Investigating the Effects of Variations in Reinforcement Learning Environments and Quantum Hardware for Qubit Routing

MSc Thesis

Joris Henstra

TU Delft

Investigating the Effects of Variations in Reinforcement Learning Environments and Quantum Hardware for Qubit Routing

MSc Thesis

by

Joris Henstra

to obtain the degree of Master of Science at the Delft University of Technology,
to be defended publicly on Wednesday, February 12, 2025, at 10:00 AM.

Student number:	4500954	
Thesis committee:	Dr. Sebastian Feld,	TU Delft, supervisor
	Dr. Jan van Gemert,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Summary

Quantum computing holds the potential to revolutionize computation by leveraging quantum mechanical principles to solve problems intractable for classical computers. However, current noisy intermediate-scale quantum (NISQ) devices are constrained by limited qubit connectivity, high gate error rates, and hardware-specific noise patterns. One of the key challenges in quantum circuit execution is qubit routing—efficiently mapping logical qubits to physical qubits while adhering to connectivity constraints and minimizing error accumulation. Traditional heuristic and rule-based transpilation techniques struggle to generalize across different hardware architectures and noise conditions, motivating the exploration of machine learning approaches for more adaptive and scalable routing strategies.

This study investigates reinforcement learning (RL)-based methods for qubit routing, focusing on how different RL formulations (primitive vs. hierarchical action spaces) and environment configurations (e.g., lookahead depth, training-circuit sizes) impact routing performance. The designed RL environment models quantum hardware constraints through coupling graphs and noise parameters, providing configurable state representations and flexible action spaces.

Key findings indicate that a moderate lookahead (e.g., 4 gates) offers the best balance between performance and computational complexity. Training on circuits with 8–16 gates yields optimal generalization to larger circuits. Comparing RL policy formulations, hierarchical approaches converge faster and perform robustly on complex topologies, while primitive approaches occasionally achieve higher reliability given sufficient training. Larger hardware graphs diminish RL's relative advantage over heuristic transpilers, whereas higher-connectivity topologies improve RL routing efficiency.

Benchmarks against Qiskit's standard transpilers reveal that RL methods are competitive on small-scale and highly connected devices but face challenges on larger and more constrained architectures. While RL-based qubit routing can effectively integrate hardware noise and connectivity constraints, scalability and adaptability to dynamic error rates remain open challenges.

Future research should explore integrating more comprehensive noise models, adaptive noise tracking, concurrent gate execution, hybrid heuristic–RL approaches, and benchmarking with real hardware implementations. As quantum devices scale, RL-driven, noise-aware compilation strategies may become integral to optimizing practical quantum computations.

Contents

1 Introduction

Quantum computing leverages the principles of quantum mechanics—such as superposition and entanglement—to offer computational speedups on problems that are intractable for classical computers [1], [2], [3]. Although rapid progress has been made in designing and building noisy intermediate-scale quantum (NISQ) devices, these systems are constrained by limited qubit numbers, noisy operations, and non-uniform gate connectivity [4]. Such constraints introduce the qubit routing challenge: efficiently mapping a quantum algorithm onto the physical hardware so that gate and measurement operations remain valid while minimizing detrimental effects from noise and circuit overhead.

1.1. Context

Research on quantum circuit compilation has produced many approaches to tackling qubit routing, including constraint-based optimizations, heuristic routing, and more recently, machine learning methods [5], [6], [7], [8]. Reinforcement learning (RL), in particular, has shown promise by adaptively learning routing policies that account for hardware constraints, circuit interactions, and noise factors. However, most RL research for quantum compilation has relied on simplified or custom-tailored environments, limiting applicability to diverse topologies and noise conditions. Noise-awareness and scalability also remain significant challenges, underscoring the need for more flexible, robust, and hardware-aware RL approaches.

1.2. Problem Statement

Existing qubit routing strategies frequently overlook the interplay between noise, hardware variability, and circuit structure. Methods that do account for noise often scale poorly to larger NISQ devices. This research therefore aims to develop and evaluate an RL environment for qubit routing that flexibly represents diverse hardware topologies and connectivity constraints while aiming to be more scalable and noise-aware than current approaches.

1.3. Objectives

A goal is the design of an RL environment that systematically models quantum hardware constraints, noise parameters, and circuit interactions in an extensible and scalable manner. Another objective is to investigate how different RL policy formulations—such as hierarchical action spaces—impact routing outcomes when tested on varied topologies. The performance of these RL agents is then assessed through experiments that measure circuit reliability, compared against heuristic and standard compiler baselines.

1.4. Scope

Superconducting quantum processor architectures with limited qubit connectivity and moderate noise levels anchor the scope of this research. Although the proposed environment is extensible to other

quantum platforms, the experiments and evaluation metrics focus on gate-based superconducting devices. The study emphasizes two-qubit gate connectivity and qubit routing in compilation, without explicitly integrating advanced error mitigation or error-correction techniques.

1.5. Significance

Investigating how different reinforcement learning (RL) policy formulations—such as hierarchical action spaces—impact qubit routing outcomes is crucial for optimizing quantum circuit compilation. Traditional heuristic and rule-based approaches often struggle to generalize across diverse hardware topologies and noise conditions. By exploring alternative RL policy structures, this research aims to uncover strategies that enhance adaptability, efficiency, and scalability in qubit routing.

1.6. Structure

The remainder of this report is organized as follows. Chapter 2 provides theoretical background on quantum computing, including qubit measurement, quantum gates, gate commutation, and hardware constraints, then introduces fundamental concepts in reinforcement learning. Chapter 3 surveys the literature on qubit routing, covering exact formulations, heuristic approaches, and Machine Learning (ML)-based methods and discusses the current research gap. Chapter 4 details the methodology, including problem formulation, environment design, and evaluation metrics as well as the implementation. Chapter 5 presents experimental results that compare RL approaches against baseline transpilers and heuristics. Finally, Chapter 6 summarizes key findings, limitations, and directions for future work.

2

Theory

This section explains the relevant theory and concepts essential for understanding the research, including quantum computing and reinforcement learning.

2.1. Quantum Computing

Quantum computing is a paradigm of computation that exploits the laws of quantum mechanics to process information. This approach leverages quantum phenomena such as superposition and entanglement, which enable computational tasks that can surpass the capabilities of classical systems. Compared to conventional bits, quantum systems can encode information in more complex ways, providing new avenues for algorithm design. At a high level, quantum computing involves manipulating quantum states through specialized operations and measurements, executed on physical devices subject to hardware-specific constraints. The following sections explore the essential principles of quantum computing, including Hilbert spaces and quantum states, qubits, qubit measurement, quantum gates, and gate commutation.

2.1.1. Hilbert Space and Quantum States

Quantum states are represented as vectors in a complex vector space known as a *Hilbert space*.

A Hilbert space $\mathcal{H}$ is a complete vector space equipped with an inner product that allows the computation of norms and angles between vectors.

In quantum computing, the state of a single qubit is described as a unit vector in a two-dimensional Hilbert space $\mathcal{H}_2$:

$$\mathcal{H}_2 = \mathbb{C}^2 = \text{span}\{|0\rangle, |1\rangle\}. \tag{2.1}$$

In bra-ket notation, these basis states can be written explicitly as column vectors:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{2.2}$$

A general quantum state $|\psi\rangle$ in a two-dimensional Hilbert space is a superposition of the basis states $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.3}$$

where α and β are complex probability amplitudes satisfying the normalization condition:

$$|\alpha|^2 + |\beta|^2 = 1 \tag{2.4}$$

Quantum operations, such as quantum gates, are represented as unitary transformations on the Hilbert space. Measurement collapses the quantum state onto one of the basis vectors, $|0\rangle$ or $|1\rangle$, probabilistically based on the squared magnitudes of the amplitudes.

For a system of n qubits, the state resides in a 2^n-dimensional Hilbert space $\mathcal{H}_{2^n}$, formed by the tensor product of single-qubit spaces (Equation 2.5):

$$\mathcal{H}_{2^n} = \underbrace{\mathcal{H}_2 \otimes \mathcal{H}_2 \otimes \cdots \otimes \mathcal{H}_2}_{n \text{ times}} \tag{2.5}$$

This mathematical structure enables quantum entanglement and interference, distinguishing quantum computing from classical computing.

2.1.2. Qubits

Qubits are the fundamental units of quantum information. While classical bits can only be in one of two possible states (0 or 1), a qubit can exist in a superposition of these basis states, as described in Equation 2.3. This ability to occupy multiple states simultaneously underpins quantum phenomena such as entanglement and interference, leading to computational advantages over classical systems [9]. A distinction can be made between *virtual qubits*, which are used in quantum algorithms and theoretical models, and *physical qubits*, which refer to the actual qubits in a quantum computer. The implementation of qubits differs across quantum technologies, such as superconducting qubits [10], trapped ions [11], [12], and topological qubits [13], [14].

2.1.3. Qubit Measurement

Measuring a qubit collapses its superposition into one of the classical states. The probability of obtaining $|0\rangle$ or $|1\rangle$ is given by $|\alpha|^2$ and $|\beta|^2$, respectively [9]. This process is central to extracting classical information from quantum states. Because the measurement outcome is probabilistic, applying specific quantum gates before measurement can change the measurement basis and influence the observed results.

2.1.4. Quantum Gates

Quantum gates are the building blocks of quantum circuits, performing operations on qubits. Formally, a gate is an operator mapping the n-qubit Hilbert space $\mathcal{H}_{2^n}$ to itself, as shown in Equation 2.6. The gate set $\mathcal{G}$ is the collection of all such gates (Equation 2.7), and the indices of the qubits in the n-qubit system that are affected by the gate operation are defined by its domain $\text{dom}(U_G)$ (Equation 2.8).

$$U_G : \mathcal{H}_{2^n} \mapsto \mathcal{H}_{2^n} \tag{2.6}$$

$$\mathcal{G} = \{U_G \mid U_G : \mathcal{H}_{2^n} \mapsto \mathcal{H}_{2^n}\} \tag{2.7}$$

$$\text{dom}(U_G) : \mathcal{G} \mapsto \{\{1\}, \{2\}, \ldots, \{1,2\}, \ldots\} \tag{2.8}$$

Single-qubit gates, such as the X, Y, and Z gates, act on individual qubits, while multi-qubit gates, such as the controlled-not (CNOT) gate, can entangle qubits by applying conditional operations. Entanglement is a key resource in quantum computing that enables the representation of complex, highly correlated states beyond what is possible with classical bits [9].

Single-Qubit Gates

Single-qubit gates are quantum gates that operate on a single qubit. Any arbitrary single-qubit operation can be represented as a rotation on the Bloch sphere (Figure 2.1). The Pauli-X gate, also known as the bit-flip gate, flips the state of a qubit from $|0\rangle$ to $|1\rangle$ and vice versa by introducing a phase change of π around the X-axis of the Bloch sphere (Figure 2.1a). The Pauli-Y and Pauli-Z gates perform similar operations, introducing a phase change of π around the Y and Z axes of the Bloch sphere, respectively. The operation can be also described by a unitary matrix that acts on the qubit's state vector (Equation 2.9). The identity gate I leaves the qubit state unchanged. Another important single-qubit gate is the Hadamard gate, which creates superposition states by rotating the qubit state vector by $\pi/2$ around the axis between the X and Z axes of the Bloch sphere (Figure 2.1b) [9].

(a) Pauli-X gate action on a $|0\rangle$ state, applying a π rotation around the X axis moving the state to $|1\rangle$.

(b) Hadamard gate action on a $|0\rangle$ state, applying a $\pi/2$ rotation around the axis between the X and Z axes moving the state to $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

Figure 2.1: Bloch sphere representation of single-qubit gates. It is a unit sphere that represents the state of a qubit, with the poles corresponding to the $|0\rangle$ and $|1\rangle$ states. Any gate introduces a rotation on the Bloch sphere, changing the state of the qubit.

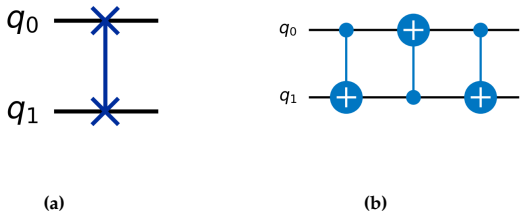
$$U = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\phi+\lambda)}\cos(\theta/2) \end{pmatrix} \quad \text{where} \quad \begin{cases} U & \text{unitary matrix} \\ \theta & \text{rotation angle} \\ \phi & \text{phase angle} \\ \lambda & \text{global phase} \end{cases} \tag{2.9}$$

Controlled Gates

Controlled gates are multi-qubit gates that perform an operation on a target qubit based on the state of a control qubit. The controlled-not (CNOT) or controlled-X (CX) gate is a two-qubit gate that flips the state of the target qubit if the control qubit is in state $|1\rangle$. This conditional operation is essential for creating entanglement between qubits and implementing quantum algorithms. All other variations of controlled gates can be constructed from the CNOT gate by applying single-qubit gates before and after the CNOT gate [9].

