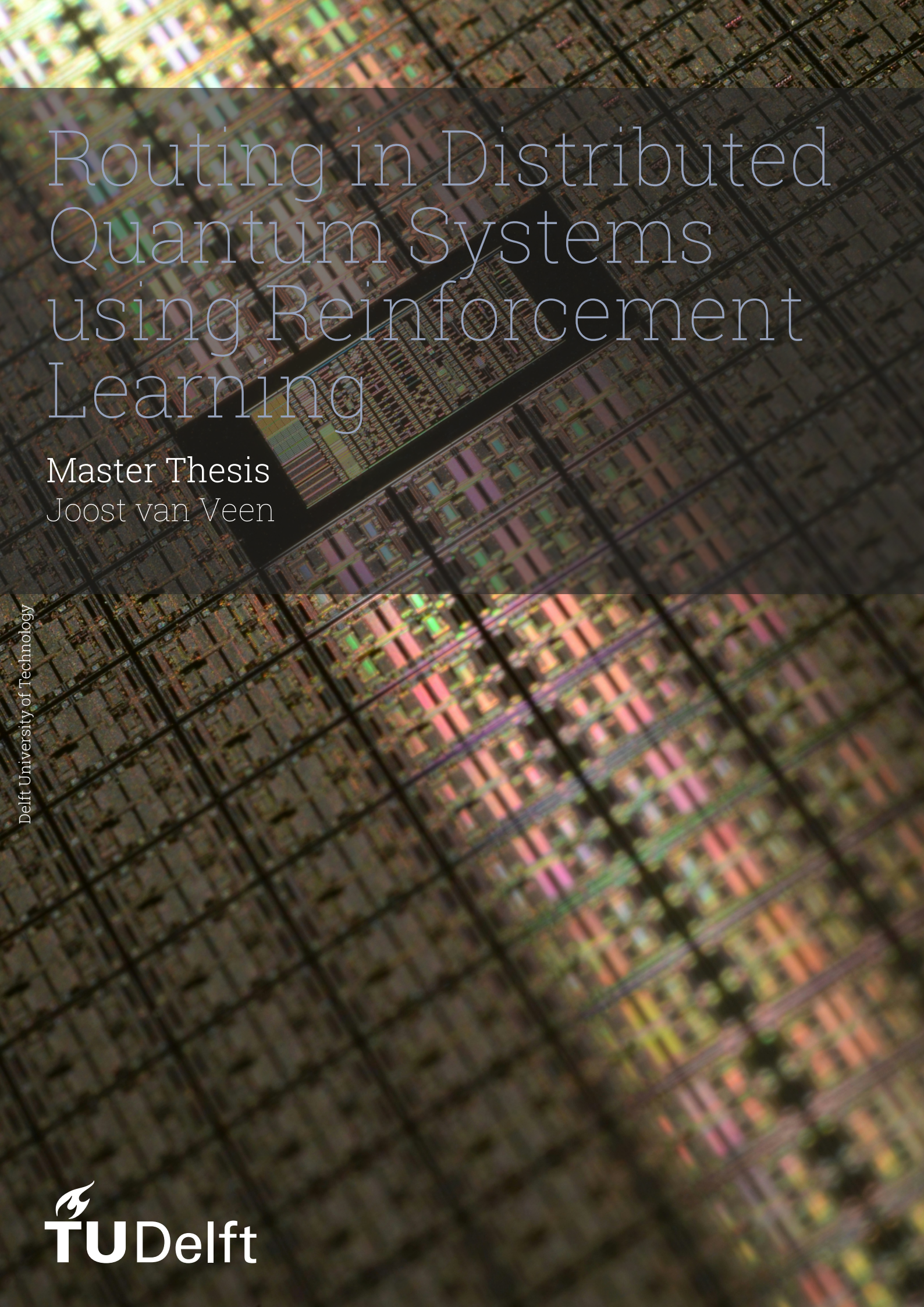


Routing in Distributed Quantum Systems using Reinforcement Learning

Master Thesis

Joost van Veen

Delft University of Technology

TU Delft

Routing in Distributed Quantum Systems using Reinforcement Learning

by

Joost van Veen

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Monday October 13, 2025 at 14:00.

Student number:	5097428
Project Duration:	Feb 2025 – Oct 2025
Thesis committee:	Dr. Q. Wang, TU Delft, chair
	Dr. S. Feld, TU Delft
	L. Prielinger, TU Delft
Daily supervisors:	Dr. S. Feld, TU Delft
	L. Prielinger, TU Delft
Faculty:	Electrical Engineering,
	Mathematics and Computer Science

TU Delft

Summary

Quantum computing holds the potential to solve problems that are intractable for classical systems. However, the physical realization of large-scale quantum systems remains a challenge due to the difficulty of scaling qubit counts. Distributed Quantum Computing (DQC) offers a promising solution by interconnecting multiple Quantum Processing Units (QPUs), effectively increasing the number of usable qubits. This interconnection introduces additional operations and exacerbates the complexity of qubit routing and entanglement management during circuit execution. While a reinforcement learning (RL) approach by Promponas et al. shows promise for qubit routing and EPR management in distributed quantum computing (DQC) environments, it suffers from inconsistent performance and is unable to reliably compile larger circuits. To address these limitations, we introduce a novel action space that allows direct operations between arbitrary qubit pairs, rather than restricting interactions to neighbouring qubits. While this significantly reduces solution depth, it increases the size of the action space, posing scalability challenges. To address this, we propose a novel neural network architecture that computes Q-values for qubit pairs based on the values of their individual qubits, considerably reducing the number of trainable parameters. Additionally, we extend the masking strategy to eliminate sub-optimal actions, effectively constraining the branching factor and accelerating learning. Together, these enhancements enable the agent to compile larger circuits with improved speed and consistency, significantly outperforming the baseline. Overall, our contributions result in better performance and reduced training time, marking a step forward in scalable quantum circuit compilation for distributed quantum systems.

Contents

1

Introduction

Quantum processing units (QPUs) represent a transformative advancement in computational technology, offering the capability to solve problems that lie beyond the computational reach of classical systems. Using the principles of quantum mechanics, such as superposition and entanglement, QPUs can tackle problems that are hard or even impossible to solve with classical computers [1]. Such as for example Shor's famous algorithm which he found in 1994 [2]. While only super-polynomial classical algorithms are known to factor large integer numbers, Shor's algorithm runs in polynomial time. Or the algorithm Grover [3] developed only two years later. It can search an unstructured database quadratically faster than any known classical algorithm. Such algorithms could be able to provide solutions in fields like cryptography, optimization, quantum chemistry, machine learning, and materials science [4][5][6][7].

However, despite their potential, the practical realization of QPUs faces significant challenges [8]. Chief among these is the difficulty of scaling up the number of qubits, the fundamental units of quantum information. As the number of qubits increases, maintaining coherence and minimizing noise become exponentially harder, posing a barrier to building larger QPUs. Additionally, the physical and engineering constraints associated with qubit integration, control, and error correction further complicate efforts to scale quantum processors [9].

An emerging solution to this challenge draws inspiration from classical computing architectures: instead of focusing on building larger single QPUs, multiple smaller QPUs can be linked together to form a distributed quantum system [10]. This approach mirrors the development of modern classical CPUs, where multiple processors are integrated into a single architecture to enhance computational power. By interconnecting multiple QPUs, it becomes possible to effectively increase the available qubit resources while mitigating some of the scaling challenges faced by individual units.

However, linking multiple QPUs introduces new technical hurdles, particularly in terms of communication, synchronization, and the management of quantum information across the network which are essential for quantum compilation [11]. The complexities associated with routing quantum states and operations between interconnected QPUs become a significant challenge for successfully compiling quantum circuits on distributed quantum systems [10][12]. Efficient initial placement and routing are essential just as in regular quantum systems, however routing is more challenging in distributed quantum systems due to their limited connectivity. This results in longer paths between qubits, requiring additional SWAP operations to bring them together. Consequently, circuit depth, noise, and the overall complexity of finding efficient routing paths all increase [10][12].

This routing issue is further complicated by how different QPUs in distributed quantum systems are linked. Unlike classical data links, which rely on deterministic data transfer, EPR pair-based connections require precise quantum operations to generate and maintain. This process is highly sensitive to noise, which leads to decoherence degrading system performance. Therefore the handling of EPR pairs inside of distributed quantum system is critical in order to optimise system performance.

To address the quantum circuit routing problem in compilers for Distributed Quantum Computing (DQC),

Promponas et al. [13] framed the task as a Markov Decision Process (MDP) and employed a reinforcement learning (RL) agent to learn efficient routing strategies. Here, execution time refers to the total time required to perform all operations in a quantum circuit, including both local gates and inter-QPU operations such as EPR generation and remote gates. The results reported in [13] demonstrate that the RL agent is able to learn policies that reduce the expected execution time, effectively improving the overall efficiency of circuit execution in distributed settings.

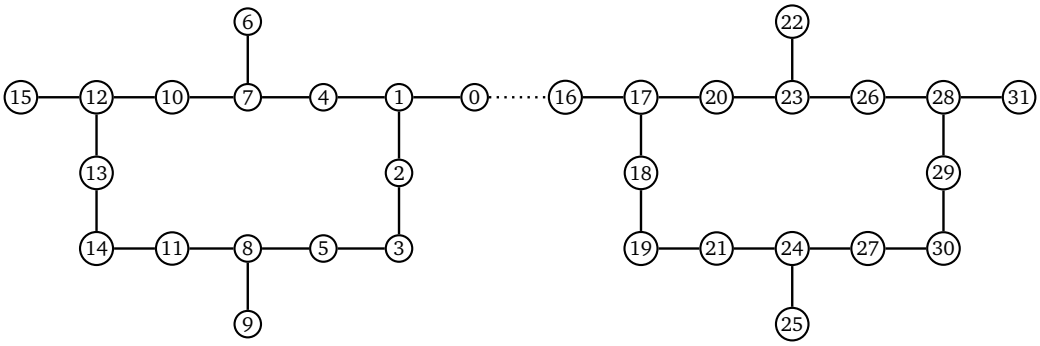


Figure 1.1: DQC system with two IBM Q Guadalupe QPUs. Node labels reflect the physical qubit number; dotted line is the quantum link connecting both QPUs.

However, the result also shows that for a system with two IBM Q Guadalupe QPUs, as shown in Figure 1.1, the agent is limited in the size of quantum circuits it is able to compile. This limitation prevents it from processing circuits that represent practical quantum algorithms since they are often significantly larger than what the agent is able to compile. For example, a Quantum Fourier Transform algorithm with 18 qubits requires 153 two-qubit gates, whereas the current system can only compile circuits up to 30 gates (with the same number of qubits).

Furthermore, the agent is unable to create a general policy which solve all circuits within the training time meaning after training it would be unable to compile almost any circuit. While such a policy could, in principle, be learned by increasing the training time, this approach is impractical for several reasons. First, longer training significantly increases computational costs, making the method inefficient and difficult to scale. Second, the reinforcement learning agent needs to be retrained for each specific hardware architecture, and if training times are too long, this severely limits adaptability to new architectures. Finally, the gains from additional training often diminish over time, meaning that a substantial increase in training duration may lead to only marginal performance improvements. Together, these factors make simply increasing training time an unsuitable solution for achieving generalization in this setting.

This performance gap underscores the need to enhance the agent's ability to generate a more general policy such that the agent is able to solve circuits more consistently, and is brought one step closer to the ability to compile larger, more realistic quantum algorithms.

Based on this gap the research goals pursued in this thesis are as follows:

- The current compiler is too inconsistent in its ability to compile quantum circuits, meaning that it is unable to create a general enough policy within the given training time. To this end, this work introduces methods which allow a reinforcement learning agent to find a more general policy within the same training time.
- The current compiler is unable to compile circuits with 40 or more gates given the used training time constraint. A comprehensive numerical investigation shows that the introduced agent is able to learn and develop a solving strategy for the given compilation problem for larger circuit sizes using similar hyperparameters and same training time.

To provide the necessary context, Section 2.1 introduces quantum compilation and outlines the challenges associated with compiling quantum circuits in distributed quantum systems. Section 2.2 presents the fundamentals of reinforcement learning, with a focus on Deep Q-Networks (DQN) and Double DQN (DDQN). Chapter 3 describes the approach proposed by Promponas et al., detailing their method for formulating the compilation task as a Markov Decision Process (MDP). Chapter 4 discusses the changes made to the original

design and the rationale behind these modifications. Chapter 5 presents the experimental results of the updated compiler and compares them with the baseline across different hardware architectures and quantum circuit sizes. It also evaluates how models trained on smaller circuits generalize to larger ones, in comparison to models trained directly on larger circuits. Finally, Chapter 6 and Chapter 7 analyse the outcomes, reflect on the implications of the findings, and provide recommendations for future work.

2

Preamble

2.1. Quantum compilation for distributed quantum systems

Quantum computation operates on the fundamental unit of quantum information known as the qubit. Unlike a classical bit, which can be either 0 or 1, a qubit can exist in a superposition of both states simultaneously, represented as

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

where α and β are complex amplitudes satisfying $|\alpha|^2 + |\beta|^2 = 1$. The states $|0\rangle$ and $|1\rangle$ are called the computational basis states, they represent the classical bit values of 0 and 1. Mathematically, they are expressed as vectors in a two-dimensional complex vector space:

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

A quantum state is a linear combination of these two states and, when measured, collapses into one of the two states with the probabilities $|\alpha|^2$ and $|\beta|^2$, respectively. The ability to exist in superposition is what enables quantum computers to process a vast amount of information in parallel.

To manipulate qubits and perform computation, quantum circuits are used. A quantum circuit is a sequence of quantum gates applied to qubits, guiding their evolution over time. These gates are the quantum analogues of classical logic gates and come in two main types: single-qubit and multi-qubit gates.

Single-qubit gates act on individual qubits to change their state. Some commonly used gates include:

- **Pauli-X gate** (bit-flip):
$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad X|0\rangle = |1\rangle, \quad X|1\rangle = |0\rangle$$
- **Pauli-Y gate** (bit- and phase-flip):
$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$
- **Pauli-Z gate** (phase-flip):
$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$
- **Hadamard gate**, which creates superposition:
$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

To enable interaction between qubits and produce quantum effects such as entanglement, multi-qubit gates are required. The most fundamental of these is the CNOT (Controlled-NOT) gate, which flips the target qubit if the control qubit is in the $|1\rangle$ state:

$$\text{CNOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \quad \text{CNOT}|10\rangle = |11\rangle$$

By combining these gates in a specific sequence, quantum circuits can implement complex quantum algorithms. An example of a quantum circuit can be seen in Figure 2.1, the single qubit gates are represented by boxes with the letter of the gate. The CNOT gates are denoted by a line going from one qubit to another, where the dot ($\cdot$) indicates the controller qubit and the $\oplus$ symbol indicates the target qubit. The design of such circuits plays a critical role in quantum computing, as it plays a crucial role in how efficient an algorithm can be executed on real hardware [14] [15].

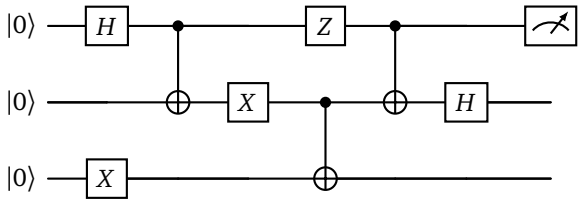


Figure 2.1: Example of a quantum circuit

In the context of quantum compilation, it is important to distinguish between logical and physical qubits. Logical qubits are abstract qubits defined by the quantum circuit or just *circuit* for short, whereas physical qubits are the actual hardware qubits on a quantum processor. To execute a quantum algorithm on real hardware, each logical qubit must be assigned to a specific physical qubit, this process is known as qubit mapping [14].

The execution of quantum circuits is often constrained by the underlying quantum hardware. Most quantum processors have a fixed physical layout and are not fully connected, meaning that not every qubit can directly interact with every other qubit. Two-qubit gates, such as the commonly used CNOT gate, can only be applied between physically connected (neighbouring) qubits. As a result, some gates in a quantum circuit may not be executable directly if the involved qubits are not adjacent on the quantum processing unit (QPU). To enable such interactions, the quantum states of the qubits must be brought closer together using a series of SWAP gates [14] [15].

