Reinforcement Learning and DEAR Framework for Solving the Qubit Mapping Problem

Ching-Yao Huang
National Tsing Hua University
Hsinchu, Taiwan
rabit66119929@gapp.nthu.edu.tw

Chi-Hsiang Lien
National Tsing Hua University
Hsinchu, Taiwan
wee62660@gapp.nthu.edu.tw

Wai-Kei Mak
National Tsing Hua University
Hsinchu, Taiwan
wkmak@cs.nthu.edu.tw

ABSTRACT

Quantum computing is gaining more and more attention due to its huge potential and the constant progress in quantum computer development. IBM and Google have released quantum architectures with more than 50 qubits. However, in these machines, the physical qubits are not fully connected so that two-qubit interaction can only be performed between specific pairs of the physical qubits. To execute a quantum circuit, it is necessary to transform it into a functionally equivalent one that respects the constraints imposed by the target architecture. Quantum circuit transformation inevitably introduces additional gates which reduces the fidelity of the circuit. Therefore, it is important that the transformation method completes the transformation with minimal overheads. It consists of two steps, initial mapping and qubit routing. Here we propose a reinforcement learning-based model to solve the initial mapping problem. Initial mapping is formulated as sequence-to-sequence learning and self-attention network is used to extract features from a circuit. For qubit routing, a DEAR (Dynamically-Extract-and-Route) framework is proposed. The framework iteratively extracts a subcircuit and uses A* search to determine when and where to insert additional gates. It helps to preserve the lookahead ability dynamically and to provide more accurate cost estimation efficiently during A* search. The experimental results show that our RL-model generates better initial mappings than the best known algorithms with 12% fewer additional gates in the qubit routing stage. Furthermore, our DEAR-framework outperforms the state-of-the-art qubit routing approach with 8.4% and 36.3% average reduction in the number of additional gates and execution time starting from the same initial mapping.

1 INTRODUCTION

Quantum algorithms can be exponentially faster than classical algorithms and can be applied to problems such as large integer factorization [16], database searching [24], and cryptography [6]. Several quantum systems have been released such as Bristlecone with 72 qubits from Google and Hummingbird with 65 qubits from IBM. Besides, Rigetti and IBM now provide public accessible cloud platforms [7] where researchers can execute quantum programs on real quantum devices.

In the NISQ (Noisy Intermediate-Scale Quantum) era, quantum architectures are not ideal. They only support a limited set of operations and are error prone. Furthermore, it is not possible to establish direct interaction between any pairs of qubits. However, a quantum program is designed under the assumption that it executes on an ideal quantum architecture where qubit has zero error rate, unlimited coherence time and operations can be applied to any pairs of qubits. To execute a quantum program, it is necessary to compile it into a functionally equivalent physical circuit. The compilation of a quantum program comprises two main stages of translation. First, the program is converted into a logical circuit composed of elementary gates. Second, ancillary gates are inserted into the logical circuit to transform it to a physical circuit with respect to the constraints imposed by target architecture.

This paper focuses on the the second stage of compilation which consists of two steps, initial mapping and qubit routing. Initial mapping, also known as qubit placement, is to decide an initial mapping from logical qubits to physical qubits. Qubit routing is to insert additional gates such as SWAP into the logical circuit to change the mapping so that all gates in the logical circuit can be executed. Additional gates inevitably results in increased gate count and execution time. Since qubits are fickle and error prone, it affects the reliability of the circuit. Thus, it is important that the qubit routing method complete the task with minimum additional gates.

Several methods have been proposed to produce initial mappings which reduce the number of inserted gates in the qubit routing phase. [28] and [2] iteratively map each logical qubit to a physical qubit. [28] only takes the gates in the front part of a logical circuit into consideration while [2] takes the gates in the whole logical circuit into account but does not consider the positional information of each gate. On the other hand, [27] and [12] use simulated annealing to search for an initial mapping that minimizes a defined cost function. Finally, [10] uses vf2 algorithm to search for a subgraph of the architecture graph isomorphic to a subgraph of the logical circuit graph in order to determine a good initial mapping.

In this work, a reinforcement learning model (RL-model) is proposed to produce high-quality initial mappings. Although reinforcement learning has been proposed to address the qubit routing problem[17][13], it has never been used for the initial mapping problem. In our RL-model, the agent encodes a quantum circuit by self-attention neural network and outputs actions which map logical qubits to physical qubits to generate an initial mapping. The generated mapping is judged by a deterministic qubit routing

This work was supported in part by the National Science and Technology Council under grants 110-2221-E-007-122 and 111-2221-E-007-119-MY3.

ICCAD '22, October 30-November 3, 2022, San Diego, CA, USA

ACM ISBN 978-1-4503-9217-4/22/10…$15.00
https://doi.org/10.1145/3508352.3549472

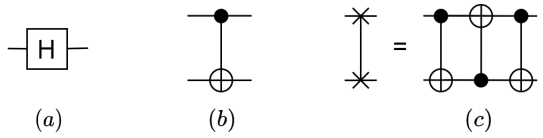

Figure 1: (a)H gate.(b) CNOT gate.(c) SWAP gate and its implementation by CNOT gates.

algorithm, t|ket> [19] and the number of added gates by t|ket> is used as the feedback reward.

Various qubit routing approaches have been proposed aiming at different objectives like the minimization of additional gate counts [10][27][9][28][26], circuit depth [3][25], or circuit error [11][12]. These methods can be classified into two categories. The first category reformulates the problem as an optimization or planning problem and solve it using off-the-shelf solvers [20][11][18]. However, it is shown in [18] that qubit routing problem is NP-complete. As the complexity of these methods usually grows exponentially, they are not scalable and lead to unacceptable execution time when the size of input circuit is large. Another category applies search algorithms guided by some heuristic cost function [27][9][28]. They exploit the intrinsic feature of the problem and construct the physical circuit iteratively. Experimental results shows that these methods are more promising in transforming large circuits.

We propose a DEAR (Dynamically Extract and Route) framework coupled with A* search to solve the qubit routing problem. The framework iteratively extracts a subcircuit of the logical circuit and uses A*-search to perform qubit routing on the subcircuit. Different from [15] which pre-splits the circuit into layers, the subcircuits in our framework are not predefined but are dynamically extracted during qubit routing such that a greater lookahead capacity can be maintained all the time.

Both the proposed RL-model and DEAR-framework are evaluated with various benchmarks on the IBM QX20 architecture. Experimental results shows that the initial mappings generated by RL-model reduce the required SWAP gate insertion by over 12% compared to [2] and [28]. The DEAR framework is able to outperform the state-of-art work[26] on almost all benchmarks. The number of additional gates is reduced by 8.4% and execution time is reduced by 36.3% on average.