SWAP Gate

The SWAP gate is a two-qubit gate that exchanges the quantum states of two qubits. This operation is used in qubit routing to move qubit states between physical qubits. SWAP gates are usually not native to quantum hardware and need to be decomposed into elementary gates, such as CNOT gates, to be implemented on physical qubits. Most superconducting qubit architectures support the implementation of SWAP gates using a sequence of three CNOT gates (Figure 2.2) [9].



(a) **(b)**

Figure 2.2: Decomposition of the SWAP gate into CNOT gates. (a) SWAP gate circuit. (b) SWAP gate decomposition into CNOT gates.

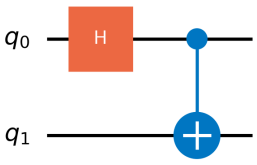


Figure 2.3: Quantum Circuit for Creating the $|\Psi^+\rangle$ Bell state with a Hadamard gate and a CNOT gate

2.1.5. Gate Commutation

Gate commutation is the property of quantum gates that allows them to be rearranged without changing the final state of the system. The mathematical representation of gate commutation between two gates U_i and U_j is given by $[U_i, U_j] = 0$, where $[\cdot, \cdot]$ denotes the commutator (Equation 2.10). When two gates commute, they can be executed in any relative order. The Hadamard gate and CNOT gate are examples of gates that do not commute, leading to different outcomes depending on their relative order [9].

$$[A, B] \equiv AB - BA, \quad \text{so} \quad [A, B] = 0 \quad \Leftrightarrow \quad AB = BA. \tag{2.10}$$

2.1.6. Quantum Circuits

Quantum circuits are visual representations of quantum algorithms that consist of qubits, gates, and measurements. Qubits are represented as lines with gates and measurements as boxes that act on qubits. The final state of a quantum circuit is obtained by applying gates sequentially from left to right, starting from the initial state of the qubits. An example of a quantum circuit which creates the $|\Psi^+\rangle$ state is shown in Figure 2.3. It uses a Hadamard gate to create superposition and a CNOT gate to entangle two qubits such that its final state is:

$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle). \tag{2.11}$$

This state is one of the four Bell states that are maximally entangled and play a crucial role in quantum communication and quantum error correction. The entanglement in the $|\Psi^+\rangle$ state means that the qubits cannot be separated into individual states, and the measurement of one qubit will determine the state of the other qubit.

An example of a state which would not be entangled is a separable state such as:

$$\left|\phi\right\rangle = |0\rangle \otimes |1\rangle = |01\rangle\,. \tag{2.12}$$

In this case, the measurement of one qubit does not provide any information about the state of the other qubit. Unlike entangled states, separable states can be described as a simple tensor product of individual qubit states [9].

2.1.7. Quantum Hardware

The physical implementation of quantum computing hardware introduces constraints and challenges that impact the design and execution of quantum algorithms. Quantum hardware can be characterized in part by its topology and noise, which influence the performance and reliability of quantum computations.

Topology

The topology of quantum hardware refers to the arrangement of qubits and their connectivity. The topology defines the possible interactions between qubits and the constraints on gate operations. Different quantum architectures have distinct topologies, such as linear, 2D grid, or fully connected layouts. The choice of topology affects the efficiency of quantum algorithms and the complexity of qubit routing. Usually the topology is dictated by the physical implementation of the qubits. For example, superconducting qubits are typically arranged in a grid-like structure [15] and trapped-ion qubits in a linear chain [16].

Noise

In practice, the quantum gates introduced in subsection 2.1.4 and the measurement described in subsection 2.1.3 can deviate from their ideal, unitary-based descriptions due to environmental factors, control electronics, and hardware imperfections, collectively called noise. Such deviations introduce errors in quantum operations, which can be quantitatively described using error rates.

For a quantum gate G, ideally represented by a unitary matrix U_G, noise causes the actual operation to deviate from U_G. A common approach to model this deviation is to introduce a *noise channel* $\mathcal{E}$, such that the actual operation applied to a quantum state ρ is:

$$\rho' = \mathcal{E}(U_G \rho U_G^\dagger). \tag{2.13}$$

One standard noise model is the *depolarizing channel*, which assumes that with probability $1 - r_G$, the gate operation is a completely depolarizing channel, and with probability r_G, the gate operation is the ideal unitary operation U_G [9]:

$$\mathcal{E}_{\text{depol}}(\rho) = r_G U_G \rho U_G^\dagger + (1 - r_G)\frac{I}{d}, \tag{2.14}$$

where d is the dimension of the Hilbert space of the system. The parameter r_G is the reliability of the gate G, and $1 - r_G$ is the error rate of the gate.

For many quantum devices, two-qubit gate errors dominate in many quantum devices, and error mitigation strategies often focus on reducing their impact. In this work the focus is on the *two-qubit gate error*, and single-qubit gate errors are assumed to be negligible.

Physical quantum processors exhibit spatial variations in error rates, where certain qubits or specific gate locations may have significantly higher error rates. Additionally, error rates are subject to temporal fluctuations due to changes in system calibration and environmental conditions [17]. These factors make error characterization and mitigation essential for reliable quantum computation.

2.1.8. Quantum Compiling

Quantum compiling is the process of translating a quantum algorithm into a form that can be executed on quantum hardware. One of the key challenges in quantum compiling is the mapping of virtual qubits to physical qubits, taking into account the connectivity constraints of the hardware. This process involves qubit placement and qubit routing to ensure that the quantum circuit can be executed. Other relevant aspects of quantum compiling include gate scheduling, which determines the order of gate operations, and circuit optimization, which reduces the number of gates in a circuit [6].

2.2. Reinforcement Learning

Reinforcement Learning (RL) is a machine learning paradigm in which an *agent* interacts with an *environment* over a sequence of discrete *steps*. At each step, the agent receives a (partial) observation of the environment's state s_t, selects an *action* a_t, and receives a numerical *reward* r_{t+1} along with a transition to the next state s_{t+1}. This sequence of interactions continues until an *episode* ends, which can happen upon reaching a terminal state or after a maximum number of steps. Figure 2.4 illustrates the interaction between an RL agent and its environment.

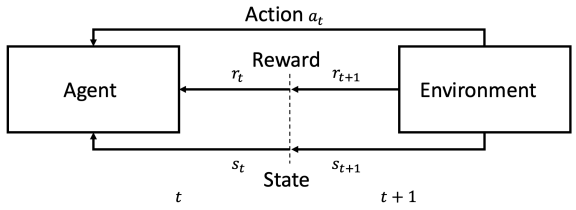


Figure 2.4: The interaction between a Reinforcement Learning (RL) agent and its environment follows a cyclic process. At each time step t, the agent selects an action a_t based on the current state s_t. The environment processes this action, determines the next state s_{t+1}, and provides a reward r_{t+1} to the agent.

2.2.1. Markov Decision Process

An *MDP* provides a formal framework for RL problems. It comprises a set of possible states S, a set of possible actions A, a transition function $P(s' \mid s, a)$ that gives the probability of moving to state s' after taking action a in state s, and a reward function $R(s, a)$ that specifies the immediate reward upon making that transition. The *Markov property* holds that the next state and reward depend only on the current state and action, not on the history of previous states or actions [18].

2.2.2. Training Process

The training process of an RL agent involves iteratively updating its *policy* or *value function* based on the rewards received from the environment. A fundamental aspect of training is balancing *exploration* (trying actions that may yield more information about the environment) and *exploitation* (selecting actions that currently appear to maximize rewards). This interplay helps the agent discover strategies that optimize long-term returns. Once training is complete, the agent can use its learned policy to make effective decisions in real or simulated scenarios [19].

Model-Free vs. Model-Based Learning
When training an RL agent, one distinguishes between *model-free* and *model-based* approaches. Model-free RL learns the policy or value function directly through environmental interactions, making no attempt to build a model of the transition or reward functions. By contrast, model-based RL either learns or has access to an explicit model of the environment's dynamics (via P and R) and can use them to simulate outcomes or plan future actions [19].

Value-Based vs. Policy-Based Decision-Making
In value-based methods, the agent learns a value function—often a *Q-function* $Q(s, a)$—that estimates the expected return of taking action a in state s [19]. The policy is then implicitly derived by selecting the action that maximizes the estimated value. By contrast, policy-based methods directly parameterize $\pi(a \mid s)$ and adjust those parameters to maximize long-term returns, rather than first learning a separate value function. In practice, both value-based and policy-based approaches often leverage deep neural networks to approximate functions and optimize decision-making.

On-Policy vs. Off-Policy Training
Another important distinction is whether the agent learns from the same policy it is currently using (*on-policy*) or from data generated by a different policy (*off-policy*) [19]. On-policy methods, such as Proximal Policy Optimization (PPO) [20], continually update the policy that generates the training experience. Off-policy methods, like Q-Learning [21], can learn from experience generated by a behavior policy that may differ from the current target policy.

2.2.3. Q-Learning
Q-Learning is a classic model-free algorithm where the agent explicitly learns a Q-function $Q(s, a)$. This function represents the expected return of taking action a in state s and thereafter following an optimal policy. The update rule is:

$$Q(s, a) \leftarrow Q(s, a) \; + \; \alpha\Big(R(s, a) + \gamma \max_{a'} Q(s', a') - Q(s, a)\Big), \tag{2.15}$$

where α is the learning rate, γ is the discount factor, and s' is the next state. Over time, Q-learning (QL) converges to optimal values for each state-action pair, from which an optimal policy can be derived by selecting $\arg\max_a Q(s, a)$ [21].

2.2.4. Function Approximation
Modern RL systems often use neural networks to approximate the value function (or the policy in policy-based methods). Let θ denote the parameters of a neural network. In a value-based approach like Deep Q-Network (DQN) [22], the network outputs an estimate $\hat{Q}_\theta(s, a)$ of the Q-function for each state-action pair (s, a). To update θ, one typically minimizes a mean squared error (MSE) loss of the form:

$$L(\theta) = \Big(R(s, a) + \gamma \max_{a'} \hat{Q}_{\theta^-}(s', a') - \hat{Q}_\theta(s, a)\Big)^2, \tag{2.16}$$

where θ^- represents a set of *target network* parameters periodically synchronized with θ, and s' is the next state. The parameters θ are then updated via stochastic gradient descent to minimize $L(\theta)$.

By contrast, policy-based methods parameterize $\pi_\theta(a \mid s)$ and use a gradient *ascent* update to directly maximize the expected return. Here, the objective function $J(\theta)$ typically measures the long-term reward under policy π_θ, and gradients are computed using backpropagation through the network:

$$\nabla_\theta J(\theta) = \mathbb{E}_{(s,a)\sim\pi_\theta}\big[\nabla_\theta \log \pi_\theta(a \mid s)\, G_t\big], \tag{2.17}$$

where G_t represents an estimator of the future returns [23]. Thus, whether in value-based or policy-based RL, neural networks are trained using gradient-based optimization procedures that leverage large-scale data collection from the environment.

2.2.5. Policy Optimization
Policy optimization aims to directly optimize a parameterized policy by adjusting its parameters in the direction of the performance gradient. These methods, such as REINFORCE [24] and actor-critic

algorithms [25], leverage gradient-based updates to maximize expected cumulative rewards. Policy optimization is particularly effective in high-dimensional action spaces and continuous control tasks, where explicit value function estimation may be challenging.

2.2.6. Proximal Policy Optimization

PPO [20] is a popular policy optimization method that improves stability and sample efficiency by employing a *clipped surrogate objective*, which constrains large deviations from the previous policy update. It is a model-free, on-policy algorithm that adjusts policy parameters to maximize expected rewards while balancing stability and exploration. PPO typically uses a mini-*batch* approach with stochastic gradient ascent, where each iteration involves multiple epochs of optimization using data sampled from a replay buffer. The update step minimizes the clipped loss function, ensuring balanced exploration and exploitation while maintaining stable learning dynamics. This makes PPO particularly effective for continuous control tasks and other complex RL problems.

2.2.7. Hierarchical Reinforcement Learning

To enable more efficient learning and decision-making, Hierarchical Reinforcement Learning (HRL) organizes actions into a hierarchy of subtasks [26]. In one form of such a framework, higher-level actions correspond to decisions that trigger predefined policies, which execute sequences of sub-actions to complete specific tasks. By delegating low-level decision-making to these predefined policies, the agent can operate at a higher level of abstraction, focusing on strategic choices rather than finer execution details. This approach leverages the structure of the environment to improve learning efficiency and performance in complex decision-making tasks.

2.2.8. Action Masking

Action Masking is a technique used in RL to handle environments where certain actions are invalid or undesired in particular states [27], [28], [29]. By dynamically excluding these invalid actions, the agent focuses solely on optimizing the valid ones, improving learning efficiency, stability, and performance in complex action spaces. However, this approach requires environment-specific knowledge to define the masks accurately, and errors in masking can lead to suboptimal policies or performance.

3

Literature review

3.1. Related Work

In this section, a survey of relevant existing approaches to the qubit routing problem is made by grouping them on their optimization method with noise-aware methods discussed separately.

3.1.1. Rule-based

A fixed routing scheme [30] focused on reducing communication overhead in two-dimensional architectures through deterministic swaps. Although this rule-based approach simplifies implementation, it often leads to suboptimal mappings and ignores hardware variability.