A SWAP gate exchanges the quantum states of two qubits, effectively relocating the logical qubits mapped to those physical positions. Since a SWAP gate is typically implemented using three CNOT gates, adding SWAPs increases the circuit depth, the total number of sequential operations in the circuit. This increase is significant because current quantum processors still suffer from qubit instability, where a qubit can lose its coherence over time due to unwanted interactions with its environment. When this happens, the qubit no longer maintains its quantum properties such as superposition and entanglement. Importantly, decoherence does not typically cause a qubit to collapse into a definite classical state (e.g., 0 or 1) unless a measurement is made. Instead, due to interactions with the environment and various noise processes, the qubit's state evolves into a mixed state, a statistical mixture of states that no longer preserves the phase relationships necessary for quantum computation. In the case of full decoherence, the qubit approaches a completely mixed state, where it contains no usable quantum information. This degradation of quantum coherence, referred to as decoherence, is one of the central challenges in maintaining reliable quantum computation [16].

Because decoherence occurs continuously and worsens with longer operation times, minimizing circuit depth and the total number of operations is crucial for successful circuit execution. If a circuit runs too long or contains too many gates, the quantum information can become corrupted, leading to errors and unreliable results. Therefore, managing decoherence is a central challenge in quantum compilation, as it drives the need to optimize qubit routing, reduce SWAP gates, and execute circuits as efficiently as possible [14][17].

Furthermore, quantum gates must be executed in a specific order dictated by their qubit dependencies. Some gates operate on the same qubits and thus rely on the outcomes of previous operations. Therefore some gates can not be executed in parallel or before other gates have been executed. This sequencing constraint limits how much a quantum circuit can be parallelized and directly impacts execution time and error accumulation. To represent and analyse these dependencies, Directed Acyclic Graphs (DAGs) are commonly used in quantum compilation [18]. In a DAG, nodes correspond to quantum gates, and edges indicate the required execution order between them, typically due to shared qubits or logical constraints. Figure 2.2 draws the DAG of the circuit displayed in Figure 2.1. In this DAG the so-called *frontier* would consist of the gates H on q_0 and X on q_2, that is, the first layer of execution. DAGs provide a useful abstraction for optimizing circuit scheduling, parallelism, and qubit routing.

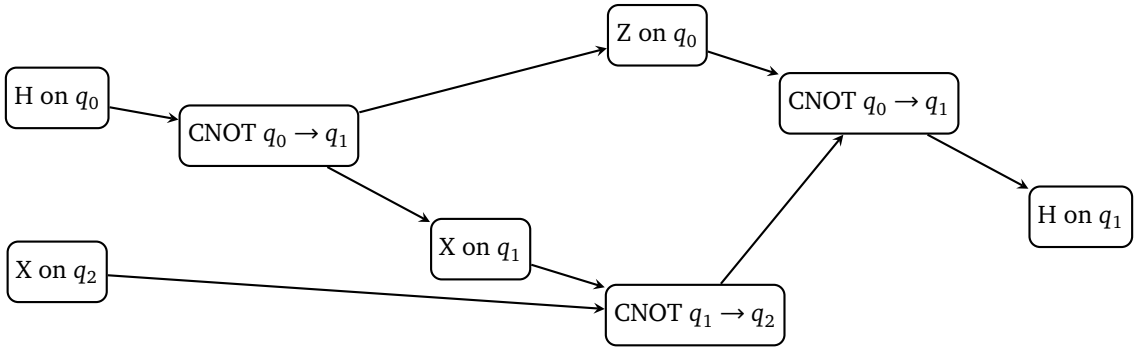


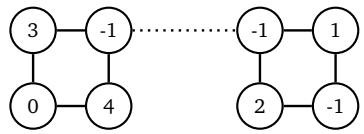
Figure 2.2: DAG of the quantum circuit displayed in Figure 2.1

Limiting the amount of SWAPs when executing quantum circuits is one of the main challenges in quantum compilers [14] [19]. This is typically achieved through two main strategies: initial mapping and efficient qubit routing algorithms. Initial mapping refers to the mapping of logical qubits to physical qubits at the start of execution. An optimal initial mapping can significantly reduce the amount of routing required during computation. Qubit routing, on the other hand, involves moving logical qubits to adjacent physical qubits to enable the execution of gates. In an optimal routing, qubits are moved along the most efficient paths which lead to the fastest possible circuit execution. The qubit routing problem is a challenging combinatorial optimization task. Notably, it has been proven to be NP-hard even in restricted settings such as linear nearest-neighbour architectures [20]. Combined, the goal of both strategies is to minimize the number of SWAP operations required to successfully execute the quantum circuit and less risk of decoherence.

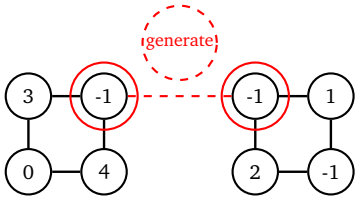
These compilation challenges become even more complex in Distributed Quantum Computing (DQC) systems due to how different QPUs are interconnected. Typically, QPUs are linked via entangled quantum states, often established through Einstein-Podolsky-Rosen (EPR) pairs [21]. EPR pairs create a shared entanglement between qubits located on different QPUs, enabling the transmission and coordination of quantum information across the distributed system.

In a DQC environment, a QPU will have one or more physical qubits linked to qubits on other QPUs through what is known as a *quantum link*. Along these quantum links, EPR pairs can be generated [22]. It is important to note that both physical qubits must be 'empty', meaning they cannot be mapped to logical qubits involved in the circuit. This is necessary because the states of qubits used in computation cannot be altered or sacrificed for establishing entanglement. After an EPR is generated both of those physical qubits will hold one half of the EPR pair. An EPR pair half can then be moved around on a QPU via SWAP operation the same way a logical qubit would. This process can be seen in Figure 2.3, which shows the state before an EPR pair is generated and which qubits are involved. The empty qubits are denoted by -1, all other numbers coincide with logical qubits. The quantum link connecting the two QPUs is the dotted line. By moving the EPR pair halves and thereby freeing the physical qubit, a new EPR pair can be generated. Importantly, this means that multiple EPR pairs can exist at once in a system.

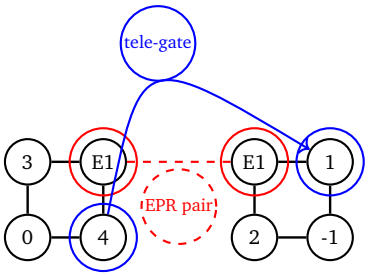
An EPR pair can be used to perform a *tele-qubit* operation, which teleports a neighbour of one half of the EPR pair to the location of the other half [14]. This mechanism enables the teleportation of a logical qubit from one QPU to another. Alternatively, a *tele-gate* operation can be performed, where a CNOT gate is executed between a neighbour of one EPR half and a neighbour of the other EPR half. Both of the operations and how



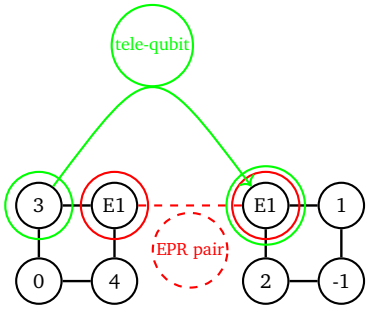
(a) Base system.



(b) Generate action, after which the two qubit highlighted will hold an EPR pair half (both -1 will become E1).



(c) Tele-gate action, executes a CNOT between qubit 4 and 1, after which the EPR is destroyed and the qubit will become empty qubits again (both E1 will become -1).



(d) Tele-qubit actions, which swaps qubit 3 with the qubit holding the other EPR pair half on the other QPU, after which the EPR is destroyed and the qubit will become empty qubits again (3 and the neighbouring E1 will be -1, and the other E1 will be 3).

Figure 2.3: DQC system with two QPUs linked by one quantum link (a) showing: red dashed line for EPR pair generation between '-1' qubits (b), blue arrow for teleported gate (c), and green arrow for teleported qubit (d). Circles highlight qubits involved in each operation.

they work can be seen in Figure 2.3. This allows for the execution of two-qubit gates across different QPUs. Both operations consume the EPR pair in the process, meaning the qubit holding the EPR pair halves become empty qubits again.

Correctly implementing these operations is a critical part of circuit compilation. Teleporting the wrong qubit, or choosing a tele-qubit operation when a tele-gate is would be more efficient (or vice versa), can significantly increase circuit depth and execution time. This is one of the key reasons why routing qubits in distributed quantum systems is substantially more complex than in single-QPU systems [12].

2.2. Reinforcement learning

Reinforcement learning (RL) is a type of machine learning where an agent interacts with an environment by taking actions to achieve a goal [23]. Unlike supervised learning, where models learn from labelled datasets, RL focuses on learning from experience through trial and error. At each step, an *agent* observes the current state of the environment, chooses an action and receives feedback in the form of a reward along with a new state. The agent's objective is to choose actions which maximize the expected sum of future rewards over time.

The formal mathematical framework used to describe many reinforcement learning problems is the Markov Decision Process (MDP). An MDP defines the environment in which the agent operates and consists of:

- S — a set of possible states,
- A — a set of possible actions,
- $P(s'|s,a)$ — a transition probability function, which gives the probability of transitioning to state $s' \in S$ after taking action $a \in A$ in state $s \in S$,
- $R(s,a)$ — a reward function, which gives the immediate reward, r, received after taking action $a \in A$ in state $s \in S$.

A policy π defines the agent's behaviour by mapping states to actions. More formally, a stochastic policy is defined as a distribution $\pi(a|s)$, which gives the probability of taking action a in state s, while a deterministic policy always selects the same action for a given state. The goal in reinforcement learning is to find an

optimal policy π^* that maximizes the expected cumulative reward, or return, over time.

To evaluate how good a policy is, RL makes use of value functions. The state-value function $V^\pi(s)$ gives the expected return when starting in state s and following policy π thereafter:

$$V^\pi(s) = E_\pi\left[\sum_{t=0}^{\infty}\gamma^t R(s_t, a_t)\,\middle|\, s_0 = s\right]$$

where $\gamma \in [0,1]$ is a discount factor that determines the present value of future rewards.

The action-value function, or Q-function, is more commonly used in many modern RL algorithms [24] [23]. It estimates the expected return of taking action a in state s, and thereafter following policy π:

$$Q^\pi(s,a) = E_\pi\left[\sum_{t=0}^{\infty}\gamma^t R(s_t, a_t)\,\middle|\, s_0 = s, a_0 = a\right]$$

The optimal policy π^* can then be obtained by acting greedily with respect to the optimal Q-function:

$$\pi^*(s) = \arg\max_{a\in A} Q^*(s,a)$$

Before the development of deep neural networks, simpler function approximators such as decision trees were used to estimate value functions [25]. A decision tree is a hierarchical model that recursively splits the input space based on feature values, assigning a prediction (e.g., a Q-value) at each leaf node. These models are interpretable, efficient to train, and well-suited for capturing non-linear patterns in low-dimensional state spaces [26].

Although less common in reinforcement learning today, decision trees can be relevant in analysing the structure and difficulty of a learning task. For example, trees can be used to estimate properties such as the *solution depth* (the minimum number of decisions needed to solve a task), or the *branching factor* (the average number of meaningful actions per state). These metrics can serve as proxies for the expected difficulty of the environment and help explain why certain RL tasks can be learned faster than others. A large branching factor or long solution path may indicate that deeper or more expressive neural networks and longer training are required for convergence.

In environments with large or continuous state spaces, it becomes computationally infeasible to represent and update the Q-function as a lookup table. To address this, Deep Q-Networks (DQN) were introduced, leveraging the function approximation capabilities of deep neural networks [27][28]. In DQN, a multi-layered neural network approximates the Q-function by taking a state s as input and outputting an estimated Q-value for each possible action $a \in A$. The parameters (*weights*) of the network are adjusted through training to minimize the difference between predicted Q-values and target values.

To improve the stability of training, DQN introduces two key techniques aimed at making the learning process more consistent and reliable. First, a separate target network is used to compute the target Q-values in the loss function. This target network is a delayed copy of the main Q-network and is updated at fixed intervals, preventing rapid fluctuations in the targets that the main network is trying to learn. Without this mechanism, the Q-values could become unstable because the network would be learning from targets that are themselves changing too quickly.

Second, experience replay is used, where the agent stores past transitions in a buffer and samples mini-batches randomly for training. The agent then learns periodically, every N steps, by sampling a random batch of experiences from this buffer. The size of the sampled batch, β, and the total buffer size, BS, are important hyperparameters. They influence not only the learning quality of the model, but also the requirements of computational resources and the training speed [29]. This breaks the correlation between consecutive experiences and improves learning efficiency. The overall learning process involves minimizing a loss function defined as:

$$\mathscr{L}(\theta) = E_{(s,a,r,s')\sim D}\left[(y - Q(s,a;\theta))^2\right]$$

where θ are the parameters of the main network, $y = r + \gamma \max_{a'} Q(s',a';\theta^-)$ is the target Q-value computed using the target network with parameters, θ^-, and experiences from the replay buffer, D.

While effective, the max operator, $\max_{a'} Q(s', a'; \theta^-)$, in the target Q-value computation can lead to overoptimistic value estimates, particularly in stochastic environments. To address this, Double DQN (DDQN) was introduced [28]. In DDQN, the action selection and evaluation are decoupled to reduce overestimation bias. Specifically, the next action is selected using the online (current) Q-network, but its value is evaluated using the target network:

$$y^{\text{DDQN}} = r + \gamma Q(s', \arg\max_{a'} Q(s', a'; \theta); \theta^-)$$

This modification improves the accuracy of value estimation and thereby enhances the stability and performance of the learning process across various environments.

In reinforcement learning, greedy epsilon exploration is a strategy used to balance exploration and exploitation during training by mostly choosing the action with the highest estimated reward, in case of DDQN the action with the highest Q-value, (greedy action) while occasionally selecting a random action with a small probability ϵ. This allows the agent to explore potentially better actions that it hasn't tried enough, preventing it from getting stuck in suboptimal behaviours. However, striking the right balance between exploration (trying new actions to discover their value) and exploitation (leveraging known high-reward actions) is challenging. Too much exploration can lead to inefficient learning, as the agent spends excessive time on poor choices, while too much exploitation can cause premature convergence to suboptimal policies, missing out on better long-term strategies. Greedy epsilon methods attempt to mediate this tension by maintaining a controlled level of randomness, this is done by tuning ϵ appropriately and decaying it over time. The rate of decay and the balance between exploitation and exploration is crucial to achieve efficient learning.

In order to train a reinforcement learning agent to learn as optimally as possible for a given problem, the choice of these and other hyperparameters is crucial. As discussed earlier, the memory buffer size, BS, batch size, β, and epsilon decay, ϵ_d, must be carefully tuned. In addition, several other hyperparameters play an important role in the performance of a reinforcement learning models:

- **Learning rate** (lr): determines how quickly the network updates its parameters during gradient descent. A high learning rate may cause instability, while a very low rate may slow convergence.
- **Epsilon decay** (ϵ_d): controls the rate at which the exploration parameter ϵ decreases over time in the ϵ-greedy strategy. A slower decay encourages longer exploration, while a faster decay makes the agent exploit its knowledge earlier.
- **Discount factor** (γ): specifies how much future rewards are valued relative to immediate rewards. A value close to 1 encourages long-term planning, while smaller values bias the agent toward short-term gains.
- **Target network update frequency** (τ): in DDQN, a separate target network is periodically updated to stabilize training. This parameter determines how often the weights of the main network are copied to the target network.
- **Number of learning iterations**: specifies how many times the network is updated during a single learning phase. More iterations allow deeper learning but increase computational cost.
- **Learning frequency**: defines how often the agent performs a learning step relative to its interactions with the environment (e.g., after every step, or after every n steps). This balances experience collection with training stability.

Together, these hyperparameters strongly influence the stability and convergence of the RL training process, and they should be tuned carefully to match the complexity of the environment.

Through this neural approximation framework and its refinements such as DDQN, deep reinforcement learning enables agents to operate effectively in complex, high-dimensional environments where traditional tabular methods are no longer viable. However, RL also faces several intrinsic challenges. Learning typically requires a large number of interactions with the environment, making training computationally expensive, especially in real word setting or domains like quantum compiling where data collection is complex. Furthermore, after training reinforcement learning agents often struggle on new environments or tasks, since the agent has difficulty creating policies which work in different and new situations [30].

