Qubit Routing using Graph Neural Network aided Monte Carlo Tree Search

Animesh Sinha,* Utkarsh Azad,† and Harjinder Singh‡
Center for Computational Natural Sciences and Bioinformatics,
International Institute of Information Technology, Hyderabad.
(Dated: May 25, 2021)

Near-term quantum hardware can support two-qubit operations only on the qubits that can interact with each other. Therefore, to execute an arbitrary quantum circuit on the hardware, compilers have to first perform the task of qubit routing, i.e., to transform the quantum circuit either by inserting additional SWAP gates or by reversing existing CNOT gates to satisfy the connectivity constraints of the target topology. We propose a procedure for qubit routing that is architecture agnostic and that outperforms other available routing implementations on various circuit benchmarks. The depth of the transformed quantum circuits is minimised by utilizing the Monte Carlo tree search to perform qubit routing, aided by a Graph neural network that evaluates the value function and action probabilities for each state.

Keywords: Qubit Routing, Reinforcement Learning, Monte Carlo Tree Search, Graph Neural Network

I. INTRODUCTION

The present-day quantum computers, more generally known as Noisy Intermediate-Scale quantum (NISQ) devices [1] come in a variety of hardware architectures [2–5], but there exist a few problems pervading across all of them. These problems constitute the poor quality of qubits, limited connectivity between qubits, and the absence of error-correction for noise-induced errors encountered during the execution of gate operations. These place a considerable restriction on the number of instructions that can be executed to perform useful quantum computation [1]. Collectively these instructions can be realized as a sequential series of one or two-qubit gates that can be visualized more easily as a quantum circuit as shown in Fig. 1a [6].

To execute an arbitrarily given quantum circuit on the target quantum hardware, a compiler routine must transform it to satisfy the connectivity constraints of the topology of the hardware [7]. These transformations usually include the addition of SWAP gates and the reversal of existing CNOT gates. This ensures that any non-local quantum operations are performed only between the qubits that are nearest-neighbors. This process of circuit transformation by a compiler routine for the target hardware is known as qubit routing [7]. The output instructions in the transformed quantum circuit should follow the connectivity constraints and essentially result in the same overall unitary evolution as the original circuit [8].

In the context of NISQ hardware, this procedure is of extreme importance as the transformed circuit will, in general, have higher depth due to the insertion of extra SWAP gates. This overhead in the circuit depth becomes more prominent due to the high decoherence rates of the qubits and it becomes essential to find the most optimal and efficient strategy to minimize it [7–9]. In this article, we present a procedure that we refer to as *QRoute*. We use Monte Carlo tree search (MCTS), which is a look-ahead search algorithm for finding optimal decisions in the decision space guided by a heuristic evaluation function [10–12]. We use it for explicitly searching the decision space for depth minimization and as a stable and performant machine learning setting. It is aided by a Graph neural network (GNN) [13], with an architecture that is used to learn and evaluate the heuristic function that will help guide the MCTS.

Structure: In Section II, we introduce the problem of qubit routing, the previous works that are done in the field, and show how our work differs from them. The methodology of the QRoute algorithm is described in Section III. In Section IV, we benchmark the performance of our algorithm against other state-of-the-art quantum compilers. Finally, we discuss our results and possible improvements in Section V.