3.1.2. Mathematical Optimization

Early exact solutions with Pseudo-boolean optimization (PBO) were proposed to achieve optimal qubit placements in linear [31] and multidimensional [32] architectures. Despite yielding optimal solutions, PBO-based methods become computationally expensive for large circuits and did not incorporate error rates. Integer linear programming (ILP) [33] is also employed to find optimal qubit permutations on specific devices, but it shares scalability limitations. Quadratic unconstrained binary optimization (QUBO) [34] extends the search space to modular architectures and can leverage specialized hardware, including Ising machines [35], for faster solutions. However, these methods ignore noise and hardware variability leading to suboptimal mappings.

3.1.3. Heuristic

Dynamic programming (DP) combined with heuristic routines [36] addresses routing through reversal, swap, or bridge operations but quickly becomes intractable for large devices. Purely heuristic strategies [6] focus on circuit depth optimization and are integrated into compilers like `t|ket⟩`, offering speed at the cost of ignoring hardware noise. Further heuristic-based mapping [37] introduces scheduling and latency minimization, yet still does not incorporate evolving device-specific constraints. In general, heuristic methods improve scalability over exact optimization but often rely on static or simplified assumptions about hardware.

3.1.4. Machine Learning

Reinforcement learning (RL) [38], [39] formulates qubit routing as a combinatorial decision process and learns a policy to optimize certain cost metrics (e.g., circuit depth). Although RL approaches can adapt to complex circuit topologies, most assume idealized hardware or uniform costs for gates and swaps. Graph neural networks (GNNs) combined with Monte Carlo Tree Search [7] approximate value functions and policies, indicating the promise of ML for complex routing scenarios. Nonetheless, dynamic hardware variation is rarely incorporated, and scalability to devices with many qubits remains an open problem.

3.1.5. Noise-aware Methods

Satisfiability modulo theory (SMT)-based methods coupled with greedy heuristics [40] take static device-specific noise data into account, while purely heuristic strategies are used in SABRE [5], one of Qiskits [41] routing strategies. Variable error rates are incorporated into A* search [17] and ILP formulations [42] handle multi-constraint settings under noisy conditions but still face scalability issues. Recently, Graph Neural Network (GNN)s are used to estimate fidelity with RL [43], demonstrating an integrated noise-aware ML approach up to moderate system sizes. It is unclear how these methods scale to larger devices or more complex noise models. Variation-aware mapping based on multi-agent search [44] and deep RL strategies for noise-adaptive routing [8] further expand the suite of noise-aware techniques. However, these methods often operate on smaller benchmarks or rely on simplified noise models, limiting their applicability to larger real-world quantum processors.

3.2. Research Gap

Despite the growing body of work on quantum circuit routing—from exact formulations and heuristics to ML–based strategies—several gaps remain unaddressed, particularly those relevant to the design of RL environments, the treatment of hardware variability, and scalability challenges.

Environment Design Choices Current qubit routing strategies based on RL often use fixed custom environment setups—without variations in state representation, reward design, and action spaces—making it difficult to isolate which aspects directly impact performance. This lack of understanding hinders the development of generalizable RL solutions that can adapt to different quantum hardware configurations.

Noise-awareness Although some hardware-aware approaches account for error rates or calibration data, most RL research still relies on static or simplified assumptions. Consequently, there is a limited understanding of how dynamic or uncertain hardware parameters (e.g., fluctuating gate fidelities, varying connectivity) can be robustly incorporated into the RL environment. This gap is especially relevant for Noisy Intermediate Scale Quantum (NISQ) devices, where noise and error rates can vary significantly over time.

Scalability While RL-based qubit routing solutions have demonstrated encouraging results on small- or mid-scale devices, scaling these methods to larger hardware remains problematic. On the one hand, noise-aware approaches tend to be tested on relatively small benchmarks due to the computational overhead of modeling device-specific or time-varying error rates, thus limiting their applicability to bigger, more complex systems. Yet, more scalable techniques often rely on simplifying assumptions about hardware. Overcoming these competing demands requires novel RL frameworks that effectively account for hardware imperfections at scale—enabling both realistic noise modeling and practical scalability on larger quantum processors.

Until methods are explored for building and evaluating RL environments, while also incorporating real hardware awareness, and designing for scalability, the full potential of RL-based qubit routing may remain out of reach. As quantum devices become larger, addressing these gaps will be crucial for developing robust routing solutions.

4

Methodology

This chapter outlines the methodology used to investigate the effects of variations in reinforcement learning environments and quantum hardware for qubit routing. First, the definitions of the quantum hardware, circuits, and placement are introduced. Subsequently, the metrics used to evaluate the performance of routing strategies are discussed. The problem formulation is presented, followed by the environment design and data generation. Finally, the evaluation and implementation details are described.

4.1. Definitions

4.1.1. Quantum Hardware

The quantum hardware is modeled as a coupling graph H, where the vertices P represent the set of physical qubits, and the edges E the physical edges as shown in Figure 4.1. Each edge $e_i = p_a, p_b \in E$ represents a bidirectional physical connection between two physical qubits p_a and p_b. The reliability of successfully executing a two-qubit gate on this edge e in either direction is denoted by r_e, where $r_e \in [0, 1]$ is a value between 0 and 1, with 1 indicating perfect reliability and 0 indicating guaranteed failure. Single-qubit gates are assumed to have a reliability of 1. Edges are assumed to have a bidirectional connection and symmetric reliability, i.e., $r_{e_{ab}} = r_{e_{ba}}$. The formal definition of the quantum hardware is given in Equation 4.1.

$$H = (P, E), P = \{p_i\}, E \subseteq P \times P, r_e \in [0, 1] \tag{4.1}$$

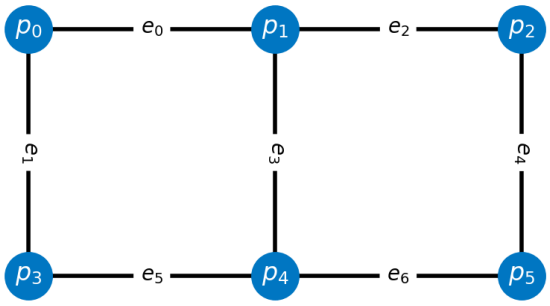


Figure 4.1: Example of a coupling graph $H = (P, E)$ of the quantum hardware with a 2×3 grid of physical qubits.

4.1.2. Circuit

A quantum circuit C with n gates is represented by an ordered sequence of gates from the gate set $\mathcal{G}$ in the order $\prec_C$:

$$C = (G_0, \ldots, G_{n-1}) = (G_i \in \mathcal{G}, \prec_C), \tag{4.2}$$

The set of qubits $Q(G)$ involved in a gate G is defined as:

$$Q(G_i) = \{q_i \mid i \in \mathrm{dom}(U_G)\}, \tag{4.3}$$

Figure 4.2b shows an example of a quantum circuit.

Gate Dependency Graph
The gate dependency graph D_G of a circuit C is a directed acyclic graph, an example of which is shown in Figure 4.2c. The vertices V_D are the gates in the circuit C, and the edges E_D denote the dependencies between gates:

$$D_G = (V_D, E_D), V_D = \{G_i | G_i \in C\}, E_D \subseteq V_D \times V_D \tag{4.4}$$

This dependency relation is described by a partial ordering $\preceq_{D_G}$, where gate G_i precedes gate G_j if G_i occurs earlier in the circuit C and their operations do not commute (Equation 4.5). An edge exists from gate G_i to gate G_j if the former precedes the latter according to the partial ordering $\prec_{D_G}$, with no intermediate gate G_k in circuit C that should be placed between them:

$$G_i \prec_{D_G} G_j \Rightarrow G_i \prec_C G_j \wedge [U_{G_i}, U_{G_j}] \neq 0 \tag{4.5}$$

An edge from gate G_i to gate G_j exists if G_i comes before G_j according to $\prec_{D_G}$, and no intermediate gate in the circuit should be placed between them:

$$G_i \overset{D_G}{\rightarrow} G_j \Leftrightarrow G_i \prec_{D_G} G_j \wedge \nexists G_k \in C, (G_i \prec_{D_G} G_k \prec_{D_G} G_j) \tag{4.6}$$

Interaction Dependency Graph
The interaction dependency graph D_I is constructed by preserving only vertices V_I that are gates with two-qubit interactions (Equation 4.7). The dependencies among gates are maintained by preserving each path $\underset{*}{\rightarrow}$ from the dependency graph D_G to the interaction dependency graph D_I using edge contractions. These contractions ensure that if a path exists between two gates in the dependency graph D_G, the same path exists in the interaction dependency graph D_I (Equation 4.8). Figure 4.2d shows an example of an interaction dependency graph.

$$D_I = (V_I, E_I), V_I = \{G_i | G_i \in V_D, |\mathrm{dom}(G_i)| = 2\}, E_I \subseteq V_I \times V_I \tag{4.7}$$

$$G_i \underset{*}{\overset{D_G}{\rightarrow}} G_j \Leftrightarrow G_i \underset{*}{\overset{D_I}{\rightarrow}} G_j \tag{4.8}$$

Interaction Circuit
Interactions are two-qubit gates that are executed on the quantum hardware. A valid execution order of interaction dependency graph D_I can be described by the interaction circuit I. It is a sequence of interactions representing a topological traversal of the interaction dependency graph D_I. This order is determined by the partial ordering $\prec_{D_I}$ (Equation 4.9). In Figure 4.2d, the interaction circuit is $I = (G_0, G_1, G_2)$ or $I = (G_0, G_2, G_1)$.

$$I = (G_i \in D_I, \prec_{D_I}) \tag{4.9}$$

Front Layer
The front layer L_t is the set of gates in the gate dependency graph D_G scheduled after timestep t with no dependencies on other gates after timestep t:

$$L_t = \{G_i \in C \mid \forall G_j \in C, (G_j \preceq_{D_G} G_t \wedge j \geq t)\} \tag{4.10}$$

Figure 4.2e shows an example of a front layer.

4.1.3. Placement
Qubit placement is defined as a bijective function π, associating each virtual qubit q uniquely to a physical qubit p:

$$\pi : Q \mapsto P, \quad \pi(q_i) = p_j \tag{4.11}$$

This bijectivity ensures that the placement is both injective, ensuring each virtual qubit is mapped to a unique physical qubit (Equation 4.12), and surjective, ensuring that every physical qubit is assigned a virtual qubit Equation 4.13.

$$\forall q_i, q_j \in Q, \quad \pi(q_i) = \pi(q_j) \Rightarrow q_i = q_j \tag{4.12}$$

$$\forall p_i \in P, \quad \exists q_j \in Q, \quad \pi(q_j) = p_i \tag{4.13}$$

The placement function $\mathcal{P}$ uses this placement π to convert the virtual qubits of a gate to their corresponding physical qubits:

$$\mathcal{P}(\pi, G_i) : \mathcal{G} \mapsto \mathcal{G}, \quad \mathcal{P}(\pi, G_i) = G_i(\pi(q_0), \ldots, \pi(q_k)) \tag{4.14}$$

Figure 4.2f shows an example of a placement.

4.1.4. Swap gates
To update the placement, swap gates S can be inserted in a circuit:

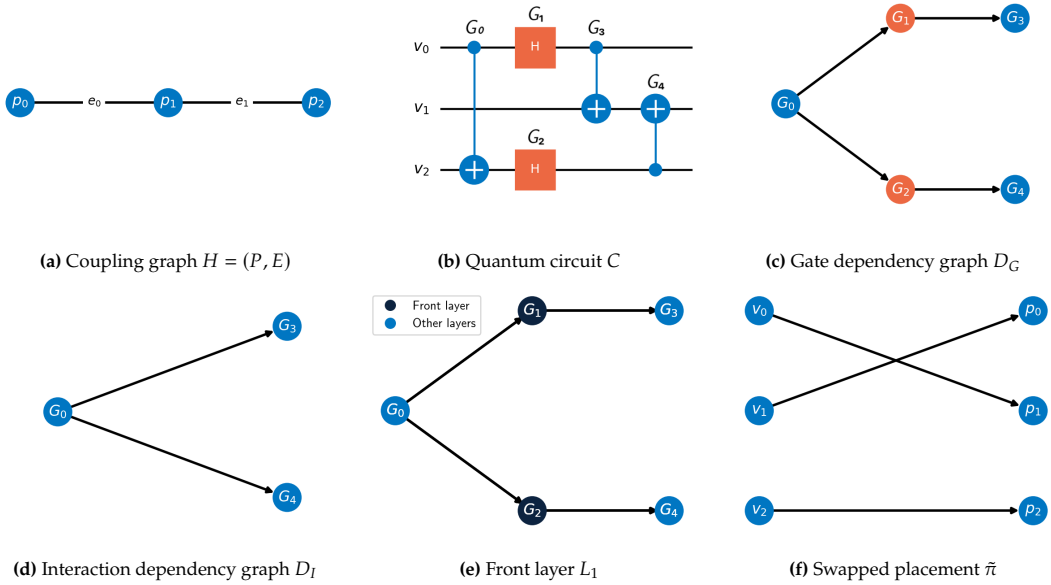
$$S = \{\sigma_e \mid e \in E\} \tag{4.15}$$

Each swap σ_e swaps the states of the physically connected qubits p_a and p_b on edge e. This operation updates the placement from π to $\tilde{\pi}$ where state of p_a is moved to p_b and vice versa, while the states of the other qubits remain unchanged:

$$\begin{aligned} \sigma_e : \pi \rightsquigarrow \tilde{\pi}, \quad & \forall e = \{p_a, p_b\} \in E, \\ \tilde{\pi}(p_a) = \pi(p_b), \quad & \tilde{\pi}(p_b) = \pi(p_a), \\ \tilde{\pi}(p) = \pi(p), \quad & \forall p \in P \setminus \{p_a, p_b\}. \end{aligned} \tag{4.16}$$

Figure 4.2f shows an example of a trivial placement after applying a swap gate.