2.3. Related Work

The compiler proposed by Promponas et al. explicitly targets minimizing expected execution time in distributed quantum computing (DQC), a goal it pursues by jointly managing EPR pair generation, remote gate scheduling, and SWAP insertion. This integrated approach directly addresses the risk of decoherence and failed runs, which are major challenges in DQC systems where communication latency can dominate the total execution time.

By contrast, many existing compilers are designed around different optimization objectives. For example, the strategy for remote scheduling proposed by Ferrari et al. [31] focuses on tele-qubit and tele-gate scheduling first. In their framework, remote gates are scheduled as a separate preprocessing step before local routing is performed. This contrasts with the approach of Promponas et al., where remote scheduling and local routing are handled jointly, enabling decisions that better reflect the true execution dynamics of distributed systems. The difference is also evident in how the two compilers determine gate scheduling: Ferrari et al. decide between tele-gate and tele-qubit operations by minimizing a generalized cost function that accounts for both execution time and resource consumption, whereas the RL-based compiler of Promponas et al. chooses based solely on which option minimizes the expected total execution time.

Another line of work, such as the circuit mapping framework by D. Ferrari et al. [12], aims to reduce circuit depth overhead. In these approaches, the compiler minimizes the number of additional gates (e.g., SWAPs) inserted during mapping, thereby constraining the increase in circuit depth. While this indirectly constrains execution time, the objective remains overhead minimization rather than directly optimizing runtime performance.

Finally,the quickest flow optimization method proposed by Cuomo et al. [32], which does explicitly address focusing on optimizing for execution time. In this formulation, the routing task is modelled as a network flow problem, where the objective is to find the fastest way to send a required amount of "flow" (in this case, qubit interactions or entanglement links) through a network with limited capacities. By minimizing this flow time, the compiler aims to reduce the overall execution time of the distributed computation. However, their results are validated primarily on high-connectivity architectures, where qubit routing is inherently simpler. As a result, the demonstrated benefits may not translate to more restrictive or heterogeneous DQC topologies, where routing bottlenecks and limited entanglement bandwidth dominate performance.

Together, these distinctions underscore the novelty of the compiler by Promponas et al., which uniquely unifies EPR management, scheduling, and routing under the single objective of minimizing expected execution time in realistic distributed settings. This while other compilers and routing solutions, like DDRoute by A. Annechini et al. [33], optimize for circuits depth. While both are related circuit depth refers to the length of the longest sequence of dependent gates, while execution time would refer to the actual time it would take to execute those gates. For example, if a circuit has 10 layers of gates that must be done in sequence, its depth $= 10$, even if each gate is extremely fast or slow.

L. Le and T. N. Nguyen [34] demonstrated that reinforcement learning can be applied to quantum routing through their Deep Quantum Routing Agent (DQRA), which optimizes entanglement distribution across a quantum network to maximize successful communication. While DQRA focuses on quantum communication, its use of RL to make adaptive, sequential routing decisions is conceptually similar to Promponas et al., who employ RL to optimize gate scheduling. Both works highlight how RL can manage complex, resource-constrained quantum systems under uncertainty.

Reinforcement learning has also been applied directly to the qubit routing problem in non-distributed quantum systems in several recent studies. Early and influential RL approaches include the deep Q-learning based qubit routing methods by Pozzi et al. [35], which demonstrate that RL can outperform classical heuristic routers on several near-term topologies. More recently, hybrid RL + search approaches (e.g., AlphaRouter) have shown additional gains by combining learned policies with tree search, reporting substantial reductions in routing overhead [36]. These RL-based methods differ from Promponas et al.'s work in their target setting and objective functions: some focus on reducing added depth or entangling-gate count on single-chip topologies, while Promponas et al. explicitly model the stochastic and time-dependent costs of inter-QPU entanglement and therefore optimize for expected execution time in a DQC setting. While routing in DQC is inherently more challenging due to the need for entanglement operations to enable cross-QPU communication, the strong performance of RL in single-chip qubit routing suggests that similar approaches could be promising for learning efficient policies in distributed quantum systems.

3

Base Model

Routing qubits and managing EPR pairs in distributed quantum computing (DQC) is a challenging task. Promponas et al. propose a reinforcement learning (RL) approach that schedules SWAP and EPR operations to minimize overall circuit execution time [13]. In this approach, an RL agent interacts with a custom DQC environment that evolves in discrete timesteps. At each timestep, the agent may select multiple operations between qubits to execute in parallel, provided they respect hardware constraints as a qubit can participate in only one operation at a time. The operations that can be executed between qubits include SWAP, EPR pair generation (*generate*), tele-qubit, tele-gate, and CNOT gate (*score*).

The *generate* operation can only be executed along a quantum link connecting two different QPUs in order to generate an EPR pair linking the two QPUs. All other operations are executed along an edge connecting two qubits which are on the same QPU. Even though the tele-gate and tele-qubit operations influence qubits on different QPUs they are executed between a logical qubit and an EPR pair half that are located on the same QPU. The other EPR pair half is then located automatically. This is done so it is clear which qubit is being teleported or which gate is executed.

The objective is to complete all CNOT gates in the quantum circuit before a predefined decoherence deadline T and thereby *solving* the circuit. This requires the agent to coordinate both spatial routing and temporal scheduling. This deadline serves as a proxy for decoherence, as compilation time has to be within a coherence time determined by the quantum hardware [14]. In reality, decoherence is a probabilistic and continuous process influenced by various hardware-specific factors, and it may occur before or after any arbitrarily chosen threshold. In the work of Promponas et al. a deadline of $T = 1500$ timesteps is used, though this value is not derived from physical constraints. Rather, it provides a simplified model that reflects the increasing likelihood of decoherence over time. While compiling explicitly against hardware-specific decoherence models is a more complex task, it is not the focus here. Instead, the compiler is designed to minimize total execution time under the assumption that faster execution reduces the risk of decoherence-related errors.

(a) Matching A (b) Matching B

Figure 3.1: Two different matchings on the same graph connectivity.

This routing problem in DQC is formalized as a Markov Decision Process (MDP) providing a general framework for decision-making under constraints. This framework is used to create reinforcement learning model. At each timestep, the agent selects an action in the form of a matching, i.e., a subset of edges in the network where no two edges share a node as shown in Figure 3.1. This selection ensures that qubit operations can be performed in parallel without conflicts. The action space consists of all possible different matchings. Each

Table 3.1: Timesteps it takes to execute each action

Action	Execution time
Score	1 timestep
SWAP	3 timesteps
Generate	5 timesteps
Tele-qubit	5 timesteps
Tele-gate	5 timesteps

edge in a matching is assigned a label indicating a specific two-qubit operation: do nothing, apply a SWAP gate, apply a CNOT gate, teleport a gate, or teleport a qubit. These actions apply only to physical edges between qubits located on the same QPU. For edges that are quantum links, the possible actions are limited to either generating an EPR pair or doing nothing. The agent continues selecting matchings at each timestep until the quantum circuit is fully executed.

Operations between qubits require different numbers of timesteps to complete, depending on how many quantum gates they consist of. The number of timesteps it takes to execute each actions can be seen in Table 3.1. To account for these differences, *cooldown* periods are introduced to indicate when a qubit becomes available again after participating in an operation. For example, when a SWAP is executed the two qubits involved are given a cooldown of 3 timesteps, meaning only after the environment has progressed 3 timesteps can they be interacted with again.

To sum up, the MDP's state includes the qubit locations, the circuit's DAG, and the qubit cooldowns and the actions are all possible matchings. However, the state and action spaces of the MDP are too large to be efficiently handled by reinforcement learning models. The size of the state space, only including the qubit mapping and cooldowns, would already be

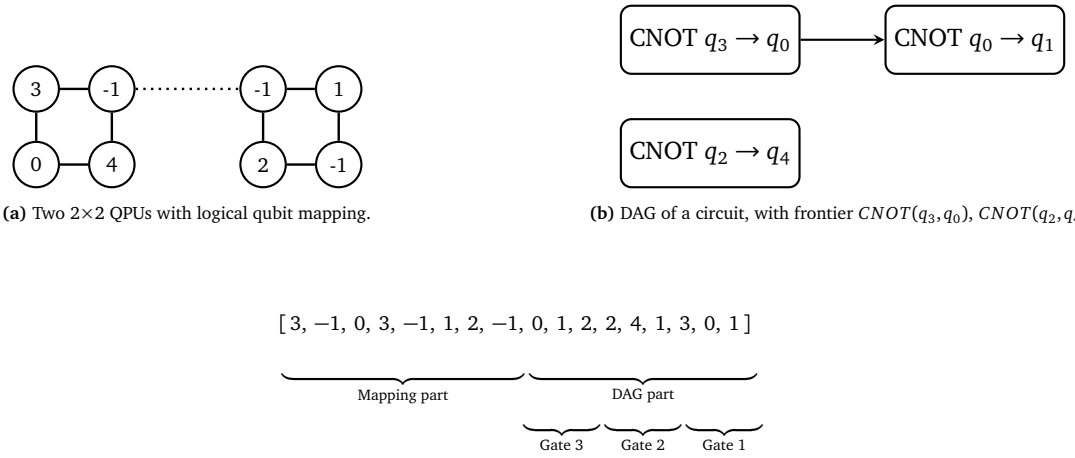
$$(|V|!/(|V|-|Q_t|)!)(|E_t| \star C_{max})$$

where V is the amount of physical qubits, Q_t the amount of logical qubits, $|E_t|$ is the set of possible operations and C_{max} is the maximum cooldown that can be set by these actions and this is without considering the different amount of quantum circuits there can be. If the layer number is ignored there can be $\binom{Q_t}{2}^G$ possible circuits, where Q_t is the amount of logical qubits and G the amount of gates. Therefore, Promponas et al. introduces approximations to reduce the complexity.

First, the action space is redefined. Instead of choosing from all possible matchings, the agent now selects a single action for any given edge. The available actions for a physical edge include: applying a SWAP gate, applying a CNOT gate, teleporting a gate, or teleporting a qubit. For a quantum link, the available action is to generate an EPR pair.

Secondly, another approximation involves omitting the exact cooldown times for each link from the state representation. Instead, a binary mask vector is used to indicate the availability of each action based on the current state. Any action involving a qubit or link that is currently on cooldown is masked out. For neural network–based agents, this mask is applied to the Q-values at the output layer, setting the Q-values of unavailable actions to zero. This approximation simplifies the state space significantly, as the exact timing of link availability offers little benefit to the agent's decision-making.

Lastly, the actions score and tele-gate, which were previously part of the action space used to execute circuit gates, are now performed automatically whenever possible. The environment monitors each gate in the frontier and checks if the involved qubits are either neighbours on the device or both connected to halves of the same EPR pair. When these conditions are met, the corresponding gate is executed automatically. Gate execution bypasses cooldown restrictions, instead the execution time for a CNOT or tele-gate is simply added to the existing cooldown timer. Furthermore, a *stop* action is introduced, allowing the agent to signal the end of the current timestep's action selection and proceed to the next step. With this modified action space, the agent effectively constructs a matching at each timestep and uses the stop action to finalize it instead of choosing a matching as an action, this reduces the actionspace.



(a) Two 2×2 QPUs with logical qubit mapping.

(b) DAG of a circuit, with frontier $CNOT(q_3, q_0)$, $CNOT(q_2, q_4)$.

Figure 3.2: Example of a state vector for a state with a qubit mapping on two 2×2 grid QPUs connected by a quantum link (a) and DAG of a simple quantum circuit (b).

The state vector is reduced to include only the qubit location vector and the circuit DAG and thus the mapping part of the state has been reduced to:

$$(|V|!/(|V|-|Q_t|)!)$$

where V is the amount of physical qubits, Q_t the amount of logical qubits. The DAG part of the state is still the same size as before as it is unchanged. Figure 3.2 shows an example state vector for an arbitrary qubit mapping and a DAG. In the qubit mapping part of the state vector the index represents the physical qubit and the value the logical qubit which is mapped to it; if that value is -1 it means there is no logical qubit mapped to that physical qubit, i.e., it is empty. In the DAG part of the state vector every three-value tuple represents a gate, where the first two values represent the controller and target (logical) qubits of the CNOT gate and the third value in which layer of the DAG the gate is in. The action space now consists of: a stop action, one swap and teleport-qubit action per physical link, and one generate action per quantum link between QPUs. This reduces the actionspace to the dimension $1 + Eq + 2En$, where Eq is the number of quantum links which connect different QPUs and En is the number of physical edges between qubits. The binary mask vector has the same length as the action space and indicates which actions are valid at a given timestep.

During training, the agent aims to maximize its future reward, so the reward for each action should reflect the agent's progress toward the overall goal. The agent therefore receives most of its reward from completing, or scoring gates: $R_{\text{score}} = 500$. Successfully completing *all* gates provides an additional reward, $R_{\text{success}} = 3000$. If the agent fails to complete the circuit (i.e., it reaches the deadline), it receives a large penalty, $R_{\text{fail}} = -3000$.

Because the agent should solve circuits as quickly as possible, the stop action is penalized. Each stop action carries a small negative reward, $R_{\text{stop}} = -20$, so minimizing unnecessary stops leads to a higher final reward.

For qubit movement (via SWAP or tele-qubit actions), the agent should be rewarded when it brings qubits that belong to the same gate in the DAG's frontier closer together. Since the goal of routing is to position qubits such that required gates can be executed, a distance-based reward shaping scheme is used. The distance between the qubits of each gate in the frontier is calculated on a weighted graph: physical edges have weight 1, while quantum links have weight w. When an entangled qubit pair is generated, an edge of weight 1 is drawn between its two halves. Summing the distances between qubits of all frontier gates yields a distance metric for the state. By comparing this metric before and after a qubit movement, a reward can be assigned:

$$R_{\text{move}} = \xi\,(d_{\text{metric}} - d'_{\text{metric}}) \quad \text{with} \quad \xi = 18. \tag{3.1}$$

with d_{metric} being the distance metric before executing the action and d'_{metric} the distance metric after.

During the training process, each episode begins with a randomly generated initial mapping and DAG, which together define the initial state and mask vector. This state is provided as input to the neural network (in the case of an NN-based agent), which outputs Q-values for all possible actions. The Q-values are then masked to eliminate any actions that may not be possible due to constraints.

At the start of training, the agent prioritizes exploration, selecting among the possible actions randomly since it has little or no prior knowledge. As training progresses and the agent accumulates experience, it gradually shifts to exploitation, choosing the action with the highest Q-value. Once an action is selected, it is executed, after which the environment determines whether any gates can be applied. If applicable, these gates are executed, and the DAG and frontier are updated accordingly.

After executing the action the agent receives a reward and saves the experience, which includes the state before, the action chosen, the state after and the reward received, to the memory buffer. It also checks whether the predefined number of steps between learning iterations has elapsed; if so, it performs the learning iterations. Finally, the agent verifies whether the episode has ended, this is either when the DAG is empty or time has passed the deadline. If the episode is not yet complete, the state and mask are updated, and the agent repeats the process until the episode concludes. An overview of the training process is shown in Figure 3.3.