The rest of the paper is organized as follows. The background of quantum computing is introduced in section 2 and the initial mapping and qubit routing problems are formulated in section 3. Section 4 gives the details of proposed RL-model and section 5 introduces the DEAR-framework. The experimental results are reported in section 6 and we conclude this article in section 7.

2 BACKGROUND

In this section, we introduce some relevant basic concepts.

Qubit In classical computing, bit is the basic information unit which has only two states, 0 and 1. In contrast, quantum computing performs computation on qubits. One qubit has two basis states, $|0\rangle$ and $|1\rangle$, and it can be in the superposition state represented as a linear combination of two basis states, $\psi = a\,|0\rangle + b\,|1\rangle$, where $a, b \in \mathbb{C}$ and $|a|^2 + |b|^2 = 1$.

Quantum Gate Quantum gate is applied to qubits to manipulate the states of qubits. A single-qubit gate operates on one target qubit. For instance, the Hadamard gate (Fig. 1(a)) maps the basis state $|0\rangle \Rightarrow \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $|1\rangle \Rightarrow \frac{|0\rangle-|1\rangle}{\sqrt{2}}$ to create a superposition state. A two-qubit gate operates on two qubits simultaneously. For example, a Control-Not (CNOT) gate (Fig. 1(b)) performs the NOT operation on the target qubit (denoted as $\oplus$) when the control qubit (denoted by $\bullet$) is $|1\rangle$. A SWAP gate (Fig. 1(c)) swaps the states of two qubits (denoted by $\times$), and can be implemented with three CNOT gates.

Quantum Circuit A quantum circuit can be represented by a quantum circuit diagram as shown in Fig. 2(a). There is a horizontal line for each logical qubit, the gates are placed from left to right on the lines according to their execution order. The circuit shown in Fig. 2(a) uses four logical qubits and seven gates. g_1 and g_4 are single qubit gates applied to qubits q_1 and q_3, respectively. g_2, g_3, g_5, g_6 and g_7 are two-qubit gates applied to $\{q_3, q_4\}$, $\{q_1, q_2\}$, $\{q_2, q_3\}$, $\{q_2, q_4\}$ and $\{q_1, q_4\}$, respectively.

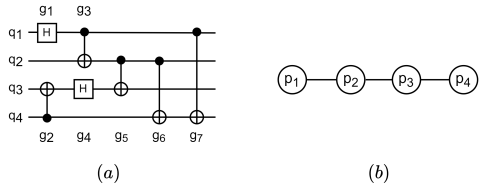

Figure 2: (a) A quantum circuit. (b) A linear architecture.

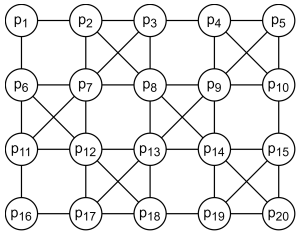

Figure 3: The architecture graph of IBM QX20.

Quantum Device Real-world quantum device can be described by a connected undirected graph called the architecture graph. Fig. 3 shows the architecture graph for IBM QX20. Each vertex on the graph represents a physical qubit and each edge represents a pair of physical qubits that a two-qubit gate can be applied to. The single qubit gates can be executed by any physical qubit. However, two-qubit interaction may only be performed between specific pairs of the physical qubits as dictated by its underlying topology which is referred to as the *connectivity constraint*.

In order to run a quantum circuit on a quantum device, logical qubits of the quantum circuit must be mapped to the physical qubits of the device. However, very often there does not exist a feasible mapping that satisfies all connectivity constraints due to the limited interconnection between the physical qubits. For example, the quantum circuit described in Fig. 2(a) does not have any feasible mapping on the quantum device of Fig. 2(b). To solve this problem,

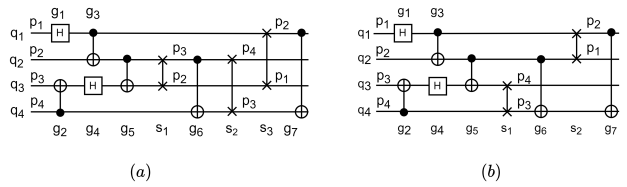

Figure 4: (a) A feasible physical circuit after initial mapping and routing with three SWAP gates inserted for the example in Fig. 2. (b) Another feasible physical circuit solution with two inserted SWAP gates.

we may change the qubit mapping during execution to make all two-qubit gates of the quantum circuit executable. Qubit mapping may be changed during execution by inserting additional SWAP operations. A SWAP operation can be realized by three CNOT gates as in Fig. 1(c).

Consider the quantum circuit composed of 7 gates in Fig. 2(a) and the architecture in Fig. 2(b). Given an initial mapping $\{q_1 \mapsto p_1, q_2 \mapsto p_2, q_3 \mapsto p_3, q_4 \mapsto p_4\}$. The first five gates, $g_1\{q_1\}$, $g_2\{q_3, q_4\}, g_3\{q_1, q_2\}, g_4\{q_3\}$, and $g_5\{q_2, q_3\}$ are executable under the initial mapping since g_1 and g_4 are single-qubit gates and the two logical qubits of $g_2/g_3/g_5$ are mapped to adjacent physical qubits in the architecture. But gate $g_6\{q_2, q_4\}$ is not executable since q_2 and q_4 are mapped to p_2 and p_4 which are not adjacent in the architecture. To execute $g_6\{q_2, q_4\}$, we may insert SWAP(q_2, q_3) to swap the physical qubits assigned to q_2 and q_3. Then, the mapping is updated to $\{q_1 \mapsto p_1, q_2 \mapsto p_3, q_3 \mapsto p_2, q_4 \mapsto p_4\}$ which allows $g_6\{q_2, q_4\}$ to execute. But gate $g_7\{q_1, q_4\}$ is still not executable on the updated mapping, two more SWAP operations SWAP(q_2, q_4) and SWAP(q_1, q_3) can be inserted to make it executable. Fig. 4(a) shows the final physical circuit[1] where all gates can be executed and three SWAP gates are inserted. Alternatively, for the same initial mapping, we can insert a SWAP operation SWAP(q_3, q_4) first to execute $g_6\{q_2, q_4\}$ and then another SWAP operation SWAP(q_1, q_2) to execute $g_7\{q_1, q_4\}$ as in Fig. 4(b).

3 PROBLEM FORMULATION

Since the initial mapping can have a huge impact on the number of SWAP insertion required, the qubit mapping problem is further divided into two subproblems. The first is how to determine a good initial mapping. The second is how to determine a minimum cost transformation by SWAP operation insertion in order to execute all the gates in the original quantum circuit starting from an initial mapping. The second subproblem is also known as the qubit routing problem. We formally define the qubit mapping problem below.