4.2. Metrics
To evaluate and compare circuits and routing strategies, several metrics are used. This section expands on the key metrics: *circuit reliability*, *circuit depth*, *interaction count*, *interaction reliability*, and *routing time*. Depending on the use case, some or all metrics may be employed to assess the quality of a solution.



(a) Coupling graph $H = (P, E)$ **(b)** Quantum circuit C **(c)** Gate dependency graph D_G

(d) Interaction dependency graph D_I **(e)** Front layer L_1 **(f)** Swapped placement $\tilde{\pi}$

Figure 4.2: Example of a quantum circuit and its corresponding graphs. (a) Coupling graph $H = (P, E)$ of the quantum hardware. (b) Quantum circuit C with gates: G_0, G_1, G_2, G_3, G_4. (c) Gate dependency graph D_G for circuit C with nodes the gates from C and the edges the dependencies between them. (d) Interaction dependency graph of D_G leaving only gates with two-qubit interactions. (e) Front layer of D_G at $t = 1$. Only gates G_1 and G_2 are in the front layer as they have no dependencies on gates after $t = 1$. The only gate they depend on is G_0 which comes before $t = 1$. (f) Trivial placement with a swap σ_{e_0} applied on edge e_0 (p_0, p_1) resulting in $\tilde{\pi} = \{v_0 \mapsto p_1, v_1 \mapsto p_0, v_2 \mapsto p_2\}$.

4.2.1. Circuit Reliability

Definition. The overall reliability of a circuit estimates the probability that the circuit executes successfully on a given quantum hardware H. As outlined in Equation 4.17, the reliability of a circuit is calculated as the product of the reliabilities associated with each *two-qubit interaction*. Since single-qubit gates are assumed to have perfect fidelity (i.e., reliability of 1), they do not affect the product.

$$R(C) = \prod_{e \in I(C)} r_e \tag{4.17}$$

Interpretation. A higher $R(\tilde{C})$ implies a higher probability that the mapped circuit completes without error (under the simplified assumption that two-qubit gates are the primary source of errors). If multiple ways exist to map the same circuit, the mapping with the highest $R(C)$ is usually preferred because it maximizes the chance of successful execution.

4.2.2. Circuit Depth

Definition. The *circuit depth* D is the minimum number of sequential time steps required to execute the circuit $\tilde{C}$, assuming as much parallelism as allowed by the gate dependencies and the hardware constraints. Formally, it is the length of the *longest path* in the gate dependency graph once the scheduling and placement constraints are accounted for:

$$D(\tilde{C}) = \max_{(G_i, G_j) \in \tilde{C}} \text{dist}(G_i, G_j), \tag{4.18}$$

where $\text{dist}(G_i, G_j)$ is the number of edges along the longest directed path from gate G_i to G_j in the a dependency graph.

Interpretation. Circuit depth measures how many time steps it takes to finish running all gates under the best possible parallelization. A smaller depth generally means faster execution, which can be crucial

in mitigating decoherence. In architectures where the coherence time is limited, achieving a low circuit depth may be as important as maximizing the reliability.

4.2.3. Interaction Count

Definition. The interaction count refers to the total number of two-qubit gates present in a circuit. Mathematically, it is $|I(C)|$, the cardinality of the set of two-qubit interactions in the circuit.

Interpretation. Adding more two-qubit interactions increases the risk of error because two-qubit gates are typically noisier than single-qubit operations. A lower interaction count is often beneficial for both reliability and execution speed.

4.2.4. Interaction Reliability

Definition. Introducing the *interaction reliability*, denoted $\bar{R}(C)$, as a derived measure that quantifies the effective per-interaction reliability of a circuit. Given that $R(C)$ is computed as the product of individual reliabilities, $\bar{R}(C)$ is defined as:

$$\bar{R}(C) \;=\; R(C)^{\frac{1}{|I(C)|}}, \tag{4.19}$$

where $|I(C)|$ represents the total number of two-qubit interactions in the circuit.

Interpretation. This measure provides insight into the reliability of individual interactions within the circuit. It represents the effective per-interaction reliability, assuming that all two-qubit interactions contribute uniformly. A higher or lower $\bar{R}(C)$ compared to the hardware average $\bar{r_e}$ suggests that the circuit preferentially uses more or fewer reliable interactions respectively.

4.2.5. Routing Time

Definition. The *routing time* measures how long it takes to compute a valid mapping (and possibly a scheduling) for a given circuit C onto the hardware H. It is typically reported as an *average* (or total) CPU or wall-clock time over multiple benchmark circuits.

Interpretation. Routing time is an indicator of how *scalable* or *efficient* the mapping algorithm is in practice. In scenarios where many circuits need to be routed rapidly (e.g., in a compiler pipeline or a real-time quantum-control setting), a fast routing algorithm may be necessary even if it occasionally sacrifices some reliability or circuit depth.

Overall, no single metric comprehensively captures "best" performance. Depending on hardware constraints, user priorities, and the nature of the quantum algorithm, one may prioritize the highest circuit reliability, the lowest circuit depth, the fewest interactions, or a balance among all criteria. Thus, a thorough evaluation often involves examining multiple metrics in tandem to gain a complete understanding of the trade-offs offered by different routing or scheduling solutions.

4.3. Problem Formulation

The mapping problem is formulated as transforming a quantum circuit C into an alternative representation, the mapped circuit $\tilde{C}$. This representation can be executed on a given quantum hardware H by adhering to the hardware's specific constraints while maximizing the circuit reliability of the mapped circuit. It is formulated as two subproblems: the routing problem and the scheduling problem. The solution to these subproblems is the set of swap gates $\mathcal{R}$ and the scheduling function $\mathcal{S}$, respectively that can be applied to the circuit C to obtain the mapped circuit $\tilde{C}$. The initial placement π_0 is assumed to be given.

4.3.1. Routing Problem

Given an initial placement π_0, the circuit C should be updated with swap gates to ensure each gate G_t is executable (Equation 4.25) under the placement π_t at every time-step t. Whenever gate G_t is

not executable under π_t, a sequence of swap gates s_t should be inserted before gate G_t to update the placement from π_t to π_{t+1}:

$$\pi_t \overset{\sigma_{t,0}}{\rightsquigarrow} \dots \overset{\sigma_{t,j}}{\rightsquigarrow} \dots \overset{\sigma_{t,m-1}}{\rightsquigarrow} \pi_{t+1}, \quad \text{with } \{\sigma_{t,0}, \dots, \sigma_{t,m-1}\} = s_t, \quad \forall t \in \{0, \dots, n-1\}. \tag{4.20}$$

The solution $\mathcal{R}$ should define a sequence of swap gates for every gate G_t:

$$\mathcal{R} = (s_0, \dots, s_{m-1}), \quad s_i = (\sigma_i, j \in S) \tag{4.21}$$

4.3.2. Scheduling Problem

A second goal is to determine the execution order of the gates in the circuit. This order is determined by a sequence of gates $\mathcal{S}$ that schedules the gates in the circuit C. The scheduling function $\mathcal{S}$ is a valid permutation of the gates in C following the dependencies in the gate dependency graph D_G:

$$\mathcal{S} = (G_i \in C, \prec_{D_G}) \tag{4.22}$$

4.3.3. Mapped Circuit

The mapped circuit $\tilde{C}$ is constructed iteratively by starting with an empty circuit $\tilde{C}_0$ and, at each iteration t, appending the swap gates s_t, updating the placement to π_{t+1}, and then adding the gate G_t with the placement π_{t+1} applied to its qubits:

$$\begin{aligned} \tilde{C}_0 &= \varnothing \\ \tilde{C}_{t+1} &= \tilde{C}_t s_t \mathcal{P}(\pi_{t+1}, \mathcal{S}_t), \quad \forall t \in \{0, \dots, n-1\} \end{aligned} \tag{4.23}$$

Figure 4.3 shows an example of the quantum circuit routing process.

4.4. Environment

The environment is defined by its state, initialization, state transitions, actions, observations, rewards, and termination conditions.

4.4.1. State

The state of the qubit routing environment is defined as in Table 4.1. It is updated according to the state transitions and actions taken by the agent.

Table 4.1: State elements in the qubit routing environment.

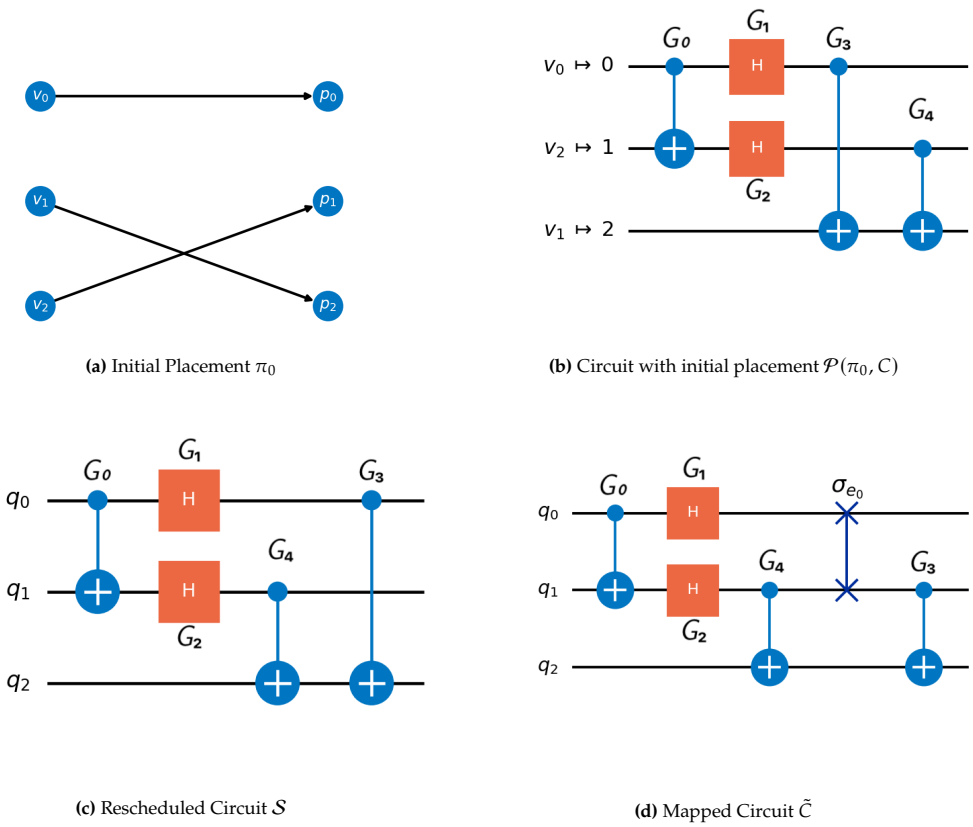
Element	Value	Update Interval
Hardware	H	Episode
Circuit	C	Episode
Position	t	Timestep
Placement	π_t	Timestep
Inserted Swap Gates	$\mathcal{R}$	Timestep
Execution Order	$\mathcal{S}$	Timestep

4.4.2. Initialization

The hardware H and circuit C are initialized at the start of each episode and stay constant throughout the episode. Additionally, the only change between episodes for the hardware is the edge reliabilities r_e. The initial position t is reset to 0 at the start of each episode. The initial placement is given by π_0 and can be either random or predefined. The inserted swap gates $\mathcal{R}$ and the execution order $\mathcal{S}$ are initialized as empty lists.

4.4.3. Actions

Two versions of actions can be configured for the environment: primitive or hierarchical actions. These two versions of the environment will be called *primitive* or just *RL*, and *hierarchical* or *RL (hierarchical)* respectively. The size of both action spaces is equal to the number of edges in the coupling graph $|E|$.