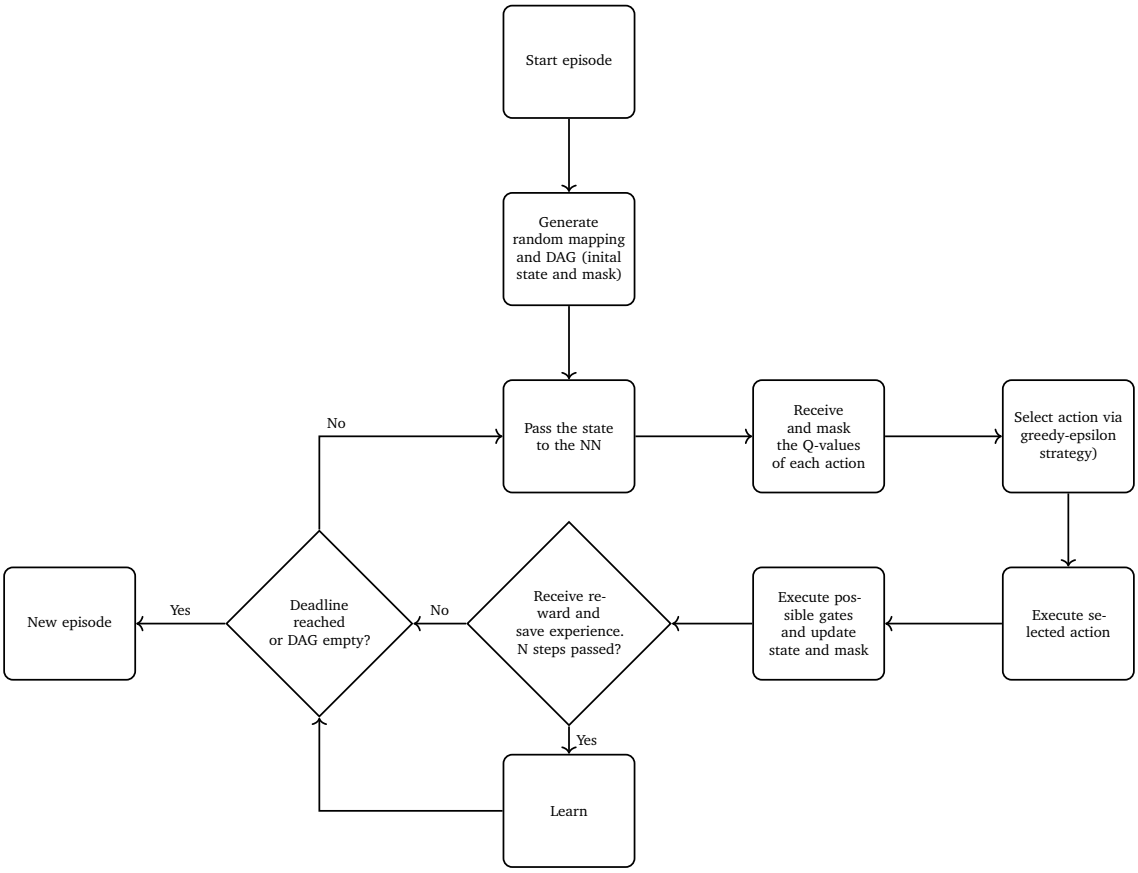


Figure 3.3: Flow diagram of the reinforcement learning cycle.

Multiple different reinforcement learning (RL) algorithms are tested by Promponas et al., most notably PPO, DQN and DDQN. The PPO agent fails to improve and learn the circuit within the set episode limit. However, DQN and DDQN progressively get better and are able to learn to solve the circuit. DDQN performs better than DQN and is therefore highlighted in the paper. This is something that can be expected given DDQN improves upon DQN by decoupling the action selection and action evaluation processes [28]. Given these findings, all subsequent design improvements and architectural changes are developed with the DDQN agent in mind, as it proved to be the most effective for the problem setting.

Promponas et al. found the hyper parameter settings listed in Table 3.2 to be the most efficient for training the DDQN agent in their report. Figure 3.4 shows the moving average of the cumulative reward and solving time of each episode during training for different circuit size (30, 40 and 50 gates). These results show that agent learns to solve circuit comprised of 30 gates. The use of a moving average helps lessen the impact of outliers and produces a smoother learning curve, making long-term trends easier to interpret. However, it can also obscure important details of the agent's performance: sharp fluctuations, occasional failures, or rare successes

Table 3.2: Hyperparameters used for training DDQN agent

Hyperparameter	Value
Learning rate (lr)	0.00001
Batch size (β)	2560
Memory buffer (BS)	100000
Epsilon decay denominator (ϵ_d)	50
Discount rate (γ)	0.99
Target network update parameter (τ)	0.001
Learning frequency	Every 5 actions
Number of learning iterations	10

may be hidden, giving the impression of more stable learning than what actually occurred. Additionally, the choice of window size for the moving average can significantly affect how the results appear; a large window smooths aggressively but may delay showing meaningful improvements, while a small window may fail to reduce noise effectively. Without complementary information such as standard deviation, it is difficult to assess the true consistency and reliability of the agent's learning progress.

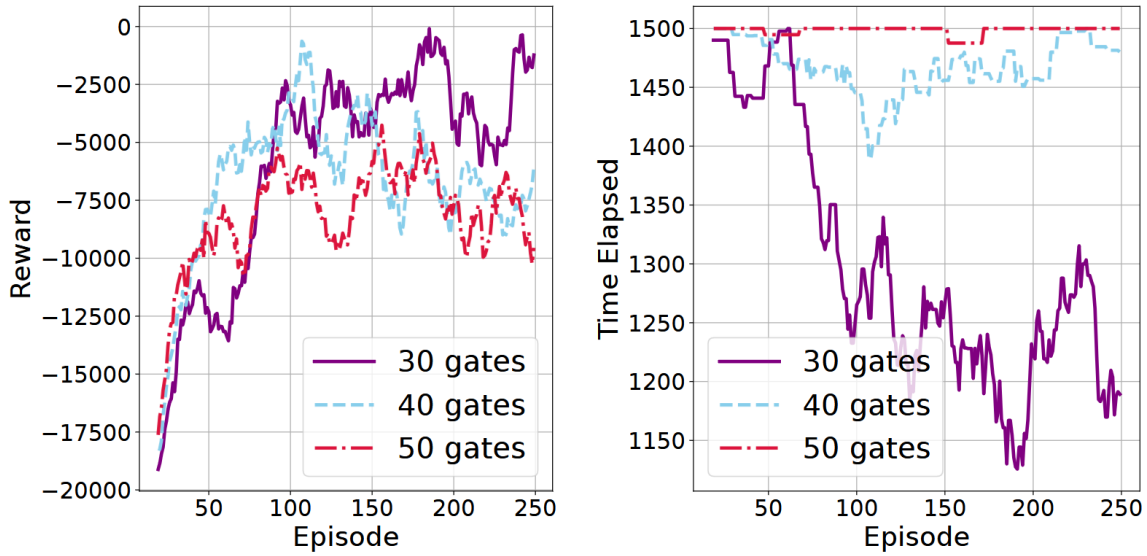


Figure 3.4: Time evolution of (a) cumulative reward and (b) time elapsed until successful quantum circuit compilation or deadline expiry, for random circuits comprising 30, 40 and 50 CNOT gates.[13]

These results are after implementing approximations and significantly reducing state and actions sizes from the MDP formulation. Consequently, the reinforcement learning-based routing model is not scalable enough to support larger, more complex quantum circuits that will be required in future quantum processors.

4

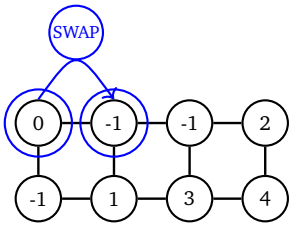
Implementation

To address the limitations in scalability, and consistency in solving times observed in Promponas et al.'s agent, several key modifications were made to the action space and the masking mechanism of the environment. The following changes aim to improve the agent's ability to generalize across circuits and handle larger problem instances by reducing the solution depth, decreasing the branching factor of the decision tree, and lowering the number of weights in the neural network. Reducing the solution depth decreases the number of actions required to reach the goal, while lowering the branching factor limits the number of available actions per state, making learning more efficient [37]. This is achieved by redefining the actions to allow the agent to directly move any qubit to any location as described in Section 4.1. Additionally, Section 4.2 introduces a new neural network layer that computes Q-values for qubit pairs based on the Q-values of the individual qubits involved. This significantly reduces the number of network parameters, which contributes to improved scalability [38]. Lastly, by restricting available actions to those that are likely to move the state closer to the goal as described in Section 4.3 the branching factor is significantly reduced.

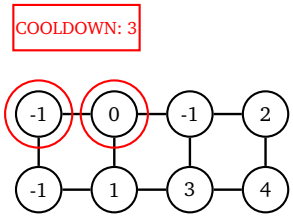
4.1. Reimagined action space

Currently, the model is able to compile circuits with up to 30 gates, and for these, the agent is likely far from achieving the optimal execution time. On average, solving a 30-gate circuit requires more than 1,100 timesteps. As circuit sizes increase, execution time grows further; for 40- or 50-gate circuits, it often exceeds the 1500 time limit and the agent is unable to learn a policy and improve upon execution time. A proposed extension by Promponas et al. is to allow an agent trained on 30-gate circuits to generalize to larger circuits. This could be achieved by dividing the circuit into smaller blocks, where the final qubit mapping of one block serves as the initial mapping for the next. However, even with this approach, the current agent would struggle due to its inefficiency. Therefore, the action space was redesigned to make the learning task easier and enable the agent to solve circuits more efficiently. If the agent is able to solve circuits more efficiently it should either be able to solve larger gate circuits within the given time limit T or should solve smaller circuits fast enough, such that solving larger circuits by dividing them into smaller circuits becomes feasible.

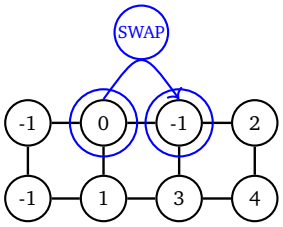
To bring logical qubits closer together, Promponas et al.'s model requires the agent to execute multiple individual actions. Consequently, achieving the final goal necessitates navigating a deep decision tree, as many intermediary steps are required. To mitigate this depth and simplify the decision-making process, the action space is redefined. Rather than selecting actions for each individual edge, the agent is now permitted to move a qubit directly towards another qubit. The latter is achieved by introducing an action for every physical qubit pair, where the qubit mapped to the first qubit in the pair is moved to the second qubit in the pair by implementing multiple consecutive SWAPs and potentially a tele-qubit operation if the qubits are on different QPUs. This modification allows the agent to perform complex sequences with a single high-level action, thereby significantly reducing the overall solution depth. For example, moving a qubit four qubits further would have taken 9 actions (swap + 3 stops + swap + 3 stops + swap) before the change as shown in Figure 4.1, since each swap has to be performed individually the agent must perform three stop actions in between every swap to reduce the cooldowns. With the new action this can be done in a single action as shown in Figure 4.2.



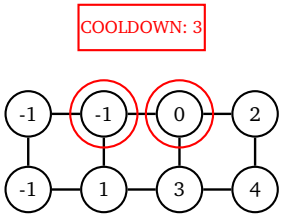
(a) First SWAP.



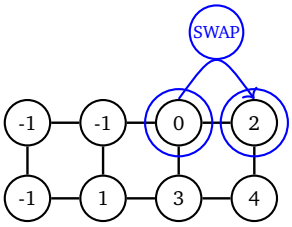
(b) Cooldowns blocking next SWAP



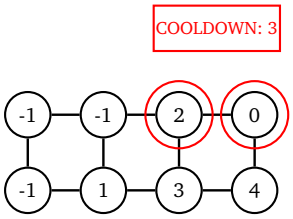
(c) Second SWAP after 3 stop actions.



(d) Cooldowns blocking next SWAP

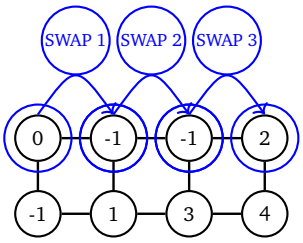


(e) Final SWAP after another 3 stop actions.

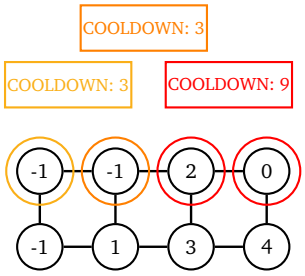


(f) Cooldowns at the end of sequence

Figure 4.1: Number of actions required to move a qubit three places and the cooldowns afterwards by the base actionspace, a, c and e show the SWAPs and b, d and f show the cooldowns after the SWAPs.



(a) All SWAPs in a single action.



(b) Cooldowns at the end of sequence

Figure 4.2: Number of actions required to move a qubit three places and the cooldowns afterwards by the novel actionspace, a show the SWAPs executed in a single action and b the cooldowns after.

When moving a qubit to another location this need to happen in the least amount of swaps possible. Therefore the actions which have to be executed is determining the shortest path, using Dijkstra's algorithm [39], from the source qubit to the target location. The action on the path consists of a series of swap operations and, if the target is on a different QPU, includes a tele-qubit operation as well. These operations are applied sequentially in a single step, with the corresponding cooldowns being applied immediately. As a result, the qubit locations in the new state reflect the completed swaps, but the affected qubits remain on cooldown and cannot be interacted with until the operations would realistically have finished.

In this design, the entire swap (and tele-qubit) chain is executed in a single step, rather than in real time. All operations are applied at once, and the cooldown of each qubit is determined by how which swaps it participates in along the chain. For example, the source qubit only takes part in the first swap and therefore receives a cooldown of 3, the second qubit participates in the first and second swaps and receives a cooldown of 6, and the third qubit participates in swaps two and three and receives a cooldown of 9, and so on as seen in Figure 4.2. Of course, in an actual device these swaps could not occur simultaneously, since each

swap depends on the previous one finishing. An alternative implementation could therefore execute only the first swap immediately and queue the remaining swaps, allowing them to play out step by step in real time. However, this queuing approach makes it harder for the agent to reason about states, because it must distinguish between changes caused by its most recent action and those still propagating from earlier ones. By executing the entire chain in a single step and applying cooldowns up front, the agent always observes a consistent and unambiguous state.

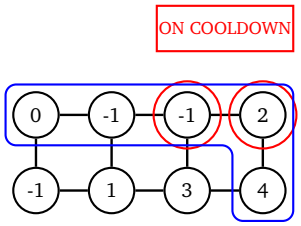


Figure 4.3: Qubit cooldowns blocking shortest path between two qubits

Actions must still be masked similar as before, since certain actions are invalid if any qubit along the path or the qubit being moved is currently on cooldown as seen in Figure 4.3. While the environment computes paths without considering cooldowns, it might seem beneficial to reroute around qubits that are temporarily unavailable. For example, in Figure 4.3 if the agent wanted to move qubit 0 to 4 another path could be taken than the one highlighted. However, the agent itself has no awareness of cooldown states. This means that the same input state and selected action could result in different outcomes, depending on which path is chosen by the environment. This introduces non-determinism into the system, which is problematic for a DDQN agent. Since DDQN relies on consistent state transitions to learn from experience, non-deterministic actions undermine its ability to associate actions with reliable outcomes and hinder effective learning. Therefore, qubits are always routed along the same shortest path, and actions that involve a qubit on cooldown are masked out.

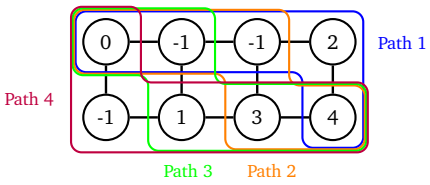


Figure 4.4: Different shortest paths between qubits mapped to 0 and 4.

In case of multiple shortest paths, something which is common in connectivities of 2D grid-like architectures as highlighted by Figure 4.4, the environment still only considers one shortest path for an action of multiple swaps. The agent is still able to route a qubit along all shortest paths that exist between source and destination, however it may have to execute more than one action in order to route qubits along the desired path. For example, consider the paths in Figure 4.4, routing a qubit along path 1 can be done in one action, but path 4 will require 2 actions, and path 2 and 3 will require 3 actions. In the worst case the agent may have to execute each swap in the path with individual actions. So in the worst case it will take the agent just as many actions to route a qubit as with the actionspace of Promponas et al. However, in the best case the agent is able to do it in a single action.

The actionspace could be increased to include actions for all possible paths between two physical qubits. This could work in cases where there are only a couple extra paths. However in high connectivity architectures, like a grid for example, this would add a significant number of actions. With the actionspace where the actions only follow one shortest path the solution depth in a worst case scenario is as deep as before but should in almost all cases be less deep. These extra actions are therefore not added to the actionspace.

To enable the agent to move empty physical qubits, for example, to generate or transport EPR pairs, the action space is defined over pairs of physical qubits rather than logical qubits. While defining actions over logical qubit pairs would result in a smaller action space, it would prevent the agent from taking actions involving unmapped or empty qubits, which are essential for reliable EPR pair generation and movement. An alternative would be to number the unmapped qubits so they can be included in a logical-qubit-based action

space. However, this would require the state space to explicitly represent these identifiers, allowing the agent to distinguish between different empty qubits. Doing so increases the complexity of the state space, leading to a larger number of possible states and thereby widening the decision tree the agent must search through during learning.