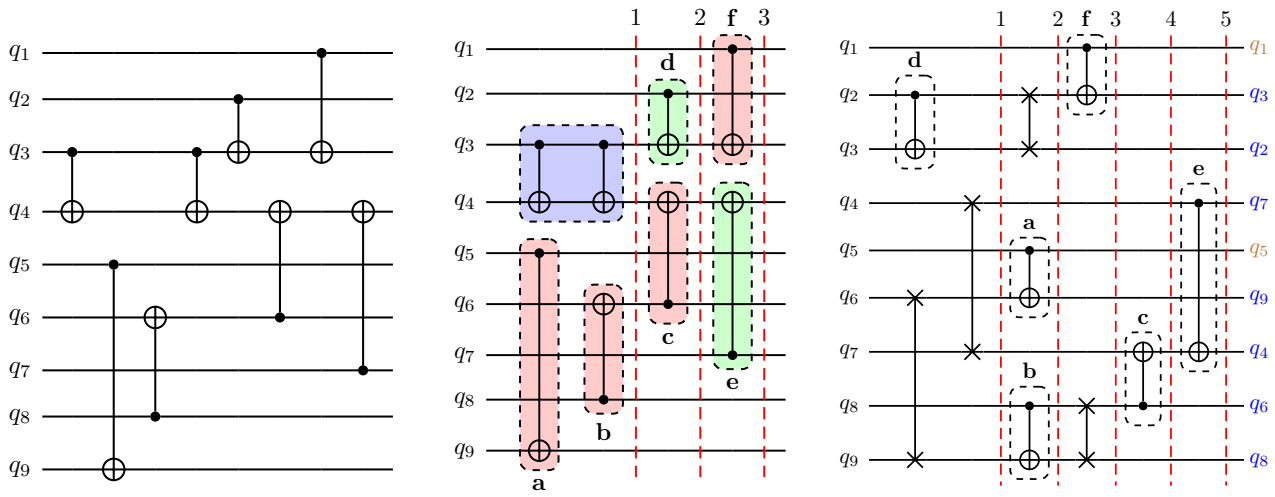
II. QUBIT ROUTING

In this section, we begin by defining the problem of qubit routing formally and discussing the work done previously in the field.

A. Describing the Problem

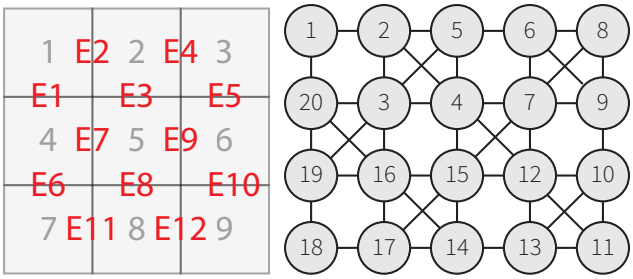
The topology of quantum hardware can be visualized as a qubit connectivity graph (Fig. 2). Each node in this graph would correspond to a physical qubit which in turn might correspond to a logical qubit. The quantum instruction set, which is also referred to as quantum circuit (Fig. 1a), is a sequential series of single-qubit and two-qubit gate operations that act on the logical qubits. The two-qubit gates such as CNOT can only be performed between two logical qubits iff there exists an

* animesh.sinha@research.iiit.ac.in
† utkarsh.azad@research.iiit.ac.in
‡ laltu@iiit.ac.in



(a) Quantum circuit (b) Decomposed circuit (c) Decomposed transformed circuit

FIG. 1: An example of qubit routing on a quantum circuit for 3×3 grid architecture (Figure 2a). (a) For simplicity, the original quantum circuit consists only of two-qubit gate operations. (b) Decomposition of the original quantum circuit into series of slices such that all the instructions present in a slice can be executed in parallel. The two-qubit gate operations: $\{d, e\}$ (green) comply with the topology of the grid architecture whereas the operations: $\{a, b, c, f\}$ (red) do not comply with the topology (and therefore cannot be performed). Note that the successive two-qubit gate operations on $q_3 \rightarrow q_4$ (blue) are redundant and are not considered while routing. (c) Decomposition of the transformed quantum circuit we get after qubit routing. Four additional SWAP gates are added that increased the circuit depth to 5, i.e., an overhead circuit depth of 2. The final qubit labels are represented at the end right side of the circuit. The qubits that are not moved (or swapped) are shown in brown ($\{q_1, q_5\}$), while the rest of them are shown in blue. Note that the overall unitary operation performed by the circuit is preserved despite the changes in the order of two-qubit gate operations.



(a) 3×3 grid architecture (b) IBMQX-20 architecture

FIG. 2: Examples of qubit connectivity graphs for some common quantum architectures

edge between the nodes that correspond to the physical qubits, [9]. This edge could be either unidirectional or bidirectional, i.e., CNOT can be performed either in one direction or in both directions. In this work, we consider only the bidirectional case, while noting that the direction of a CNOT gate can be reversed by sandwiching it between a pair of Hadamard gates acting on both control and target qubits [14].

Given a target hardware topology $\mathcal{D}$ and a quantum circuit $\mathcal{C}$, the task of qubit routing refers to transforming this quantum circuit by adding a series of SWAP gates such that all its gate operations then satisfy the connectivity constraints of the target topology (Fig. 1c). Formally, for a routing algorithm R, we can represent this process as follows:

$$\mathrm{R}(\mathcal{C},\ \mathcal{D}) \rightarrow \mathcal{C}'$$

Depth of $\mathcal{C}'$ (transformed quantum circuit) will, in general, be more than that of the original circuit due to the insertion of additional SWAP gates. This comes from the definition of the term *depth* in the context of quantum circuits. This can be understood by decomposing a quantum circuit into series of individual slices, each of which contains a group of gate operations that have no overlapping qubits, i.e., all the instructions present in a slice can be executed in parallel (Fig. 1b). The depth of the quantum circuit then refers to the minimum number of such slices the circuit can be decomposed into, i.e., the minimum amount of parallel executions needed to execute the circuit. The goal is to minimize the overhead depth of the transformed circuit with respect to the original circuit.