(a) Initial Placement π_0

(b) Circuit with initial placement $\mathcal{P}(\pi_0, C)$

(c) Rescheduled Circuit $\mathcal{S}$

(d) Mapped Circuit $\tilde{C}$

Figure 4.3: Example of the quantum circuit mapping process onto a quantum hardware H (Figure 4.2a) for a quantum circuit C (Figure 4.2b). (a) Initial placement with swapped p_1 and p_2 resulting in $\pi_0 = \{v_0 \mapsto p_0, v_1 \mapsto p_1, v_2 \mapsto p_2\}$. (b) Quantum circuit to be mapped onto hardware H with initial placement π_0 applied resulting in $\mathcal{P}(\pi_0, C)$. (c) Rescheduled circuit for direct execution of the second CNOT gate with gates G_3 and G_4 reordered: $\mathcal{S} = \{G_0, G_1, G_2, G_4, G_3\}$. (d) Mapped circuit with a swap gate for execution of the final CNOT gate. One swap is inserted before G_4 to update the placement to $\pi_4 = \{v_0 \mapsto p_0, v_1 \mapsto p_2, v_2 \mapsto p_1\}$. The final routing solution $\mathcal{R} = \{\emptyset, \emptyset, \emptyset, \emptyset, s_4\}$, with $s_4 = \{\sigma_{e_0}\}$.

Primitive Actions

A primitive consists of choosing a physical edge e from the coupling graph. An action mask is used to prevent the agent from selecting edges that do not have at least one qubit involved in the next interaction.

Hierarchical Actions

Hierarchical actions entail selecting a physical edge e from the coupling graph. An underlying policy then computes an optimal action sequence s_t that makes gate C_t executable under the current placement. This policy uses Dijkstra's shortest path algorithm [45] with edge weights $\log(1 - r_e)$ to relocate the virtual qubits of the next interaction to the chosen edge according to Algorithm 1. The state is then updated by appending the swap gates to the list of inserted swap gates and adjusting the placement accordingly. This action corresponds to selecting an interaction edge from the coupling graph and ensuring the next interaction is executed on the chosen edge.

4.4.4. State Transitions

Both explicit and implicit state transitions are possible in the environment. The agent's action triggers the explicit transitions, while the implicit transitions are triggered by the environment's state.

Implicit Transitions

The executable front layer $\tilde{L}_t$ is the set of gates in the front layer that are executable under the current placement π_t. This edge should connect physical qubits Q involved in the gate G_t placed following

placement π_t:

$$\tilde{L}_t = \{G_i \in L_t \mid \text{executable}(G_i, \pi_t)\} \tag{4.24}$$

Gate G_t is considered executable if it involves only one qubit, otherwise the corresponding physical qubits have to be connected in the coupling graph H by an edge $e \in E$:

$$\text{executable}(G_t, \pi_t) \Leftrightarrow |Q(G_t)| = 1 \vee \mathcal{P}(\pi_t, G_t) \in E \tag{4.25}$$

When the executable front layer $\tilde{L}_t$ of the circuit C is not empty, the environment executes the executable gates and transitions to the next state. The execution order is changed by appending the executable gate, and the position is incremented by one. This corresponds to the execution of a gate in the circuit.

$$\tilde{L}_t \neq \varnothing \Rightarrow t \mapsto t+1 \Rightarrow r_e$$

Explicit Transitions
If no implicit transitions occur, the agent selects an action from the action space. Depending on the chosen action, the environment transitions to the next state by applying the action to the current state. The state is then updated by appending the swap to the list of inserted swap gates and adjusting the placement accordingly. This action corresponds to inserting a swap gate into the circuit.

4.4.5. Observations

Multiple variations of observations can be configured in the environment. For each observation a predefined lookahead ℓ can be set to vary the amount of information available to the agent. The interaction circuit I is used to determine the upcoming interactions and is defined as the interaction circuit derived from the gate dependency graph D_G without the interactions that already executed in the execution order $\mathcal{S}$. Their description, value, and number of elements, as the number of integer values, are summarized in Table 4.2.

Table 4.2: Possible observations.

Observation Value	**Elements** (#int)
Interactions & placement $I_{t+1,\dots,t+\ell}$ & π_t	$2\ell + \lvert P\rvert$
Placed interactions $\pi_t(I_{t+1,\dots,t+\ell})$	2ℓ
Interaction matrix $M(C)$	$\ell \times \lvert P\rvert$

Interactions & Placement
The upcoming interactions and the current placement are observed. This includes the next ℓ interactions in the interaction circuit and the current placement π_t. The size of the observation space is the number of interactions in the interaction circuit $|V_I|$ plus the number of physical qubits $|P|$.

Placed Interactions
The *placed interactions* observation is the next ℓ interactions of the interaction circuit I with the placement applied to the virtual qubits. Each interaction is represented by the physical qubits they are mapped to. The size is equal to ℓ times two as each interaction involves two qubits. It differs from the interaction gates & placement observation by combining the interactions with the placement.

Interaction Matrix
The interaction matrix $M(C)$ is a representation of the next ℓ interactions in the interaction circuit with the placement applied to the virtual qubits. The matrix is constructed by iterating over the interaction circuit, applying the placement and setting the corresponding physical qubits to 1 in the matrix. The other elements are set to 0. Each row in the matrix represents an interaction gate, and each column represents a qubit. It differs from the placed interactions observation by including all physical qubits in the matrix, not just the ones involved in the interactions.

4.4.6. Rewards
The reward function is defined as the circuit reliability as defined in section 4.2. It is given to the agent at the end of each episode.

4.4.7. Termination
The episode terminates when the position t reaches the end of the circuit $t = |C_I|$.

4.5. Data
For training and evaluation, random quantum circuits and hardware configurations are generated.

4.5.1. Circuits
Random quantum circuits are generated by selecting gates randomly from a predefined set of operations, such as single- and multi-qubit gates, with the number of qubits involved in each gate being flexible. The circuit is constructed by adding gates layer by layer, with constraints such as the total number of gates, depth, and operands per gate. A random number generator is used to ensure reproducibility.

4.5.2. Noise
The edge reliabilities in the coupling graph are generated randomly using a shifted and scaled beta distribution. This distribution follows the probability density function of the standardized beta distribution, as defined by:

$$f(x \mid \alpha, \beta) = \frac{\Gamma(\alpha+\beta)x^{\alpha-1}(1-x)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)}, \quad x \in [0,1], \quad \alpha, \beta \in \mathbb{R}^+ \tag{4.26}$$

where Γ is the gamma function, and α and β are the shape parameters of the beta distribution:

$$\Gamma(z) = \int_0^\infty t^{z-1}e^{-t}\,dt, \quad \Re(z) > 0 \tag{4.27}$$

To shift and scale the beta distribution from $B(\alpha, \beta)$ to $B'(\alpha, \beta, \mu, \sigma)$, the following transformation is applied:

$$f(x \mid \alpha, \beta, \mu, \sigma) = \frac{1}{\sigma} f(y \mid \alpha, \beta), \quad y = \frac{x-\mu}{\sigma} \tag{4.28}$$

Reliabilities are derived from the error rates of IBM's native quantum gates, specifically those of the IBM Kyiv quantum processor. A beta distribution $\hat{B}'(\alpha, \beta, \mu, \sigma)$ is fitted to the measured error rates.

Since a CNOT gate consists of multiple native gates, its overall reliability depends on the reliability of its decomposed components. The individual single and two-qubit gate error rates are used to calculate the error rate of the CNOT gate.

To model gate reliability variations statistically, the measured error rates are first used to fit a shifted and scaled beta distribution. Given that reliability is defined as $r = 1 - \varepsilon$, the corresponding reliability distribution is obtained by transforming the fitted beta distribution. This transformed distribution can then be used to sample edge reliabilities for the coupling graphs.

Figure 4.4 shows a comparison of the actual data and the fitted data, demonstrating how well the model captures the characteristics of the gate error rates.

4.6. Evaluation
4.6.1. Baseline
Two baselines are used to evaluate the performance of the reinforcement learning model.

Random. The RL router selects actions randomly for both primitive and hierarchical actions. This approach serves as a reference to determine whether the agent is learning a more effective strategy compared to random selection.

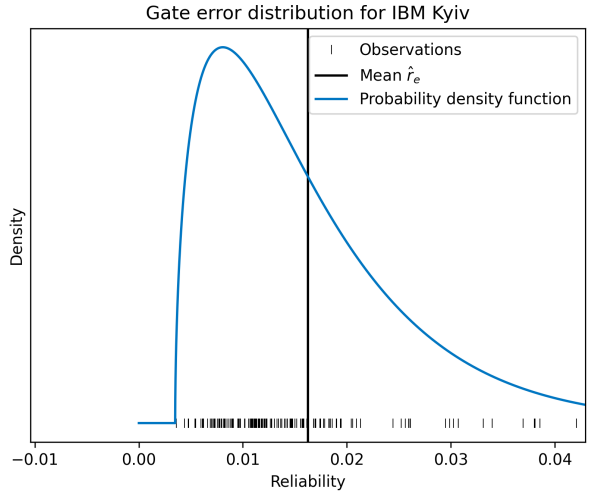


Figure 4.4: CNOT error rates and the fitted beta distribution: $\varepsilon_{\text{CNOT}} \sim \hat{B}'(\alpha = 1.67, \beta = 3.25 \times 10^{10}, \text{loc} = 3.17 \times 10^{-3}, \text{scale} = 2.32 \times 10^{8})$. The dashed vertical line represents the mean error rate of the fitted distribution: $\bar{\varepsilon}_{\text{CNOT}} = 1.514 \times 10^{-2}$. The corresponding mean reliability is $\bar{r}_{\text{CNOT}} = 1 - 1.514 \times 10^{-2} = 0.98486$.

Qiskit SWAP. The second baseline represents the optimal combination of initial placement and SWAP-based routing methods from the Qiskit library [46]. This strategy is chosen over other Qiskit routing methods because it provides a fair comparison by relying solely on SWAP gates for routing, without incorporating additional techniques. The initial placement strategies considered are `TrivialLayout`, `DenseLayout`, and `SabreLayout`. The routing methods include `BasicSwap`, `SabreSwap`, and `StochasticSwap`. The best combination is selected based on the highest circuit reliability produced for each circuit.

4.6.2. Inference

Trained models can route circuits by predicting actions for each state until the episode concludes. To enhance the reliability of the routing solution, it's common practice in reinforcement learning to run multiple trials per circuit and select the best outcome. Additionally, setting a maximum number of steps ensures that the model doesn't run indefinitely.

4.6.3. Validation

To validate the results of the trained model, the state vectors of the output circuits are compared to the state vectors of the input circuits. This ensures that the output circuits are equivalent to the input circuits, and the model has learned to map the circuits correctly.

4.7. Implementation

The implementation of the reinforcement learning model is based on the Stable Baselines3 library [47] using the PPO [20] with action masking [28] algorithm and default hyperparameters. The environment for qubit routing is implemented as a variation of the QGym library [48]. Source code is available as a package called *Q-NARR* along with the configuration files for training and evaluation [49].

5

Results

All experiments are divided into two categories: optimization and evaluation. Within each category, experiments are presented in separate subsections followed by their own discussion. For every experiment a comparison is also made between primitive and hierarchical actions referenced as *primitive* or just *RL*, and *hierarchical* or *RL (hierarchical)* respectively. Default parameters in Table 5.1 are used for each experiment unless stated otherwise. The distribution of edge reliability is set to $\hat{B}'$ as described in subsection 4.5.2 with:

$$\hat{B}'(\alpha = 1.67, \beta = 3.25 \times 10^{10}, \text{loc} = 3.17 \times 10^{-3}, \text{scale} = 2.32 \times 10^{8})$$

Optimization experiments focus on finding the best training setup for the reinforcement learning agent (section 5.1).

Evaluation experiments investigate the performance of the trained agent on different hardware configurations including size and topology (section 5.2). Where possible, the metric is normalized relative to Qiskit to provide a relative comparison.

General	Topology	Grid 3x3
	Lookahead	4
	Gate set	CX, CY, CZ
	Edge reliability	$r_e \sim \hat{B}'$
Train	Circuit gates	8
	Time steps	2×10^6
Evaluation	Circuits	256
	Interactions	32
	Qubits	9
	Trials	4

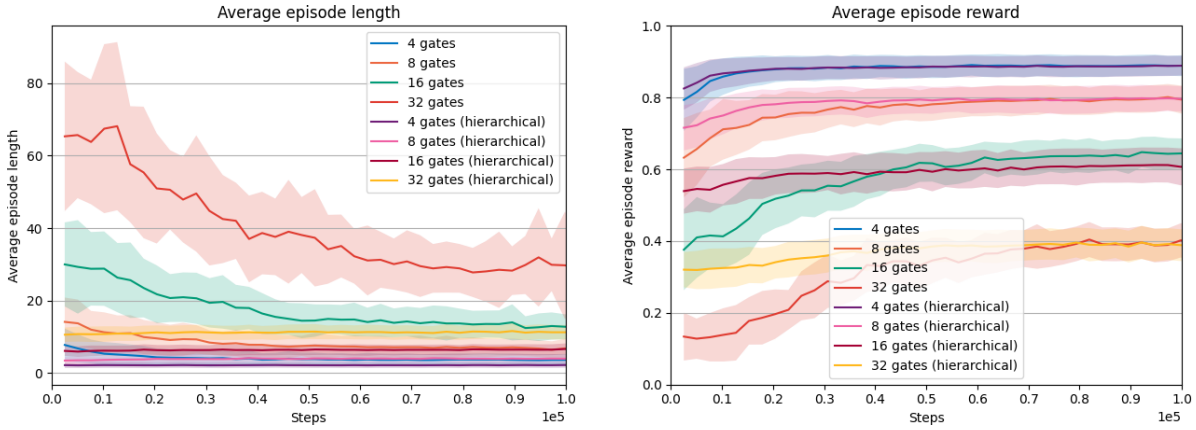
Table 5.1: Default parameter values.