To further reduce the solution depth the stop action was modified to reduce cooldowns until the action mask changes or all cooldowns reach zero. This change was introduced because, in the original setup, the stop action only reduced cooldowns by one time step per use. In cases where no additional valid actions become available after a single stop, the agent would be forced to select multiple consecutive stop actions before cooldowns of a qubits are removed, the mask changes and additional actions become available to the agent. Since the cooldown for a SWAP is three, this is something which often occurs. This artificially inflates the decision depth without contributing meaningful decision-making steps. By allowing the stop action to continue until the set of available actions changes, the agent can reach the same effective state in a single step, thereby simplifying the learning process and improving training efficiency. The reward for the stop action, as explained in Chapter 3, has been modified to account for the change in timesteps: where it was previously defined as R_stop it is now $R_stop \cdot t$, with t representing the increase in timesteps. Thus if, for example, a stop action reduces the time by three timesteps the reward is the stop reward multiplied by three.

The dimension of the previous actionspace is

$$1 + E_q + E_n$$

where E_q is the number of quantum links which connect different QPUs and E_n is the number of physical edges between qubits. The dimension of the new actionspace is

$$1 + E_q + \frac{N(N-1)}{2}$$

where E_q is the number of quantum links which connect different QPUs and N is the number total physical qubits.

4.2. Neural network adaption

The changes to the actionspace reduce the overall solution depth, which should make it easier for the agent to learn and reach better solutions by requiring fewer sequential decisions. However, the new action space is significantly larger than before. This in turn increases the branching factor of the decision tree. This broader set of possible actions makes it more difficult for the agent to explore effectively and identify the best decisions [37]. Additionally, a larger action space increases the number of output neurons in the neural network, resulting in more trainable parameters.

An increased number of weights has several important consequences. First, it raises the memory requirements and computational load during both training and inference, using the agent after training, which can slow down learning and execution times [40]. Second, it requires more training data to generalize well; otherwise, the model will only learn to solve specific circuits and not find a policy which works for any arbitrary circuit. Finally, larger networks typically converge more slowly [41].

To reduce both the size of the output values and the number of weights in the neural network, the agent is designed to output a value for each physical qubit individually, rather than for every pair of qubits. The intended behaviour is for the agent to select two qubits—the first to be moved and the second as the destination. This means the amount of actions the agent can choose are the same before, however the amount of output values the neural network has are reduced to

$$1 + E_q + N$$

Where E_q is the amount of quantum links and N is the amount of physical qubits. This significantly reduces the amount of weights of the final layer. It also makes it so the agent only has to optimize the Q-values of these output actions.

This should improve learning, not by reducing the number of distinct decisions the agent must make, but by decreasing the number of Q-values the network has to predict directly. With fewer outputs, the learning

problem becomes simpler and more data-efficient, which can lead to faster convergence and a policy which works better for all circuits [24].

However, directly implementing this approach, where the qubits with the highest Q-values determine which qubit is routed where, is not realistic, since masking individual output actions (e.g., blocking moves that involve qubits on cooldown) becomes infeasible in such a two-action selection process. For instance, if qubits 3 and 5 have the highest Q-values, the agent would select the action of moving the logical qubit mapped to qubit 3 to qubit 5. Yet, if any qubit along the path between 3 and 5 is on cooldown, the action cannot be executed and must instead be rejected by the compiler. Properly masking out these invalid actions is overly complex, making this approach impractical.

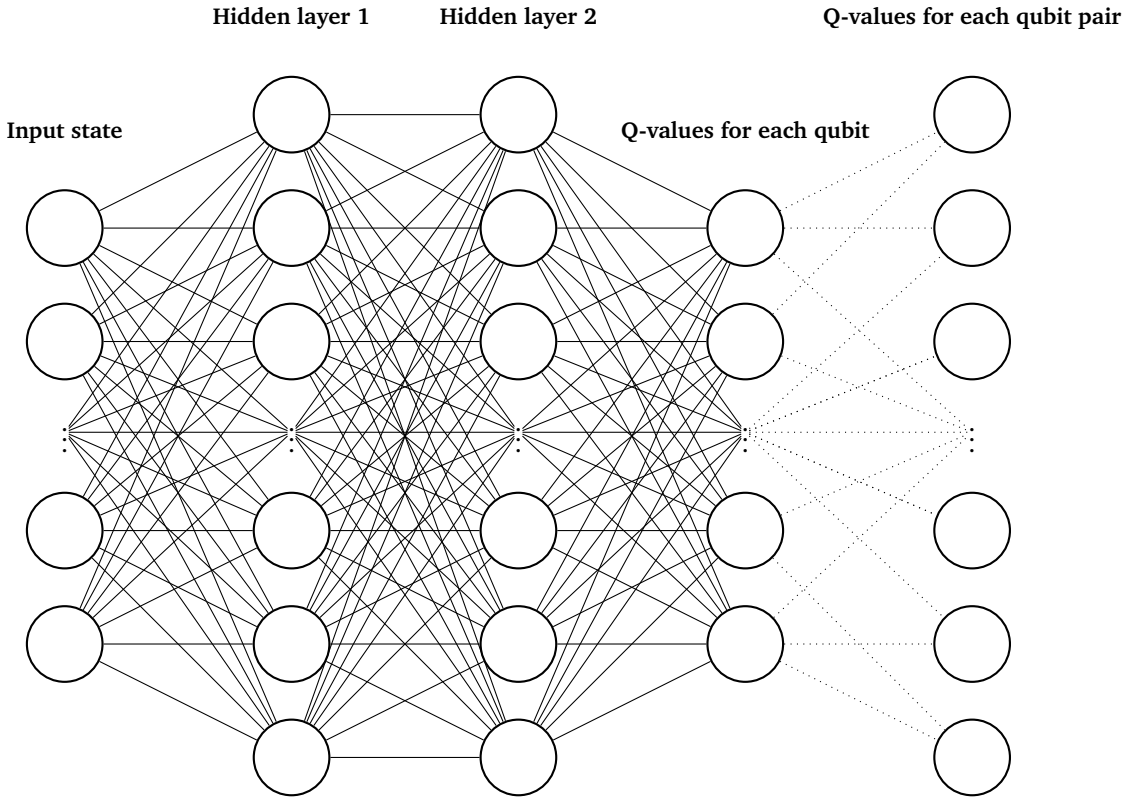


Figure 4.5: Feed-forward neural network with an additional layer. In this layer Q-values for physical qubit pairs are computed from the Q-values of individual qubits, the weights in this layer are set and not trained. Stop and generate actions are not shown, their Q-values would remain unchanged after the single-qubit Q-value layer. Each circle represents an input value, neuron, or Q-value, while the smaller dots indicate additional inputs, neurons, or Q-values not drawn explicitly. Each line is a trainable weight, except for the dotted lines.

Therefore, this change is implemented differently such that actions can still be masked. The Neural Network is still changed to output Q-values for each qubit instead of each qubit pair. Additionally, a new layer is introduced into the agent's neural network. This layer computes action values for physical qubit pairs by combining the single-qubit Q-values through a fixed, weightless function. An example of the architecture with this additional layer is shown in Figure 4.5. The Q-values of non-pair actions (e.g., stop or generate) are passed through unchanged. For a qubit pair, the Q-value is computed as

$$Q_{\text{pair}} = 0.75Q_1 + 0.25Q_2,$$

where Q_1 is the Q-value of the first qubit in the pair and Q_2 the Q-value of the second. The weighting was chosen to ensure that pairs have distinct Q-values depending on their direction (e.g., pair 1–2 does not equal pair 2–1), and is kept fixed throughout training .

In this design, the model still preserves the same pairwise action structure as before. The neural network's output dimension remains

$$1 + E_q + \frac{N(N-1)}{2},$$

where E_q is the number of quantum links connecting different QPUs and N is the total number of physical qubits. However, these outputs are now computed from a smaller set of intermediate values produced by the previous layer, which has dimension

$$1 + E_q + N.$$

Here, E_q again denotes the number of quantum links and N the number of qubits. Since the final layer has no trainable weights, the total number of neural network parameters now scales linearly with N, rather than quadratically. This architectural change should significantly improve training efficiency and learning performance, as it reduces the number of parameters that must be optimized during training.

4.3. Action masking

While the new neural network architecture improves training efficiency by reducing the number of weights, the effective branching factor of the decision tree still increases rapidly as the number of qubits grows. This large action space, combined with the high cost of certain actions, makes sampling effective experiences for training the agent difficult. In particular, actions that involve multiple SWAP operations can impose significant cooldown periods on the qubits involved. For instance, an action requiring a chain of 4 SWAPs will result in the two qubits involved in the final SWAP incurring a cooldown of 12 timesteps, limiting their availability for future operations and thus constraining the agent's options in subsequent steps.

To address this, the action mask is further refined to limit the agent's choices to only those actions that are likely to contribute to finding a solution. Specifically, the mask now allows:

- Actions which move one qubit that is part of a gate located on the DAG's frontier along all shortest paths towards or to a qubit such that it neighbours the other qubit in that gate;
- Actions that move empty qubits along all shortest paths towards or to a physical qubit which is linked to another QPU, such that EPR generation is made possible;
- Actions that move EPR pair halves and qubits involved in frontier gates along all shortest paths towards one another.

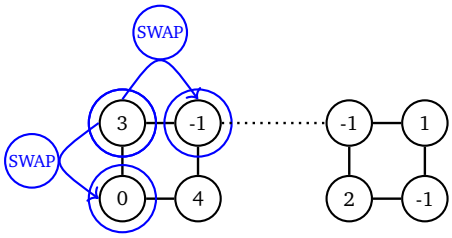
Examples of which actions are and which actions are not allowed in different scenarios can be seen in Figure 4.6.

It is important that the agent is allowed to move qubits along all shortest paths as otherwise the agent is limited to only one path. This may not be problematic for connectivities of circular architectures, where there is often only one shortest path. But could pose significant issues for 2D grid-like architectures where there may be several shortest paths.

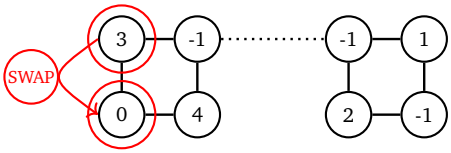
By filtering for more relevant and productive actions, this masking strategy reduces the number of poor or misleading choices available to the agent. As a result, the agent can focus its exploration on actions that are more likely to produce progress toward the goal, even if they are not strictly optimal. This encourages deeper and more effective exploration without incurring heavy penalties for slightly suboptimal moves, ultimately improving the agent's ability to learn efficient solutions [42].

This filtering mechanism is implemented by identifying all valid paths that align with the agent's overall objective, as previously discussed. Once these paths are established, the action mask is constructed to allow only those actions that either correspond to valid subpath of a shortest path or represent the entire path. All other actions that do not contribute meaningfully toward the goal, based on these defined paths, are excluded from consideration.

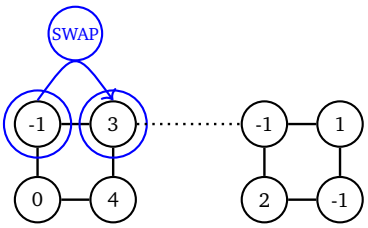
As before, actions that are inherently invalid due to system constraints, such as qubit cooldowns, are also masked out to ensure physical feasibility. Additionally, a specific constraint is enforced to block actions that would move a logical qubit to another QPU if doing so would eliminate the last remaining empty qubit on the destination QPU. This restriction prevents scenarios where a QPU becomes fully occupied by logical qubits and can no longer generate EPR pairs, rendering it incapable of interacting with other QPUs. Such cases could leave parts of the circuit unsolvable, effectively causing the environment to reach a dead-end. If the agent reaches such a dead end state, all subsequent states are also dead end states, and the experience it receives from actions after reaching a dead end state negatively impacts the agents ability to learn [43].



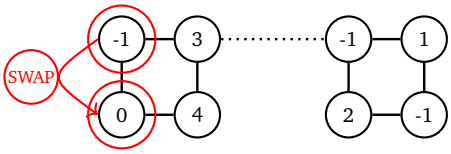
(a) Example of valid actions for moving qubit three if CNOT(3,4) is in the frontier (towards gate qubit).



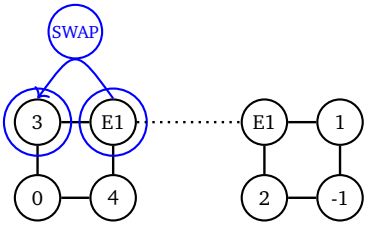
(b) Example of invalid action for moving qubit three if CNOT(3,1) is in the frontier (away from gate qubit).



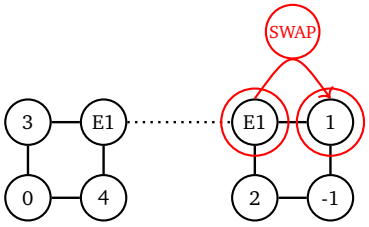
(c) Example of valid actions for moving empty qubit (towards quantum link).



(d) Example of invalid action for moving unmapped qubit (away from quantum link).



(e) Example of valid actions for moving EPR pair half if CNOT(0,1) is in the frontier (towards gate qubit).



(f) Example of invalid actions for moving EPR pair half if CNOT(0,1) is in the frontier (away from gate qubit).

Figure 4.6: Examples of valid (a),(c),(e) and invalid (b), (d), (f) actions for moving logical qubits, empty qubits and EPR pair halves.

An alternative implementation would be to allow dead-end actions to occur but instantly give a negative reward when this happens and end the game. These negative rewards could theoretically teach the agent to avoid them and ending the game instantly would mean the agent does not learn from actions in a dead-end state. However this approach relies on frequent enough encounters and may result in unproductive episodes. Blocking ensures that the agent remains in valid, solvable states throughout training, accelerating learning.

Combined, these updates have reduced the number of neural network weights, decreased the depth of the decision tree, and did not negatively affect the effective branching factor. As a result, the agent has fewer Q-values to optimize and encounters a simpler action selection process, which should lead to faster training times, more stable convergence, and improved scalability for larger circuits.

5

Evaluation

In order to demonstrate the effectiveness of the new implementation and design [1], the model is tested with different architectures and constraints. Most of the hyper-parameters are based on the baseline implementation [13] as listed in Table 3.2 in Chapter 3. The only change is in the size of the neural network, which is still a multi-layer perceptron with two hidden layers, but the first layer is now 90 neurons and the second layer 80 neurons compared to the 150 and 140 neurons used in the previous work. Although the network size differs from the baseline implementation, this is considered a fair comparison since the objective is to evaluate the performance of the complete system rather than individual implementations in isolation. The modified network size was selected to allow a meaningful assessment of its overall performance. The network size is a hyperparameter that can be tuned and is not a change to the fundamental design or implementation of the model. The network also has the additional final layer which converts the q-values for every physical qubit into q-values for every qubit pair. This training setup, including the hyperparameters, is used for all results unless stated otherwise. In the results that follow, we report a moving average of 10 timesteps and corresponding standard deviation. The moving average helps reveal the overall learning trend by smoothing out short-term fluctuations, while the standard deviation indicates the variability and consistency of the agent's performance. This is done to properly evaluate the models capabilities.

This chapter is structured as follows: Section 5.1 evaluates different reward strategies for qubit movement actions. Section 5.2 examines how the model performs when trained on circuits of varying sizes and compares these results to those of the previous model. In Section 5.3, the generalization ability of models trained on 30-, 40- and 50-gate circuits is analysed by comparing their performance on compiling circuits after training and comparing with the base model. Section 5.4 assesses the model's performance across various distributed quantum computing (DQC) architectures, again benchmarking against the baseline model. Finally, Section 5.5 investigates the impact of different action elimination strategies on circuit depth and compilation efficiency.

For the greater part of the evaluation (Sections 4.1, 4.2, 4.3 and 4.5), the model is trained on two connected IBM Q Guadalupe QPUs as shown in Figure 1.1, which is also used to train the original model. This system has 32 physical qubits and training is done with 18 logical qubits such that not all logical qubits fit on one QPU. All results are obtained on the same workstation, equipped with an AMD Ryzen 7 5700G CPU (8 cores, 3.8 GHz) and an NVIDIA RTX 3070 GPU.

