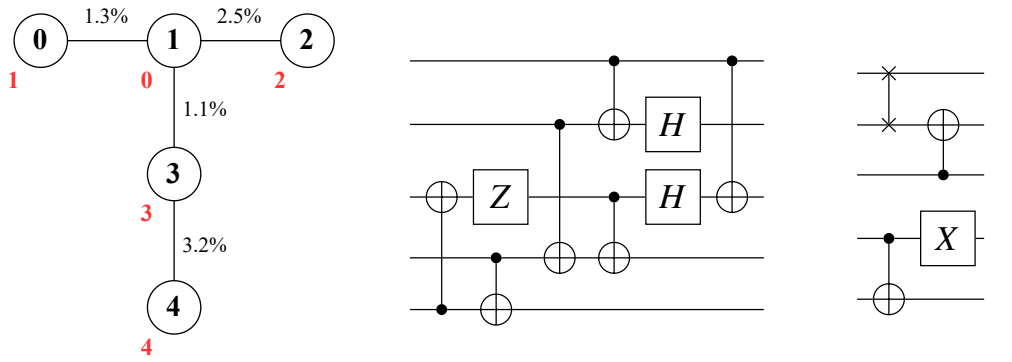
- The qubit routing problem has been proven to be NP-complete [29], and RL has proven to be effective at solving challenging optimization problems;
- The problem involves making a series of sequential decisions that affect its state, meaning it can be adequately modeled as an RL environment;
- Due to the problem's complexity, obtaining optimal solutions is infeasible when targeting quantum computers of a large enough size. This characteristic favors RL over other machine learning techniques (namely supervised learning) since it means that learning can only occur via interaction with the environment;
- Choices made earlier in the routing process can dictate later opportunities for optimization. The ability of RL agents to consider long-term rewards in addition to short-term gain is valuable in this situation.

4.2.1 Qubit Routing as a Reinforcement Learning Problem

The three main steps when formulating a task intended to be solved using reinforcement learning consist of defining the state (and observation) space, establishing the set of possible actions, and developing an appropriate reward function. The state should encompass all information required to fully characterize the environment at any given time step, which, for our purposes, includes:

- The current mapping between physical and logical qubits;
- A list of gates from the original circuit that have yet to be routed;
- A partial solution containing already scheduled gates as well as any additional SWAP or BRIDGE operations;
- The current error rates associated with each valid two-qubit gate.

The device coupling graph is regarded as an intrinsic aspect of the environment rather than being part of the state. Figure 4.1 provides a visualization of the different state components. The numbers in red denote the logical qubits currently occupying each physical node. A state is terminal if there are no more gates to schedule, such that each episode involves routing a single circuit.



(a) Current mapping and error rates (b) Gates to schedule (c) Partial solution

Figure 4.1: A sample state of our qubit routing environment

As stated in the literature review, most articles on qubit routing with reinforcement learning resort to one of two approaches when specifying the action space: an action can correspond to a single SWAP gate or an entire layer of parallelizable SWAP operations. We have previously remarked that the latter approach leads to a combinatorial action set that grows exponentially with the size of the target quantum device and is more suitable when trying to minimize the depth of compiled circuits. For these reasons, we opted for the more straightforward gate-by-gate approach. However, unlike the methods in the literature, we also allow the use of BRIDGE gates. Enumerating all valid BRIDGE gates is as simple as finding every pair of physical qubits whose shortest path distance in the device's coupling graph is exactly two. These actions do not need to specify

the direction of the BRIDGE gate, as that can be determined contextually from the environment state.

Designing a reward function that can effectively guide the agent toward the desired behavior is often one of the most challenging aspects of formulating RL problems. Our optimization objective is calculated as shown in Equation 4.1, where C_{2q} denotes the two-qubit gates in the routed circuit (auxiliary or not), and $r(x,y)$ equates to the reliability of the edge linking physical qubits x and y. It is important to keep in mind that this value is only an estimate of the probability that the circuit will execute without errors, as it does not capture the full complexity of the effects of noise on the computation and also assumes that errors are independent.

$$r_{\text{circuit}} = \prod_{g_{x,y} \in C_{2q}} r(x,y) \tag{4.1}$$

Reinforcement learning agents follow the principle of maximizing expected return, computed as a sum of discounted rewards. Hence, our reward function uses the logarithms of two-qubit gate reliabilities instead of the probability values, and the goal is shifted to maximizing the sum in Equation 4.2. Since the reliability of a gate is a probability between 0 and 1, its logarithm will fall within the interval $]-\infty, 0]$, with lower values indicating an increased chance for errors. In practice, however, we impose a lower bound on log reliabilities to prevent extreme rewards from destabilizing the training process.

$$\log r_{\text{circuit}} = \sum_{g_{x,y} \in C_{2q}} \log r(x,y) \tag{4.2}$$

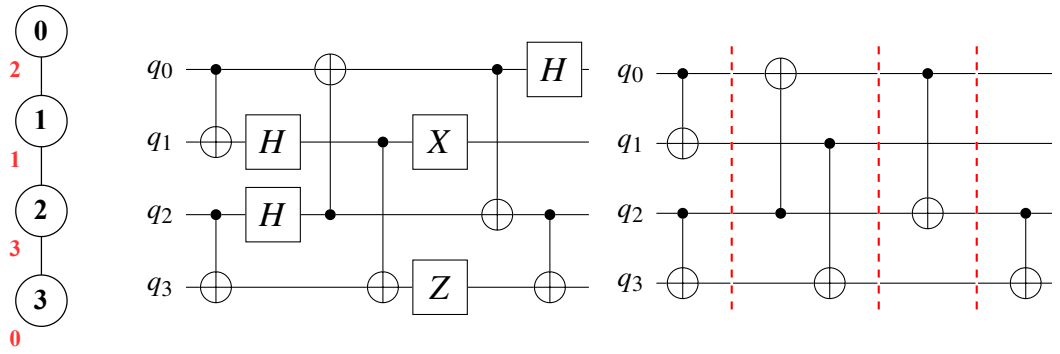
The formulas for calculating the different components of our reward function are presented in Equation 4.3. At each time step, the agent selects the next SWAP or BRIDGE gate to perform according to its policy. The environment state is updated according to the chosen operation, and any gates from the original circuit that become executable are immediately scheduled. The reward for that time step will be R_{SWAP} or R_{BRIDGE} (depending on the type of action that was picked) plus the sum of rewards for two-qubit gates made feasible due to the agent's action ($\sum R_{\text{CNOT}}$). Gates automatically routed at the start of an episode do not contribute to the reward, as the agent had no influence on their execution.

$$\begin{aligned} R_{\text{SWAP}}(x,y) &= 3 \cdot \log r(x,y) \\ R_{\text{BRIDGE}}(c,m,t) &= 2 \cdot \big(\log r(c,m) + \log r(m,t)\big) + R_{\text{bonus}} \\ R_{\text{CNOT}}(c,t) &= \log r(c,t) + R_{\text{bonus}} \\ R_{\text{bonus}} &= |\min_{x,y} \log r(x,y)| + \varepsilon \end{aligned} \tag{4.3}$$

The rewards associated with SWAP and BRIDGE gates are virtually always negative due to the use of log reliabilities. Originally, R_{CNOT} was merely equal to the log reliability of the routed two-qubit gate, meaning it was actually a penalty. Despite being in line with the concept that the total

rewards received during an episode should correspond to the log reliability of the resulting circuit, this practice resulted in poorer training performance. Thus, we included an extra component R_{bonus} equal to the absolute value of the minimum log reliability across the entire device plus a small positive value ε. This change ensures that the reward for routing a two-qubit gate will always be positive, providing better feedback to the agent throughout the episode. This bonus is also issued when the agent selects a BRIDGE operation, as it implicitly executes a gate from the original circuit.

Applying a deep reinforcement learning algorithm (such as PPO) requires encoding states as observations that can be fed to a neural network, generally fixed-length arrays of numeric values. In light of how the constituent parts of the state were identified, the most pressing concern is that the length of the input circuit can vary. Our solution consists of a modified variant of the strategy used in [37], an article already covered in more depth in the state of the art. In essence, the agent receives a partial, fixed-size view of the circuit, modified to only include two-qubit gates (as single-qubit gates do not affect routing). The lookahead capability of the agent is adjusted by changing the depth of this view: higher values can help improve its routing decisions at the expense of making training more difficult due to the increased number of input neurons.



(a) Current mapping (b) Circuit to route (c) Circuit (two-qubit gates and explicit layers)

$$\begin{bmatrix} q_1 & q_2 & q_2 & \times \\ q_0 & q_3 & \times & \times \\ q_3 & q_0 & q_0 & q_3 \\ q_2 & q_1 & \times & q_2 \end{bmatrix}$$

(d) Matrix representation (logical qubits)

$$\begin{matrix} \mathbf{q_0} \to \mathbf{p_3} \\ \mathbf{q_1} \to \mathbf{p_1} \\ \mathbf{q_2} \to \mathbf{p_0} \\ \mathbf{q_3} \to \mathbf{p_2} \end{matrix} \begin{bmatrix} p_1 & p_0 & p_0 & \times \\ p_3 & p_2 & \times & \times \\ p_2 & p_3 & p_3 & p_2 \\ p_0 & p_1 & \times & p_0 \end{bmatrix}$$

(e) Matrix representation (physical qubits)

$$\begin{matrix} \mathbf{p_0} \\ \mathbf{p_1} \\ \mathbf{p_2} \\ \mathbf{p_3} \end{matrix} \begin{bmatrix} 2 & 3 & 3 & 2 \\ 3 & 2 & -1 & -1 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & -1 \end{bmatrix}$$

(f) Final matrix representation (reordered rows)

Figure 4.2: Constructing a circuit matrix observation

Figure 4.2 shows the steps involved in generating a matrix representation from a given circuit and qubit mapping. As in the original paper, we begin by extracting the two-qubit gates from the original circuit and grouping them into layers based on their dependency relationships (Figure 4.2c). Then, a matrix is produced from the interactions between qubits at every time step (Figure 4.2d), with one row per physical qubit and one column per layer (up to a maximum depth, which, in this case, is equal to 4). The element in the i^{th} row and j^{th} column is q_x if and only if

qubit q_i is involved in a two-qubit gate with q_x at time step j. Figure 4.2e depicts an additional step present in our approach, where we replace each logical qubit q with its corresponding physical qubit p according to the present mapping. Finally, the rows of the matrix are rearranged to conform with the ordering of physical qubits (Figure 4.2f).

The decision to use the indices of physical nodes instead of logical qubits in the circuit observation makes it simple to identify gates that are executable under the current mapping. For instance, the interaction between p_1 and p_2 in the second layer (which actually corresponds to $\text{CNOT}_{1,3}$ from the input circuit) would be valid if not for the infeasible gates in the first layer, as there is an edge connecting p_1 and p_2 in the coupling graph. Another advantage of this representation is that it simultaneously conveys information about sections of the circuit left to route and the current layout of logical qubits within the device. A complete observation also includes the log reliabilities of each edge in the coupling graph, allowing the agent to tailor its routing decisions to the device's error rates.

4.2.2 Enhancements and Optimizations

This section covers a few aspects of our solution intended to improve its performance and robustness but which are not integral to the reinforcement learning formulation. Returning to the matter of circuit observations, one must acknowledge that while integer values are often used to identify physical qubits, such indices actually behave as categorical labels. There is generally no guarantee that qubits with consecutive indices will be linked or even close to each other in the coupling graph. As an example, consider the topology of the 16-qubit `ibmq_guadalupe` quantum computer, illustrated in Figure 4.3, where the shortest path between qubits q_5 and q_6 has length six.

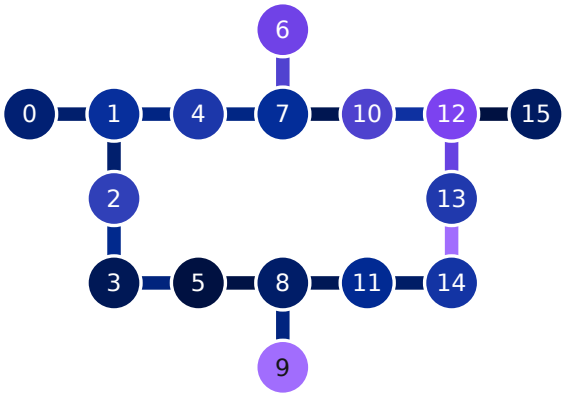


Figure 4.3: Coupling graph of IBM Q Guadalupe [9]

Due to the lack of a clear ordinal relationship between qubits, simply using integer indices in the circuit matrix representation could be misleading to the neural network and hinder its learning. One way to encode categorical features in machine learning is with one-hot encoding, which replaces each variable with a binary array of length equal to the number of values it can take. This

approach has the obvious drawback of rendering the input layer unreasonably large for devices with a higher qubit count. Using embeddings is an alternative strategy where each discrete value is expressed as a vector in a continuous n-dimensional space. Embeddings are frequently used in natural language processing as they allow words with similar semantics to be mapped to similar regions of the vector space. We employ the same strategy for circuit observations, as exemplified in Equation 4.4.

$$\begin{bmatrix} 2 & 3 & 3 & 2 \\ 3 & 2 & -1 & -1 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & -1 \end{bmatrix} \longrightarrow \begin{bmatrix} \begin{bmatrix} 0.561 \\ -0.449 \end{bmatrix} & \begin{bmatrix} -1.304 \\ 0.233 \end{bmatrix} & \begin{bmatrix} -1.304 \\ 0.233 \end{bmatrix} & \begin{bmatrix} 0.561 \\ -0.449 \end{bmatrix} \\ \begin{bmatrix} -1.304 \\ 0.233 \end{bmatrix} & \begin{bmatrix} 0.561 \\ -0.449 \end{bmatrix} & \begin{bmatrix} -0.246 \\ 0.376 \end{bmatrix} & \begin{bmatrix} -0.246 \\ 0.376 \end{bmatrix} \\ \begin{bmatrix} -0.157 \\ 1.899 \end{bmatrix} & \begin{bmatrix} -1.848 \\ -0.828 \end{bmatrix} & \begin{bmatrix} -0.246 \\ 0.376 \end{bmatrix} & \begin{bmatrix} -0.157 \\ 1.899 \end{bmatrix} \\ \begin{bmatrix} -1.848 \\ -0.828 \end{bmatrix} & \begin{bmatrix} -0.157 \\ 1.899 \end{bmatrix} & \begin{bmatrix} -0.157 \\ 1.899 \end{bmatrix} & \begin{bmatrix} -0.246 \\ 0.376 \end{bmatrix} \end{bmatrix} \tag{4.4}$$

This method allows the neural network to learn suitable representations for each qubit while avoiding the high space complexity associated with one-hot encoding. If we let V be the set of physical qubits, E the set of links, d the depth of the circuit matrix, and e the dimensionality of the embedding space, the input layer will contain $|V| \cdot d \cdot e + |E|$ neurons. Assuming that d and e are constants and the device has a sparse coupling graph ($|E| \approx |V|$), the input layer will have size $O(|V|)$.

The background on reinforcement learning referenced the technique of invalid action masking in policy gradient algorithms, which we used for three distinct purposes. The first and most straightforward is to prevent the execution of invalid BRIDGE gates, which are only allowed when there is a CNOT in the front layer between qubits with a shortest path distance of two. We also use action masking to ensure that the agent can only apply SWAP operations involving qubits from the front layer of gates. In this way, we stop the agent from scheduling SWAPs that do not contribute towards routing gates in the front layer. Such actions are not strictly invalid but, in practice, result in slower learning. Under this same viewpoint, we also forbid the agent from executing the same SWAP gate twice, assuming no other gates were routed between them.