5.1. Optimization

5.1.1. Experiment: Circuit Gate Count for Training

How does the number of gates in the training circuits affect training time and evaluation performance?

The number of gates in the training circuits is set to 4, 8, 16, and 32 gates. From the training data in Figure 5.1, the training time increases with the number of gates, and the primitive routing algorithm takes longer to converge than the hierarchical routing algorithm. The absolute number of steps is higher for the primitive routing algorithm, but the hierarchical inserts more gates per step, so this is expected. Figure 5.2 shows worse performance than Qiskit for both the primitive and hierarchical, with primitive outperforming hierarchical except when only 4 gates are used for training.



(a) **(b)**

Figure 5.1: Training metrics for the number of gates in the training circuits. The shaded area is the standard deviation of every 10 update batches.

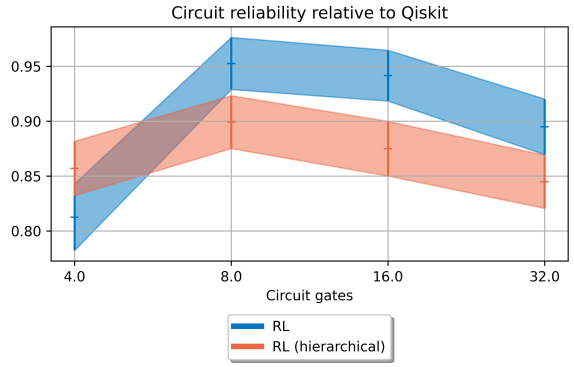


Figure 5.2: Circuit reliability versus the number of gates in the training circuits relative to Qiskit. The shaded area is the 95% confidence interval. Note the non-linear scale on the x-axis.

5.1.2. Experiment: Lookahead

How does the lookahead affect training time and evaluation performance?

The lookahead is set to 1, 2, 4, and 8 with the optimal number of gates (8) used for training from the previous experiment. The training data in Figure 5.3 shows that the training time increases with the lookahead parameter, which is expected as the observation space grows. Furthermore, the primitive routing algorithm converges slower than the hierarchical routing algorithm. The evaluation results in Figure 5.4 show that the optimal lookahead for both routing algorithms is 4, with the primitive outperforming the hierarchical approach.

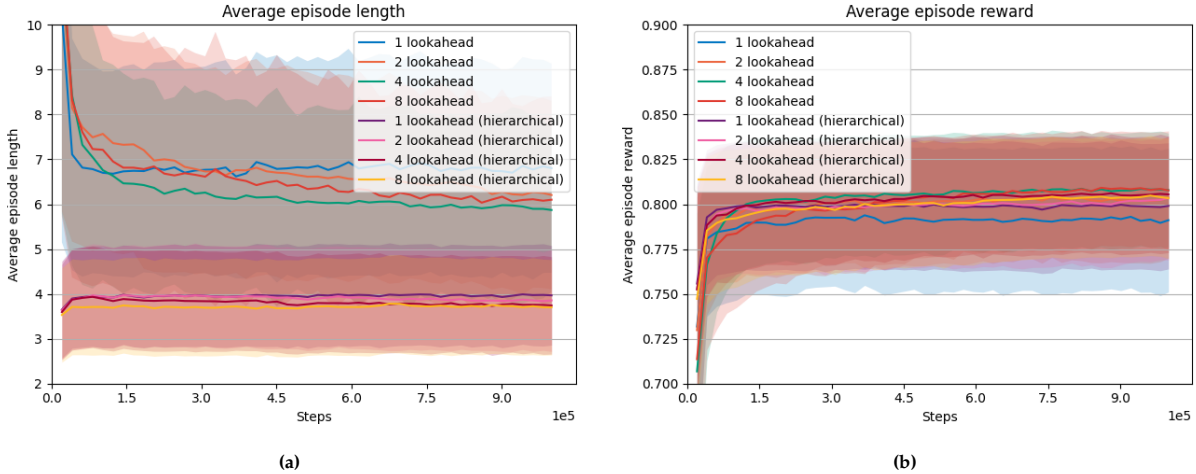


Figure 5.3: Training metrics for the lookahead parameter. The shaded area is the standard deviation of every 10 update batches..

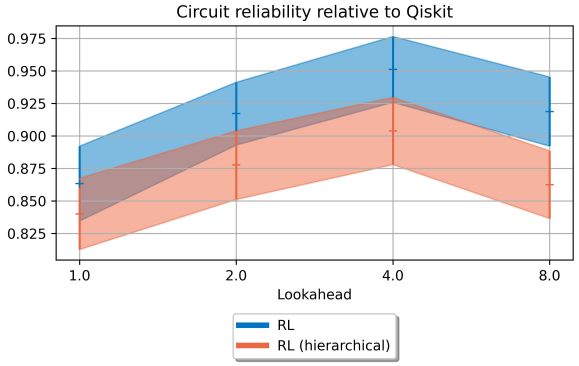
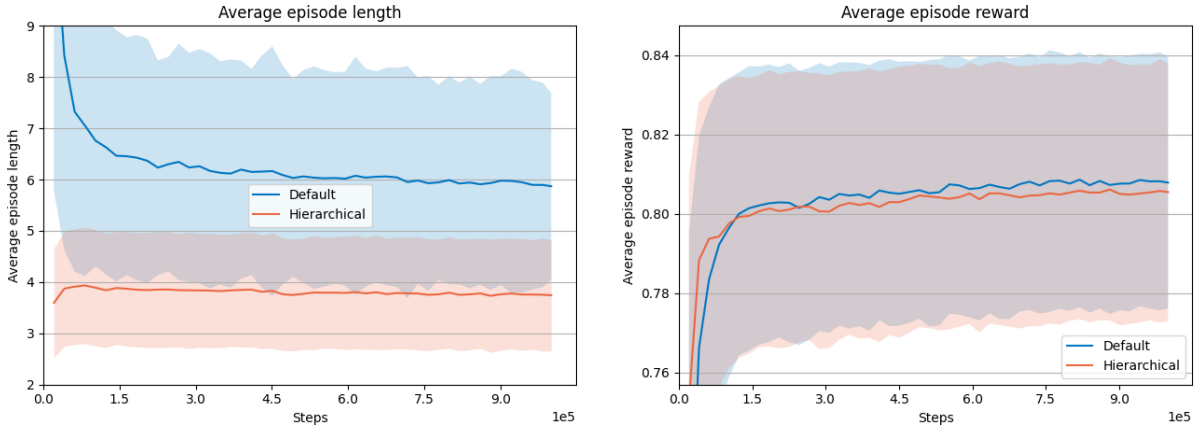


Figure 5.4: Circuit reliability versus the lookahead parameter. The shaded area is the 95% confidence interval.

5.1.3. Experiment: Number of Trials

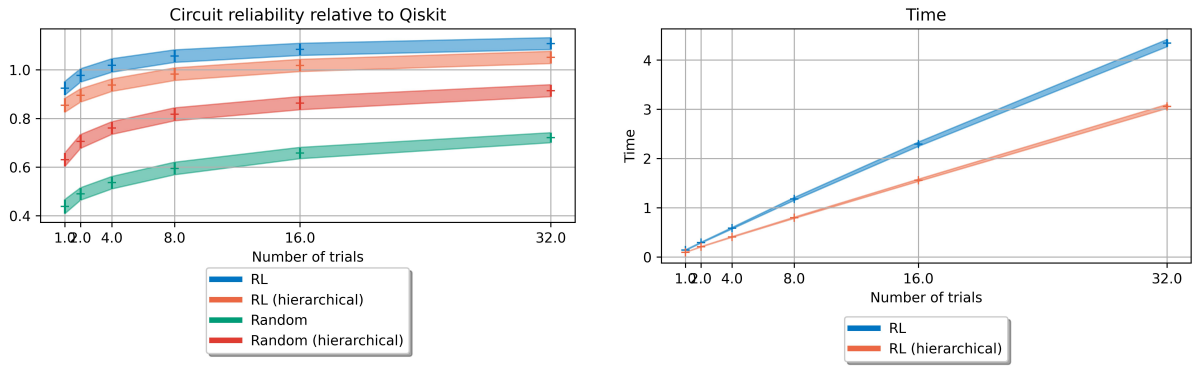
How does the number of trials affect routing time and evaluation performance?

The number of trials is set to 1, 2, 4, 8, 16, and 32 with the optimal lookahead parameter (4) from the previous experiment used. The training data in Figure 5.5 gives a more detailed comparison of the two routing algorithms with the primitive approach converging slower than the hierarchical approach but achieving a higher total reward in the end. The evaluation results in Figure 5.6a show that the circuit reliability increases with the number of trials. Both trained models outperform their random counterparts, but the primitive action space shows a larger improvement. Additionally, Qiskit is outperformed by the trained models starting from 8 trials. No significant difference is observed when using more than 8 trials ($p > 0.05$). Figure 5.6b shows a linear increase in routing time with the number of trials, which is expected.



(a) (b)

Figure 5.5: Training metrics for the number of trials. The shaded area is the standard deviation of every 10 update batches..



(a) (b)

Figure 5.6: Evaluation metrics for the number of trials. The shaded area is the 95% confidence interval. (a) Circuit reliability versus the number of trials. (b) Routing time versus the number of trials.

5.1.4. Discussion

Across the optimization experiments, it is observed that training circuit size, lookahead, and the number of trials each significantly influence both the routing performance (circuit reliability) and the training and runtime overhead.

Effect of circuit gates for training. From Figure 5.1 and Figure 5.2, the overall reliability of both the primitive and hierarchical approaches is worse than Qiskit when using only 4 gates. However, reliability improves considerably for both algorithms as the gate count increases to 8 or 16, which emerge as

the best-performing configurations (see Table 5.2). At 32 gates, the reliability drops slightly again and remains below the level at 8 or 16 gates. Regarding training dynamics, the hierarchical approach generally converges faster than the primitive approach, although the latter may reach a higher total reward toward the end of training.

Effect of lookahead. Varying the lookahead parameter has a marked effect on both training time and final performance (Figure 5.3 and Figure 5.4). Increasing lookahead expands the observation space, which slows training but can improve the agent's ability to plan. In practice, a lookahead of 4 yields the best balance: it consistently outperforms lookahead 1 or 2 while avoiding the excessive complexity at lookahead 8. Thus, for both primitive and hierarchical action spaces, lookahead 4 is chosen as the optimal setting.

Effect of number of trials. Finally, increasing the number of trials improves circuit reliability for both routing algorithms (Figure 5.5 and Figure 5.6a). However, Figure 5.6b also shows that routing time scales roughly linearly with the number of trials. Although 8 trials already achieve most of the reliability benefits, 16 or 32 trials can offer marginally higher reliability if the increased computational cost is acceptable.

The results of the optimization experiments are summarized in Table 5.2. The values chosen for the best performance are shown in bold and are used for the evaluation experiments.

Parameter	**Action**	**Best Value(s)**
Circuit gates	Primitive	**8**, 16
	Hierarchical	**8**, 16
Lookahead	Primitive	**4**
	Hierarchical	**4**
Number of trials	Primitive	**8**, 16, 32
	Hierarchical	**8**, 16, 32

Table 5.2

5.2. Evaluation

After optimizing the training setup, the trained agents are evaluated on different hardware configurations to investigate their performance in terms of circuit reliability. The parameters chosen are the best performing from the optimization experiments.

5.2.1. Experiment: Hardware Size

How does the hardware size affect training time and evaluation performance?

The number of timesteps for training is set to 4×10^6 for the largest hardware size. The hardware varies from a grid of 6 to 36 qubits to investigate the effect on training time and evaluation performance. Figure 5.7 shows primitive routing outperforming hierarchical routing for all topologies less than 16 qubits. Qiskit is only outperformed for circuit smaller than 9 qubits with relative perfromance decreasing as the number of qubits increases.

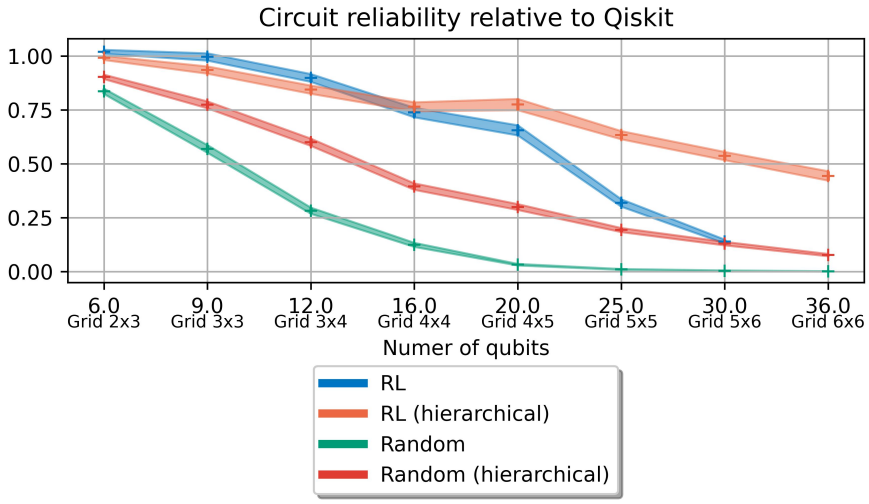


Figure 5.7: Circuit reliability versus the number of qubits in the hardware. The shaded area is the 95% confidence interval. Note the non-linear scale on the x-axis.