5.1. Different SWAP rewards

First, the model is tested with different reward shapings for qubit movement. Previously, Promponas et al. implemented a reward system for SWAPs which gave positive reward to actions that move qubits in gates of the frontier closer together and negative reward if they move further away. This was determined via a distance metric, a score for how close qubits of gates in the frontier are to each other, where the difference between this score before and after an action determines the reward as described in Equation 3.1. This was done to encourage the agent to move a qubit in a certain direction and speed up learning.

[1] The codebase can be found at: `https://github.com/joost-vanveen/CompilerDQC`

However, this scoring may be redundant with the changes introduced to the masking strategy since the agent is now only able to select actions which should provide progress to the goal. Therefore encouraging or punishing certain actions may not make sense since all actions may provide progress to the goal even if the actions return negative reward. For example, in some cases when moving empty qubits for EPR generation qubits in the frontier are moved further away from each other. This may be necessary in order to generate an EPR pair, but the agent will still get a negative reward for this action.

For this reason three different reward implementations for SWAPs are compared:

- **Distance metric reward** - Rewards for SWAPs are the same as before by comparing the distance metric before and after executing the action.
- **Positive only reward** - Rewards for SWAPs are calculated using distance metric difference, however any negative difference would not result in negative reward but zero reward.
- **Zero reward** - Reward for all SWAPs are zero. This way the only rewards the agent receives are from scores, stop actions and reaching the deadline or solving the circuit.

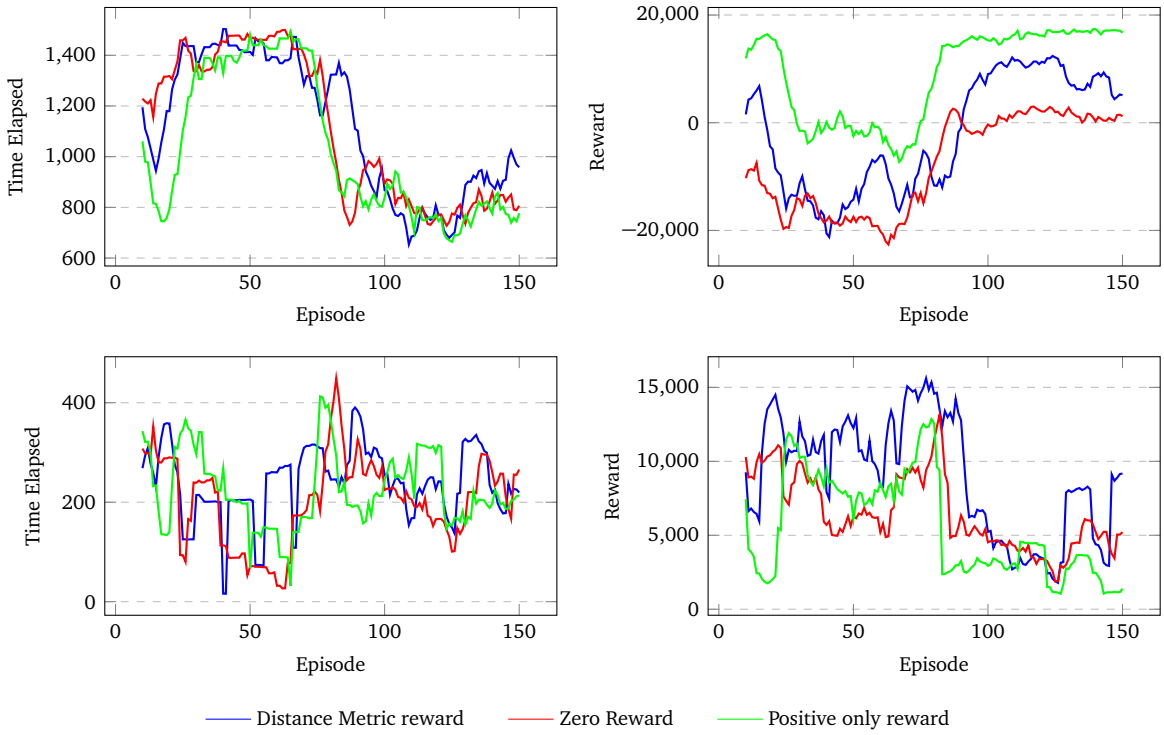


Figure 5.1: Comparison of moving average (top row) and standard deviation (bottom row) of time and reward evolution across reward shaping variants.

In Figure 5.1 the circuit execution time and cumulative reward per episode for training with different reward shapings are plotted. From these results, it can be concluded that the agent is able to learn to solve circuits regardless of which reward structure is used. This means that the agent understands that completing gates and circuits maximizes long-term reward, regardless of the specific reward assigned to individual SWAP actions. Consequently, the negative reward the agent sometimes receives for moving empty qubits (for example, moving empty qubits for EPR generation) does not discourage those actions, since they may ultimately lead to a higher overall reward. It can be seen that there is a slight increase in learning speed with zero and positive only reward. However, this improvement is minor and could be due to other factors like complexity of the randomly generated circuit. For all subsequent experiments the positive only reward is used.

It is important to note that, for both variants of the distance-metric reward, a larger cumulative reward does not always correspond to a faster solution time when comparing different problems (e.g., different circuits or initial mappings). In some cases, slower solutions actually yield larger cumulative rewards. This happens because they require more SWAP actions, which are positively rewarded due to the distances between qubits

and the amount of cross-QPU communication involved. In such cases, the additional positive reward from these SWAPs outweighs the negative penalty associated with the longer solution time.

However, for the same problem instance, a faster solution would generally produce a larger cumulative reward than a slower one, since the number of SWAPs is fixed and extra delays only increase the time penalty. This effect does not occur with zero-reward shaping, as SWAP actions do not contribute to the cumulative reward in that case.

5.2. Different circuit sizes and comparison to the previous model

First, the introduced model is compared to the baseline model by training both on randomly generated 30-gate circuits. Because the reward function for SWAP actions has changed in the new model, the cumulative reward values are not directly comparable to those of the baseline. A higher cumulative reward in the introduced model does not necessarily correspond to better performance in absolute terms, since SWAPs now contribute differently to the total reward. Instead, comparisons focus on metrics such as circuit execution time and consistency in execution time, which provide a meaningful measure of improvement independent of the reward scaling.

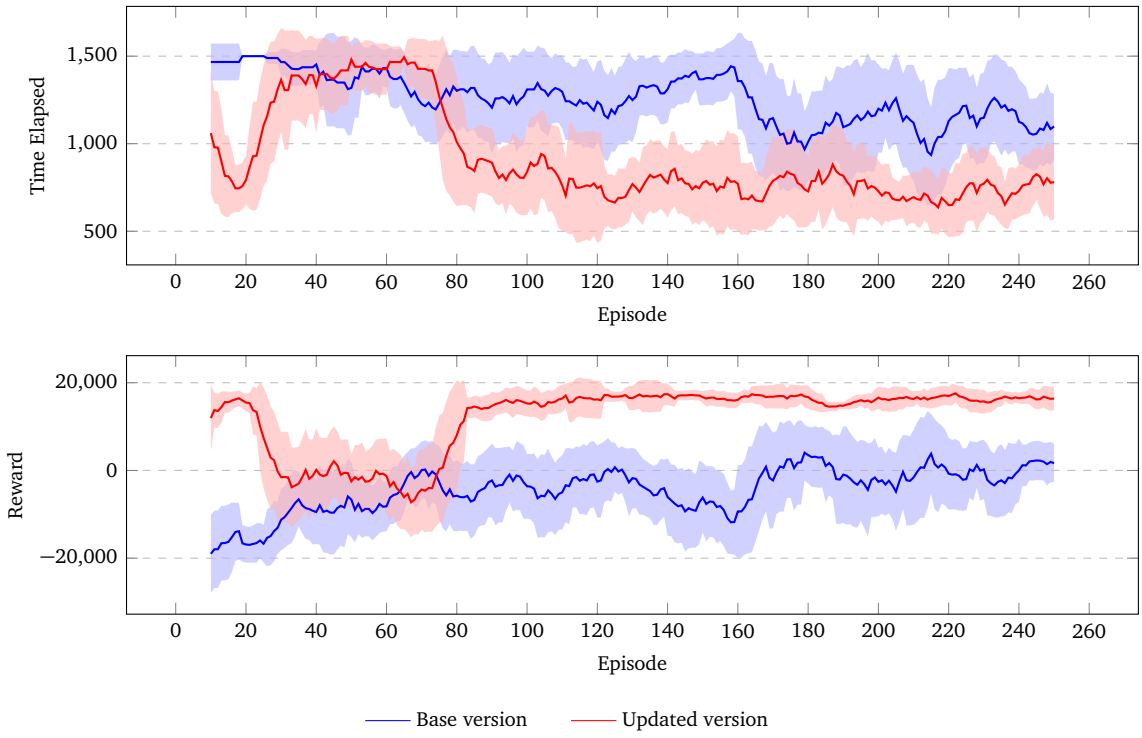


Figure 5.2: Comparison of moving average and standard deviation of time and reward evolution between the original and introduced model trained on 30-gate circuits compiled on two IBM Q Guadalupe QPUs.

Figure 5.2 shows the circuit execution time and cumulative reward per episode for both implementations. The introduced model achieves consistently lower execution times, solving circuits on average $\sim 32\%$ (over the last 50 episodes) faster than the baseline within the training time limit, with average solving times dropping from $\sim$ 1100 timesteps on average to $\sim$ 750 timesteps. Moreover, the introduced model exhibits lower variance in execution time across episodes, indicating that it learns a more general policy compared to the base model.

Lastly, it is important to note the difference in training times between the two models. Both are trained on a workstation equipped with an AMD Ryzen 7 5700G (8 cores, 3.8 GHz) CPU and an NVIDIA RTX 3070 GPU. The baseline model required approximately 66 hours to train, whereas the introduced model completed training in about 23.5 hours. This reduction in training time is largely due to the redesigned action space. By reducing the solution depth, the new action space allows the agent to solve circuits using fewer actions per episode. Because learning updates are performed at the same fixed frequency (every five actions), the

introduced model undergoes fewer learning updates per episode, and therefore fewer updates overall. Since these updates are the most computationally expensive part of training, this results in a significant reduction in total training time.

To further evaluate scalability, the introduced implementation is trained on larger circuits containing 40 and 50 gates. The baseline model was able to solve 30-gate circuits but failed to converge on the larger instances. The results of this experiment reveal whether the introduced changes improve scalability. In the following, the models trained on 30-, 40-, and 50-gate circuits are referred to as the 30G, 40G, and 50G agents, respectively. For the 40G and 50G agents, the neural network architecture is expanded to 120 and 150 neurons per hidden layer, this is done in order to achieve better performance. Based on the 30G results, where the model completed circuits in approximately 800 timesteps on average, we would expect, under an assumption of linear growth, that 40-gate circuits would require roughly 1050 timesteps and 50-gate circuits around 1300 timesteps.

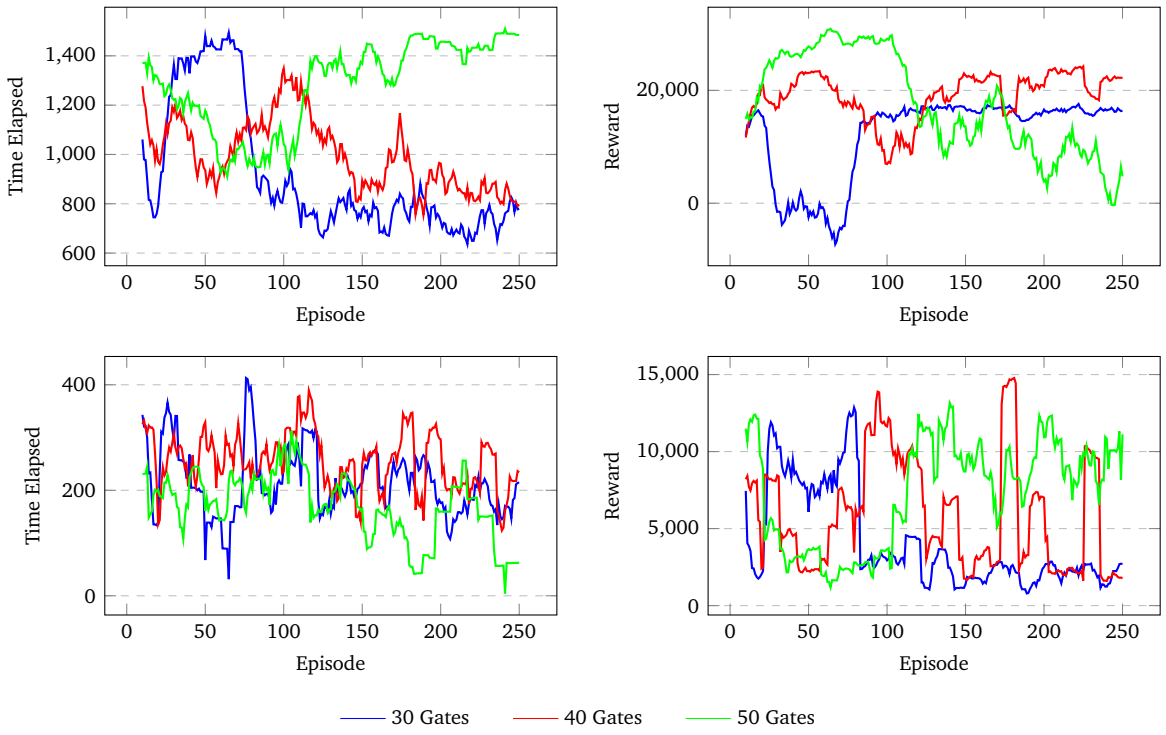


Figure 5.3: Comparison of moving average of 10 timesteps (top row) and associated standard deviation (bottom row) of time and reward evolution across different circuit sizes.

Figure 5.3 shows the circuit execution time and cumulative reward per episode for both implementations. From these results, it is evident that the introduced model is able to learn a policy to complete 40-gate circuits, whereas the previous model failed to learn such a policy within the same episode limit as seen in Figure 3.4. The solving times for the 40G model are better than expected, with the agent completing circuits in nearly the same number of timesteps as for 30-gate circuits.

For 50-gate circuits, however, the model is unable to learn a policy that successfully completes the circuits. In fact, the average number of timesteps required to compile a circuit increases with the number of episodes, contrary to the desired downward trend. The hyperparameters for the introduced models were not extensively optimized and were mostly based on the original work, and may therefore be suboptimal for the 50G case. Still, the results demonstrate a significant improvement in performance of the model over the previous version. These results suggest that a model could potentially be applied to larger circuits than it was trained on by partitioning those circuits into smaller blocks.

5.3. Trained model compiling unseen circuits and larger circuits

Promponas et al. proposed that models which are trained on smaller circuits could be used to compile larger circuits by partitioning circuits into blocks. This would mean the initial mapping of each block would be the final mapping of the previous block. For example consider the circuit shown in Figure 5.4, which is split into two parts. The mapping when the last gate of the first part is executed would be the initial mapping when executing the next part of the circuit. One problem with this implementation would be that the end of one block the agent could not make decisions on the first gates in the next block. For example the decision of moving a qubit to another QPU can be influenced by future gates. If the agent is unable to see future gates, it cannot make decisions with those gates in mind. This could then lead to slower execution times; for example, teleporting a qubit to another QPU when it would have been more efficient to have that qubit remain in the QPU and do a tele-gate instead.

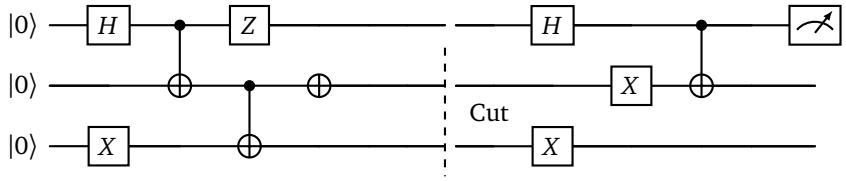


Figure 5.4: Circuit divided into two blocks. The qubit mapping at the end of the first block serves as the initial mapping for the second block.

In order to mitigate this issue, a sliding window approach is implemented. In a sliding window implementation the gates which are in the state and passed as input to the agent would be the first N gates of the circuit which still have to be completed. So after a gate is completed the window would move and include the next gate to be solved. This means that the agent would always see the next N gates which have to be executed. If the smaller circuit model is used with a sliding window implementation, it could potentially be possible for the agent to solve larger circuits without slowing down execution time. This size of the sliding window is determined by how large the circuits are the agent is trained on. For example an agent trained on solving 30 gate circuits can at most see 30 gates of a circuit, therefore its sliding window will have size $N = 30$.