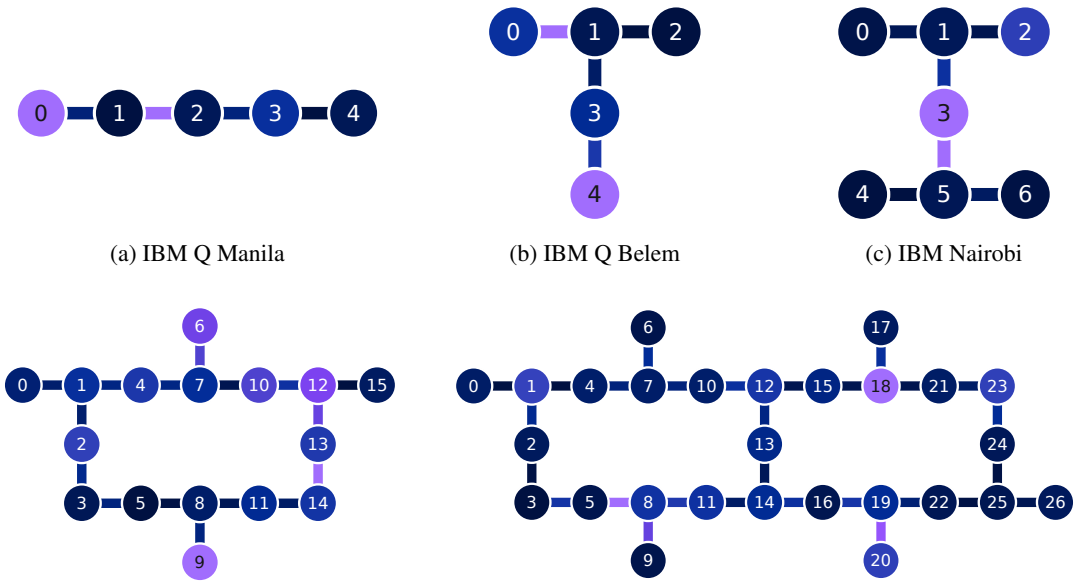
The final supporting component of our solution relates to exploiting gate commutation rules for further improvements during routing, analogously to the techniques discussed in a few articles from the state of the art [57, 58]. At the beginning of each episode, the input circuit is analyzed to identify groups of gates that commute amongst each other. Then, at every time step, any gates that are valid under the current mapping and can be moved to the front layer by virtue of commutation rules are immediately scheduled. This procedure is performed automatically and outside of the agent's direct control. Nevertheless, it successfully reduces the number of SWAP and BRIDGE gates required, resulting in more compact and reliable circuits.

4.2.3 Scope

Having defined the problem being studied and described the individual components of our proposed solution, we next delineate the scope of our research. We acknowledge the limiting factors of our approach, expose our reasoning behind our choice of concrete problem instances, identify the data sources used, and explain our evaluation methodology.

4.2.3.1 Quantum Architectures

The device topology is undoubtedly one of the external factors with the most impact on the qubit routing process. As such, a key facet in assessing the capability of any routing algorithm is analyzing how it performs across differing device architectures and qubit counts. Our aim of developing a noise-adaptive routing method further warrants access to device error rates. To our knowledge, IBM is the only manufacturer that openly discloses both the topology and calibration data associated with their devices. Rigetti and Google do not report error probabilities of individual qubits or gates [5, 3], while Quantinuum's trapped ion computers possess all-to-all connectivity and hence do not need routing [6].



(a) IBM Q Manila (b) IBM Q Belem (c) IBM Nairobi

(d) IBM Q Guadalupe (e) IBM Q Mumbai

Figure 4.4: Topology of the devices considered in our work [9]

The IBM Quantum Platform website displays calibration data for all active IBM devices, including one and two-qubit gate error rates, measurement errors, and qubit coherence times, among other metrics [9]. This information is also available through an API. One important but unfortunate characteristic of this API is that historical calibration data is only available for the devices included in the access plan of the account in use. At the time of writing, only five and seven-qubit devices are included in the free open plan. For larger devices, only the most recent calibration

figures can be obtained, meaning that we cannot use past metrics to train the corresponding RL models.

Figure 4.4 depicts the architectures used in our experiments. These include all distinct topologies for active IBM computers with up to 27 qubits, a limit imposed by time and computation constraints. Analyzing several devices with the same topology was not deemed relevant to our research.

4.2.3.2 Deep Reinforcement Learning Algorithm

When deciding which deep RL algorithm to use in our solution, we considered two established approaches: Deep Q-Networks (DQN) and Proximal Policy Optimization (PPO). Q-learning stands as one of the most established reinforcement learning methods, and many extensions to the base DQN formula have been proposed over the years to boost its effectiveness and stability. Being an off-policy algorithm that learns by sampling from a large replay buffer of past experiences, its sample efficiency makes it suitable for environments that are costly to step through, such as those that demand substantial computing power or require interaction with the real world.

On the other hand, PPO's main strengths lie in its training stability and robustness to changes in its hyperparameters. These benefits make it appealing for our use case, given that the qubit routing environment we have devised is fast and straightforward to simulate, meaning that sample efficiency becomes less of a concern. Another detail that influenced our decision is the ease of implementation and theoretical basis for invalid action masking in policy gradient algorithms [48], already covered in greater detail in the background.

4.2.3.3 Hyperparameter Selection

Arriving at a reasonable set of hyperparameters is an important step in many machine learning applications. Hyperparameters are used to control the flow of learning or other characteristics of the algorithm being used. Some, such as the learning rate, discount factor, or batch size, are common to almost every reinforcement learning algorithm. In deep RL, the neural network architecture (such as the chosen activation function or the number and size of the hidden layers) can also be considered part of the set of hyperparameters. Others, such as the clipping parameter and entropy coefficient, are specific to PPO.

As previously noted, PPO is typically less sensitive to hyperparameter tuning than other RL algorithms. From the experiments conducted, adjusting the hyperparameters did not significantly impact the models' performance as long as values were kept within an adequate range. Instead, changes to the components of our solution, such as the reward function, the observations, and the commutation analysis, appeared to have a more distinguished effect on training and evaluation. We did attempt hyperparameter tuning for five-qubit devices using population-based training [68], but improvements were minimal compared to configurations derived from sensible defaults and hand-tuning. Table 4.1 lists the hyperparameter values used to train the RL models for each IBM

device, including the neural network architecture, PPO-specific parameters, and the overall training duration. The learning rate varies according to a piecewise linear function defined by the points presented in Table 4.2.

	DEVICES				
	Manila	Belem	Nairobi	Guadalupe	Mumbai
Qubits	5		7	16	27
Edges	4		6	16	28
Circuit matrix depth	8				
Embedding dimensions	2			3	4
Hidden layers	$(128, 128)$			$(256, 256)$	$(512, 512)$
Activation function	SiLU				
Discount factor (γ)	0.99				
Batch size	8,192			32,768	65,536
Minibatch size	512			2,048	4,096
SGD[1] iterations	5			10	
Value function coefficient	0.5				
Entropy coefficient	0			5.0×10^{-3}	1.0×10^{-2}
GAE[2] λ parameter	0.95				
Training iterations	1,500		2,000	5,000	4,000
Total time steps	12,288,000		16,384,000	163,840,000	262,144,000

Table 4.1: Hyperparameters used for each device architecture

DEVICE(S)	TIME STEPS	LEARNING RATE
Manila Belem Nairobi	0	3.0×10^{-4}
	500,000	1.0×10^{-4}
	1,500,000	5.0×10^{-5}
	2,500,000	3.0×10^{-5}
Guadalupe	0	1.0×10^{-4}
	3,000,000	5.0×10^{-5}
	9,000,000	2.0×10^{-5}
	18,000,000	1.0×10^{-5}
Mumbai	0	1.0×10^{-4}
	10,000,000	5.0×10^{-5}
	20,000,000	2.0×10^{-5}
	40,000,000	1.0×10^{-5}

Table 4.2: Learning rate schedule used for each device architecture

[1] Stochastic gradient descent
[2] Generalized advantage estimation

4.2.3.4 Evaluation Strategy

The main decisions associated with the evaluation procedure consist of identifying the algorithms to compare our strategy against, the methods for obtaining the input data, and the metrics used to measure performance. Several approaches examined in the literature review either lack an open-source implementation or would require significant effort to incorporate into our evaluation scheme (for instance, some were implemented using a different quantum computing framework). In contrast, the close integration between the Qiskit SDK and the IBM Quantum Platform greatly facilitated the comparison with the routing algorithms implemented by the Qiskit compiler, considering the dissertation's time frame. These methods include the SABRE algorithm (`SabreSwap`), also referenced in the state of the art, and two simpler techniques known as `StochasticSwap` and `BasicSwap`.

At the time of writing, a large, unified dataset of frequently used quantum circuits does not appear to exist. We follow the example of most articles in the existing literature, which utilize benchmark circuits from the *RevLib* dataset, intended to aid research into reversible computing [69]. Figure 4.5 shows an example circuit from the dataset, specifically a realization of the Fredkin (controlled SWAP) gate. With the help of Qiskit, we converted these circuits from RevLib's own `.real` file format to OpenQASM and decomposed them into the native gate set for IBM computers. Any duplicates or circuits using more than 27 qubits or containing more than 1,000 CNOT operations were excluded from the final dataset. Additionally, we generate random circuits by iteratively applying CNOTs between random pairs of qubits until a target number of gates is reached. Randomly generated circuits are mainly used during training since the size of the benchmark set is substantially lower than the number of distinct circuits that can be created this way.

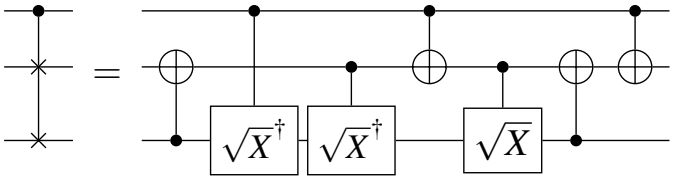


Figure 4.5: Quantum circuit that implements a Fredkin gate [69]

During training, error rates are sampled from distributions created using real calibration data from IBM devices. When assessing the performance of the reinforcement learning models, we employ a recent snapshot of the error probabilities of the target device. Alongside circuit reliability, which is our main evaluation criterion, we consider secondary metrics such as the number of added CNOT gates and the depth of routed circuits.

Another essential aspect of the evaluation process is the number of episodes per circuit. Being a policy gradient algorithm, PPO is able to produce stochastic policies, unlike Q-value methods such as DQN. Since the agent only receives a partial view of the circuit at any given point, it is often possible to achieve better results by routing the circuit multiple times and selecting the episode with the highest total reward.

Although we are only concerned with investigating the performance of the previously mentioned routing strategies, the choice of initial mapping can significantly affect the quality of the solutions. Since our primary goal is to increase circuit reliability, we used Qiskit's `NoiseAdaptiveLayout` pass [55], which implements one of the qubit placement algorithms mentioned in our literature review [54]. In some extremely rare instances where the noise-adaptive procedure failed, our code fell back to using a trivial initial mapping.

4.2.3.5 Experimental Setup

The training process was conducted on a remote server provided by FEUP's Artificial Intelligence and Computer Science Laboratory (LIACC)[3], possessing the following specifications:

- **CPU**: Intel Xeon W-2295 @ 3.00 GHz (18-core)
- **GPU**: 2x NVIDIA Quadro RTX 8000 (48 GB GDDR6)
- **RAM**: 128 GB DDR4
- **Operating System**: Ubuntu 20.04.6 LTS
- **Python Version**: 3.11.5

We used 18 rollout workers for experience collection, with 8 vectorized environments per worker. A single GPU was used for training since, for our workloads, the overhead of using multiple GPUs outweighed any improvements in training speed. The device calibration data used to evaluate our models and to compute the probability distribution for sampling error rates during training was obtained from the IBM Quantum website on 2023-08-18. Unless stated otherwise, we use 10 evaluation episodes per circuit.

[3] https://liacc.fe.up.pt/

Chapter 5

Implementation

This chapter delves deeper into specific implementation details of our solution, building on the higher-level methodological approach presented earlier. We begin with a brief overview of the main tools and environments used. Following that, we describe the various modules that make up our system and the relationships between them.

5.1 Technologies and Platforms

Among the tools used for implementing and validating our approach, the quantum computing and deep reinforcement learning frameworks stand out as having the greatest impact on the project's development. For these technologies, we succinctly compare the alternatives considered before exposing the logic behind our final decision. We conclude this section by referencing libraries that are less critical but still relevant to our work.

5.1.1 Quantum Computing Framework

Qiskit [7] is an open-source Python SDK for quantum software development, maintained by IBM Research and contributors. Qiskit can be used to design quantum circuits, compile them for a particular device architecture, run programs on classical simulators or real quantum computers over the cloud, and collect and visualize analytics from computation results. In addition to a comprehensive set of quantum gates, the library also includes template circuits, implementations of well-known algorithms such as QAOA and VQE, and tools for visualizing qubit states. The framework is modular: Qiskit Terra encompasses the core functionality, Qiskit Aer is a C++ quantum simulator with support for emulating noisy systems, and there are other high-level modules for applications such as optimization and machine learning. Users can also convert circuits to the OpenQASM language [70], which is practical for sharing designs in a portable way.

With strong links to Qiskit, **IBM Quantum** [9] is an online platform providing tools to facilitate quantum application development and access to IBM's cloud-based quantum hardware and simulators. Examples include Quantum Composer, a graphical interface for building circuits, and Qiskit Runtime, which can be used to plan and carry out workloads more efficiently. At the time

of writing, IBM provides free, queue-based access to multiple five and seven-qubit processors and a 32-qubit simulator with adjustable noise settings.

Cirq [8] is another open framework for quantum programming, developed by Google. The Python library supports building circuits, running noiseless or noisy simulations, and executing programs on quantum hardware platforms such as Microsoft Azure Quantum, IonQ, or Rigetti.

The **Microsoft Quantum Development Kit** [71] features Q#, a special-purpose language for quantum development that seeks to provide a higher degree of abstraction, forgoing details concerning qubits, quantum states, and circuits in favor of standard programming constructs. Additionally, there is some level of interoperability with Cirq and Qiskit. Access to quantum hardware through Azure uses a credit-based system with specific pricing schemes for each provider (ranging from circuit complexity and shot count to overall execution time). Microsoft offers an initial volume of credits to new users of the platform, and it is possible to request additional funding for certain projects.

Since we seek to evaluate our routing algorithm by targeting IBM devices, using Qiskit in conjunction with IBM's cloud platform is the most suitable choice for our use case. It is important to point out that an IBM Quantum account and API key are required in order to access IBM's quantum computers, regardless of whether or not we wish to run actual programs on the hardware. Interaction with these devices via the API is handled by a separate Python package named `qiskit_ibm_provider` [72].

5.1.2 Deep Reinforcement Learning Framework

The two reinforcement learning frameworks considered for our application were Ray RLlib and Stable Baselines3 (SB3). We used SB3 during the early stages of development but eventually transitioned to RLlib for our final solution.

RLlib [73] is an open-source library for reinforcement learning with a strong emphasis on scalability and distributed computing. It is a component of Ray [74], a framework for scaling Python applications and machine learning workloads, alongside other libraries such as Tune (for distributed hyperparameter tuning) and Serve (for deploying machine learning models). RLlib provides numerous RL algorithm implementations and variants featuring a highly modular architecture, allowing individual algorithm components to be replaced with user-defined code. Other benefits include robust logging and reporting functionality, as well as the ability to utilize multiple GPUs or distribute workloads across several nodes. These advantages come at the expense of raising the library's complexity and making it somewhat cumbersome for smaller RL projects.