Definition. *(Qubit Mapping Problem) Given a logical circuit and the architecture graph of a quantum device. The qubit mapping problem is to (i) compute an initial mapping of logical qubits to physical qubits, and (ii) derive the intermediate qubit mapping transition (i.e., qubit routing) by SWAP gate insertion to satisfy the connectivity constraints of the quantum device such that the number of SWAP gates inserted is minimized in order to successfully execute the logical circuit on the quantum device.*

[1]To distinguish the original quantum circuit from the final transformed circuit with an initial mapping that can be executed on a specific quantum device, we call the former the *logical circuit* and the latter the *physical circuit.*

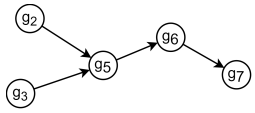

Figure 5: The dependency graph of the circuit in Fig. 2(a).

We also define the dependency graph of a logical circuit which will used in the following sections.

Definition. *(Dependency Graph) The dependency graph is a directed acyclic graph that can be constructed to represent the dependency between two-qubit gates in a quantum circuit. Each vertex of the dependency graph corresponds to a two-qubit gate. If two-qubit gates g_i and g_j share at least one common logical qubit and g_i is executed earlier than g_j with no other two-qubit gate in between, then there is a directed edge from the vertex for g_i to the vertex for g_j.*

Fig. 5 shows the dependency graph of the circuit in Fig. 2(a). Note that single-qubit gates are not included in the dependency graph here since they can always be executed on any physical qubit irrespective of the mapping.

4 INITIAL MAPPING

Given a logical circuit that has to be executed on a quantum device with architecture graph $\mathcal{AG} = (P, E)$, it is desired to find an initial mapping Π which maps logical qubits $Q = \{q_1, q_2, ..., q_{|Q|}\}$ to physical qubits $P = \{p_1, p_2, ..., p_{|P|}\}$ such that the expected number of additional gates after qubit routing phase can be minimized.

Sequence-to-sequence models are widely used in machine learning fields such as translation [4], image captioning [14] and speech recognition [1]. These models usually consist of two components, an encoder and a decoder. The encoder encodes an input sequence and the decoder takes the encoded sequence as input and decodes it as a transformed sequence. We propose an innovative approach to solve the initial mapping problem by sequence-to-sequence learning.

We partition the two-qubit gates of a logical circuit into layers $\{L_1, L_2, ...\}$ where the gates in each layer act on distinct logical qubits. We iteratively put a gate into layer L_i where i is as small as possible according to the topological order in the corresponding dependency graph. For example, the two-qubit gates of the logical circuit in Fig. 2(a) can be partitioned into four layers $\{L_1, L_2, L_3, L_4\}$ according to its dependency graph in Fig. 5 where $L_1 = \{g_2, g_3\}$, $L_2 = \{g_5\}$, $L_3 = \{g_6\}$, $L_4 = \{g_7\}$.

We sort the two-qubit gates in a logical circuit according to their layers in ascending order and call it the *gate sequence* $GS = (g_1, g_2, ..., g_n)$. In addition, we form a *layer sequence* $LS = (\ell_1, \ell_2, ..., \ell_n)$ where g_i is in layer L_{ℓ_i} of the circuit. The layer sequence provides the positional information of the corresponding gates in the gate sequence. For example, given the logical circuit in Fig. 2(a). GS and LS can be obtained as $(\{q_1, q_2\}, \{q_3, q_4\}, \{q_2, q_3\}, \{q_2, q_4\}, \{q_1, q_4\})$ and $(1, 1, 2, 3, 4)$, respectively, where a two-qubit gate is represented by the two logical qubits that it acts on. We note that GS may not be unique for a logical circuit, but it makes no difference to the final

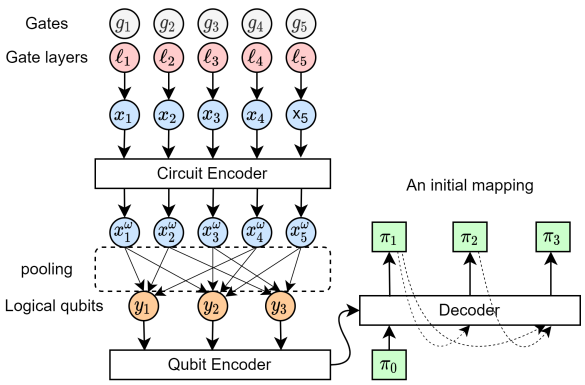

Figure 6: Overview of our encoder-decoder model.

result of our model. We define an initial mapping as a permutation of a subset of physical qubits, $\Pi = (\pi_1, \pi_2, ..., \pi_{|Q|})$ where $\pi_i \in P$. For each logical qubit q_i in Q, q_i is mapped to π_i.

We set our encoder-decoder as the agent and an existing qubit routing tool as the environment. For a given GS and LS, the agent takes several actions to produce Π according to the learned stochastic policy. Each action is determined by a probability distribution. These actions result in a reward $R(\Pi)$ that is the negative number of additional gates computed by the environment. The policy of the agent is updated according to the routing results and the goal is to maximize the reward $R(\Pi)$.

4.1 Our Neural Network Architecture

In this subsection, we propose a neural network model to encode a circuit and output probability distributions to select actions. The high level structure is illustrated in Fig. 6 which consists of two encoders and one decoder. The encoders and decoder contain multi-head attention mechanism and are adapted from [21]. Given the gate sequence and the layer sequence of a logical circuit, the circuit encoder encodes each gate into a hidden feature vector. Then, for each logical qubit q_i, hidden features of all gates in the circuit that act on q_i are aggregated to obtain the feature vector of the qubit. This stage is called gate pooling. The second encoder, qubit encoder, encodes logical qubits again. Afterwards, the decoder produces a series of probability distributions to determine actions.

4.1.1 *The Circuit Encoder.*

Since we adopt the encoder and decoder model in [21], the positional information of each input element is needed. The circuit encoder has two inputs, gate sequence and layer sequence. The layer sequence is used to specify the positional information for the corresponding gates in the gate sequence.