The size of the sliding window is an important parameter as it directly affects what influences the agent's decision-making. For instance, if the agent is trained on a 50-gate circuit, it is unclear whether the final gates meaningfully impact early routing decisions when solving 50-gate circuits. Or if those decisions are driven primarily by the first 10, 20, or 30 gates. This influence likely depends on several factors:

- **The number of logical qubits** - With more logical qubits, it is more likely that there are large gaps between consecutive uses of a single qubit. For example, logical qubit 1 might be involved in the very first gate but not appear again until gate 45. In such cases, the agent would need a longer lookahead, i.e. access to more of the upcoming circuit, to make informed routing decisions early on.
- **The specific structure of the circuit** - The structure of the circuit naturally plays a role as it determines gate order and qubit usage patterns, potentially leading to similar scenarios where qubits must be routed long in advance of their next use.
- **The underlying hardware architecture** - It plays a crucial role, as different connectivity topologies determine how costly routing mistakes are. For example, in IBM's Q Guadalupe QPU, which has a primarily circular architecture, the lack of alternative paths means that correcting a bad routing decision can require multiple sequential SWAP operations to move a qubit back into position. In contrast, in a 2D grid-like architecture, there are usually more routing options and shortcuts available due to higher local connectivity, which can make recovery from poor routing decisions less expensive in terms of time and gate overhead. Thus in architectures with richer connectivity, the agent can often afford to reason over a shorter window, since routing is more forgiving and mistakes can be corrected with less cost.

The models trained on 30-, 40-, and 50-gate circuits, referred to as the 30G, 40G, and 50G agents, are compared based on their ability to solve circuits after training. This evaluation serves two purposes: first, to investigate whether a model trained on smaller circuits can successfully solve larger circuits, and whether later gates have influence on early decision-making; second, to assess how well the trained models generalize to new, unseen problems.

Note, however, that it is not expected that the 50G agent will be able to solve most 50-gate circuits, since it was unable to learn a policy to do so during training. Assuming execution time scales linearly with circuit size and the models perform on new data as well as on training data, the expected average timesteps are shown in Table 5.1.

Table 5.1: Expected average execution time (timesteps) for different agents on circuits of varying sizes.

Agent	30-gate	40-gate	50-gate
30G	750	1000	1350
40G	675	900	1125

For the evaluation, three sets of randomly generated circuits are created, containing 30-, 40-, and 50-gate circuits, respectively. Each set is paired with a corresponding set of random initial qubit mappings, forming a collection of problems for the agents to solve in the same manner as during training. This ensures that each agent attempts exactly the same problems. The baseline model is also evaluated on the 30-gate set, allowing for a direct comparison between the introduced model and the baseline.

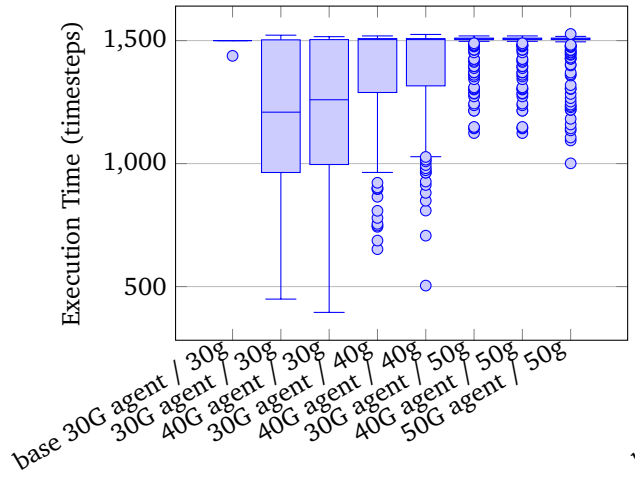


Figure 5.5: Execution times for various agents solving circuits with N gates (agent / gates).

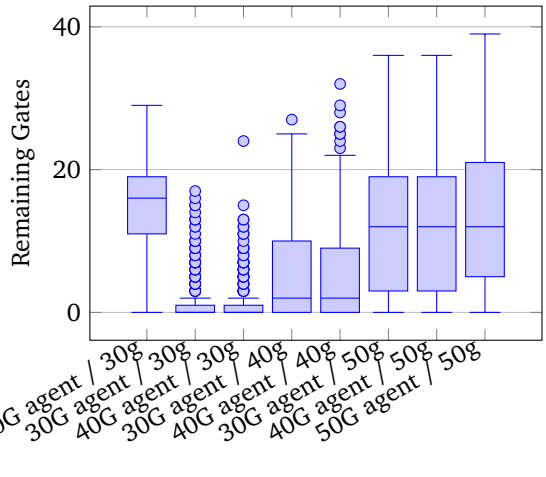


Figure 5.6: Gates left at end of execution for various agents solving circuits with N gates (agent / gates).

Figure 5.5 shows the distribution of execution times of each different agents attempting to complete circuits of a certain size and Figure 5.6 shows the distribution of the amount of gates left at the end of those attempts. First of the evaluation shows that the base model performs significantly worse on new data compared to training, as now it is unable to solve any circuits and on average is only able to complete about 15 of the 30 gates before the deadline. The new model performs significantly better with execution times of 30-gate circuits mostly ranging between 1000 to 1500 timesteps with around 25% of circuits failing to compile before the deadline. This means that the policy created by the introduced model generalizes significantly better than the policy created by the base model. However, the results for the introduced model are also considerably worse compared to training where the 30G agent compiled all circuits with completion times mostly ranging between 600 to 1000 timesteps. This performance drop is also seen with the 40G agent where it now fails $\sim$ 55% of the circuits with most execution times of successful compilations ranging between 1100 to 1500. These results are not as expected, but they can all be attributed to the fact that the agents underperform compared to their training performance when evaluated on new, unseen circuits.

This means that the 30G and 40G agents are also unable to compile 50 gate circuits most of the time. Both are only able to compile about 15% of all circuits. This performance is similar to the 50G agent. Additionally, the performance of the 30- and 40-gate agents is also similar in other cases, suggesting that the choice of training circuit size does not significantly affect an agent's ability to handle circuits of that same size after training. In other words, training on larger circuits than 30 gates does not appear to yield a clear performance advantage in this setup, meaning the gates at the end of the circuit have no significant impact in the agents decision making at the start of compilation.

5.4. Different DQC architectures

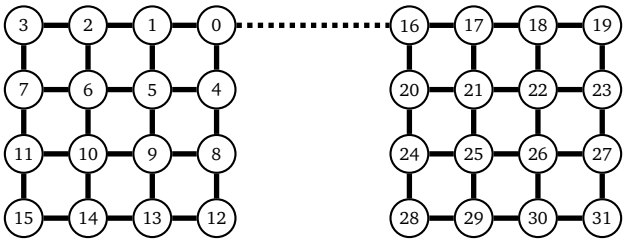


Figure 5.7: Qubit connectivity of DQC architecture consisting of two 4x4 grid connected QPUs, the dotted line indicates the quantum link between QPUs.

To verify that the introduced model can learn a generalizable policy capable of completing circuits on different hardware architectures, it is also trained on a system composed of two 16-qubit QPUs, each arranged in a 4×4 grid connectivity. In this setup, the quantum link connects qubit 0 of each QPU. An overview of this architecture is shown in Figure 5.7. Its capability is directly compared to that of the original model. Given this connectivity, it is important to observe consistency in the time it takes the introduced agent to solve circuits. If the agent consistently solves the circuit within a similar time frame, it indicates that the model has effectively learned to utilize alternative paths. Even when doing so requires selecting a sequence of multiple actions instead of choosing the action which directly moves a qubit to the destination.

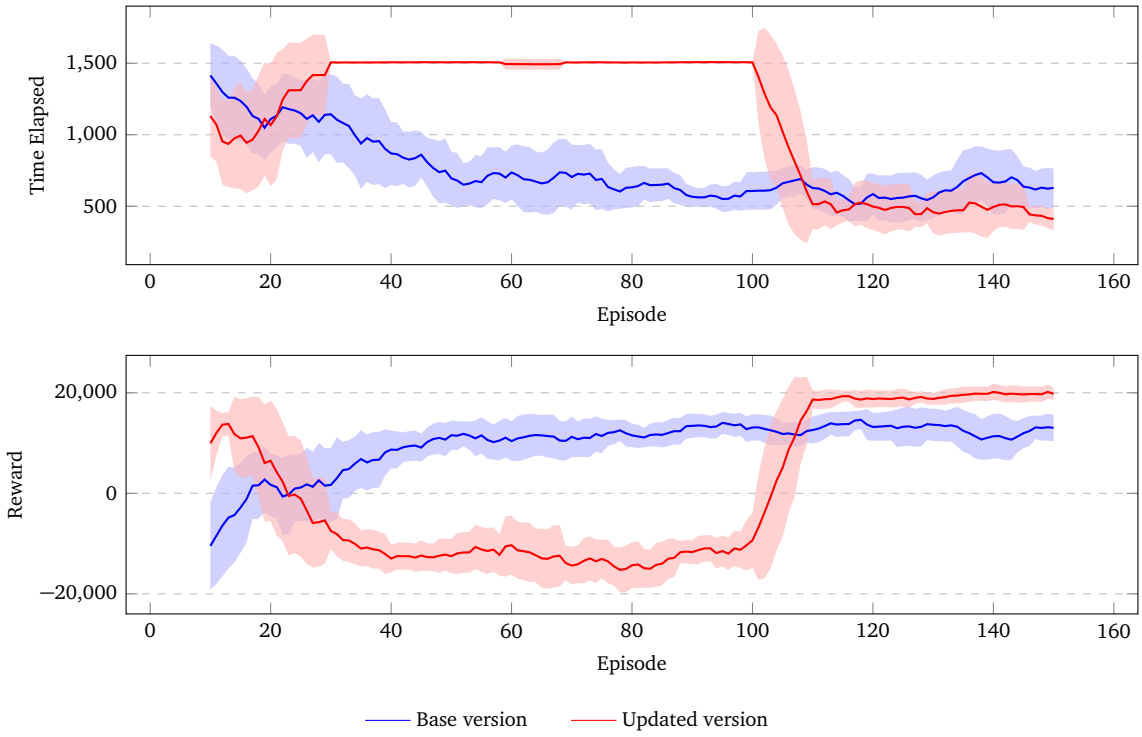


Figure 5.8: Comparison of moving average and standard deviation of time and reward evolution between the original and introduced model trained on 30-gate circuits compiled on two 4x4 grid QPUs.

Figure 5.8 shows the moving average and standard deviation of the execution time and cumulative reward per episode for both the base and introduced model. From these results it can be seen that the new model takes longer to develop a policy that can consistently solve the circuits. However, once learned, this policy enables the model to solve circuits faster than the previous model. However, it requires slightly more episodes to reach that level of performance.

Interestingly, the previous model demonstrates more consistent solving times, compared to its results of training on the IBM Q Guadalupe system, despite operating in a larger action space. This is likely because the higher connectivity of the grid connectivity system reduces overall problem complexity, with more possible

paths available, the cost of making a suboptimal move is lower. Additionally, fewer actions are needed to bring qubits closer together.

The increased connectivity also results in each SWAP action excluding more possible future actions; for example, executing a single SWAP can invalidate up to seven other SWAPs in this system, compared to a maximum of four in the Guadalupe system. This is since two neighbouring qubits in the Quadalupe system have a most four edges, one between the pair and three connecting to other qubits. After executing an actions between the pair all actions for those edges will be masked out thereby masking out four SWAPS. In the grid system however a neighbouring pair has at most 7 edges, one between the pair and six to other qubits.

Overall, this means that the improvement of the introduced model in solving time and consistency in solving time across circuits over the base model is less significant, but still valuable for practical performance as it shows the introduced model is also able to create a policy for systems with higher connectivity.

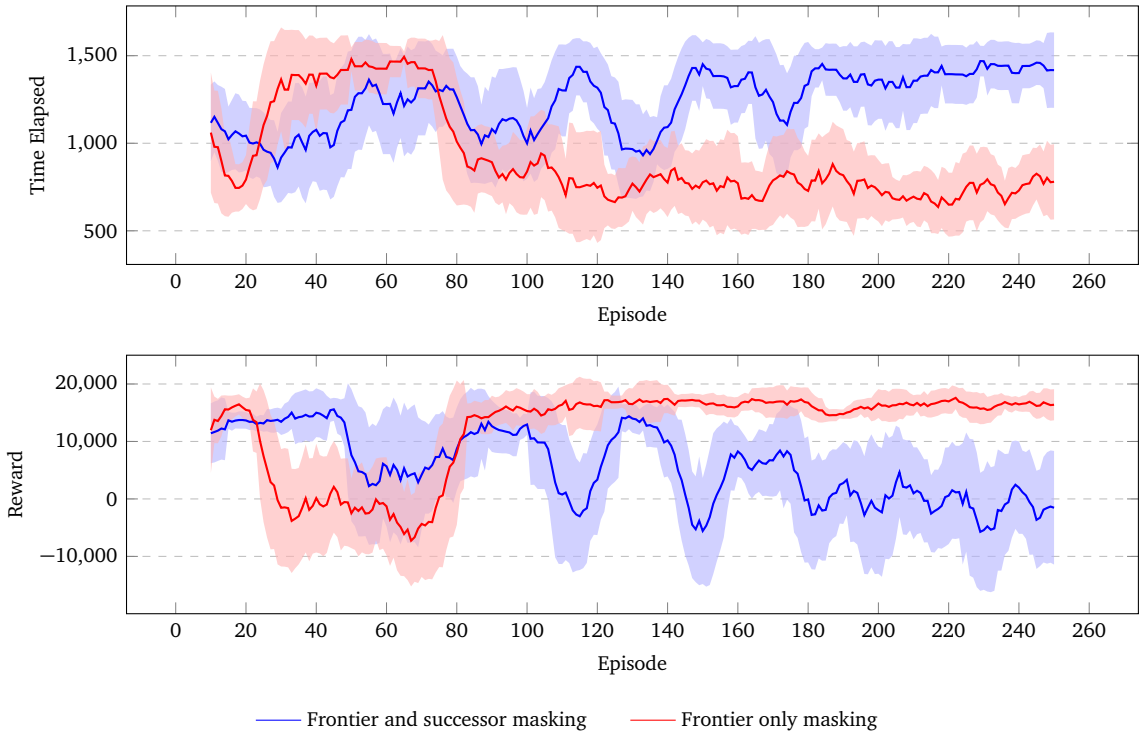


Figure 5.9: Comparison of moving average and standard deviation of time and reward evolution between different masking strategies for models trained on 30-gate circuits compiled on two IBM Q Guadalupe QPUs.

5.5. Different action elimination strategies

The action mask was refined not only to enforce qubit availability constraints but also to eliminate actions deemed suboptimal. In the current implementation, the agent is restricted to selecting actions involving qubits that participate in gates located at the frontier of the circuit's directed acyclic graph (DAG). To assess whether this restriction is effective or overly limiting, the model is trained under different levels of action elimination. In particular, the standard masking approach is compared to a more relaxed strategy that also permits actions involving gates that are immediate successors of frontier gates.

This relaxation is expected to be particularly beneficial in the later stages of compilation, where the frontier often consists of only a few remaining gates that are tightly dependent on one another. In such cases, allowing actions related to successor gates enables the agent to reposition qubits in anticipation, potentially executing a successor gate shortly after its parent frontier gate. While this could accelerate circuit execution, it also increases the branching factor by expanding the action space, which may make learning more challenging. However, this increased flexibility could also allow the agent to plan more effectively, resulting in better qubit placements and more efficient scheduling of gate sequences.

However, the numerical evaluation shows the opposite trend to the improvement that is expected: the relaxation to successor gates leads to significantly worse performance. Figure 5.9 shows the results of training with moving qubits part of successor gates allowed compared to the masking strategy used previously. The additional actions that move successors results in a significantly larger number of actions available at every state. This makes learning more difficult as is evident by the results. Despite the intuitive benefits listed earlier, the agent is simply unable to learn within such a large pool of actions and thus how to solve the circuits within the episode limit. The increase in available actions is likely around double on average, however it depends on the distance between qubits in successor gates as well as the amount of successor gates and which qubits are part of those gates. However if we consider the same average path length and that each frontier gate has one successor gate then it would be double. This increase is too much for the agent to learn how to solve the circuits within the episode limit.