Stable Baselines3 [75] is an actively maintained version of OpenAI Baselines, a collection of RL algorithm implementations written in Python. In SB3, the deep learning framework of choice is PyTorch, unlike the original Baselines, which required TensorFlow. SB3's key strengths lie in its simplicity and ease of use, but compared to RLlib, it supports fewer algorithms and has less sophisticated out-of-the-box logging capabilities. It does, however, provide SB3 Contrib [76], an additional package containing less popular or experimental algorithms and variants. Among these

is an implementation of PPO with support for action masking, which we used for our initial experiments. Our decision to use RLlib for our final models was primarily motivated by its improved logging and scalability. The framework's more flexible design made it simpler to develop a custom model with action masking functionality.

5.1.3 Other Tools

Another essential library used by our solution is Gymnasium [77], an up-to-date fork of OpenAI Gym. Its goal is to provide a standard interface for defining reinforcement learning environments, and it is favored by most RL frameworks. In addition to letting developers build custom environments, Gymnasium also features several premade environments, including toy problems, physics simulations, and Atari video games. Some of these are frequently used to benchmark RL algorithms.

PyTorch [78] and TensorFlow [79] are decidedly the two most well-known frameworks for deep learning. Core features found in both libraries include efficient methods for operating on tensors (multidimensional arrays), constructs for defining neural networks, automatic differentiation for computing gradients, and GPU acceleration (primarily through the CUDA platform). We ultimately stuck with PyTorch during the project's development, as it is supported by both RLlib and SB3. RLlib is also capable of handling TensorFlow models, but we found no benefits to that approach that would justify the effort and time investment of learning a new framework.

The progress of our experiments and the various metrics reported by RLlib were tracked through TensorBoard [80], a web dashboard designed for visualizing the training process of machine learning models. Figure 5.1 exemplifies how a model's performance can be monitored via the web-based GUI. The project was written entirely in Python, but any training or evaluation configurations were defined as YAML files. To a lesser extent, we also leveraged conventional Python libraries for data analysis and visualization, such as NumPy [81], Pandas [82], and Seaborn [83, 84].

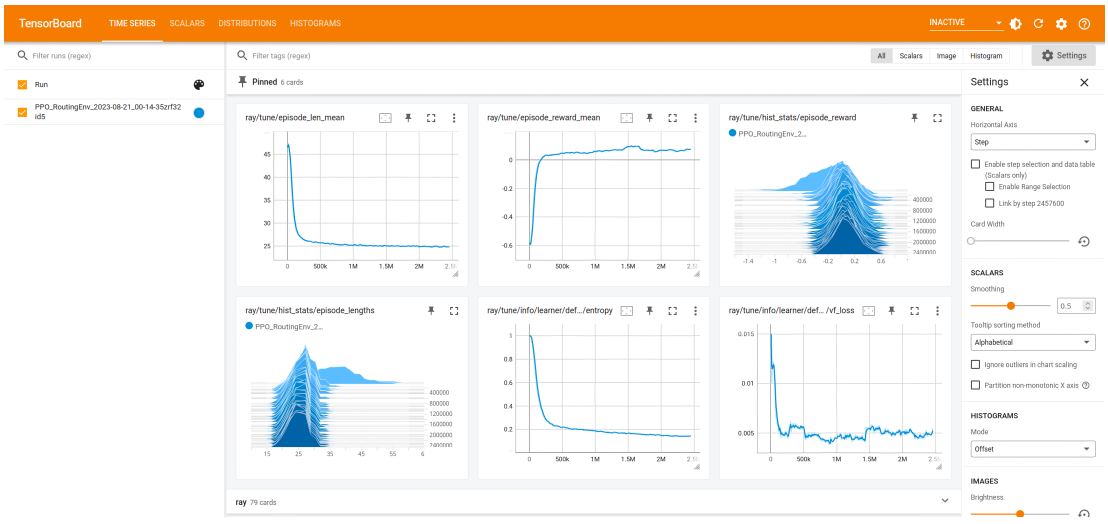


Figure 5.1: The TensorBoard web interface

5.2 Implementation Details

5.2.1 Overview

The software developed for this dissertation consists of a collection of Python modules that implement our qubit routing solution and a few auxiliary scripts. The complete source code, including the input data and configuration files used for our experiments, is openly available under the Apache 2.0 license and is hosted at the following GitHub repository: `https://github.com/GoncaloPascoal/dissertation`. The repository includes instructions for executing the scripts and visualizing the results to make our experiments easier to replicate.

The fundamental modules that make up our application are:

- **Environment**: contains our definition of qubit routing as an RL environment according to the Gym API standard (`env.py`), as well environment wrappers to aid training and evaluation (`wrappers.py`). The environment also encapsulates the logic for gate commutation analysis and determining which actions are invalid in the current state;
- **Generators**: provides methods for generating random quantum circuits (`circuit.py`) and sampling error rates (`noise.py`), which are mainly used to produce input data to be used during the training process;
- **Orchestration**: contains classes for setting up and managing the training and evaluation procedures (`orchestration.py`), as well as interfacing with the reinforcement learning framework (RLlib);
- **RLlib**: consists of custom components intended to replace the default RLlib implementations, namely a neural network with support for invalid action masking and qubit embeddings, but also a modified logging callback;
- **Parsing**: includes functions for parsing YAML files containing environment, training, or evaluation configurations (`parsing.py`);
- **Utilities**: comprises functionality that is not specific to any of the previously listed modules.

5.2.2 Gym Environment

The RL environment is undoubtedly the most intricate and essential aspect of our system. For conciseness, we refer to the Gymnasium package by the name of its predecessor, Gym, but note that all referenced functionality still pertains to Gymnasium. The library's focal point is the `Env` class, which establishes the API that any Gym environment should follow. Users can define custom environments by inheriting from this class and implementing the necessary methods [77].

The first requirement when implementing a Gym environment is to set the observation and action spaces in the constructor. In its `spaces` module, Gym provides several space classes

corresponding to different kinds of mathematical sets. These include, among others, `Box`, a fixed-size n-dimensional array of real numbers, and `Discrete`, representing a finite set of values. It is also possible to combine multiple spaces using a composite space, such as `Tuple`, a fixed-length sequence of subspaces, or `Dict`, which represents a mapping from keys to spaces [77].

The action space for our qubit routing environment is of type `Discrete`, with the number of actions being equal to the number of edges in the coupling graph (i.e., the different possible SWAP operations) plus the number of distinct BRIDGE operations (as stated before, the direction of the BRIDGE gate is inferred by context). Actions are defined in ascending order of the physical qubits involved in the operation. Table 5.1 shows how actions would be encoded if targeting a 5-qubit device with linear nearest neighbor connectivity.

Action	Operation	Qubits
0	SWAP	$(0,1)$
1	SWAP	$(1,2)$
2	SWAP	$(2,3)$
3	SWAP	$(3,4)$
4	BRIDGE	$(0,1,2)$
5	BRIDGE	$(1,2,3)$
6	BRIDGE	$(2,3,4)$

Table 5.1: Encoding actions as discrete values

A quirk of invalid action masking in RLlib is that the binary mask that signals which actions are valid must be part of the observation, even though it is not supplied to the neural network. Hence, our *outer* observation space is a `Dict` that encapsulates the action mask and the actual observation. The inner observation space is also implemented as a `Dict` containing several *observation modules*, specified in the environment configuration. This decision allows users to create and combine different types of observations without having to modify the environment source code. Our approach uses two modules: one for the circuit matrix representation (`CircuitMatrix`) and another for providing edge log reliabilities to the agent (`LogReliabilities`). A sample observation is shown in Listing 5.1. At this point, the circuit matrix observation is still using physical qubit indices because the conversion to embeddings happens in one of the modules of the neural network.

The `Env` interface consists of four methods: `reset`, `step`, `render`, and `close`. The `render` method provides a visual representation of the environment, while the `close` method is responsible for cleaning up any external resources. Since these are not essential to our use case, we only discuss the implementation of the `reset` and `step` methods.

The `reset` method is called at the beginning of every episode and should reset the environment to its initial state, returning the first observation the agent will encounter [77]. Listing 5.2

```
{
    'action_mask': [1, 0, 1, 0, 1, 0, 0],
    'true_obs': {
        'circuit_matrix': [
            [ 2, -1, -1,  2],
            [-1,  2,  3, -1],
            [ 0,  1,  4,  0],
            [-1, -1,  1,  4],
            [-1, -1,  2,  3],
        ],
        'log_reliabilities': [-0.0142, -0.0117, -0.0092, -0.0103],
    },
}
```

Listing 5.1: Structure of a routing environment observation

presents the method's signature. The `seed` and `options` arguments are not used in our implementation, and neither is the dictionary within the return value (which is for supplementary information). The pseudocode for `reset` is presented in Algorithm 1. Both the DAG creation and gate commutation analysis are achieved with the help of the Qiskit framework. These actions are performed at every reset because the input circuit will change during training, potentially every episode. Throughout the source code, we use "qubit" to refer to logical qubits and "node" to refer to physical qubits, hence the naming of the `node_to_qubit` and `qubit_to_node` mappings.

```
def reset(
    self,
    *,
    seed: Optional[int] = None,
    options: Optional[dict[str, Any]] = None,
) -> tuple[ObsType, dict[str, Any]]:
    # Method implementation
```

Listing 5.2: Signature of the `reset` method

Algorithm 1 Pseudocode for `reset` method

procedure RESET
    Construct DAG from input circuit
    Run commutation analysis
    node_to_qubit ← initial_mapping
    qubit_to_node ← INVERSEMAPPING(node_to_qubit)
    SCHEDULEAVAILABLEGATES
    obs ← CURRENTOBS
    return obs, {}
end procedure

The `step` method (see Listing 5.3 and Algorithm 2) is responsible for evolving the environment state according to the actions chosen by the agent. Its return value is a tuple consisting of the next observation, the reward, a boolean that is true if the environment reached a terminal state, a boolean indicating whether or not the episode has been truncated (e.g., due to reaching a maximum number of time steps), and a dictionary for additional data [77].

```
def step(
    self,
    action: ActType,
) -> tuple[ObsType, SupportsFloat, bool, bool, dict[str, Any]]:
# Method implementation
```

Listing 5.3: Signature of the `step` method

Algorithm 2 Pseudocode for `step` method

```
1:  procedure STEP(action)
2:      if action is SWAP then
3:          SWAP(action.qubits)
4:          reward ← SWAPREWARD(action.qubits)
5:      else                                              ▷ action is BRIDGE
6:          BRIDGE(action.qubits)
7:          reward ← BRIDGEREWARD(action.qubits)
8:      end if
9:      SCHEDULEAVAILABLEGATES                  ▷ also calculates scheduling_reward
10:     reward ← reward + scheduling_reward
11:     obs ← CURRENTOBS
12:     terminated ← unscheduled_gates = ∅
13:     return obs, reward, terminated, False, {}
14: end procedure
```

The SCHEDULEAVAILABLEGATES procedure (see Algorithm 3) present in both the `reset` and `step` methods executes all gates from the original circuit made possible under the current mapping and commutation rules. It also computes the reward associated with the two-qubit gates that were scheduled. Scheduling a gate involves removing it from the DAG, converting its qubits from logical to physical, and adding it to the list of routed gates.

There are two details in the pseudocode worth highlighting. First, the COMMUTINGGATES procedure is actually abstracting away a significant amount of computation since Qiskit's `CommutationAnalysis` pass groups commuting gates through the lens of individual qubits rather than within the context of the entire circuit. When the commuting gate is a two-qubit gate, our code must also verify that the commutation rules hold for the other qubit involved in the operation, thus ensuring that the circuit's logic is preserved. Second, commuting gates are not immediately scheduled; instead, they are added to the `to_schedule` set to preserve the dependency relationships between operations.

Algorithm 3 Pseudocode for SCHEDULEAVAILABLEGATES

```
procedure SCHEDULEAVAILABLEGATES
    scheduling_reward ← 0
    locked_nodes ← ∅                          ▷ physical qubits blocked by a previous gate
    to_schedule ← ∅                           ▷ gates to schedule due to commutation rules
    for all gate ∈ unscheduled_gates do
        if gate ∈ to_schedule then
            SCHEDULEGATE(gate)
            scheduling_reward ← scheduling_reward + GATEREWARD(gate)
        else if |locked_nodes| ≠ num_qubits then          ▷ not all qubits are blocked
            if ISSCHEDULABLE(gate, locked_nodes) then
                SCHEDULEGATE(gate)
                scheduling_reward ← scheduling_reward + GATEREWARD(gate)
            else
                to_schedule ← to_schedule ∪ COMMUTINGGATES(gate)
                locked_nodes ← locked_nodes ∪ gate.nodes
            end if
        end if
    end for
end procedure
```

The routing environment can be configured via the arguments passed to its constructor. These include the coupling graph and observation modules but also flags for enabling or disabling specific features, such as noise awareness, commutation analysis, or action masking for SWAP operations.

5.2.3 Training and Evaluation

To avoid having to directly change or extend the qubit routing environment for the purposes of training and evaluation, we use wrappers, a Gym library feature that helps customize the behavior of an environment in a modular way. Environments can be wrapped multiple times, and Gym offers general-purpose wrappers for normalizing or transforming observations, actions, and rewards [77].

The `TrainingWrapper` class takes five arguments: the environment to wrap, a circuit generator, a noise generator, the recalibration interval, and the number of episodes per generated circuit. The circuit and noise generators supply the quantum circuits and two-qubit gate error rates used to train the RL agent. The recalibration interval corresponds to the number of training episodes before the error rates are resampled. Conversely, the `EvaluationWrapper` class uses a single set of edge error rates originating from calibration data. The `evaluation_episodes` parameter controls how many episodes the agent has for routing the current circuit (as explained in the methodology, only the best attempt will be preserved). After reaching this limit, a new circuit will be generated upon resetting the environment.

The training and evaluation procedures are governed by their respective orchestrator classes. The `TrainingOrchestrator` is in charge of setting up and seeding the environments, configuring the reinforcement learning algorithm using the RLlib API, initiating the training process, and saving the trained policy to disk. In addition to the routing environment and generators, it is possible to adjust the PPO hyperparameters and the CPU and GPU resources via the orchestrator's constructor. Similarly, the `EvaluationOrchestrator` loads and applies the model to the evaluation data set, computes performance metrics using the routed circuits, and serializes the results to a file using `pickle`. The `routing_methods` parameter is used to specify one or more Qiskit routing algorithms for comparison, which will be tested on the same set of circuits.

5.2.4 Data Generation

The training process relies on randomly generated data for both the input circuits and gate error rates. By extending the `CircuitGenerator` class, it is possible to control how the quantum circuits are constructed. Our project includes a few concrete implementations, one being `RandomCircuitGenerator`, which repeatedly inserts CNOT gates between random qubit pairs until the circuit reaches the desired gate count. All our models were trained using circuits generated this way, though we also provide `LayeredCircuitGenerator`, which produces circuits with denser layers and a fixed maximum depth. When evaluating our models on realistic circuits, we used `DatasetCircuitGenerator`, which features no random generation and simply yields circuits from an input list. One can also specify a path to a folder containing the circuits as OpenQASM [70] files.

On the other hand, the error probabilities that determine the rewards received by the agent are managed by subclasses of `NoiseGenerator`. While `UniformNoiseGenerator` assigns error rates to edges by sampling from a normal distribution, `KdeNoiseGenerator` uses kernel density estimation to approximate a probability distribution from a finite set of samples, which is then used to compute the artificial error rates. As mentioned before, it is impossible for us to obtain past calibration data for IBM quantum computers that are not part of the open-access plan. Thus, to keep our methodology consistent, we use a `KdeNoiseGenerator` created from a single calibration data point in all experiments.

We acknowledge that reproducibility is a significant challenge in many applications of deep reinforcement learning, which is why the circuit and noise generators use independent random number generators, which are seeded alongside the default random number generators for Python, NumPy, and PyTorch. Despite this, it is practically impossible to guarantee identical results between separate runs since RLlib uses multiple concurrent workers to collect experiences from the environment, and this parallelism introduces an uncontrollable source of randomness.