For each kind of two-qubit gates in GS specified by its operation logical qubits[2], it can be embedded as a feature vector. For gate sequence $GS = (g_1, ..., g_n)$, an embedding feature vector sequence $GS' = (g'_1, ..., g'_n)$ where $g'_i \in \mathbb{R}^{d_{model}}$ is obtained (d_{model} is set to 128). The layer sequence is a sequence of scalars which can also be embedded in the same way. For layer sequence $LS = (\ell_1, ..., \ell_n)$, an embedding feature vector sequence $LS' = (\ell'_1, ..., \ell'_n)$ where $\ell'_i \in \mathbb{R}^{d_{model}}$ is obtained as well. We add GS' and LS' to obtain $X^0 \in \mathbb{R}^{n \times d_{model}}$ which comprises the information of the gates and their positions.

The circuit encoder is composed of N (N is set to 3) identical layers and a layer is illustrated on the left side of Fig. 7. Each layer consists of two sub-layers, a multi-head attention mechanism and a feed-forward network. For X^i where $i > 0$, X^i stands for the outputs of the i-th layer. The encoder takes X^0 as input and uses the output of the i-th layer as the input of the $(i+1)$-th layer. After N layers, X^N is the output of the encoder.

- *Multi-head attention mechanism*: We explain the single head attention function before introducing the multi-head attention mechanism. The inputs of a single head attention function are U, K, and $V \in \mathbb{R}^{n \times d_s}$ where U, K and V stand for queries, keys and values, respectively. The output of a single head attention function is a weighted sum of the values computed by scaled dot-product by eq. (1).

$$Attention(U, K, V) = softmax(\frac{UK^T}{\sqrt{d_s}})V \tag{1}$$

  The i^{th} row in $Attention(U, K, V) \in \mathbb{R}^{n \times d_s}$ is a weighted sum of all rows in V and the weights is determined by the dot-product of U and K divided by $\sqrt{d_s}$ with softmax function.
  Instead of using a single head attention function eq. (1) with $d_s = d_{model}$, the multi-head attention mechanism computes h (h is set to 8) single head attention functions in parallel and concatenates the results of these single head attention functions. For each single head attention function, the inputs are the queries, keys, and values projected by the trainable weights with dimensions reduced to $d_s = d_{model}/h = 16$.

$$MultiHead(U, K, V) = Concat(head_1, ..., head_h)W^O \tag{2a}$$

$$head_i = Attention(UW_i^U, KW_i^K, VW_i^V) \tag{2b}$$

  where $W_i^U \in \mathbb{R}^{d_{model} \times d_s}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_s}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_s}$, and $W_O \in \mathbb{R}^{hd_s \times d_{model}}$ are trainable weights. For the i-th layer of the circuit encoder, the queries, keys, and values inputs are identical and are equal to X^{i-1}.
- *Add and Norm*: To avoid the problem of vanishing gradients, the output of each sub-layer is added with its input and the sum is normalized. The output of the multi-head attention mechanism in layer i of circuit encoder is $Norm(X^{i-1} + MultiHead(X^{i-1}, X^{i-1}, X^{i-1}))$.
- *Feed-Forward network*: The feed-forward network consists of two linear transformations with a ReLU activation in between.

$$Feed\_Forward(T) = max(0, TW_1 + b_1)W_2 + b_2 \tag{3}$$

  where T is the output from the previous sub-layer which is the multi-head attention mechanism in the circuit encoder and W_1, W_2, b_1, and b_2 are trainable parameters.

4.1.2 *Gate Pooling.*

The encoded feature vector sequence $X^N = (x_1^N, x_2^N, ..., x_n^N)$ is transformed to a logical qubit feature vector sequence $Y = (y_1, y_2, ..., y_{|Q|})$ where element y_i represents the feature vector of logical

[2]There is no need to distinguish the control qubit and the target qubit of a CNOT gate.

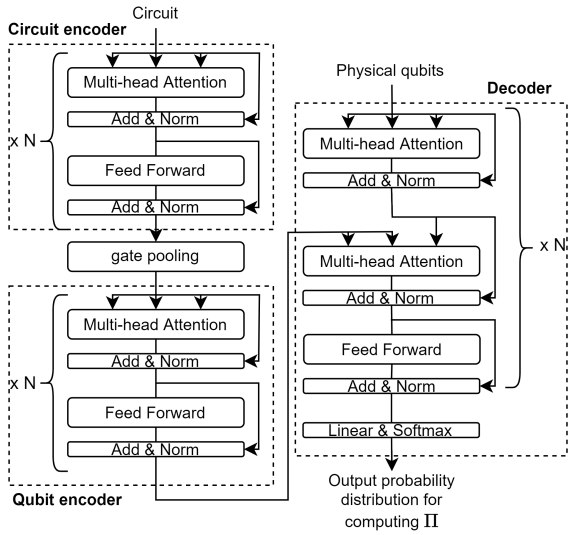

Figure 7: Detailed structures of the encoders and decoder.

qubit q_i. For each logical qubit q_i, y_i is obtained by aggregating the encoded feature vectors corresponding to the gates that act on q_i,

$$y_i = mean(\{x_j^N | x_j^N \in X^N \wedge g_j \in GS \wedge q_i \in g_j\}) \tag{4}$$

where GS is the input gate sequence.

4.1.3 *The Qubit Encoder.*

An additional encoder is proposed here to further encode the feature vectors of logical qubits. The qubit encoder's input is the sum of $Y = (y_1, y_2, ...y_{|Q|})$ and the positional information of the elements Y. The positional information of y_i is i. The qubit encoder has the same structure as the circuit encoder. We denote the output of the qubit encoder by Y'.

4.1.4 *The Decoder.*

The decoding process runs for $|Q|$ iterations. In iteration t, π_t (the physical qubit to be assigned to logical qubit q_t) is chosen by a probability distribution computed according to Y' (the encoding feature vector of logical qubits) and must be different from $\pi_1, ..., \pi_{t-1}$. Since a physical qubit can only be occupied by one logical qubit, the output physical qubit sequence $\Pi = (\pi_1, \pi_2, \ldots, \pi_{|Q|})$ must not contain duplicate entries. Thus, an action mask is maintained to set the probabilities of the physical qubits already chosen in previous iterations to 0.

The structure of the decoder is illustrated on the right side of Fig. 7. It is similar to the circuit encoder but with an additional multi-head attention mechanism. In iteration t, the input of the decoder is the sum of the embedding feature vector sequence $Z = (z_0, z_1, ..., z_{t-1})$ of the chosen physical qubits $\pi_1, \pi_2..., \pi_{t-1}$, and the positional information of the elements of Z. π_0 is the start signal token which causes the model to start decoding at the first iteration. The positional information of z_i is i. The first multi-head attention mechanism encodes Z to Z'. For the second multi-head attention mechanism, the query is Z', the key and value are both the output Y' of the qubit encoder. The third sub-layer, a feed-forward network, is similar to the one inside the circuit encoder and the fourth sub-layer does a linear transformation to obtain the output probability distribution.