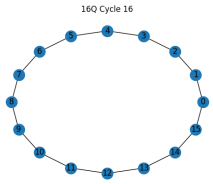
5.2.2. Experiment: Topology

How does the topology affect training time and evaluation performance?

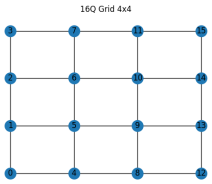
The topologies used in this experiment, illustrated in Figure 5.8, are designed to investigate their impact on both training time and evaluation performance. These topologies vary in connectivity degree k, ranging from an average of 2 to 4 connections per node, allowing us to assess how different levels of connectivity influence routing efficiency and circuit reliability.

IBM's Falcon and Eagle architectures feature large-cell tessellation, forming loops of various sizes [50]. In contrast, Google's Sycamore and Willow processors exhibit a connectivity approaching 4 [51], [52]. To maintain compatibility with the smallest topology, random circuits with 15 qubits are used, which results in some qubits being redundant in larger topologies. Additionally, the diameter δ of these topologies ranges from 4 to 8, a factor that may impact routing efficiency.

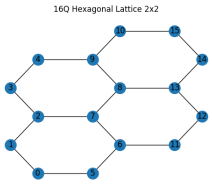
The results presented in Figure 5.9 indicate that circuit reliability improves with higher node degree, while the ladder topology exhibits a significant decline in performance. Moreover, the hierarchical routing algorithm consistently outperforms the primitive routing algorithm across all topologies, likely due to its more structured approach to path selection. Notably, for the cycle topology, the primitive routing algorithm fails to converge during training, potentially due to inefficient routing paths or poor optimization dynamics, leading to a missing value in the results.



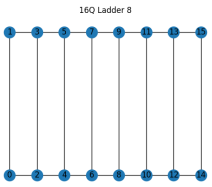
(a) Cycle 16 ($k = 2, \quad \delta = 8$)



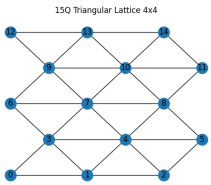
(b) Grid 4x4 ($k = 3, \delta = 6$)



(c) Hexagonal lattice 2x2 ($k = 2.4, \delta = 7$)



(d) Ladder 8 ($k = 2.75, \delta = 8$)



(e) Triangular lattice 4x4 ($k = 4, \delta = 4$)

Figure 5.8: Topologies used in the evaluation experiments.

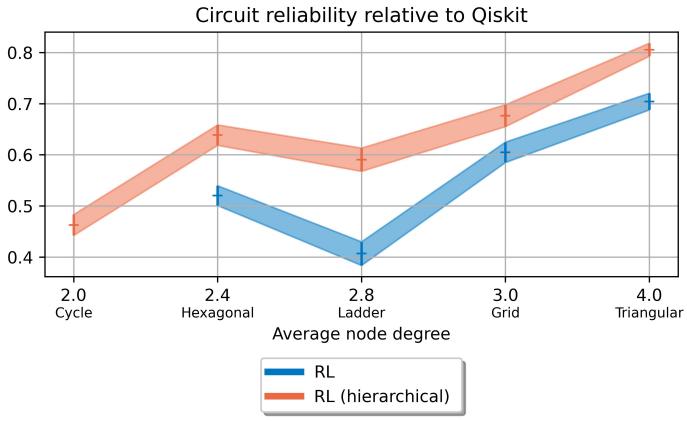


Figure 5.9: Circuit reliability versus the topology. The shaded area is the 95% confidence interval.

5.2.3. Discussion

Effect of hardware size. The results of the hardware size experiment (Figure 5.7) show that the primitive routing algorithm outperforms the hierarchical routing algorithm for all topologies with fewer than 16 qubits. Qiskit is only outperformed for circuits smaller than 9 qubits, with relative performance decreasing as the number of qubits increases. This suggests that the hierarchical routing algorithm may be more effective for larger circuits, while the primitive routing algorithm is better suited for smaller circuits.

Effect of topology. The topology experiment (Figure 5.9) reveals that circuit reliability is highest for topologies with the greatest node degree, while the ladder topology experiences a significant decrease in performance. The hierarchical routing algorithm consistently outperforms the primitive routing algorithm across all topologies. Notably, the primitive routing algorithm fails to converge during training for the cycle topology, leading to a missing value in the results. These findings suggest that the hierarchical routing algorithm may be more effective for complex topologies with a high node degree, while the primitive routing algorithm may struggle with simpler topologies.

6

Conclusion

This chapter concludes the report by summarizing the key findings and assessing whether the overarching research question has been addressed. It then identifies potential avenues for future research that build on the work presented here.

6.1. Summary of Findings & Research Question

The central goal of this research was to design and evaluate a reinforcement learning (RL) environment for qubit routing on noisy, near-term quantum hardware. Specifically, the study aimed to explore how different RL formulations (primitive vs. hierarchical action spaces) and environment configurations (lookahead, training-circuit sizes, number of trials) influence routing outcomes in terms of circuit reliability, gate overhead, and routing time.

Environment Design Choices. An environment is proposed that flexibly models hardware constraints (via coupling graphs and noise parameters) and circuit interactions (via gate dependency graphs). The environment was equipped with configurable state representations (e.g., interaction matrices and lookahead windows) and two types of action spaces (primitive and hierarchical).

- *Lookahead parameter:* The experiments showed that a moderate lookahead (e.g., 4 steps) offered the best performance—outperforming smaller lookahead values, while avoiding the excessive complexity associated with a large lookahead.
- *Training-circuit size:* Training on small circuits (fewer than 4 gates) led to suboptimal mappings for larger, more complex circuits. Training with 8–16 gates offered a strong balance between learning speed and generalization performance on larger circuits during evaluation.

RL Policy Formulations. Primitive and hierarchical action spaces are compared in terms of routing performance, scalability, and robustness:

- *Primitive Actions (Swap-based):* The agent learned how to insert individual swap gates at each step. While this approach often converged more slowly, it occasionally achieved a higher total reward once sufficiently trained, particularly for smaller circuit or hardware sizes.
- *Hierarchical Actions (Edge-based):* The agent selected an entire path-planning solution (via Dijkstra's or a similar method) for relocating qubits. This method converged faster and performed more robustly on larger or more complex topologies—though, at times, its ultimate reliability was slightly below that of the best-tuned primitive approach.

Scalability and Hardware Variation. In extending the environment to different hardware sizes and topologies (e.g., from a 6-qubit up to a 36-qubit grid, and from ladder-like to highly connected triangular-lattice topologies), the following trends were observed:

- Larger hardware graphs tended to reduce RL's performance significantly compared to heuristic transpilers, as the action and state spaces grew exponentially.
- Using hierarchical actions, RL methods were more robust and scalable on complex topologies, converging faster and achieving higher reliability than primitive approaches.
- Topologies with higher connectivity (i.e., higher node degree) generally allowed both RL approaches to maintain higher reliability, as they could leverage more diverse routing paths.

Comparison to Standard Compilers. In benchmark tests against Qiskit's swap-based transpiler passes (e.g., `BasicSwap`, `SabreSwap`, `StochasticSwap`) on random circuits, the best RL policies were competitive for smaller (up to 9 qubits) or highly connected topologies. For larger or more linear/ladder-type devices, Qiskit baselines often performed on par or better, especially with more than 16 physical qubits.

Answer to the Research Question. The study demonstrates that a flexible RL environment, incorporating hardware noise, gate connectivity constraints, and varied observation spaces, can indeed learn effective qubit-routing policies—occasionally surpassing or matching standard compiler heuristics under specific conditions (smaller hardware, moderate circuit size, or higher connectivity). The choice of action space (primitive vs. hierarchical), lookahead, and training-circuit complexity significantly affects routing performance and scalability. This confirms that properly designed RL approaches can be extended to more realistic, noise-aware scenarios on near-term quantum devices, although further work is needed to make such methods fully competitive on large-scale hardware.

6.2. Directions for Future Research

Although the proposed environment and RL formulations represent an advance in noise-aware, hardware-adaptive qubit routing, a number of open challenges remain. Below are several promising directions for future investigation:

1. **Integrating Single-Qubit Noise & More Detailed Error Models.** The current framework largely focuses on two-qubit gate noise, assuming single-qubit gates are error-free or have negligible noise. Incorporating realistic single-qubit error rates, measurement noise, idle errors (decoherence), and time-dependent calibration data would yield a more comprehensive noise model.
2. **Adaptive/Online Noise Tracking.** As real quantum devices exhibit time-varying error rates, particularly for two-qubit gates, an RL agent could be periodically retrained or updated online with new noise data. Techniques like meta-learning and continual learning may help the policy adapt quickly to shifts in hardware calibration.
3. **Advanced Scheduling & Concurrent Gate Execution.** Future work could refine the scheduling problem to allow parallel gate operations, respecting hardware-specific timing constraints and crosstalk. Multi-level scheduling, which integrates classical control signals and readout timing, remains largely unexplored in RL-based routing.
4. **Hybrid Heuristic–RL Methods.** Combining heuristic-based initial placements or partial solutions with RL planning might mitigate the large action and state spaces that hamper pure RL. For instance, short heuristic routes could be used where device connectivity is limited, while the RL agent handles more complex or globally impactful decisions.
5. **Scalability to Large Systems.** To push beyond 30–50 qubits, RL frameworks will require more efficient function approximators (e.g., graph neural networks) and distributed training. Investigating how RL solutions scale—and how to best represent large coupling graphs—remains an essential next step. Also reusing by training on smaller subgraphs or subproblems could help manage the combinatorial explosion of state-action pairs.
6. **Benchmarking With Real Hardware Runs.** While simulation-based metrics (circuit reliability, gate counts, etc.) provide insight, verifying the gains on real hardware experiments would strengthen the case for RL-based routing. Comparing actual circuit success probabilities with simulator predictions would also validate or refine the noise model.

Pursuing these directions will further clarify the viability of RL-based qubit routing on near-term quantum devices. As quantum processors expand in size and complexity, robust, noise-aware, and

scalable compilation methods—potentially leveraging advanced RL—are poised to play a key role in maximizing practical quantum computational power.

References

[1] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997, ISSN: 0097-5397, 1095-7111. DOI: 10.1137/S0097539795293172. arXiv: quant-ph/9508027. [Online]. Available: http://arxiv.org/abs/quant-ph/9508027.

[2] L. K. Grover, "Quantum Computers Can Search Arbitrarily Large Databases by a Single Query," *Physical Review Letters*, vol. 79, no. 23, pp. 4709–4712, Dec. 8, 1997, ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.79.4709. arXiv: quant-ph/9706005. [Online]. Available: http://arxiv.org/abs/quant-ph/9706005.

[3] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum Algorithm for Solving Linear Systems of Equations," *Physical Review Letters*, vol. 103, no. 15, p. 150 502, Oct. 7, 2009, ISSN: 0031-9007, 1079-7114. DOI: 10.1103/PhysRevLett.103.150502. arXiv: 0811.3171 [quant-ph]. [Online]. Available: http://arxiv.org/abs/0811.3171.

[4] J. Preskill, "Quantum Computing in the NISQ Era and Beyond," *Quantum*, vol. 2, p. 79, Aug. 6, 2018, ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79. arXiv: 1801.00862 [cond-mat, physics:quant-ph]. [Online]. Available: http://arxiv.org/abs/1801.00862.

[5] G. Li, Y. Ding, and Y. Xie, "Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices," May 7, 2019. arXiv: 1809.02573 [quant-ph]. [Online]. Available: http://arxiv.org/abs/1809.02573.

[6] A. Cowtan, S. Dilkes, R. Duncan, A. Krajenbrink, W. Simmons, and S. Sivarajah, "On the Qubit Routing Problem," 2019. DOI: 10.4230/LIPIcs.TQC.2019.5. arXiv: 1902.08091 [quant-ph]. [Online]. Available: http://arxiv.org/abs/1902.08091.

[7] A. Sinha, U. Azad, and H. Singh, "Qubit Routing Using Graph Neural Network Aided Monte Carlo Tree Search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 9, pp. 9935–9943, 9 Jun. 28, 2022, ISSN: 2374-3468. DOI: 10.1609/aaai.v36i9.21231. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/21231.

[8] G. Pascoal, J. P. Fernandes, and R. Abreu, "Deep Reinforcement Learning Strategies for Noise-Adaptive Qubit Routing," in *2024 IEEE International Conference on Quantum Software (QSW)*, Jul. 2024, pp. 146–156. DOI: 10.1109/QSW62656.2024.00030. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10646540.

[9] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, 10th anniversary edition. Cambridge: Cambridge University Press, 2010.