5.2.5 Configuration Files

YAML configuration files have several benefits compared to directly specifying the experiment parameters within a Python script or passing them as command-line arguments. Specifically, it

makes experiments easier to reproduce and tweak without having to understand or interact with the underlying Python modules. Configuration files are used to customize the routing environment (Listing 5.4) and the training (Listing 5.5) and evaluation (Listing 5.6) orchestrators. It is possible to use different environment configurations in conjunction with the same training configuration and vice-versa.

```yaml
name: 'manila'

coupling_map:
  type: 'linear'
  args:
    num_qubits: 5

obs_modules:
  circuit_matrix:
    depth: 8

commutation_analysis: false
```

Listing 5.4: Example environment configuration

```yaml
circuit_generator:
  type: 'random'
  args:
    num_gates: 64
noise_generator:
  type: 'kde_calibration'
  args:
    path: 'data/calibration/manila.json'

seed: 0

# Model
hidden_layers: [128, 128]
embedding_dim: 2
minibatch_size: 512
batch_size: 8192
sgd_iters: 5
lr_schedule:
    - [0, 3.0e-4]
    - [500_000, 1.0e-4]
    - [1_500_000, 5.0e-5]
    - [2_500_000, 3.0e-5]

# Resources
num_workers: 18
```

```
envs_per_worker: 8
num_gpus: 1
```

Listing 5.5: Example training configuration

```
circuit_generator:
  type: 'random'
  args:
    num_gates: 64

error_rates: 'data/calibration/manila.json'
num_circuits: 100
use_tqdm: true
seed: 0
```

Listing 5.6: Example evaluation configuration

5.2.6 Utility Scripts

Our core library-like code is accompanied by the following Python scripts, which can be used to carry out several common tasks:

- `train.py`: used for training RL models. Environment and training configuration files must be provided as command-line arguments, as well as the number of training iterations and the directory for saving the trained model. Some configuration options, such as the computational resources available to RLlib, can be overridden through the command-line;
- `eval.py`: evaluates a model's performance using several metrics and saves the results to a file. Can override configurations such as the number of evaluation circuits and the number of iterations per circuit;
- `register_account.py`: locally stores an IBM Quantum API key so that it can be used in combination with the `qiskit_ibm_provider` package to access IBM quantum computers;
- `current_calibration.py`: fetches the latest calibration data for one or more IBM devices and stores it as a JSON file. This script relies on the API for the IBM Quantum Platform website rather than `qiskit_ibm_provider`, meaning it does not require an API key and can retrieve data from any device regardless of access plan restrictions;
- `real_to_qasm.py`: converts circuits defined using RevLib's `.real` file format [69] to OpenQASM [70], so that they can be parsed by Qiskit;
- `analyze_results.py`: generates the plots and data presented in Chapter 6 (Results and Discussion).

Chapter 6

Results and Discussion

In accordance with the methodology laid out in Chapter 4 and the implementation described in Chapter 5, this chapter evaluates our application of reinforcement learning to the qubit routing problem, comparing it with other strategies and examining what insights our results can offer regarding the research questions we initially posed.

6.1 Results

In this section, we describe each of our experiments and comment on the results obtained. We begin by assessing the effectiveness of our deep reinforcement learning agents against the techniques implemented in the Qiskit compiler. This analysis is performed for both random (Section 6.1.1) and realistic (Section 6.1.2) circuits and contemplates all of the architectures under study. Afterward, we measure the performance benefits from including the optimizations described in the methodology (Section 6.1.3 and from using multiple episodes for evaluation (Section 6.1.4). Finally, we discuss the scalability of our approach by comparing the training and routing times for each model (Section 6.1.5).

6.1.1 Random Circuits

The first benchmark involved routing 200 randomly generated circuits consisting of 64 CNOT gates and having a qubit count equal to the number of physical qubits in the target device. The circuits were generated using the same method as the random circuits used to train the reinforcement learning models (that is, by iteratively inserting CNOT gates between random qubits). For each of the five IBM quantum computers identified in the methodology, we gauge the performance of the compiled circuits according to three metrics: log reliability, number of additional CNOT gates introduced by the routing process, and depth.

- **IBM Q Manila** (Figure 6.2, Table 6.2) – On this 5-qubit device with linear nearest neighbor connectivity, the RL strategy significantly outperforms Qiskit's routing algorithms, achieving, on average, 26.1% better log reliability and using 36.9% fewer CNOTs compared to

SABRE. The resulting circuit depth is also reduced by 23.5%. We observe that excluding edge log reliabilities from the agent's observations has a minimal ($< 1\%$) impact on the performance metrics (see results for the noise-unaware RL variant);

- **IBM Q Belem** (Figure 6.3, Table 6.3) – Apart from Manila, all of IBM's active 5-qubit devices share the same T-shape topology as Belem. In this case, reinforcement learning also pulled ahead of the Qiskit compiler, beating SABRE by 23.9% in log reliability and 36.7% in added CNOTs. In practice, this translates to a 28.7% success rate for random circuits routed using RL, compared to 19.4% for SABRE and 14.0% for `StochasticSwap`. The same set of circuits was used for the Manila and Belem experiments, making their results directly comparable. Despite these two devices having the same number of qubits and edges, we find that Belem's topology makes routing easier (averaging 66.77 extra CNOTs instead of 85.64);

- **IBM Nairobi** (Figure 6.4, Table 6.4) – Our approach also fared well when applied to the 7-qubit Nairobi computer, improving on Qiskit's results by 26.6% and 33.6% in terms of log reliability and two-qubit gates. However, we still witness nearly identical results between noise-aware and noise-unaware RL models;

- **IBM Q Guadalupe** (Figure 6.5, Table 6.5) – RL's lead becomes more modest when looking at the 16-qubit Guadalupe device, but it still surpasses the other routing methods by at least 20% in log reliability and additional CNOTs. As the device and circuit sizes grow, exact routing methods become less and less feasible, thus amplifying the value of these improvements;

- **IBM Q Mumbai** (Figure 6.6, Table 6.6) – Lastly, for the 27-qubit Mumbai, we measured 9.7% (log reliability) and 9.2% (added CNOTs) leads over SABRE. The standard RL model was around 1.3% better than the noise-unaware one, but this slight gap is likely a consequence of training variance.

In summary, our reinforcement learning approach yielded encouraging results across all five quantum architectures when compiling randomly generated circuits, especially when targeting 5 and 7-qubit devices. Figure 6.1 shows the relative proportion of SWAP and BRIDGE operations used by each RL agent. The percentage of BRIDGE gates is highest for Manila, likely owing to the device's poor connectivity.

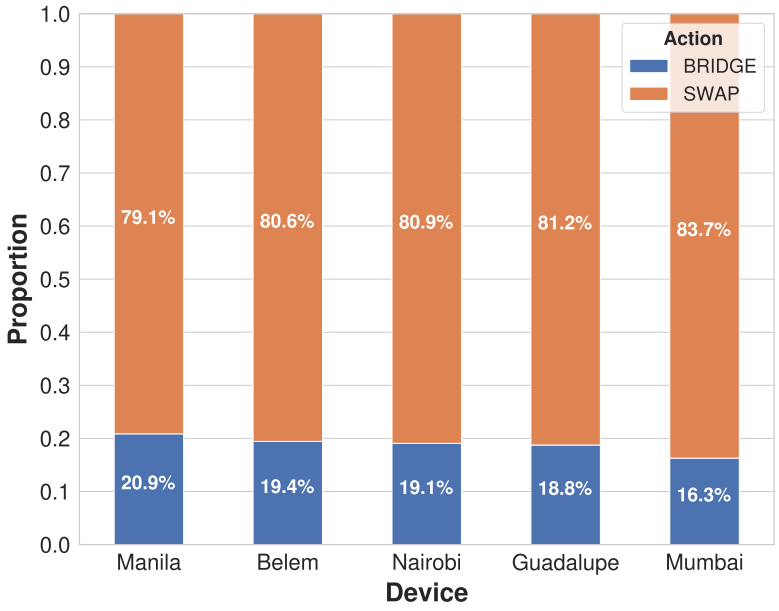
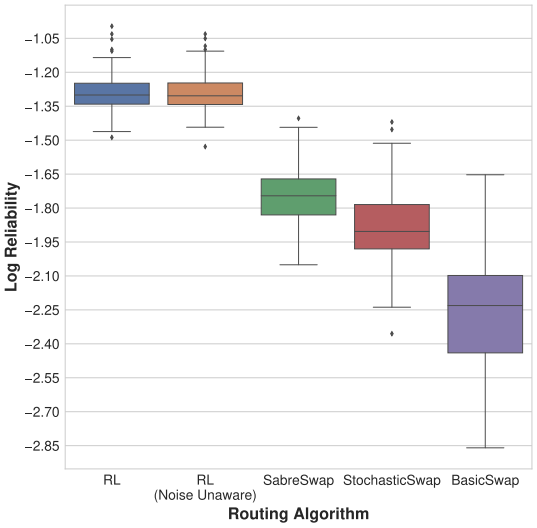


Figure 6.1: SWAP vs. BRIDGE gate usage for each device

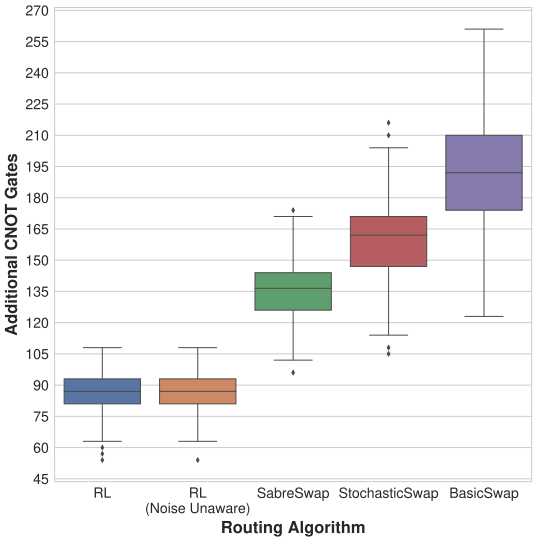
	DEVICES				
	Manila	Belem	Nairobi	Guadalupe	Mumbai
LOG RELIABILITY	26.1%	23.9%	26.6%	20.0%	9.7%
ADDED CNOTS	36.9%	36.7%	33.6%	18.7%	9.2%
DEPTH	23.5%	21.2%	22.4%	6.2%	-6.5%

Table 6.1: Relative improvements over best Qiskit algorithm (random circuits)

(a) Log reliability



(b) Additional CNOT gates



(c) Circuit depth

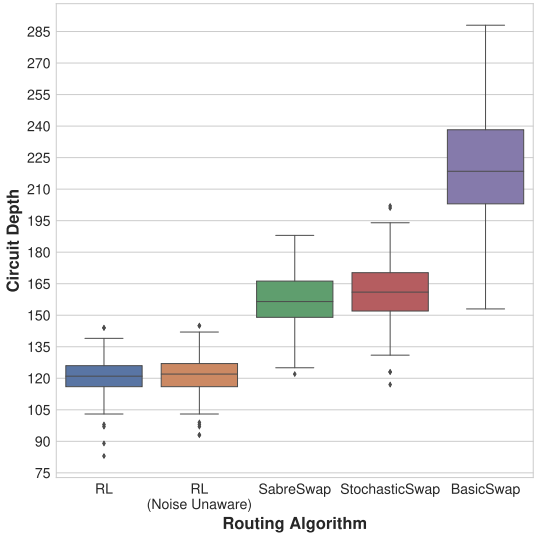
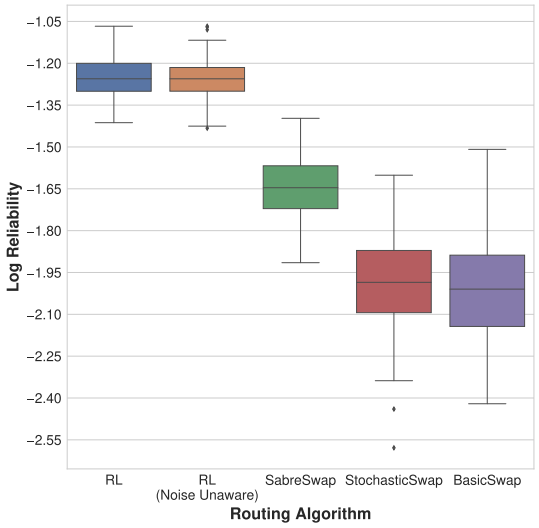


Figure 6.2: IBM Q Manila results (random circuits)

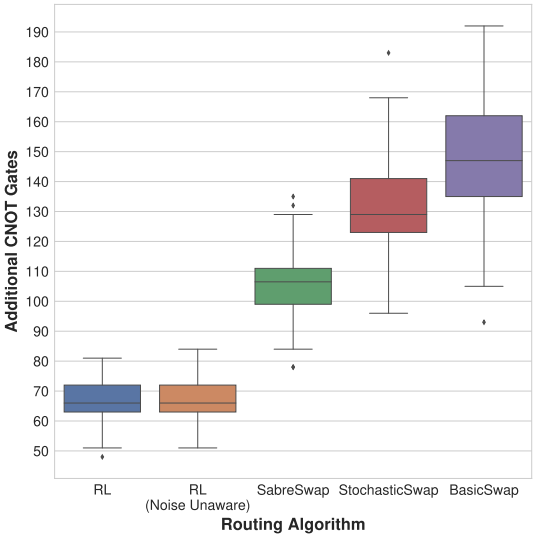
		Routing Algorithms				
		RL	RL (Noise Unaware)	SabreSwap	StochasticSwap	BasicSwap
Log reliability	Mean	**-1.291**	-1.293	-1.746	-1.888	-2.254
	Std	0.078	0.079	0.130	0.155	0.229
Added CNOTs	Mean	**85.64**	85.67	135.79	160.41	191.29
	Std	8.72	8.97	14.74	19.19	25.73
Circuit depth	Mean	**120.57**	121.18	157.56	161.62	220.49
	Std	9.02	9.13	12.96	14.03	23.80

Table 6.2: IBM Q Manila metrics (random circuits)

(a) Log reliability



(b) Additional CNOT gates



(c) Circuit depth

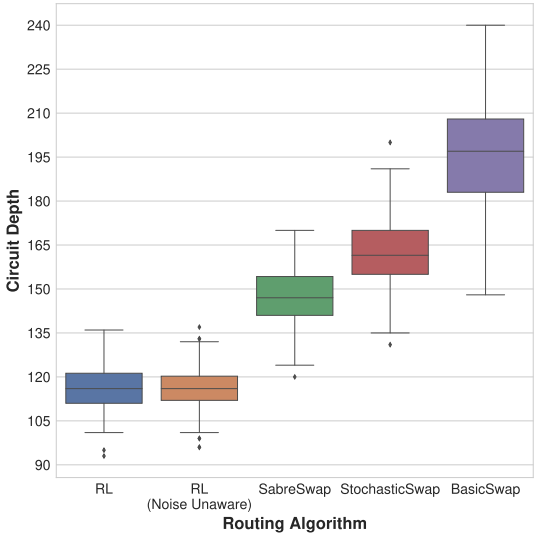
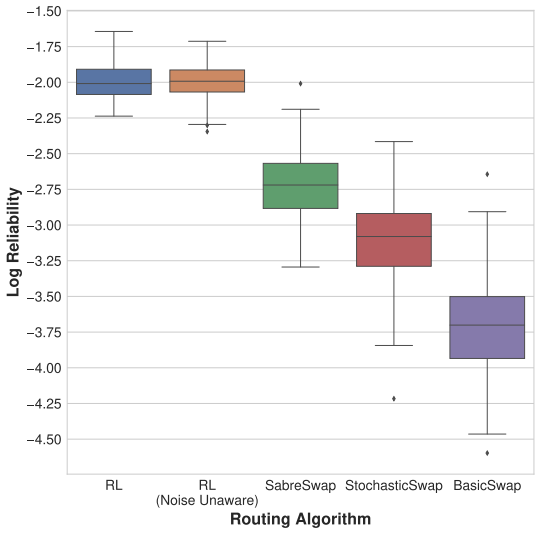