The probability ψ_θ of generating a particular initial mapping $M = \{m_1, m_2, \ldots, m_{|Q|}\}$ for a given circuit C with neural network parameters θ can be computed as follows.

$$\psi_\theta(M|C) = \prod_{t=1}^{|Q|} prob_\theta(\pi_t = m_t | C, (\pi_0, m_1, \ldots, m_{t-1})) \tag{5}$$

where $prob_\theta(\pi_t = m_t | C, (\pi_0, m_1, \ldots, m_{t-1}))$ is the conditional probability that the physical qubit m_t is assigned to logical qubit q_t for circuit C when physical qubits $m_1, \ldots, m_{t-1}$ have been assigned to logical qubits $q_1, \ldots, q_{t-1}$ by the decoder. ψ_θ helps update the neural network parameters so that the agent will learn to output a better initial mapping with higher probability.

4.2 Reinforcement Learning

Our goal is to maximize the reward $R(\Pi)$. We propose an objective function which represents the negative expected reward for a specific circuit C.

$$\mathcal{L}(\theta|C) = -\mathbb{E}_{\psi_\theta(\Pi|C)}[R(\Pi)] \tag{6}$$

where ψ_θ is the stochastic policy defined in Eq 5. $\mathcal{L}$ is minimized by gradient descent, using Adam optimizer [8] based on the REINFORCE algorithm [22] with a baseline. The gradient of $\mathcal{L}$ is computed as follows.

$$\nabla\mathcal{L}(\theta|C) = \mathbb{E}_{\psi_\theta(\Pi|C)}[(r(\Pi) - R(\Pi))\nabla log\psi_\theta(\Pi|C))] \tag{7}$$

where $r(\Pi)$ is a baseline that can reduce gradient variance and accelerate the training time. To find a good baseline $r(\Pi)$, we adopt the actor-critic model that trains an additional model, the critic, to predict the value of $R(\Pi)$. On the other hand, the model that produces an initial mapping is called the actor.

The critic has an encoder whose structure is the same as the circuit encoder described in section 4.1.1. It also takes GS and LS as input and encodes them to a hidden feature vector X^ω. The encoder is followed by a linear layer which takes the mean of encoded feature vectors in X^ω as input and outputs the predicted value $r(\Pi)$. We train our critic model to minimize the mean square error between $r(\Pi)$ and $R(\Pi)$ and update the parameters of critic with Adam optimizer.

5 QUBIT ROUTING

The qubit routing problem can be solved by customized heuristic search algorithms [9][28][12][27][10][26] but most have very shallow search depth. A* search was applied in [29]. To obtain a solution in reasonable time, [29] first partitions a given logical circuit into layers as described in section **??**. For each layer i, it conducts a A* search to determine a set of SWAP gates, $SWAP_{layer\ i}$ to be inserted at the end of layer i. A goal state for layer i is reached when all gate in layer i become executable. It assumes that no gates in layer i will be executed before reaching a goal state for layer i. In other words, it would find and insert $SWAP_{layer\ 1}$ before executing all the gates in layer 1, then repeat the same process for layer 2,

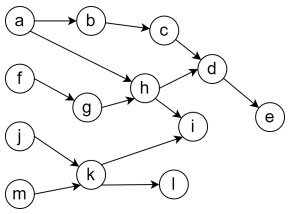

Figure 8: A dependency graph.

Algorithm 1 DEAR framework

Input: Logical circuit LC_{in}, initial mapping Π_{in}, architecture graph $\mathcal{AG}$

Output: A physical circuit with initial mapping Π_{in} equivalent to LC_{in} and satisfies $\mathcal{AG}$

1: $PC \leftarrow$ circuit of all executable gates in LC_{in} under mapping Π_{in}
2: $LC \leftarrow LC_{in}$ with all executable gates under mapping Π_{in} removed
3: $\Pi \leftarrow \Pi_{in}$
4: **while** LC is not empty **do**
5: $PC', \Pi' \leftarrow$ Extract-and-Route(LC, Π, $\mathcal{AG}$)
6: Update PC by concatenating PC'
7: Update LC by removing all gates covered by PC'
8: $\Pi \leftarrow \Pi'$
9: **return** PC

Procedure 1 Extract-and-Route

Input: Logical circuit LC, current mapping Π, architecture graph $\mathcal{AG}$

Output: A partial transformation of LC with added SWAP gates to satisfy $\mathcal{AG}$, an updated mapping

1: **if** $|LC| > 200$ **then**
2: $LC_{sub} \leftarrow$ extract(LC)
3: $min\_\#lookahead\_gates \leftarrow 30$
4: **else**
5: $LC_{sub} \leftarrow LC$
6: $min\_\#lookahead\_gates \leftarrow 0$
7: $s_{init} \leftarrow$ (empty circuit, Π, LC_{sub})
8: $S \leftarrow \phi$
9: INSERT(S, s_{init})
10: **while** S is not empty **do**
11: $s \leftarrow$ EXTRACT_MIN(S)
12: **if** $|s.LC| \leq min\_\#lookahead\_gates$ **then**
13: **return** $s.PC$, $s.\Pi$
14: **for** $SWAP(q_i, q_j) \in SWAP_{s.LC}$ **do**
15: $s'.\Pi \leftarrow s.\Pi$ with the mapping of q_i and q_j swapped
16: $s'.PC \leftarrow s.PC$ concatenated with $SWAP(q_i, q_j)$ and all gates in $s.LC$ made executable under mapping $s'.\Pi$
17: $s'.LC \leftarrow s.LC$ with all gates made executable under mapping $s'.\Pi$ removed
18: INSERT(S, s')

so on and so forth. To improve the solution quality, it introduces a lookahead scheme to consider the effect of $SWAP_{layer\ i}$ on the gates in layer $i+1$.

We note that the order of execution of the two-qubit gates in a logical circuit can follow any topological ordering in the corresponding dependency graph. For example, if a logical circuit has a dependency graph as shown in Fig. 8, the execution order can be $abcfg\ldots$ or $fabg\ldots$, etc. But [29] imposes a needless restriction that the gates must be executed in a layer-wise fashion which will ignore many better solutions. This greatly affects the quality of the final solution that can be found. Moreover, the number of gates in each layer is very limited[3]. So, the lookahead capacity of [29] is actually limited.