6

Discussion and Future work

The results demonstrate a clear improvement in performance compared to the previous implementation. The introduced model exhibits greater consistency in execution times and achieves a lower average time to solve a circuit. Notably, these gains were achieved without major changes to the training setup, the model was trained using approximately the same hyperparameters as before, with the only significant difference being a decrease in network size. This means that increase in performance was due to the design changes. That said, further tuning of hyperparameters could potentially enhance the learning process and lead to even better results and/or more consistency.

That said, there is still aspects of the current implementation that holds potential for future improvement. Below, several paths for further development of i) the variation in the training, ii) shaping of the rewards, iii) the scalability of the model, iv) the masking strategy.

Variation in the training

When a trained model is used to compile a new set of randomly generated circuits, it does not perform as well as during training. The most consistent model, the one trained on 30 gate circuits, does not perform as well as at the end of its training even failing to compile around 25% of circuits. This is likely due to training being limited in variety, as the agent is only exposed to 250 episodes during training. Both the initial qubit mappings and the training circuits were randomly generated, introducing some variation, but this was not sufficient to ensure adequate diversity. Expanding the training dataset to include a greater number and variety of circuits could help address this issue by exposing the model to more diverse topologies and challenges.

Given the substantial reduction in training time achieved with the introduced model, simply increasing the number of training episodes is a feasible way to enhance dataset diversity, doubling the episodes would still result in a shorter overall training time compared to the baseline agent. Alternatively, constructing a training dataset with guaranteed diversity would eliminate the reliance on chance introduced by purely random circuit generation. Refining the training strategy in this way could strengthen the model's ability to generalize and successfully compile a wider range of circuits. Prior work has shown that increasing data diversity improves both generalization and learning efficiency [44], [45]. One promising approach would be to blend observations from multiple training environments, further broadening the diversity of experiences available to the agent [44].

Shaping of the rewards

From the results it is seen that a larger cumulative reward does not always correspond to a faster solution time when comparing different problem. This happens because in some cases the additional positive reward from SWAPs outweighs the additional negative penalty associated with the longer solution time leading to a larger reward even tough the solving time is slower. Increasing the negative reward associated with stop actions could make the number of timesteps required to solve a circuit more influential in determining the total cumulative reward. Therefore, adjusting the reward the agent receives for swap actions could make solving time and reward more closely correlated. This adjustment would encourage the agent to further

optimize for reducing the time it takes to solve circuits instead of optimizing to solve the circuits, since time would then have a larger impact on the overall reward.

Scalability of the model

Although the new model is able to solve 30-gate circuits more consistently and can also solve 40-gate circuits within the episode limit, it still fails to learn to solve 50-gate circuits within the limit. Post-training results indicate that agents trained on a given circuit size can be applied to compile larger circuits for the DQC system containing two IBM Q Guadalupe QPUs and 18 logical qubits. However, this may not generalize to other DQC systems or to cases with more logical qubits. As the number of logical qubits increases, circuit structures become more complex, leading to a larger search space and more intricate gate dependencies. If the agent only observes a limited portion of the circuit, it may miss critical contextual information required to make decent long-term decisions. Consequently, showing the agent only part of the circuit is likely effective only up to a certain circuit size, with the minimum required observation length depending on both the system architecture and the number of logical qubits involved.

This limitation becomes more pronounced in larger architectures, which typically employ more logical qubits. In such cases, the state representation may not scale well, as larger circuits substantially increase the length of the state vector. This not only expands the number of possible states but also inflates the number of parameters the agent must learn and optimize. Improving scalability may therefore require reducing the dimensionality of the state representation. One possible approach is to remove the explicit layer number for each gate from the state, so that the state vector length grows with only two features per gate instead of three. The order of gates could then be inferred from their position within the vector, with gates at the rightmost indices representing the front of the circuit. Such a modification would allow the state to scale more efficiently with increasing circuit size and could improve the agent's learning performance on larger systems.

Masking strategy

Experiments were conducted with different levels of masking. The results showed that when movements along paths between qubits of successor gates were permitted, the action space expanded so significantly that the agent was unable to learn to solve the circuits within the episode limit. Nevertheless, if the agent could exploit such moves, it would likely be able to complete circuits faster. As discussed earlier, this advantage would be most pronounced near the end of compilation, when only a few gates remain, most of them successors, and the frontier is small. A reasonable compromise, therefore, could be to allow movement between qubits of successor gates only when the frontier is small. This would keep the action space manageable during the early stages of compilation while still enabling the agent to benefit from potential speed-ups later on.

Overall, the introduced changes significantly enhance the agent's performance, both during training and in its final compilation results. The recommendations provided focus primarily on improving scalability and generalization, as these remain the most prominent limitations of the current implementation. Addressing these aspects is essential for developing a more robust and practically applicable compiler for distributed quantum computing.

7

Conclusion

This thesis introduced several key improvements to a reinforcement learning–based compiler for distributed quantum computing (DQC) environments [13].

First, the action space was redesigned to enable direct operations between arbitrary qubit pairs, rather than restricting interactions to neighbouring qubits. This significantly increased the flexibility of the agent while also posing a greater computational challenge as it increased the size of the actionspace. To address this, a novel neural network architecture was developed that computes Q-values for qubit pairs using Q-values of individual qubits. This design drastically reduces the number of trainable parameters compared to traditional pairwise representations, improving scalability and training efficiency.

Second, a dynamic masking strategy was introduced to filter out a majority of actions during decision-making. By allowing only actions which potentially provide progress to the goal the branching factor is manageable, allowing the agent to scale more effectively as problem size increases.

The experimental results demonstrate clear improvements over the baseline model. The new approach reduced the average circuit execution time during training by approximately 32% and shortened the total training time by 64%. Moreover, the introduced agent successfully learned to compile 40-gate circuits, whereas the baseline implementation failed to do so.

For training on compiling 30-gate circuits, the introduced agent does not fully retain its training performance when evaluated after training: the average execution time increases by roughly 60%, and approximately 25% of circuits fail to compile within the allowed time. Nevertheless, this still represents a significant improvement over the baseline agent, which was unable to compile any 30-gate circuits after training.

The agent trained on 30-gate circuits is therefore also unable to solve 40- and 50-gate circuits. However, its performance on these larger circuits is comparable to that of agents trained directly on 40- and 50-gate circuits. This suggests that training on circuits larger than 30 gates does not provide a significant performance advantage for the current model architecture. In particular, it indicates that gates appearing later in the circuit have little influence on the agent's decision-making at the start of the compilation process.

In summary, the modifications introduced in this thesis substantially improve both training efficiency and post-training performance of the RL-based compiler. These results suggest that the redesigned approach scales better to larger problems while requiring fewer computational resources. The improvements show that the compiler could be used to compile larger, more complex quantum circuits that will be required in future quantum processors.

References

[1] T. F. Rønnow, Z. Wang, J. Job, *et al.*, "Defining and detecting quantum speedup", *Science*, vol. 345, no. 6195, pp. 420–424, 2014. DOI: 10.1126/science.1252319.

[2] T. Monz, D. Nigg, E. A. Martinez, *et al.*, "Realization of a scalable shor algorithm", *Science*, vol. 351, no. 6277, pp. 1068–1070, 2016. DOI: 10.1126/science.aad9480.

[3] L. K. Grover, *A fast quantum mechanical algorithm for database search*, 1996. arXiv: quant-ph/9605043 [quant-ph]. [Online]. Available: https://arxiv.org/abs/quant-ph/9605043.

[4] B. Bauer, S. Bravyi, M. Motta, and G. K.-L. Chan, "Quantum algorithms for quantum chemistry and quantum materials science", *Chemical reviews*, vol. 120, no. 22, pp. 12 685–12 717, 2020.

[5] A. Montanaro, "Quantum algorithms: An overview", *npj Quantum Information*, vol. 2, no. 1, pp. 1–8, 2016.

[6] V. Mavroeidis, K. Vishi, M. D. Zych, and A. Jøsang, "The impact of quantum computing on present cryptography", 2018.

[7] S. Lloyd, M. Mohseni, and P. Rebentrost, *Quantum algorithms for supervised and unsupervised machine learning*, 2013. [Online]. Available: https://arxiv.org/abs/1307.0411.

[8] J. Preskill, *Quantum computing 40 years later*, 2023. arXiv: 2106.10522 [quant-ph]. [Online]. Available: https://arxiv.org/abs/2106.10522.

[9] A. Kole, K. Datta, and R. Drechsler, "Design automation challenges and benefits of dynamic quantum circuit in present nisq era and beyond: (invited paper)", in *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2024, pp. 601–606. DOI: 10.1109/ISVLSI61997.2024.00114.

[10] M. Caleffi, M. Amoretti, D. Ferrari, J. Illiano, A. Manzalini, and A. S. Cacciapuoti, "Distributed quantum computing: A survey", *Computer Networks*, vol. 254, pp. 110–672, 2024. DOI: https://doi.org/10.1016/j.comnet.2024.110672.

[11] M. Caleffi, A. S. Cacciapuoti, and G. Bianchi, "Quantum internet: From communication to distributed computing!", in *Proceedings of the 5th ACM International Conference on Nanoscale Computing and Communication*, ser. NANOCOM '18, ACM, Sep. 2018. DOI: 10.1145/3233188.3233224.

[12] D. Ferrari, A. S. Cacciapuoti, M. Amoretti, and M. Caleffi, "Compiler design for distributed quantum computing", *IEEE Transactions on Quantum Engineering*, vol. 2, pp. 1–20, 2021. DOI: 10.1109/TQE.2021.3053921.

[13] P. Promponas, A. Mudvari, L. D. Chiesa, P. Polakos, L. Samuel, and L. Tassiulas, *Compiler for distributed quantum computing: A reinforcement learning approach*, 2024. [Online]. Available: https://arxiv.org/abs/2404.17077.

[14] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge university press, 2010.

[15] A. Cowtan, S. Dilkes, R. Duncan, A. Krajenbrink, W. Simmons, and S. Sivarajah, "On the qubit routing problem", *arXiv preprint arXiv:1902.08091*, 2019.

[16] L. Chirolli and G. Burkard, "Decoherence in solid-state qubits", *Advances in Physics*, vol. 57, no. 3, pp. 225–285, 2008.

[17] E. G. Rieffel and W. H. Polak, *Quantum computing: A gentle introduction*. MIT press, 2011.

[18] Y. Ge, W. Wenjie, C. Yuheng, *et al.*, "Quantum circuit synthesis and compilation optimization: Overview and prospects", *arXiv preprint arXiv:2407.00736*, 2024.

[19] A. Botea, A. Kishimoto, and R. Marinescu, "On the complexity of quantum circuit compilation", in *Proceedings of the International Symposium on Combinatorial Search*, vol. 9, 2018, pp. 138–142.

[20] T. Ito, N. Kakimura, N. Kamiyama, Y. Kobayashi, and Y. Okamoto, "Algorithmic theory of qubit routing", in *Algorithms and Data Structures Symposium*, Springer, 2023, pp. 533–546.

[21] W. Kozlowski, S. Wehner, R. Van Meter, *et al.*, *Rfc 9340: Architectural principles for a quantum internet*, USA, 2023.

[22] M. Caleffi, D. Chandra, D. Cuomo, S. Hassanpour, and A. S. Cacciapuoti, "The rise of the quantum internet", *Computer*, vol. 53, no. 06, pp. 67–72, 2020.

[23] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey", *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.

[24] R. S. Sutton and A. G. Barto, "Reinforcement learning: An introduction. mit press cambridge", *Massachusetts London, England*, 2018.

[25] J. Terven, "Deep reinforcement learning: A chronological overview and methods", *AI*, vol. 6, no. 3, p. 46, 2025.

[26] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-based batch mode reinforcement learning", *Journal of Machine Learning Research*, vol. 6, 2005.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, *et al.*, "Human-level control through deep reinforcement learning", *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[28] J. F. Hernandez-Garcia and R. S. Sutton, *Understanding multi-step deep reinforcement learning: A systematic study of the dqn target*, 2019. arXiv: 1901.07510 [cs.LG]. [Online]. Available: https://arxiv.org/abs/1901.07510.

[29] A. Witte, "Ddqn: Metrics for measuring stability using the example of replay buffer size and minibatch size",

[30] A. Srinivasan, "Reinforcement learning: Advancements, limitations, and real-world applications", *International Journal of Scientific Research in Engineering and Management (IJSREM)*, vol. 7, p. 08, 2023.

[31] D. Ferrari, S. Carretta, and M. Amoretti, "A modular quantum compilation framework for distributed quantum computing", *IEEE Transactions on Quantum Engineering*, vol. 4, pp. 1–13, 2023. DOI: 10.1109/TQE.2023.3303935.

[32] D. Cuomo, M. Caleffi, K. Krsulich, *et al.*, "Optimized compiler for distributed quantum computing", *ACM Transactions on Quantum Computing*, vol. 4, no. 2, Feb. 2023. DOI: 10.1145/3579367.

[33] A. Annechini, M. Venere, D. Sciuto, and M. D. Santambrogio, "Ddroute: A novel depth-driven approach to the qubit routing problem", in *2025 62nd ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–7. DOI: 10.1109/DAC63849.2025.11133018.

[34] L. Le and T. N. Nguyen, "Dqra: Deep quantum routing agent for entanglement routing in quantum networks", *IEEE Transactions on Quantum Engineering*, vol. 3, pp. 1–12, 2022. DOI: 10.1109/TQE.2022.3148667.

[35] M. G. Pozzi, S. J. Herbert, A. Sengupta, and R. D. Mullins, *Using reinforcement learning to perform qubit routing in quantum compilers*, 2020. arXiv: 2007.15957 [quant-ph]. [Online]. Available: https://arxiv.org/abs/2007.15957.

[36] W. Tang, Y. Duan, Y. Kharkov, R. Fakoor, E. Kessler, and Y. Shi, "Alpharouter: Quantum circuit routing with reinforcement learning and tree search", in *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, vol. 01, 2024, pp. 930–940. DOI: 10.1109/QCE60285.2024.00112.

[37] C. Mastrogiuseppe and R. Moreno-Bote, "Deep imagination is a close to optimal policy for planning in large decision trees under limited resources", *Scientific reports*, vol. 12, no. 1, p. 10 411, 2022.

[38] G. Li, Z. Wang, S. He, *et al.*, "Balancing depth for robustness: A study on reincarnating reinforcement learning models", *Applied Sciences*, vol. 15, no. 7, p. 3830, 2025.

[39] E. W. Dijkstra, "A note on two problems in connexion with graphs", *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959. DOI: 10.1007/BF01386390.

[40] N. C. Thompson, K. Greenewald, K. Lee, G. F. Manso, *et al.*, "The computational limits of deep learning", *arXiv preprint arXiv:2007.05558*, vol. 10, p. 2, 2020.

[41] K. Ota, D. K. Jha, and A. Kanezaki, "Training larger networks for deep reinforcement learning", *arXiv preprint arXiv:2102.07920*, 2021.

[42] T. Zahavy, M. Haroush, N. Merlis, D. J. Mankowitz, and S. Mannor, "Learn what not to learn: Action elimination with deep reinforcement learning", *Advances in neural information processing systems*, vol. 31, 2018.

[43] M. Fatemi, S. Sharma, H. Van Seijen, and S. E. Kahou, "Dead-ends and secure exploration in reinforcement learning", in *International Conference on Machine Learning*, PMLR, 2019, pp. 1873–1881.

[44] K. WANG, B. Kang, J. Shao, and J. Feng, "Improving generalization in reinforcement learning with mixture regularization", in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33, Curran Associates, Inc., 2020, pp. 7968–7978. [Online]. Available: `https://proceedings.neurips.cc/paper_files/paper/2020/file/5a751d6a0b6ef05cfe51b86e5d1458e6-Paper.pdf`.

[45] K. Cobbe, O. Klimov, C. Hesse, T. Kim, and J. Schulman, "Quantifying generalization in reinforcement learning", in *International conference on machine learning*, PMLR, 2019, pp. 1282–1289.