Figure 6.3: IBM Q Belem results (random circuits)

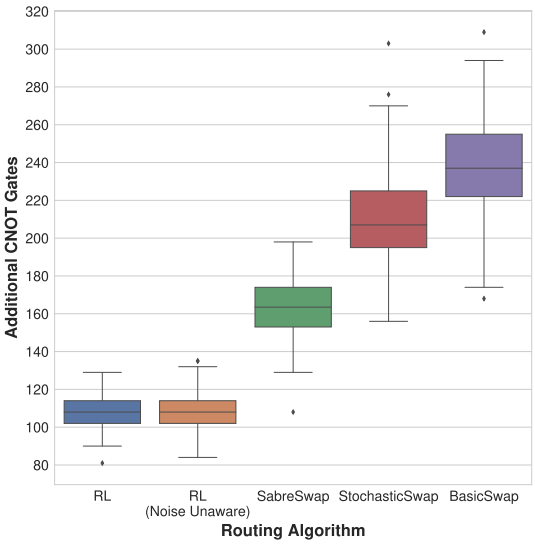
		Routing Algorithms				
		RL	RL (Noise Unaware)	SabreSwap	StochasticSwap	BasicSwap
Log reliability	Mean	**-1.251**	-1.256	-1.644	-1.982	-2.019
	Std	0.070	0.065	0.106	0.170	0.179
Added CNOTs	Mean	**66.77**	67.48	105.56	131.21	148.05
	Std	6.85	6.66	10.07	14.39	18.42
Circuit depth	Mean	**116.18**	116.57	147.48	162.44	196.32
	Std	7.23	7.01	9.61	12.24	17.94

Table 6.3: IBM Q Belem metrics (random circuits)

(a) Log reliability



(b) Additional CNOT gates



(c) Circuit depth

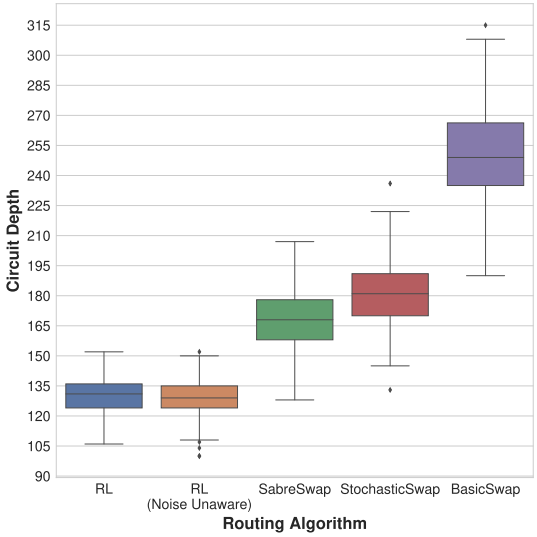
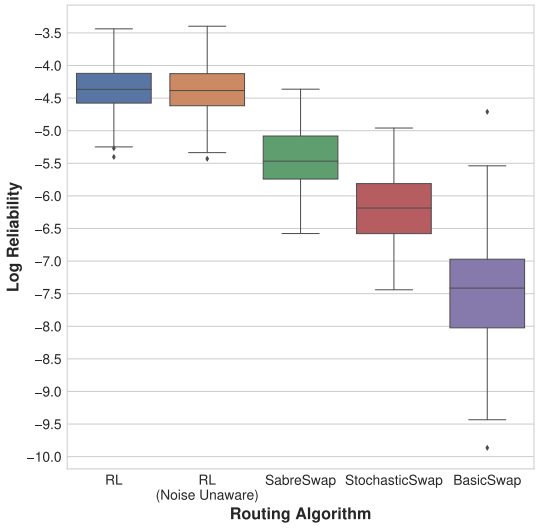


Figure 6.4: IBM Nairobi results (random circuits)

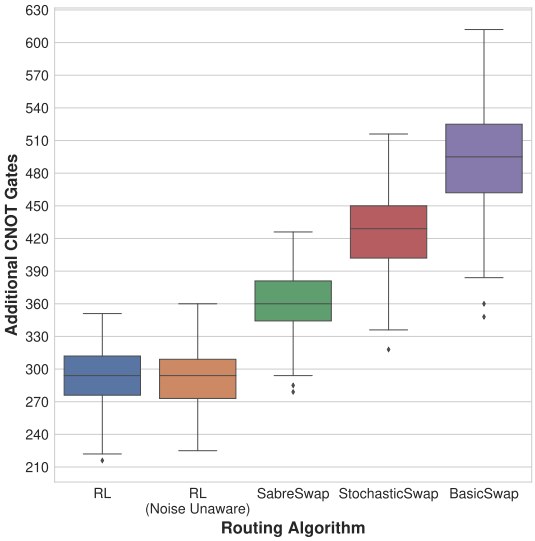
		Routing Algorithms				
		RL	RL (Noise Unaware)	SabreSwap	StochasticSwap	BasicSwap
Log reliability	Mean	-1.996	**-1.992**	-2.719	-3.105	-3.713
	Std	0.117	0.121	0.221	0.291	0.322
Added CNOTs	Mean	**108.63**	108.90	163.51	210.87	238.49
	Std	8.63	9.07	15.62	24.35	24.01
Circuit depth	Mean	130.35	**129.37**	167.87	180.94	250.16
	Std	8.88	9.50	14.02	16.03	23.06

Table 6.4: IBM Nairobi metrics (random circuits)

(a) Log reliability



(b) Additional CNOT gates



(c) Circuit depth

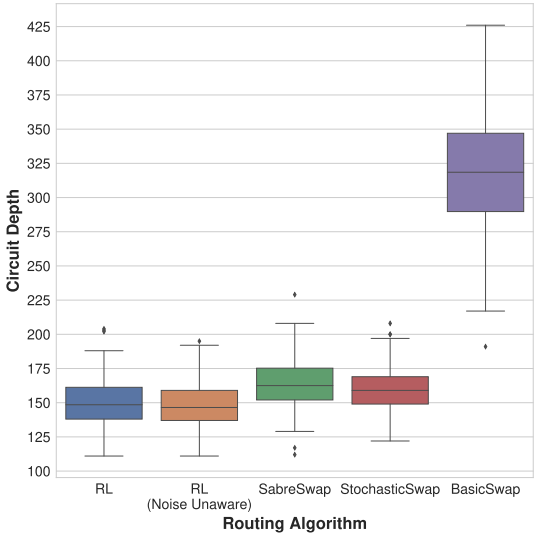


Figure 6.5: IBM Q Guadalupe results (random circuits)

		Routing Algorithms				
		RL	RL (Noise Unaware)	SabreSwap	StochasticSwap	BasicSwap
Log Reliability	Mean	**-4.357**	-4.385	-5.447	-6.199	-7.485
	Std	0.345	0.357	0.467	0.520	0.818
Added CNOTs	Mean	**292.90**	293.22	360.40	427.68	492.74
	Std	25.75	25.94	27.44	36.88	47.71
Circuit Depth	Mean	149.56	**148.20**	163.87	159.38	317.76
	Std	17.26	16.37	18.20	16.11	43.02

Table 6.5: IBM Q Guadalupe metrics (random circuits)

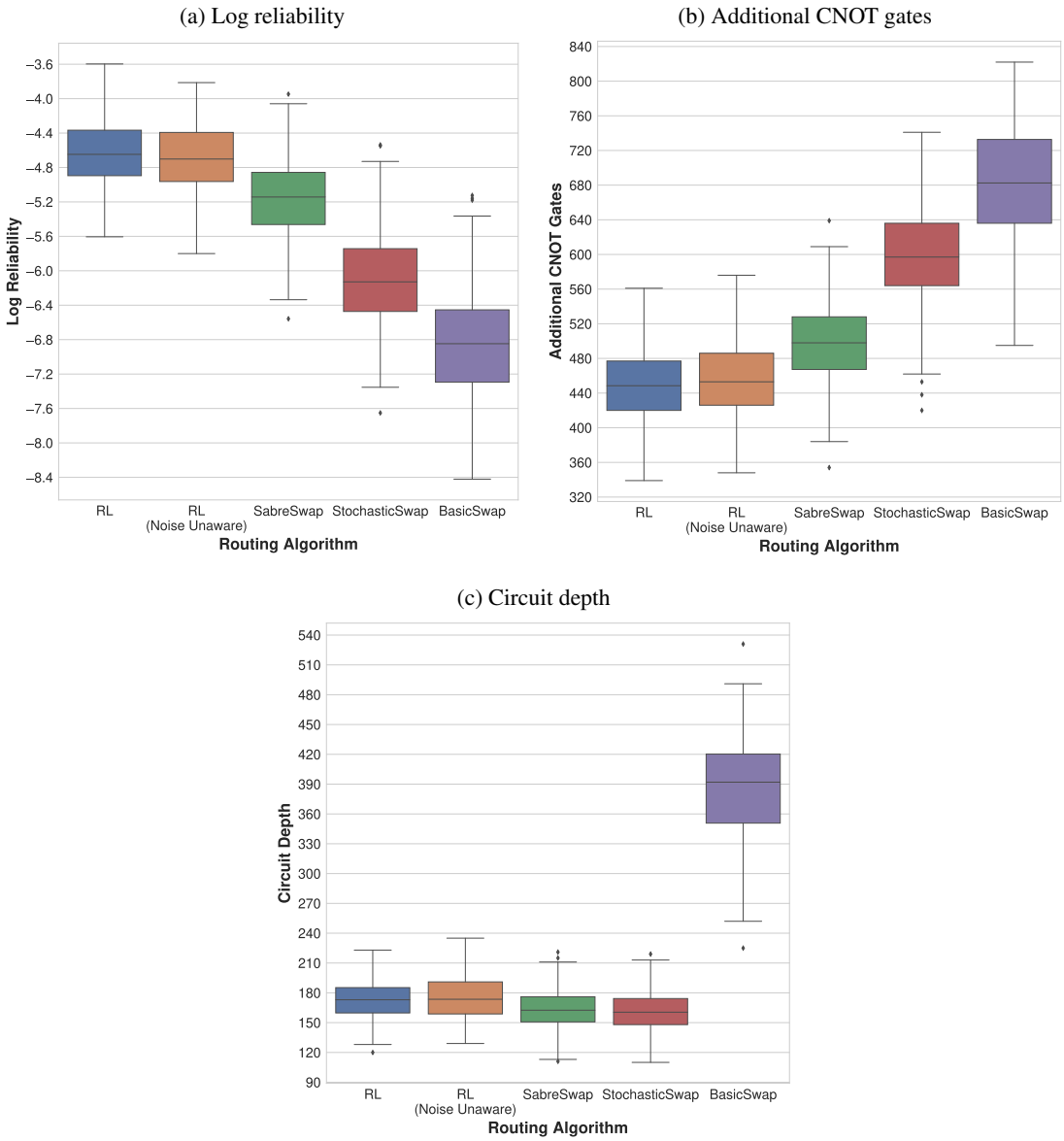


Figure 6.6: IBM Q Mumbai results (random circuits)

		Routing Algorithms				
		RL	RL (Noise Unaware)	SabreSwap	StochasticSwap	BasicSwap
Log reliability	Mean	**-4.638**	-4.701	-5.137	-6.123	-6.857
	Std	0.384	0.406	0.458	0.551	0.659
Added CNOTs	Mean	**448.90**	454.63	494.64	597.32	682.30
	Std	40.32	42.09	44.22	54.74	65.12
Circuit depth	Mean	172.22	175.51	163.47	**161.65**	386.66
	Std	19.60	21.96	19.41	19.78	50.87

Table 6.6: IBM Q Mumbai metrics (random circuits)

6.1.2 Realistic Circuits

The following experiments leveraged the collection of realistic quantum circuits made available by RevLib [69]. After preprocessing the dataset as described in the methodology, we were left with 163 circuits ranging from 3 to 27 qubits and 5 to 900 CNOT gates. From these, 101 use up to 5 qubits, 125 use up to 7 qubits, and 150 use up to 16 qubits. We employed the same performance metrics as in the previous benchmark but also normalized them according to the depth or number of CNOT gates in the original circuits to prevent larger circuits from disproportionately impacting the final results. It can be hard to estimate the difficulty of routing a given circuit since it depends not only on its two-qubit gate count but also on the qubits are involved in those operations. Nonetheless, circuits with more CNOTs will, on average, require more SWAP or BRIDGE operations.

- Regarding the **Manila** (Figure 6.7, Table 6.8), **Belem** (Figure 6.8, Table 6.9), and **Nairobi** (Figure 6.9, Table 6.10) devices, our RL technique managed to improve the mean log reliability by 14.9%, 12.3%, and 18.5%, as well as reduce the number of additional CNOTs by 24.3%, 21.7%, and 28.4% relative to the best-performing Qiskit algorithm. These performance gains are smaller than those measured in the experiments using random circuits, which could be attributed to the fact that the realistic circuits were generally easier to compile and thus offered fewer opportunities for optimization;

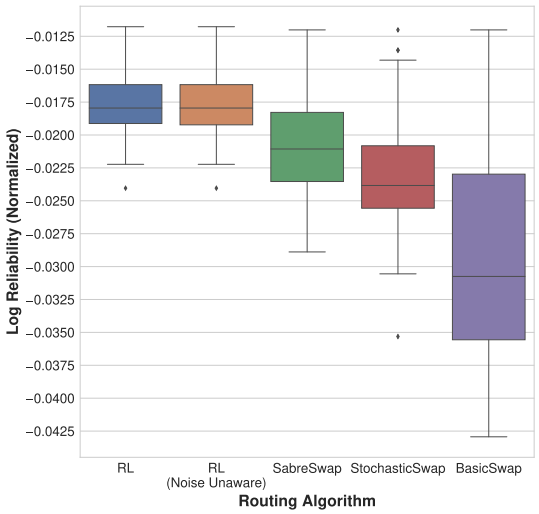
- On **Guadalupe** (Figure 6.10, Table 6.11), RL delivered performance uplifts of 7.4% (log reliability) and 6.8% (added CNOTs) over SABRE, but still significantly surpassed Qiskit's other routing methods;

- **Mumbai** (Figure 6.11, Table 6.12) was the only device where our approach failed to outperform the Qiskit compiler. Given that the RL approach was successful at routing random circuits, one possible reason for its poor performance in this scenario is that a majority ($\approx 77\%$) of the circuits in the dataset only use up to 7 qubits. Since the model for Mumbai was trained exclusively on 27-qubit random circuits, it will rarely experience situations where most of its qubits are not being used. Thus, experimenting with differently-sized circuits during training could be a relevant avenue for future research.