To address the above drawbacks, we propose a DEAR (Dynamically Extract And Route) framework coupled with A* search. The pseudo-codes are shown in Algorithm 1 and Procedure 1. In our framework, instead of pre-splitting the logical circuit into layers, we dynamically extract a targeted subcircuit of the remaining logical circuit and apply our A* search method to perform qubit routing until the remaining logical circuit becomes empty. As a result, a greater lookahead capacity can be provided all the time compared to [29] to enable the search process to go deeper. In addition, the heuristic cost in our A* search is based on quick simulation to estimate how many SWAP gates are needed to execute the targeted subcircuit which is more accurate than [29]. We will elaborate on Procedure 1 and the computation of the heuristic cost below.

[3]For quantum circuits to be executed on IBM QX20, the number of gates in a single layer is no more than 10.

In each call to Procedure 1, if the size of the remaining logical circuit is larger than a threshold (we set the threshold to 200 in our implementation for circuits to be executed on IBM QX20), then we carefully extract a targeted subcircuit LC_{sub} (lines 1-3) of the remaining logical circuit and apply A* search (lines 7 to 18) to determine when and where to insert SWAP gates one by one to transform LC_{sub} till the number of two-qubit gates that remain un-executable is no more than $min\_\#lookahead\_gates$. $min\_\#lookahead\_gates$ is the minimum desired number of lookahead gates during A* search and is set in line 3 (it is set to 30 in our implementation for circuits to be executed on IBM QX20). In this case, we stop transforming the current targeted subcircuit LC_{sub} when we reach a goal state with no more than $min\_\#lookahead\_gates$ un-executable gates remaining (lines 12-13). However, if the size of the remaining logical circuit is already small enough, we just set the targeted subcircuit LC_{sub} as the remaining logical circuit and set $min\_\#lookahead\_gates$ to 0 (lines 4-6) so that a goal state with all gates in LC_{sub} become executable will be found (lines 12-13).

It is also important how we extract a suitable targeted subcircuit LC_{sub} in line 2 of Procedure 1. The targeted subcircuit extracted should not be too small because it will degrade the final solution quality, nor should it be too large because it will increase the runtime of A* search. To find an appropriate targeted subcircuit, we run a SABRE-like[9] heuristic[4] to insert 40 SWAP gates into the remaining logical circuit, then we take the set of gates which could be made executable in this way as our targeted subcircuit LC_{sub}.

[4]A SABRE-like heuristic is very fast but its quality is not comparable to A* search, so we do not use it for actual routing and only use it to help determine a suitable subcircuit.

Procedure 2 Simulation based on Probability Distribution

Input: State s, architecture graph $\mathcal{AG}$

Output: The number of SWAP gates required to make all gates in $s.LC$ executable.

1: $n \leftarrow 0$
2: $s_{sim} \leftarrow s$
3: **while** $s_{sim}.LC$ is not empty **do**
4: $SWAP(q_i, q_j) \leftarrow$ SAMPLE($SWAP_{s_{sim}.LC}$)
5: $s_{sim}.\Pi \leftarrow s_{sim}.\Pi$ with the mapping of q_i and q_j swapped
6: $s_{sim}.LC \leftarrow s_{sim}.LC$ with all gates made executable under mapping $s_{sim}.\Pi$ removed
7: $n \leftarrow n + 1$
8: **return** n

A* search is conducted in lines 7 to 18 of Procedure 1. A state $s = (s.PC, s.\Pi, s.LC)$ in the A*-search is defined as follows. $s.PC$ is a physical circuit that corresponds to a partial transformation of the current targeted subcircuit LC_{sub}, $s.\Pi$ is the mapping at the end of $s.PC$, and $s.LC$ is obtained by removing all gates covered by $s.PC$ from LC_{sub}. Each state s is assigned a cost $f(s) = g(s) + h(s)$. $g(s)$ represents the number of SWAP gates required to get to state s from the initial state s_{init}. $h(s)$ is a heuristic cost function that estimates the number of SWAP gates required to make all gates in $s.LC$ executable.

A* search uses a priority queue to keep track of the unexpanded states. In each iteration, it chooses an unexpanded state v with the lowest cost $f(v)$ and expands from it until reaching one of the preferred goal states (lines 10 to 18 of Procedure 1). For a state $s = (s.PC, s.\Pi, s.LC)$, $SWAP_{s.LC}$ denotes the set of valid SWAP operations that affect at least one logical qubit in a two-qubit gate in the front layer of $s.LC$ (i.e., a two-qubit gate in $s.LC$ that does not have any predecessor). Only SWAP operations in $SWAP_{s.LC}$ are used to expand from s (line 14). It is an efficient strategy to avoid applying useless SWAP operations and has been employed in [9] [27] [29]. For each child state s' of s, $g(s') = g(s) + 1$ since one SWAP gate is inserted to move from s to s' and the computation of $h(s')$ which estimates the number of SWAP operations required to make all gates in $s'.LC$ executable is discussed next.

We note that the more accurate is the estimated value of $h(s)$, the sooner A* search will stop. However, an overestimation of $h(s)$ will affect the optimality of the final solution. Similar to [26], given a state s, we performs simulation (Procedure 2) β times to get an upper bound on the number of SWAP operations required to make all gates in $s.LC$ executable and deduce $h(s)$ accordingly. Note that $s.LC$ is limited in size in our DEAR framework since we only deal with a small subcircuit at a time, so simulation will be fast. In each simulation, relevant SWAP operations are inserted interatively based on a probability distribution until all gates in $s.LC$ can be executed. Let the total number of SWAP gates inserted in the i-th simulation be n_i, then we set $h(s)$ as $min(n_1, n_2, ..., n_\beta) * \lambda$ where λ is a user-defined parameter less than one to prevent overestimation. (λ is 0.17 in our experiments.)

In line 3 of Procedure 2, a relevant SWAP gate g_{swap} is sampled from $SWAP_{s.LC}$ based on a probability distribution PD defined below.

$$PD(X = g_{swap}) = \frac{IF(g_{swap})}{\sum_{g'_{swap} \in SWAP_{s.LC}} IF(g'_{swap})} \tag{8a}$$

$$IF(g'_{swap}) = e(\sum_{g \in F}(D[\Pi(g.q1)][\Pi(g.q2)] - D[\Pi'(g.q1)][\Pi'(g.q2)])) \tag{8b}$$

where F is the front layer of $s.LC$, Π is $s.\Pi$, Π' is the mapping after applying g'_{swap} to Π, $D[p_i][p_j]$ is the shortest distance between physical qubits p_i and p_j in the architecture graph and $e(x) = 2^x$. So, the SWAP gates that will reduce the total distance of the two logical qubits in each gate in the front layer of $s.LC$ have exponential higher chance to be selected than the ones that will increase the total cost. In actual implementation, we store the heuristic costs of all interim states. If a state has been computed before, then the heuristic cost can be obtained directly. This mechanism reduces the execution time.