[10] M. H. Devoret and R. J. Schoelkopf, "Superconducting Circuits for Quantum Information: An Outlook," *Science*, vol. 339, no. 6124, pp. 1169–1174, Mar. 8, 2013. DOI: 10.1126/science.1231930. [Online]. Available: https://www.science.org/doi/10.1126/science.1231930.

[11] H. Haeffner, C. F. Roos, and R. Blatt, "Quantum computing with trapped ions," *Physics Reports*, vol. 469, no. 4, pp. 155–203, Dec. 2008, ISSN: 03701573. DOI: 10.1016/j.physrep.2008.09.003. arXiv: 0809.4368 [quant-ph]. [Online]. Available: http://arxiv.org/abs/0809.4368.

[12] C. Monroe and J. Kim, "Scaling the Ion Trap Quantum Processor," *Science*, vol. 339, no. 6124, pp. 1164–1169, Mar. 8, 2013. DOI: 10.1126/science.1231298. [Online]. Available: https://www.science.org/doi/10.1126/science.1231298.

[13] C. Nayak, S. H. Simon, A. Stern, M. Freedman, and S. Das Sarma, "Non-Abelian anyons and topological quantum computation," *Reviews of Modern Physics*, vol. 80, no. 3, pp. 1083–1159, Sep. 12, 2008, ISSN: 0034-6861, 1539-0756. DOI: 10.1103/RevModPhys.80.1083. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.80.1083.

[14] J. Alicea, "New directions in the pursuit of Majorana fermions in solid state systems," *Reports on Progress in Physics*, vol. 75, no. 7, p. 076 501, Jul. 1, 2012, ISSN: 0034-4885, 1361-6633. DOI: 10.1088/0034-4885/75/7/076501. arXiv: 1202.1293 [cond-mat]. [Online]. Available: http://arxiv.org/abs/1202.1293.

[15] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver, "A Quantum Engineer's Guide to Superconducting Qubits," *Applied Physics Reviews*, vol. 6, no. 2, p. 021 318, Jun. 1, 2019, ISSN: 1931-9401. DOI: 10.1063/1.5089550. arXiv: 1904.06560 [quant-ph]. [Online]. Available: http://arxiv.org/abs/1904.06560.

[16] R. Blatt and D. Wineland, "Entangled states of trapped atomic ions," *Nature*, vol. 453, no. 7198, pp. 1008–1015, Jun. 2008, ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature07125. [Online]. Available: https://www.nature.com/articles/nature07125.

[17] S. S. Tannu and M. K. Qureshi, "Not All Qubits Are Created Equal: A Case for Variability-Aware Policies for NISQ-Era Quantum Computers," *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pp. 987–999, Apr. 4, 2019. DOI: 10.1145/3297858.3304007. [Online]. Available: https://doi.org/10.1145/3297858.3304007.

[18] M. L. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (Wiley Series in Probability and Statistics v.414). Hoboken: John Wiley & Sons, Inc, 2009, 680 pp., ISBN: 978-0-471-72782-8.

[19] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. 2015.

[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. "Proximal Policy Optimization Algorithms." arXiv: 1707.06347 [cs]. [Online]. Available: http://arxiv.org/abs/1707.06347, pre-published.

[21] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1, 1992, ISSN: 1573-0565. DOI: 10.1007/BF00992698. [Online]. Available: https://doi.org/10.1007/BF00992698.

[22] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015, ISSN: 1476-4687. DOI: 10.1038/nature14236. [Online]. Available: https://www.nature.com/articles/nature14236.

[23] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy Gradient Methods for Reinforcement Learning with Function Approximation," in *Advances in Neural Information Processing Systems*, vol. 12, MIT Press, 1999. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1999/hash/464d828b85b0bed98e80ade0a5c43b0f-Abstract.html.

[24] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3, pp. 229–256, 3 May 1, 1992, ISSN: 1573-0565. DOI: 10.1007/BF00992696. [Online]. Available: https://link.springer.com/article/10.1007/BF00992696.

[25] V. Konda and J. Tsitsiklis, "Actor-Critic Algorithms," in *Advances in Neural Information Processing Systems*, vol. 12, MIT Press, 1999. [Online]. Available: https://papers.nips.cc/paper_files/paper/1999/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html.

[26] S. Pateria, B. Subagdja, A.-h. Tan, and C. Quek, "Hierarchical Reinforcement Learning: A Comprehensive Survey," *ACM Comput. Surv.*, vol. 54, no. 5, 109:1–109:35, Jun. 5, 2021, ISSN: 0360-0300. DOI: 10.1145/3453160. [Online]. Available: https://dl.acm.org/doi/10.1145/3453160.

[27] S. Huang and S. Ontañón, "A Closer Look at Invalid Action Masking in Policy Gradient Algorithms," *The International FLAIRS Conference Proceedings*, vol. 35, May 4, 2022, ISSN: 2334-0762. DOI: 10.32473/flairs.v35i.130584. arXiv: 2006.14171 [cs, stat]. [Online]. Available: http://arxiv.org/abs/2006.14171.

[28] C.-Y. Tang, C.-H. Liu, W.-K. Chen, and S. D. You, "Implementing Action Mask in Proximal Policy Optimization (PPO) Algorithm," *ICT Express*, vol. 6, no. 3, pp. 200–203, Sep. 2020, ISSN: 24059595. DOI: 10.1016/j.icte.2020.05.003. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2405959520300746.

[29] A. Zouitine. "Masking in Deep Reinforcement Learning - Boring Guy. "[Online]. Available: https://boring-guy.sh/posts/masking-rl/.

[30] A. Shafaei, M. Saeedi, and M. Pedram, "Qubit Placement to Minimize Communication Overhead in 2D Quantum Architectures," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2014, pp. 495–500. DOI: 10.1109/ASPDAC.2014.6742940.

[31] R. Wille, A. Lye, and R. Drechsler, "Optimal SWAP gate insertion for nearest neighbor quantum circuits," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2014, pp. 489–494. DOI: 10.1109/ASPDAC.2014.6742939. [Online]. Available: https://ieeexplore.ieee.org/document/6742939.

[32] A. Lye, R. Wille, and R. Drechsler, "Determining the Minimal Number of Swap Gates for Multi-Dimensional Nearest Neighbor Quantum Circuits," in *The 20th Asia and South Pacific Design Automation Conference*, Jan. 2015, pp. 178–183. DOI: 10.1109/ASPDAC.2015.7059001.

[33] A. A. A. de Almeida, G. W. Dueck, and A. C. R. da Silva, "Finding Optimal Qubit Permutations for IBM's Quantum Computer Architectures," in *Proceedings of the 32nd Symposium on Integrated Circuits and Systems Design - SBCCI '19*, São Paulo, Brazil: ACM Press, 2019, pp. 1–6, ISBN: 978-1-4503-6844-5. DOI: 10.1145/3338852.3339829. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3338852.3339829.

[34] M. Bandic *et al.*, *Mapping Quantum Circuits to Modular Architectures With QUBO*. May 11, 2023.

[35] S. Naito, Y. Hasegawa, Y. Matsuda, and S. Tanaka. "ISAAQ: Ising Machine Assisted Quantum Compiler." arXiv: 2303.02830 [quant-ph]. [Online]. Available: http://arxiv.org/abs/2303.02830, pre-published.

[36] M. Y. Siraichi, V. F. dos Santos, C. Collange, and F. M. Q. Pereira, "Qubit Allocation," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, Vienna Austria: ACM, Feb. 24, 2018, pp. 113–125, ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168822. [Online]. Available: https://dl.acm.org/doi/10.1145/3168822.

[37] L. Lao, H. van Someren, I. Ashraf, and C. G. Almudever, "Timing and Resource-Aware Mapping of Quantum Circuits to Superconducting Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 2, pp. 359–371, Feb. 2022, ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2021.3057583. [Online]. Available: https://ieeexplore.ieee.org/document/9349092/.

[38] S. Herbert and A. Sengupta. "Using Reinforcement Learning to find Efficient Qubit Routing Policies for Deployment in Near-term Quantum Computers." arXiv: 1812.11619 [quant-ph]. [Online]. Available: http://arxiv.org/abs/1812.11619, pre-published.

[39] M. G. Pozzi, S. J. Herbert, A. Sengupta, and R. D. Mullins, "Using Reinforcement Learning to Perform Qubit Routing in Quantum Compilers," *ACM Transactions on Quantum Computing*, vol. 3, no. 2, 10:1–10:25, May 16, 2022, ISSN: 2643-6809. DOI: 10.1145/3520434. [Online]. Available: https://dl.acm.org/doi/10.1145/3520434.

[40] P. Murali, J. M. Baker, A. J. Abhari, F. T. Chong, and M. Martonosi. "Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers." arXiv: 1901.11054 [quant-ph]. [Online]. Available: http://arxiv.org/abs/1901.11054, pre-published.

[41] Qiskit. "Qiskit | Transpiler. "[Online]. Available: https://qiskit.org/documentation/apidoc/transpiler.html.

[42] D. Bhattacharjee, A. A. Saki, M. Alam, A. Chattopadhyay, and S. Ghosh, "MUQUT: Multi-Constraint Quantum Circuit Mapping on NISQ Computers: Invited Paper," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Westminster, CO, USA: IEEE, Nov. 2019, pp. 1–7, ISBN: 978-1-72812-350-9. DOI: 10.1109/ICCAD45719.2019.8942132. [Online]. Available: https://ieeexplore.ieee.org/document/8942132/.

[43] V. Saravanan and S. M. Saeed, "Noise Adaptive Quantum Circuit Mapping Using Reinforcement Learning and Graph Neural Network," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2023, ISSN: 1937-4151. DOI: 10.1109/TCAD.2023.3340608. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10360454.

[44] P. Zhu, W. Ding, L. Wei, X. Cheng, Z. Guan, and S. Feng, "A Variation-Aware Quantum Circuit Mapping Approach Based on Multi-Agent Cooperation," *IEEE Transactions on Computers*, vol. 72, no. 8, pp. 2237–2249, Aug. 2023, ISSN: 1557-9956. DOI: 10.1109/TC.2023.3242208. [Online]. Available: https://ieeexplore.ieee.org/document/10035992.

[45] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1, 1959, ISSN: 0945-3245. DOI: 10.1007/BF01386390. [Online]. Available: https://doi.org/10.1007/BF01386390.

[46] Qiskit. "Qiskit. "[Online]. Available: https://qiskit.org.

[47] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html.

[48] S. van der Linde, W. de Kok, T. Bontekoe, and S. Feld. "Qgym: A Gym for Training and Benchmarking RL-Based Quantum Compilation." arXiv: 2308.02536 [quant-ph]. [Online]. Available: http://arxiv.org/abs/2308.02536, pre-published.

[49] J. Henstra. "Q-NARR. "[Online]. Available: https://github.com/jhenstra/Q-NARR.
[50] "IBM Quantum Documentatin | Processor types," IBM Quantum Documentation. [Online]. Available: https://docs.quantum.ibm.com/guides/docs.quantum.ibm.com/guides/processor-types.
[51] F. Arute *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, Oct. 2019, ISSN: 1476-4687. DOI: 10.1038/s41586-019-1666-5. [Online]. Available: https://www.nature.com/articles/s41586-019-1666-5.
[52] A. Paetznick *et al.* "Demonstration of logical qubits and repeated error correction with better-than-physical error rates." arXiv: 2404.02280 [quant-ph]. [Online]. Available: http://arxiv.org/abs/2404.02280, pre-published.

A

Algorithms

Algorithm 1 sub_actions: Determining sub-actions for hierarchical actions. Uses algorithm 2 for calculating the distance of a path.

Input: action, position, interaction_circuit, edges, placement
if state is done **then**
    return []
end if
$q1, q2 \leftarrow$ convert action to edge ▷ get physical qubits from action
$v1, v2 \leftarrow$ interaction_circuit[position] ▷ get virtual qubits from state
$p1, p2 \leftarrow$ placement[$v1, v2$] ▷ get physical qubits from mapping
$path\_1 \leftarrow$ get shortest path from $p1$ to $q1$
$path\_2 \leftarrow$ get shortest path from $p2$ to $q2$
$path\_3 \leftarrow$ get shortest path from $p1$ to $q2$
$path\_4 \leftarrow$ get shortest path from $p2$ to $q1$
if distance($path\_1$) + distance($path\_2$) > distance($path\_3$) + distance($path\_4$) **then**
    $actions\_1 \leftarrow$ convert $path\_3$ to actions
    $actions\_2 \leftarrow$ convert $path\_4$ to actions
else
    $actions\_1 \leftarrow$ convert $path\_1$ to actions
    $actions\_2 \leftarrow$ convert $path\_2$ to actions
end if
if action is in $actions\_2$ **then** ▷ swap actions if the action is in the second path
    swap $actions\_1$ and $actions\_2$
end if
    return $actions_1 \,||\, actions_2$ ▷ Concatenate actions from both paths.

Algorithm 2 distance: Calculate the distance of a path using edge weights.

Input: path, weights
distance $\leftarrow 0$
for each edge in path **do**
    distance $\leftarrow$ distance + weights[edge]
end for
    return distance