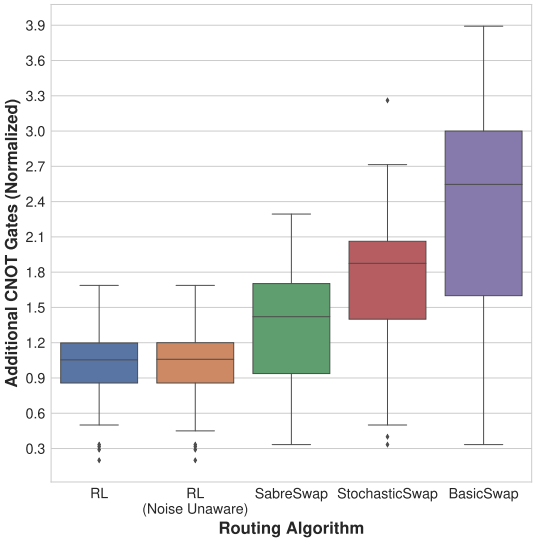
	DEVICES				
	Manila	Belem	Nairobi	Guadalupe	Mumbai
LOG RELIABILITY	14.9%	12.3%	18.5%	7.4%	-32.4%
ADDED CNOTS	24.3%	21.7%	28.4%	6.8%	-46.4%
DEPTH	4.8%	-0.9%	3.0%	-2.8%	-12.7%

Table 6.7: Relative improvements over best Qiskit algorithm (realistic circuits)

(a) Log reliability (normalized)



(b) Additional CNOT gates (normalized)



(c) Circuit depth (normalized)

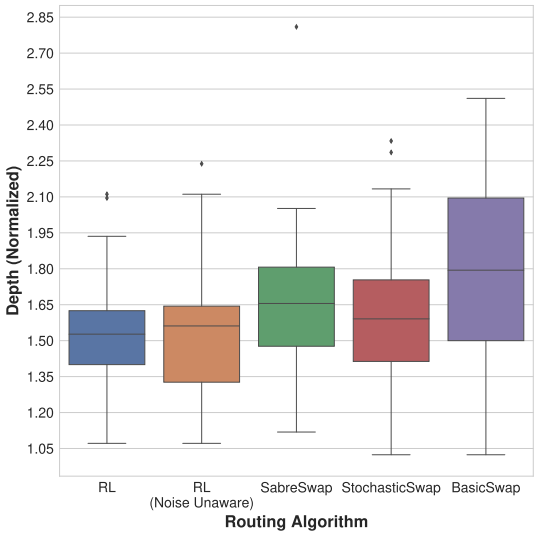
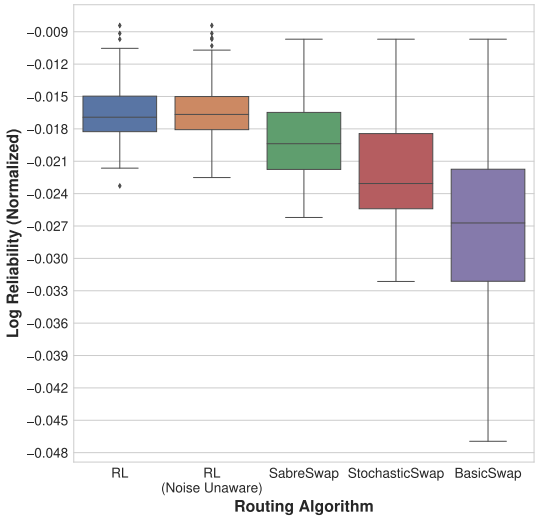


Figure 6.7: IBM Q Manila results (realistic circuits)

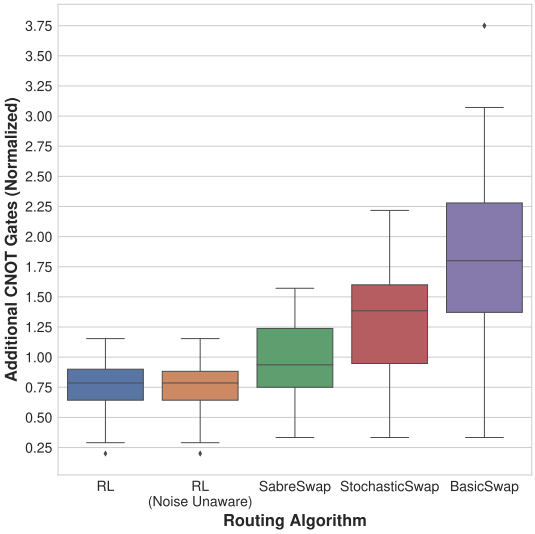
		Routing Algorithms				
		RL	RL (Noise Unaware)	SabreSwap	StochasticSwap	BasicSwap
Log reliability	Mean	**-0.0177**	-0.0177	-0.0208	-0.0229	-0.0295
	Std	0.0022	0.0024	0.0037	0.0042	0.0075
Added CNOTs	Mean	**1.011**	1.018	1.335	1.715	2.306
	Std	0.284	0.292	0.470	0.548	0.854
Circuit depth	Mean	**1.511**	1.521	1.642	1.588	1.811
	Std	0.205	0.226	0.250	0.247	0.380

Table 6.8: IBM Q Manila normalized metrics (realistic circuits)

(a) Log reliability (normalized)



(b) Additional CNOT gates (normalized)



(c) Circuit depth (normalized)

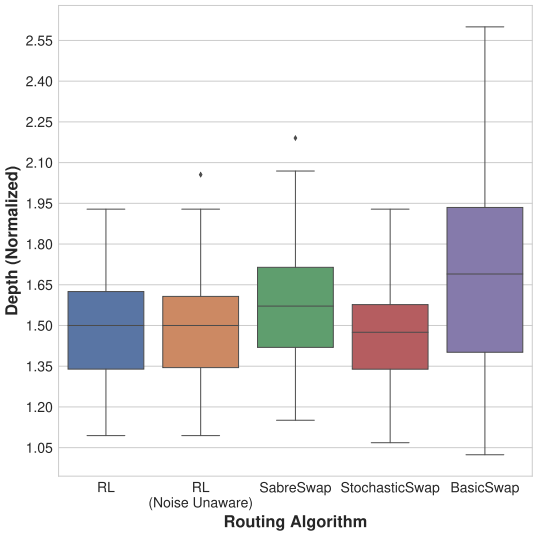
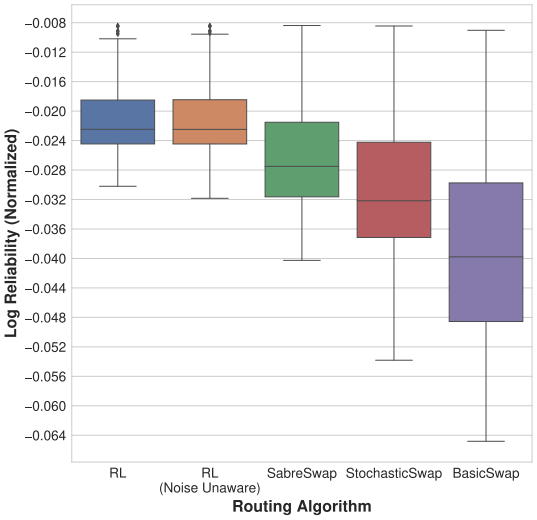


Figure 6.8: IBM Q Belem results (realistic circuits)

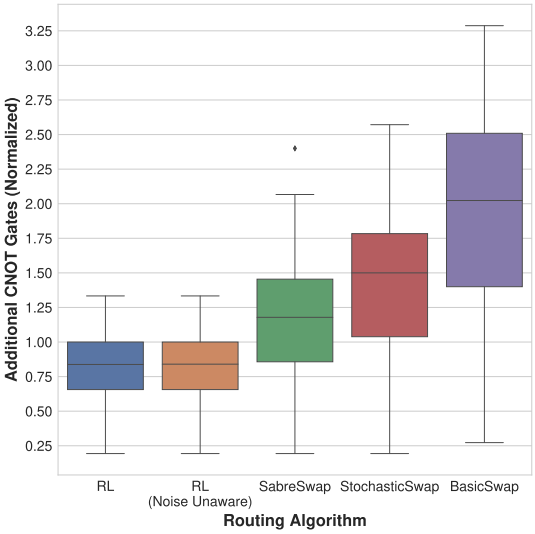
		Routing Algorithms				
		RL	RL (Noise Unaware)	`SabreSwap`	`StochasticSwap`	`BasicSwap`
Log reliability	Mean	-0.0164	**-0.0163**	-0.0187	-0.0216	-0.0265
	Std	0.0029	0.0029	0.0041	0.0052	0.0079
Added CNOTs	Mean	0.768	**0.763**	0.981	1.270	1.830
	Std	0.188	0.194	0.312	0.420	0.724
Circuit depth	Mean	1.481	1.488	1.567	**1.468**	1.679
	Std	0.178	0.190	0.212	0.167	0.348

Table 6.9: IBM Q Belem normalized metrics (realistic circuits)

(a) Log reliability (normalized)



(b) Additional CNOT gates (normalized)



(c) Circuit depth (normalized)

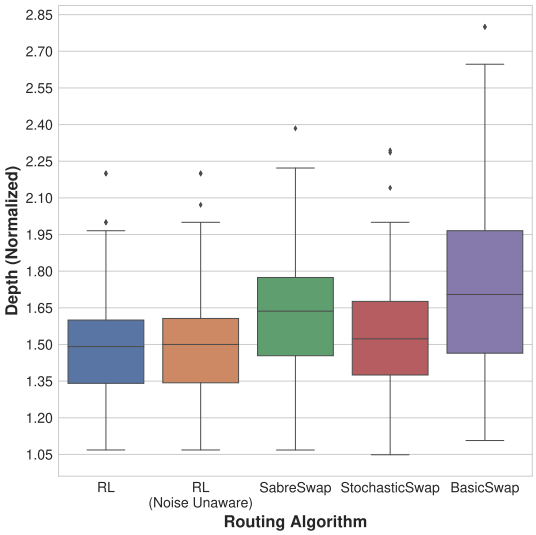
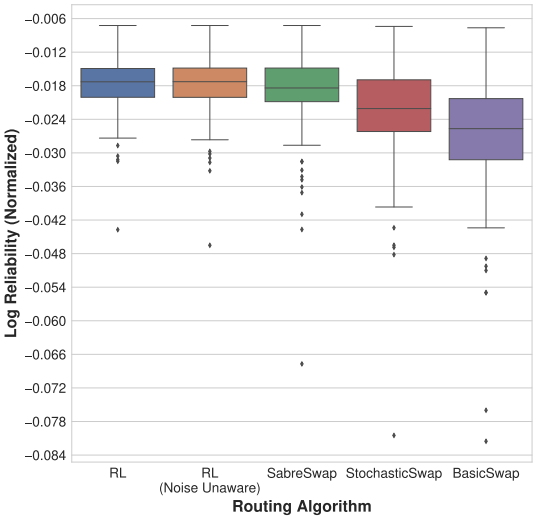


Figure 6.9: IBM Nairobi results (realistic circuits)

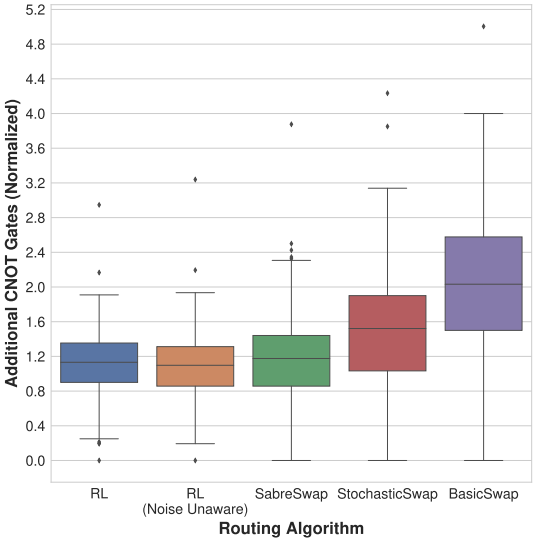
		Routing Algorithms				
		RL	RL (Noise Unaware)	SabreSwap	StochasticSwap	BasicSwap
Log reliability	Mean	**-0.0212**	-0.0213	-0.0260	-0.0303	-0.0384
	Std	0.0052	0.0053	0.0080	0.0094	0.0144
Added CNOTs	Mean	**0.821**	0.826	1.147	1.442	1.936
	Std	0.255	0.259	0.459	0.533	0.774
Circuit depth	Mean	**1.494**	1.500	1.628	1.541	1.741
	Std	0.221	0.222	0.268	0.228	0.365

Table 6.10: IBM Nairobi normalized metrics (realistic circuits)

(a) Log reliability (normalized)



(b) Additional CNOT gates (normalized)



(c) Circuit depth (normalized)

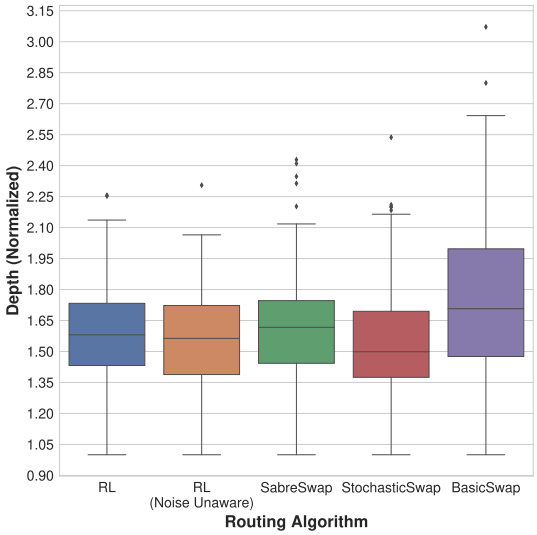
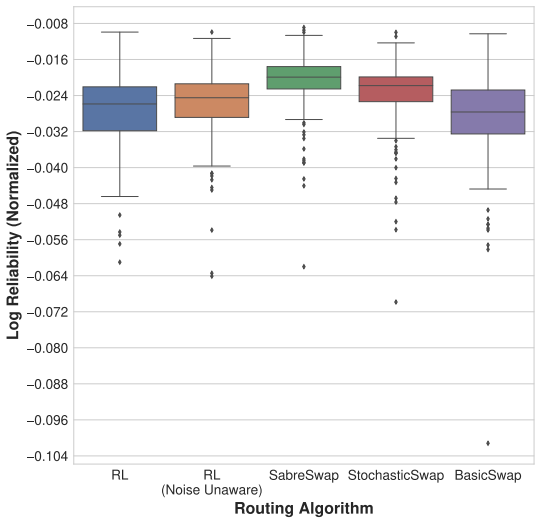


Figure 6.10: IBM Q Guadalupe results (realistic circuits)

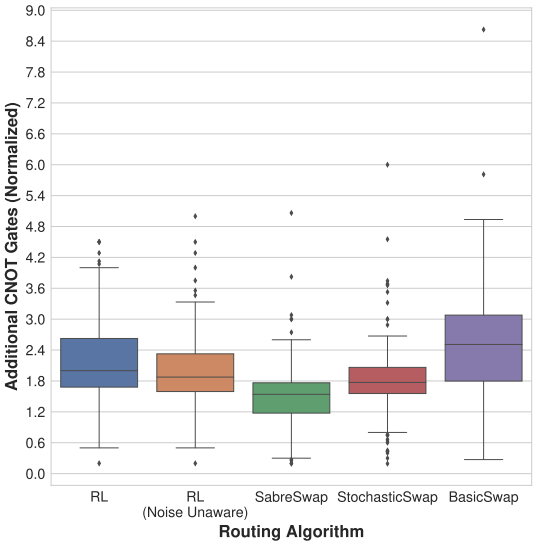
		Routing Algorithms				
		RL	RL (Noise Unaware)	`SabreSwap`	`StochasticSwap`	`BasicSwap`
Log reliability	Mean	-0.0175	**-0.0175**	-0.0189	-0.0226	-0.0263
	Std	0.0051	0.0054	0.0076	0.0092	0.0112
Added CNOTs	Mean	1.114	**1.082**	1.195	1.526	2.015
	Std	0.412	0.424	0.543	0.667	0.844
Circuit depth	Mean	1.586	1.572	1.622	**1.543**	1.739
	Std	0.241	0.237	0.269	0.281	0.368

Table 6.11: IBM Q Guadalupe normalized metrics (realistic circuits)

(a) Log reliability (normalized)



(b) Additional CNOT gates (normalized)



(c) Circuit depth (normalized)