Table 1: Comparison of the quality of the initial mappings by [2], [28], and our RL approach using two different qubit routing algorithms. (In this experiment, we executed t|ket> and DEAR on a Linux machine.)

		[2]	[28]	RL	[2]	[28]	RL
		with t|ket>			with DEAR		
Circuits	#gates	g	g	g	g_{avg}	g_{avg}	g_{avg}
radd_250	1405	993	1065	**900**	501	519.6	**432.6**
rd73_252	2319	1344	**1272**	1323	778.2	780.6	**610.8**
cycle10_2_110	2648	1440	1611	**1293**	778.2	877.8	**711**
hwb6_56	2952	1665	1740	**1143**	880.8	865.8	**825.6**
cm85a_209	4986	3075	2907	**2706**	1415.4	1267.2	**1160.4**
rd84_253	5960	3912	3804	**3381**	2196.6	2141.4	**1717.2**
root_255	7493	4554	4164	**3531**	2686.2	2731.2	**2050.2**
mlp4_245	8232	**4881**	5862	5511	2862.6	2668.8	**2500.2**
urf2_277	9408	6168	**6108**	6129	4431.6	4446.6	**4410**
sym9_148	10066	4200	3366	**3033**	1446	1459.2	**1338**
hwb7_59	10681	5226	5619	**4950**	2932.8	2962.2	**2775.6**
clip_206	14772	9339	9444	**7959**	4842	4446	**4213.2**
sym9_193	15232	9252	9462	**8676**	4510.8	4032.6	**3795.6**
dist_223	16624	10668	10569	**8082**	5522.4	4912.2	**4369.2**
sao2_257	16864	11268	9999	**7812**	5200.8	4785	**4610.4**
urf5_280	23764	15261	15201	**13791**	8629.8	8631	**8482.2**
urf1_278	26692	**17106**	17217	17391	10167	10191	**10068.6**
sym10_262	28084	16899	17079	**13833**	7491	7764.6	**6619.8**
hwb8_113	30372	17886	20007	**17319**	8993.4	8687.4	**8080.2**
Normalized		1	1.066	**0.88**	1	0.98	**0.881**

6 EXPERIMENTAL RESULTS

We implemented our RL-model and DEAR framework in Python and C++, respectively. We choose 140 and 19 circuits from IBM-QX circuit set [5] to be our training set and testing set, respectively. The circuits in the training set are used to train our RL-model and are not used in testing. The circuits in the testing set are the largest in the IBM-QX circuit set and each contains more than 1000 two-qubit gates. They are used to evaluate our RL-model and DEAR framework on the IBM QX20 architecture.

Table 2: Comparison of our qubit routing results with those by MCTS[26]. (The implementation of MCTS[23] is not compatible with Linux. For fairness, we executed both MCTS and DEAR on a Windows machine.)

	MCTS[26]			DEAR		
	g_{avg}	g_{min}	$t_{avg}(s)$	g_{avg}	g_{min}	$t_{avg}(s)$
radd_250	450.6	426	23.25	**436.8**	**405**	**13.09**
rd73_252	704.4	654	38.26	**612.6**	**591**	**18.14**
cycle10_2_110	748.8	**672**	39.32	**712.2**	684	**22.79**
hwb6_56	862.2	825	46.28	**827.4**	**801**	**27.09**
cm85a_209	1466.4	1398	95.72	**1163.4**	**1116**	**47.68**
rd84_253	1940.4	1857	134.34	**1720.2**	**1683**	**78.54**
root_255	2358.6	2172	178.21	**2051.4**	**1974**	**96.77**
mlp4_245	3177.6	3066	252.54	**2500.8**	**2394**	**146.93**
urf2_277	4582.8	4524	382.17	**4408.2**	**4320**	**340.97**
sym9_148	**1202.4**	**1107**	127.34	1342.2	1314	**35.44**
hwb7_59	2806.2	2691	234.09	**2771.4**	**2655**	**151.67**
clip_206	4851.6	4674	477.12	**4214.4**	**4041**	**332.03**
sym9_193	4016.4	3849	389.32	**3795**	**3654**	**235.28**
dist_223	5235	4842	534.87	**4368**	**4200**	**324.01**
sao2_257	5088.6	**4482**	553.21	**4614.6**	4560	**350.67**
urf5_280	8965.8	8535	1148.79	**8481.6**	**8391**	**1104.28**
urf1_278	11197.8	11151	1604.71	**10063.8**	**9996**	**1559.31**
sym10_262	7348.8	7125	1143.87	**6622.8**	**6450**	**701.30**
hwb8_113	8392.8	8172	1389.15	**8086.2**	**7989**	**1095.94**
Normalized	1	1	1	**0.916**	**0.938**	**0.637**

For the training of the RL-model, an open source routing tool, t|ket> [19], is selected to be the environment. For each circuit in the training set, we shuffle the logical qubits in different ways to obtain more training data. We also split the circuits containing more than 256 gates into subcircuits of less than 256 two-qubit gates to obtain more training data and reduce GPU memory usage. As a result, the final training set consists of 14k circuits. For each epoch, we update the parameters of model with batch size 128 and constant learning rate of 10^{-4}.

In the first experiment, we compare the initial mappings produced by our RL-model with the ones produced by [2] and [28]. We evaluate the initial mappings produced by different methods with two routing tools, t|ket> and DEAR. The initial mappings produced by our RL-model were obtained by greedily selecting the action with the largest probability. Both t|ket> and DEAR were executed on a 64-bit Ubuntu server with AMD ThreadRipper 3970X CPU and 256GB memory. The results are listed in Table 1. The *#gates* column gives the number of two-qubit gates in each benchmark originally. For t|ket>, we ran it once and list the number of additional gates, g, for routing. For DEAR, we ran it 5 times and list the average number of additional gates, g_{avg}, for routing.

Table 1 clearly shows that our RL-model is able to produce better initial mappings than both [2] and [28]. Compared to [2], our RL-model produces better initial mappings in 17 out of 19 benchmarks with 12% reduction in the number of additional gates on average after routing by t|ket>. When compared to [28], our RL-model produces better initial mappings in 16 out of 19 benchmarks with 12.5% reduction in the number of additional gates on average after routing by t|ket>. The number of additional gates is 34.4% less for benchmark hwb6_56 with our initial mapping compared to [2] using t|ket> as the evaluating routing tool. We can see that the advantage of our RL-model is preserved when using DEAR as the evaluating routing tool. On average, the number of additional gates required in routing is reduced by 11.9% and 10.2% using our RL-model compared to using initial mappings produced by [2] and [28], respectively. It is worthy to note that the initial mappings by our RL-model are better than those by [2] and [28] for all benchmarks when evaluated by DEAR even though DEAR was not used as the environment in the RL-model. Finally, we can see that DEAR produces significantly better routing results over t|ket> using much smaller number of additional gates.

In the second experiment, we compare DEAR framework with Monte Carlo Tree Search (MCTS) [26] which produced the best result among all previous works. We downloaded the source code of MCTS from [23] and ran it. Since [23] is not compatible with Linux, we executed both MCTS and DEAR on a 64-bit Windows machine with AMD Ryzen 5600X CPU and 16GB memory. The experimental results are shown in Table 2. Similar to the experiments in [26], we run our DEAR framework and MCTS five times for each benchmark with the same initial mapping produced by our RL-model. We list the average required number of additional gates g_{avg}, the minimum required number of additional gates g_{min}, and the average execution time t_{avg} in seconds.

Table 2 shows that DEAR can solve the qubit routing problem with less execution time and better solution quality compared to MCTS. The execution time (column t_{avg}) decreased by 36.3% on average. For the average number of additional gates (column g_{avg}), DEAR performs better than MCTS in 18 out of 19 benchmarks. DEAR requires 8.4% fewer additional gates than MCTS on average, which means DEAR leads to better results than MCTS in general. In addition, DEAR achieves an improvement of 21.3% for benchmark "$mlp4\_245$". For the minimum number of additional gates (column g_{min}), DEAR is able to obtain better results than MCTS in 16 out of 19 benchmarks and obtains an improvement up to 22% for benchmark "$mlp4\_245$". In summary, DEAR outperforms MCTS both in terms of solution quality and execution time.

7 CONCLUSION

In this paper, we proposed a reinforcement learning model for solving the initial mapping problem and a DEAR framework for solving the qubit routing problem. Compared to [2] and [28], the initial mappings produced by our RL-model result in 12% and 17.5% reduction in the number of additional gates after routing by t|ket>, respectively. Moreover, the advantage of the initial mappings produced by our RL-model over other initial mapping algorithms is maintained when using DEAR as the routing tool. In addition, the DEAR framework saves 36.3% execution time and 8.4% additional gates compared with the state-of-the-art Monte Carlo tree search-based qubit routing algorithm[26].

REFERENCES

[1] Dario Amodei, Rishita Anubhai, and Eric Battenberg et al. 2015. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. *arXiv preprint arXiv:1512.02595.*

[2] Xueyun Cheng, Zhijin Guan, and Pengcheng Zhu. 2020. Nearest Neighbor Transformation of Quantum Circuits in 2D Architecture. *IEEE Access* 8 (2020), 222466–222475.

[3] Haowei Deng, Yu Zhang, and Quanxi Li. 2020. Codar: A Contextual Duration-Aware Qubit Mapping for Various NISQ Devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*.

[5] Johannes Kepler University Linz Institute for Integrated Circuits. 2019. IIC JKU - IBMQX QASM Circuits. https://github.com/iic-jku/ibm_qx_mapping/tree/master/examples

[6] Nicolas Gisin, Grégoire Ribordy, Wolfgang Tittel, and Hugo Zbinden. 2002. Quantum cryptography. *Rev. Mod. Phys.* 74 (2002), 145–195. Issue 1.

[7] IBM. 2021. IBM Quantum Services. https://quantum-computing.ibm.com/services?services=systems

[8] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. In *arXiv preprint arXiv:1412.6980*.

[9] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. 1001–1014.

[10] Sanjiang Li, Xiangzhen Zhou, and Yuan Feng. 2021. Qubit Mapping Based on Subgraph Isomorphism and Filtered Depth-Limited Search. *IEEE Trans. Comput.* 70, 11 (2021), 1777–1788.

[11] Prakash Murali, Jonathan M. Baker, Ali Javadi Abhari, Frederic T. Chong, and Margaret Martonosi. 2019. Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers. *arXiv preprint arXiv:1901.11054*.

[12] Siyuan Niu, Adrien Suau, Gabriel Staffelbach, and Aida Todri-Sanial. 2020. A Hardware-Aware Heuristic for the Qubit Mapping Problem in the NISQ Era. *IEEE Transactions on Quantum Engineering* 1 (2020), 1–14.

[13] Matteo G. Pozzi, Steven J. Herbert, Akash Sengupta, and Robert D. Mullins. 2020. Using Reinforcement Learning to Perform Qubit Routing in Quantum Compilers. *arXiv preprint arXiv:2007.15957*.

[14] S. J. Rennie, E. Marcheret, Y. Mroueh, J. Ross, and V. Goel. 2017. Self-Critical Sequence Training for Image Captioning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 1179–1195.

[15] Abdullah Ash Saki, Aakarshitha Suresh, Rasit Onur Topaloglu, and Swaroop Ghosh. 2021. Split Compilation for Security of Quantum Circuits. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–7.

[16] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (1997), 1484–1509.

[17] Animesh Sinha, Utkarsh Azad, and Harjinder Singh. 2021. Qubit Routing using Graph Neural Network aided Monte Carlo Tree Search. *arXiv preprint arXiv:2104.01992*.

[18] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintao Pereira. 2018. Qubit Allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*. 113–125.

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

[20] Bochen Tan and Jason Cong. 2020. Optimal Layout Synthesis for Quantum Computing. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017).

[22] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8, 3 (1992), 229–256.

[23] Yuan Feng Xiangzhen Zhou and Sanjiang Li. 2020. Circuit-Transformation-via-Monte-Carlo-Tree-Search. https://github.com/iic-jku/ibm_qx_mapping/tree/master/examples

[24] Christof Zalka. 1999. Grover's quantum searching algorithm is optimal. *Physical Review A* 60, 4 (1999), 2746–2751.

[25] Chi Zhang, Ari B. Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Eddy Z. Zhang. 2021. Time-Optimal Qubit Mapping. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2021)*. 360–374.

[26] Xiangzhen Zhou, Yuan Feng, and Sanjiang Li. 2020. A Monte Carlo Tree Search Framework for Quantum Circuit Transformation. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–7.

[27] Xiangzhen Zhou, Sanjiang Li, and Yuan Feng. 2020. Quantum Circuit Transformation Based on Simulated Annealing and Heuristic Search. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4683–4694.

[28] Pengcheng Zhu, Zhijin Guan, and Xueyun Cheng. 2020. A Dynamic Look-Ahead Heuristic for the Qubit Mapping Problem of NISQ Computers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 12 (2020), 4721–4735.

[29] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2019. An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 7 (2019), 1226–1236.