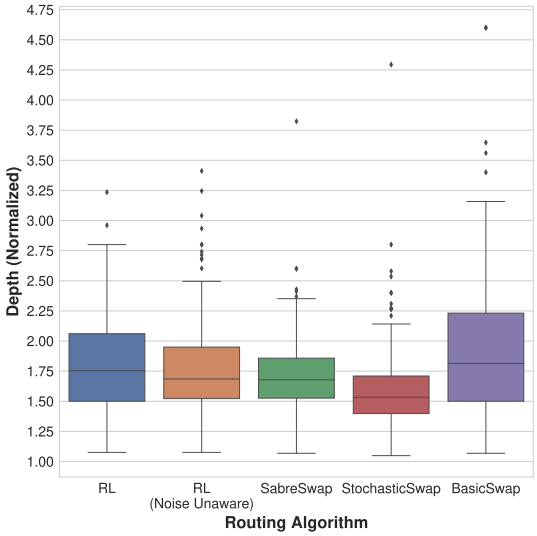


Figure 6.11: IBM Q Mumbai results (realistic circuits)

		Routing Algorithms				
		RL	RL (Noise Unaware)	SabreSwap	StochasticSwap	BasicSwap
Log reliability	Mean	-0.0278	-0.0260	**-0.0210**	-0.0238	-0.0288
	Std	0.0084	0.0078	0.0067	0.0079	0.0107
Added CNOTs	Mean	2.209	1.994	**1.509**	1.824	2.464
	Std	0.835	0.718	0.632	0.737	1.076
Circuit depth	Mean	1.814	1.792	1.718	**1.609**	1.929
	Std	0.402	0.412	0.337	0.371	0.571

Table 6.12: IBM Q Mumbai normalized metrics (realistic circuits)

6.1.3 Impact of Enhancements

To judge the individual contribution of the enhancements to the core reinforcement learning formulation (detailed in Section 4.2.2) towards the quality of the solutions found, we performed the experiment described in Section 6.1.1 several times with different optimizations disabled. Due to time constraints, we only performed this analysis for the 5-qubit T-like architecture (IBM Q Belem) since changing the set of active enhancements requires training an entirely new model. Thus, it should be noted that these findings may not necessarily extend to the complete set of topologies studied. Figure 6.12 demonstrates the relative performance of the models. The results confirm that, even without enhancements, the RL approach can outperform `SabreSwap` on the 5-qubit device. Furthermore, we see that gate commutation analysis and BRIDGE gates are the components that cause the largest drop in performance when removed. Finally, we can conclude that masking out SWAP actions according to the gates in the front layer is not required in this case, as the variant without these restrictions achieved similar performance to the default model, likely due to the simplicity of the device topology.

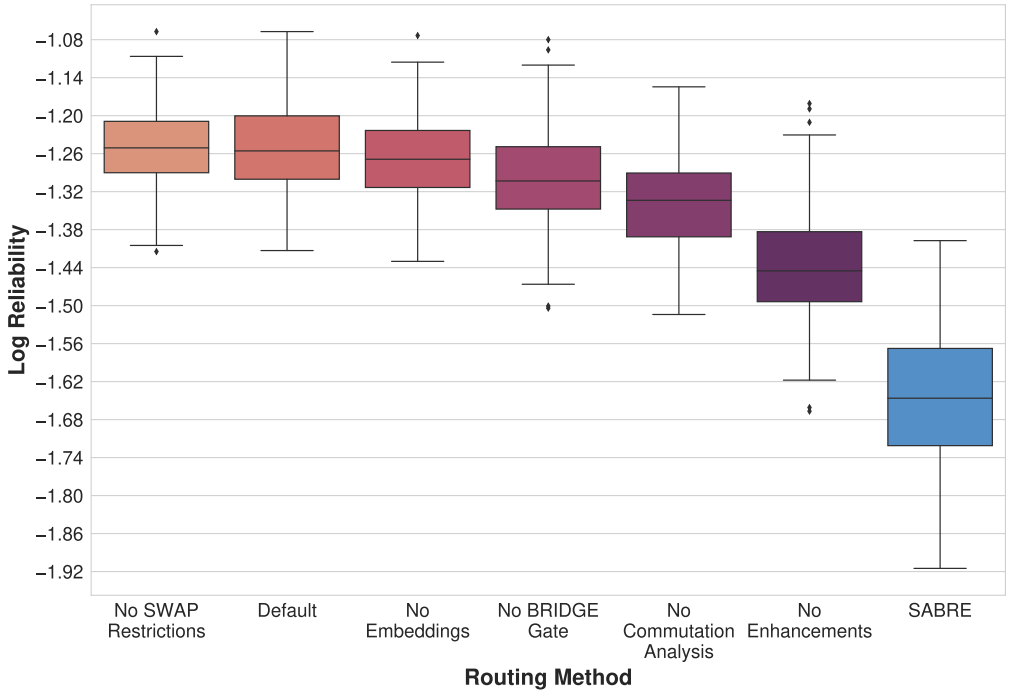


Figure 6.12: Model performance with different enhancements disabled

6.1.4 Impact of Evaluation Episodes

We previously discussed how routing the same circuit several times can yield better results due to the inherent randomness of policies trained using PPO. To investigate this property, we executed the random circuits benchmark for IBM Nairobi using different numbers of evaluation episodes, as depicted in Figure 6.13. In this instance, we found that even routing circuits once is enough to

beat the SABRE algorithm, but we can obtain even better results by allowing multiple attempts. Nevertheless, increasing the number of episodes has rapidly diminishing returns: as shown in the plot, going from 8 to 16 episodes has a minimal impact on the log reliability of routed circuits. We also measured the effectiveness of a deterministic policy that simply selects the action with the highest probability, finding it to be slightly superior to a stochastic policy that uses a single evaluation episode.

It is important to keep in mind that the magnitude of these gains may vary depending on the target architecture, the complexity of the input circuits, and the randomness of the learned policy. Still, it is reasonable to expect that as device and circuit sizes grow, so will the opportunities for improvement, as optimal solutions will be much harder to find.

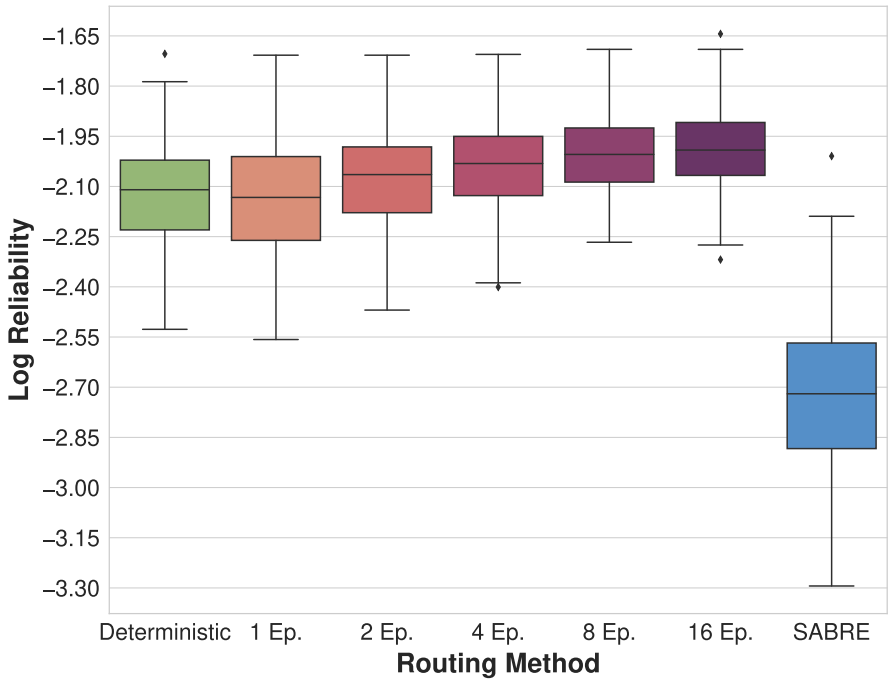


Figure 6.13: Routing performance relative to the number of evaluation episodes

6.1.5 Training and Routing Times

The total training time for each model is presented in Table 6.13, along with the average time needed to route a circuit from the random circuits benchmark. It is worth mentioning that optimizing for execution time was not one of our primary concerns during development, so while our RL strategy is considerably slower compared to the SABRE algorithm, there are still opportunities for reducing the routing time, such as running multiple episodes in parallel.

The models for Guadalupe and Mumbai proved much more challenging to train than those for 5 and 7-qubit devices, which required little training to achieve good results. Moreover, the training process was somewhat less stable when targeting these larger computers, which may indicate that further adjustments to the reward function or hyperparameters are required. Since our models are

	DEVICES				
	Manila	Belem	Nairobi	Guadalupe	Mumbai
Training time (hours)	0.55	0.57	0.76	5.34	8.90
Training iteration (s)	1.31	1.34	1.30	3.84	8.01
RL mean routing time (s)	0.342	0.273	0.450	1.295	2.493
SABRE mean routing time (s)	4.80×10^{-3}	4.73×10^{-3}	5.01×10^{-3}	6.85×10^{-3}	7.82×10^{-3}

Table 6.13: Training and routing times for each device

relatively small, a substantial portion of the training time was spent collecting experiences from the environment, which is generally a CPU-intensive task.

6.2 Discussion

Having presented and analyzed our experimental results, we now return to the three research questions raised in Chapter 4:

> **RQ1**: *How does a strategy based on deep reinforcement learning stack up against other qubit routing algorithms employed by existing quantum compilers in terms of the fidelity and compactness of resulting circuits?*

In general, the proposed qubit routing strategy performed remarkably well compared to the three algorithms currently offered by the Qiskit compiler, significantly improving on the solutions found by the `StochasticSwap` and `BasicSwap` methods and also surpassing SABRE across almost all of our tests. As expected, the models for 5 and 7-qubit computers achieved the best results, but we still observed notable performance gains when compiling for larger devices. Circuits routed using RL were consistently more reliable and used fewer error-prone two-qubit operations. In some cases, our strategy also managed to reduce circuit depth (mainly when routing randomly generated circuits).

> **RQ2**: *When targeting realistic quantum architectures, to what extent can awareness about device error rates help reinforcement learning models route circuits more reliably?*

Throughout our experiments, the noise-aware and unaware reinforcement learning models performed similarly, meaning the inclusion of edge reliabilities in the observation did not ultimately result in higher-fidelity circuits. We can think of a few potential explanations for this fact. First, the relatively low connectivity of the IBM devices studied could hinder the ability to optimize the routing process for reliability as, in most cases, there is only a single shortest path between any two qubits. If we instead consider a grid-like topology, we can readily find multiple ways to route the same pair of qubits that, despite involving the same number of operations, could have varying

fidelities. Without these alternative paths, maximizing reliability effectively becomes equivalent to minimizing the number of two-qubit gates.

Another possibility is that the additional reward from picking more reliable device edges is too subtle compared with the weight of using more SWAP or BRIDGE operations. Given how the observations and the reward function have been defined, the intended behavior may be too complex for the agent to learn. Something we did notice when examining the metrics logged by RLlib was that the value loss and explained variance were consistently better in noise-aware models, suggesting that the value network is better at predicting the expected return of a given state if the agent has access to the device calibration data. In any case, both of these justifications point to future research directions, either by investigating other types of architectures or by modifying the RL environment.

> **RQ3**: *How does the reinforcement learning approach fare across different device topologies with respect to both the routing performance and the resources required to train the model?*

Our RL strategy performed exceptionally well when compiling both random and realistic circuits for the simpler 5 and 7-qubit architectures. In these environments, PPO quickly converged toward an effective policy and appeared more resilient to changes in its hyperparameters. In contrast, the models for Guadalupe and Mumbai warranted additional tweaking. To improve stability and promote exploration, we increased the batch size and the entropy coefficient. Due to the additional complexity of mapping circuits onto these larger devices, the associated models were trained for longer but still managed favorable results, particularly for the 16-qubit Guadalupe computer.

The added computational resources needed to train these reinforcement learning agents can be considered a drawback of our approach compared to heuristic routing algorithms. That said, we emphasize that each model is associated with a particular topology, not the specific IBM computers we chose to study. For example, our 7-qubit agent is able to route circuits for IBM's Nairobi, Jakarta, Lagos, and Perth devices since all of them share the same coupling graph. More importantly, given the strict constraints of the current NISQ era, any technique that can mitigate the impact of noise or lower the consumption of quantum resources is highly valuable, even if it comes at the expense of classical processing time. Moreover, unlike exact algorithms, our methodology can still be applied to larger devices and circuits where the optimal set of routing operations cannot be discovered within a reasonable time frame.

Chapter 7

Conclusion

Quantum computing is an exciting new frontier in technology. There are high expectations regarding its potential impact and relevancy to several domains, and we have seen significant investments from large corporations such as IBM and Google to drive progress in the field. Despite remarkable hardware advancements, existing quantum computers are still constrained by low qubit counts and susceptibility to errors.

Before fault-tolerant quantum computing can be realized, we must operate within the limitations of the NISQ era and develop approaches and tools that enable us to fully exploit the potential of current devices. The ability to construct efficient circuits with careful consideration for hardware details is an essential step in the pursuit of quantum advantage with NISQ resources, motivating research efforts into quantum compiling.

This dissertation sought to design and evaluate a reinforcement learning-based technique for tackling the qubit routing problem in the context of realistic NISQ architectures. Our work builds upon similar approaches from the literature, proposing an enhanced RL formulation and incorporating device calibration data and commutation analysis. In nearly all experiments, our method produced significantly shorter and more reliable circuits compared to the routing algorithms offered by Qiskit, one of the most widely used quantum computing frameworks. These results help to further consolidate reinforcement learning as an effective qubit routing strategy in the context of NISQ computing. On another note, factoring in device calibration data when training our RL models did not deliver a substantial advantage when routing circuits for IBM's quantum computers, a finding that, from our perspective, warrants further experimentation.

Improvements to the compilation pipeline (such as the proposed routing technique) primarily help mitigate the adverse effects of noise on the execution of quantum programs. By itself, this is a valuable advantage, but it can have other knock-on benefits. Because quantum hardware is generally only accessible via the cloud through a shared queue or credit-based system, shorter circuits can also reduce wait time and save resources.

We consider that the main objectives established for this dissertation were successfully achieved. Nevertheless, we can pinpoint several aspects that constrained or hampered our work. First, research on quantum computing remains in its early stages and is rapidly developing. Although

there are many opportunities for discovery and innovation, it can be challenging to identify contributions of a suitable scope without a deep understanding of the field. From a software engineering perspective, quantum computing has yet to become a well-established topic, amplifying the background knowledge requirements for research in this area. On a separate note, certain key articles from the literature did not disclose their source code, making it harder to expand on existing ideas that aligned with our goals. Finally, given that we could not access past calibration data for some of IBM's quantum computers, we resorted to randomly generating this data (to an extent) in order to train our RL models.

Potential directions for future research include gauging the performance of our RL routing technique when targeting other quantum device topologies, as well as refining the reinforcement learning environment by adjusting the structure of the agent's observations or better shaping the reward function. Despite being beyond the scope of this work, conducting a systematic hyperparameter search might be beneficial, particularly for the larger RL models, which tended to be less stable during training. Designing noise-adaptive variants of other qubit routing approaches from the literature (such as Monte Carlo tree search) by integrating calibration data such as gate error rates or qubit coherence times is also a suitable angle to explore.

Additionally, it would be relevant to expand the evaluation methodology by compiling circuits for near-term quantum algorithms (for example, variational algorithms such as VQE or QAOA) and either executing them directly on NISQ hardware (through IBM's cloud services) or simulating their execution on a classical computer and approximating the behavior of the target device using a noise model. Since the output of a quantum algorithm is often not a single value but a distribution of values assembled over multiple runs, one would have to compare different routing strategies using a metric such as the KL divergence relative to the noiseless distribution or via algorithm-specific performance metrics (for example, the estimated ground state energy in the case of VQE).

References

[1] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C. Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando G. S. L. Brandao, David A. Buell, Brian Burkett, Yu Chen, Zijun Chen, Ben Chiaro, Roberto Collins, William Courtney, Andrew Dunsworth, Edward Farhi, Brooks Foxen, Austin Fowler, Craig Gidney, Marissa Giustina, Rob Graff, Keith Guerin, Steve Habegger, Matthew P. Harrigan, Michael J. Hartmann, Alan Ho, Markus Hoffmann, Trent Huang, Travis S. Humble, Sergei V. Isakov, Evan Jeffrey, Zhang Jiang, Dvir Kafri, Kostyantyn Kechedzhi, Julian Kelly, Paul V. Klimov, Sergey Knysh, Alexander Korotkov, Fedor Kostritsa, David Landhuis, Mike Lindmark, Erik Lucero, Dmitry Lyakh, Salvatore Mandrà, Jarrod R. McClean, Matthew McEwen, Anthony Megrant, Xiao Mi, Kristel Michielsen, Masoud Mohseni, Josh Mutus, Ofer Naaman, Matthew Neeley, Charles Neill, Murphy Yuezhen Niu, Eric Ostby, Andre Petukhov, John C. Platt, Chris Quintana, Eleanor G. Rieffel, Pedram Roushan, Nicholas C. Rubin, Daniel Sank, Kevin J. Satzinger, Vadim Smelyanskiy, Kevin J. Sung, Matthew D. Trevithick, Amit Vainsencher, Benjamin Villalonga, Theodore White, Z. Jamie Yao, Ping Yeh, Adam Zalcman, Hartmut Neven, and John M. Martinis. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, October 2019.

[2] IBM. IBM Quantum Computing | Roadmap. https://www.ibm.com/quantum/roadmap, 2023. Accessed: January 2023.

[3] Google Quantum AI. Hardware | Google Quantum AI. https://www.quantinuum.com/hardware. Accessed: August 2023.

[4] D-Wave. The Advantage™ Quantum Computer | D-Wave. https://www.dwavesys.com/solutions-and-products/systems. Accessed: August 2023.

[5] Rigetti Computing. Quantum Computing. https://www.rigetti.com. Accessed: August 2023.

[6] Quantinuum. Quantinuum | Hardware. https://www.quantinuum.com/hardware. Accessed: August 2023.

[7] Qiskit Contributors. Qiskit: An Open-source Framework for Quantum Computing. https://qiskit.org, 2023. Accessed: August 2023.

[8] Google. Cirq. https://quantumai.google/cirq, 2023. Accessed: February 2023.

[9] IBM. IBM Quantum. https://quantum-computing.ibm.com, 2023. Accessed: February 2023.

[10] Paul Benioff. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *Journal of Statistical Physics*, 22(5):563–591, May 1980.

[11] Ethan Bernstein and Umesh Vazirani. Quantum Complexity Theory. *SIAM Journal on Computing*, 26(5):1411–1473, October 1997.

[12] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences*, 439(1907):553–558, December 1992.

[13] Daniel R. Simon. On the Power of Quantum Computation. *SIAM Journal on Computing*, 26(5):1474–1483, October 1997.

[14] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Santa Fe, NM, USA, 1994. IEEE Comput. Soc. Press.

[15] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing - STOC '96*, pages 212–219, Philadelphia, Pennsylvania, United States, 1996. ACM Press.

[16] Peter W. Shor. Scheme for reducing decoherence in quantum computer memory. *Physical Review A*, 52:R2493–R2496, Oct 1995.

[17] Barbara M. Terhal. Quantum error correction for quantum memories. *Reviews of Modern Physics*, 87:307–346, Apr 2015.

[18] John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018.

[19] M. Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C. Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R. McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, and Patrick J. Coles. Variational quantum algorithms. *Nature Reviews Physics*, 3(9):625–644, September 2021.

[20] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1):4213, July 2014.

[21] Edward Farhi and Hartmut Neven. Classification with Quantum Neural Networks on Near Term Processors. *arXiv e-prints*, February 2018.

[22] J. S. Otterbach, R. Manenti, N. Alidoust, A. Bestwick, M. Block, B. Bloom, S. Caldwell, N. Didier, E. Schuyler Fried, S. Hong, P. Karalekas, C. B. Osborn, A. Papageorge, E. C. Peterson, G. Prawiroatmodjo, N. Rubin, Colm A. Ryan, D. Scarabelli, M. Scheer, E. A. Sete, P. Sivarajah, Robert S. Smith, A. Staley, N. Tezak, W. J. Zeng, A. Hudson, Blake R. Johnson, M. Reagor, M. P. da Silva, and C. Rigetti. Unsupervised Machine Learning on a Hybrid Quantum Computer. *arXiv e-prints*, December 2017.

[23] Owen Lockwood and Mei Si. Reinforcement Learning with Quantum Variational Circuit. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 16(1):245–251, October 2020. Number: 1.

[24] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A Quantum Approximate Optimization Algorithm. *arXiv e-prints*, November 2014.

[25] Xiao Yuan, Suguru Endo, Qi Zhao, Ying Li, and Simon C. Benjamin. Theory of variational quantum simulation. *Quantum*, 3:191, October 2019. Publisher: Verein zur Förderung des Open Access Publizierens in den Quantenwissenschaften.

[26] Eric Anschuetz, Jonathan Olson, Alán Aspuru-Guzik, and Yudong Cao. Variational Quantum Factoring. In Sebastian Feld and Claudia Linnhoff-Popien, editors, *Quantum Technology and Optimization Problems*, Lecture Notes in Computer Science, pages 74–85, Cham, 2019. Springer International Publishing.

[27] Carlos Bravo-Prieto, Ryan LaRose, M. Cerezo, Yigit Subasi, Lukasz Cincio, and Patrick J. Coles. Variational Quantum Linear Solver. *arXiv e-prints*, June 2020.

[28] Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation, and Jay M. Gambetta. Validating quantum computers using randomized model circuits. *Physical Review A*, 100(3):032328, September 2019.

[29] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintao Pereira. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 113–125, New York, NY, USA, February 2018. Association for Computing Machinery.

[30] Bochen Tan and Jason Cong. Optimality Study of Existing Quantum Computing Layout Synthesis Tools. *IEEE Transactions on Computers*, 70(9):1363–1373, September 2021.

[31] Amira Abbas, Stina Andersson, Abraham Asfaw, Antonio Corcoles, Luciano Bello, Yael Ben-Haim, Mehdi Bozzo-Rey, Sergey Bravyi, Nicholas Bronn, Lauren Capelluto, Almudena Carrera Vazquez, Jack Ceroni, Richard Chen, Albert Frisch, Jay Gambetta, Shelly Garion, Leron Gil, Salvador De La Puente Gonzalez, Francis Harkins, Takashi Imamichi, Pavan Jayasinha, Hwajung Kang, Amir h Karamlou, Robert Loredo, David McKay, Alberto Maldonado, Antonio Macaluso, Antonio Mezzacapo, Zlatko Minev, Ramis Movassagh, Giacomo Nannicini, Paul Nation, Anna Phan, Marco Pistoia, Arthur Rattew, Joachim Schaefer, Javad Shabani, John Smolin, John Stenger, Kristan Temme, Madeleine Tod, Ellinor Wanzambi, Stephen Wood, and James Wootton. Learn Quantum Computation Using Qiskit. https://qiskit.org/textbook, 2020.

[32] Eric R. Johnson, Nic Harrigan, and Mercedes Gimeno-Segovia. *Programming Quantum Computers: Essential Algorithms and Code Samples*. O'Reilly Media, Inc., 1st edition, 2019.

[33] Frederic T. Chong, Diana Franklin, and Margaret Martonosi. Programming languages and compiler design for realistic quantum hardware. *Nature*, 549(7671):180–187, September 2017.

[34] Qiskit Contributors. Transpiler (qiskit.transpiler) — Qiskit 0.43.2 documentation. https://qiskit.org/documentation/apidoc/transpiler.html. Accessed: July 2023.

[35] Christopher M. Dawson and Michael A. Nielsen. The Solovay-Kitaev algorithm. *Quantum Information & Computation*, 6(1):81–95, January 2006.

[36] A. Yu Kitaev. Quantum computations: algorithms and error correction. *Russian Mathematical Surveys*, 52(6):1191, December 1997.

[37] Hongxiang Fan, Ce Guo, and Wayne Luk. Optimizing quantum circuit placement via machine learning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, pages 19–24, New York, NY, USA, August 2022. Association for Computing Machinery.

[38] G. Vidal and C. M. Dawson. Universal quantum circuit for two-qubit transformations with three controlled-NOT gates. *Physical Review A*, 69(1):010301, January 2004.

[39] Kishor Bharti, Alba Cervera-Lierta, Thi Ha Kyaw, Tobias Haug, Sumner Alperin-Lea, Abhinav Anand, Matthias Degroote, Hermanni Heimonen, Jakob S. Kottmann, Tim Menke, Wai-Keong Mok, Sukin Sim, Leong-Chuan Kwek, and Alán Aspuru-Guzik. Noisy intermediate-scale quantum algorithms. *Reviews of Modern Physics*, 94(1):015004, February 2022.

[40] Lennart Bittel and Martin Kliesch. Training Variational Quantum Algorithms Is NP-Hard. *Physical Review Letters*, 127(12):120502, September 2021.

[41] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. The MIT Press, Cambridge, Massachusetts, 2nd edition, 2018.

[42] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply Modern RL Methods, with Deep Q-Networks, Value Iteration, Policy Gradients, TRPO, AlphaGo Zero and More*. Packt Publishing, 2nd edition, 2020.

[43] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[44] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, December 2013.

[45] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv e-prints*, August 2017.

[46] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust Region Policy Optimization. *arXiv e-prints*, April 2017.

[47] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, March 1951.

[48] Shengyi Huang and Santiago Ontañón. A Closer Look at Invalid Action Masking in Policy Gradient Algorithms. *The International FLAIRS Conference Proceedings*, 35, May 2022.

[49] Matteo G. Pozzi, Steven J. Herbert, Akash Sengupta, and Robert D. Mullins. Using Reinforcement Learning to Perform Qubit Routing in Quantum Compilers. *ACM Transactions on Quantum Computing*, 3(2):10:1–10:25, May 2022.

[50] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Science and Technology*, 6(1):014003, November 2020.

[51] Xiangzhen Zhou, Yuan Feng, and Sanjiang Li. A monte carlo tree search framework for quantum circuit transformation. In *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, pages 1–7, New York, NY, USA, December 2020. Association for Computing Machinery.

[52] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, December 2018. Publisher: American Association for the Advancement of Science.

[53] Animesh Sinha, Utkarsh Azad, and Harjinder Singh. Qubit Routing Using Graph Neural Network Aided Monte Carlo Tree Search. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(9):9935–9943, June 2022.

[54] Prakash Murali, Jonathan M. Baker, Ali Javadi-Abhari, Frederic T. Chong, and Margaret Martonosi. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 1015–1029, New York, NY, USA, April 2019. Association for Computing Machinery.

[55] Qiskit Contributors. NoiseAdaptiveLayout — Qiskit 0.43.2 documentation. https://qiskit.org/documentation/stubs/qiskit.transpiler.passes.NoiseAdaptiveLayout.html, 2023. Accessed: June 2023.

[56] Shin Nishio, Yulu Pan, Takahiko Satoh, Hideharu Amano, and Rodney Van Meter. Extracting Success from IBM's 20-Qubit Machines Using Error-Aware Compilation. *ACM Journal on Emerging Technologies in Computing Systems*, 16(3):32:1–32:25, May 2020.

[57] Ji Liu, Peiyi Li, and Huiyang Zhou. Not All SWAPs Have the Same Cost: A Case for Optimization-Aware Qubit Routing. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 709–725, April 2022.

[58] Toshinari Itoko, Rudy Raymond, Takashi Imamichi, and Atsushi Matsuo. Optimization of quantum circuit mapping using gate transformation and commutation. *Integration*, 70:43–50, January 2020.

[59] Davide Venturelli, Minh Do, Eleanor Rieffel, and Jeremy Frank. Temporal Planning for Compilation of Quantum Approximate Optimization Circuits. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, pages 4440–4446, 2017.

[60] Davide Venturelli, Minh Do, Eleanor Rieffel, and Jeremy Frank. Compiling quantum circuits to realistic hardware architectures using temporal planners. *Quantum Science and Technology*, 3(2):025004, February 2018.

[61] Kyle E. C. Booth, Minh Do, J. Christopher Beck, Eleanor Rieffel, Davide Venturelli, and Jeremy Frank. Comparing and Integrating Constraint Programming and Temporal Planning for Quantum Circuit Compilation. *arXiv e-prints*, March 2018.

[62] Abhoy Kole, Stefan Hillmich, Kamalika Datta, Robert Wille, and Indranil Sengupta. Improved Mapping of Quantum Circuits to IBM QX Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2375–2383, October 2020.

[63] Abtin Molavi, Amanda Xu, Martin Diges, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. Qubit Mapping and Routing via MaxSAT. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1078–1091, October 2022.

[64] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. Qubit allocation as a combination of subgraph isomorphism and token swapping. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):120:1–120:29, October 2019.

[65] 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*, ASPLOS '19, pages 1001–1014, New York, NY, USA, April 2019. Association for Computing Machinery.

[66] Sumeet Khatri, Ryan LaRose, Alexander Poremba, Lukasz Cincio, Andrew T. Sornborger, and Patrick J. Coles. Quantum-assisted quantum compiling. *Quantum*, 3:140, May 2019.

[67] Kunal Sharma, Sumeet Khatri, M. Cerezo, and Patrick J. Coles. Noise resilience of variational quantum compiling. *New Journal of Physics*, 22(4):043006, April 2020.

[68] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks. *arXiv e-prints*, November 2017.

[69] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *Int'l Symp. on Multi-Valued Logic*, pages 220–225, 2008. RevLib is available at https://www.revlib.org.

[70] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. *arXiv e-prints*, July 2017.

[71] Microsoft. Q# and the Quantum Development Kit | Microsoft Azure. https://azure.microsoft.com/en-us/resources/development-kit/quantum-computing, 2023. Accessed: February 2023.

[72] Qiskit Contributors. Qiskit IBM Quantum Provider. https://github.com/Qiskit/qiskit-ibm-provider, August 2023. Accessed: August 2023.

[73] Eric Liang, Richard Liaw, Philipp Moritz, Robert Nishihara, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for Distributed Reinforcement Learning, June 2018.

[74] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A Distributed Framework for Emerging AI Applications, September 2018.

[75] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-Baselines3: Reliable Reinforcement Learning Implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021.

[76] Stable-Baselines Team. Stable-Baselines3 - Contrib (SB3-Contrib). https://github.com/Stable-Baselines-Team/stable-baselines3-contrib, August 2023. Accessed: August 2023.

[77] Mark Towers, Jordan K. Terry, Ariel Kwiatkowski, John U. Balis, Gianluca de Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Arjun KG, Markus Krimmel, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Andrew Tan Jin Shen, and Omar G. Younis. Gymnasium, March 2023.

[78] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[79] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: a system for large-scale machine learning. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, OSDI'16, pages 265–283, USA, November 2016. USENIX Association.

[80] TensorFlow Contributors. TensorBoard. https://github.com/tensorflow/tensorboard, August 2023. Accessed: August 2023.

[81] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[82] Pandas Development Team. pandas-dev/pandas: Pandas. https://doi.org/10.5281/zenodo.8092754, June 2023.

[83] Michael L. Waskom. seaborn: statistical data visualization. *Journal of Open Source Software*, 6(60):3021, 2021.

[84] J. D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.