This goal involves solving two subsequent problems of (i) qubit allocation, which refers to the mapping of program qubits to logic qubits, and (ii) qubit movement, which refers to routing qubits between different locations such that interaction can be made possible [15]. In this

work, we focus on the latter problem of qubit movement only and refer to it as qubit routing. However, it should be noted that qubit allocation is also an important problem and it can play an important role in minimizing the effort needed to perform qubit movement.

B. Related Work

The first major attraction for solving the qubit routing problem was the competition organized by IBM in 2018 to find the best routing algorithm. This competition was won by Zulehner *et al.* [16], for developing a Computer Aided Design-based (CAD) routing strategy. Since then, this problem has been presented in many different ways. These include graph-based architecture-agnostic solution by Cowtan *et al.* [7], showing equivalence to the travelling salesman problem by Paler *et al.* [17], machine learning methods by Paler *et al.* [18], etc. A reinforcement learning in a combinatorial action space solution was proposed by Herbert and Sengupta [9], which suggested used simulated annealing to search through the combinatorial action space, aided by a Feed-Forward neural network to judge the long-term expected depth. This was further extended to use Double Deep Q-learning and prioritized experience replay by Pozzi *et al.* [8].

Recently, Monte Carlo tree search (MCTS), a popular reinforcement learning algorithm [19] previously proven successful in a variety of domains like playing puzzle games such as Chess and Go [20], and was used by Zhou *et al.* [21] to develop a qubit routing solution. However, they used MCTS in the context of minimizing the total volume of quantum circuits (i.e., the number of gates ignoring the parallelization).

C. Our Contributions

We propose to use the Monte Carlo tree search (MCTS), a reinforcement learning (RL) algorithm, for the task of total depth minimization. In the context of RL, we use an array of mutex locks to represent the information that an agent needs to know to make effective use of parallelization. Moreover, we propose a Graph neural network (GNN) that aids the MCTS in its exploration by evaluating function and action probabilities throughout the routing procedure. From these additional features, we aim to gain an agent that is not greedy immediately since it can only see the current state of the system and not effectively plan for the future. This also stabilizes the decision process which would have suffered from randomness if a method such as simulated annealing was used. Finally, we provide a simple python package that can be used to implement variants of QRoute with different neural net architectures, combining algorithms, reward structures, etc. [22]

III. METHOD

The QRoute algorithm takes in an input circuit and an injective map, $\mathcal{M} : Q \to N$, from logical qubits to nodes (physical qubits). Iteratively, over multiple timesteps, it tries to schedule the gate operations that are present in the input circuit onto the target hardware. To do so, from the set of unscheduled gate operations, $\mathcal{P}$, it takes all the current operations, which are the first unscheduled operation for both the qubits that they act on, and tries to make them into local operations, which are those two-qubit operations that involve qubits that are mapped to nodes connected on the target hardware.

In every timestep t, QRoute starts by greedily scheduling all the operations that are both current and local in $\mathcal{P}$. To evolve $\mathcal{M}$, it then performs a Monte Carlo tree search (MCTS) to find an optimal set of SWAPs by the evaluation metrics described in the Section III B such that all operations in the current timestep put together form a parallelizable set, i.e., a set of local operations such that no two operations in the set act on the same qubit. The number of states we can encounter in the action space explodes exponentially with the depth of our search, therefore an explicit search till the circuit is done compiling is not possible. Therefore we cut short our search at some shallow intermediate state, and use a neural network to get its heuristic evaluation.

The following subsections describe in greater detail the working of the search and the heuristic evaluation.

A. State and Action Space

Definition III.1 (State) *It captures entire specification of the state of compilation at some timestep t. Abstractly, it is described as:*

$$s_t = (\mathcal{D},\ \mathcal{M}_t,\ \mathcal{P}_t,\ \mathcal{L}_t)$$

where, $\mathcal{D}$ is the topology of target hardware, and $\mathcal{M}_t$ and $\mathcal{P}_t$ represents the current values of $\mathcal{M}$ and $\mathcal{P}$ respectively. $\mathcal{L}_t$ is the set of nodes that are locked by the gate operations from the previous timestep and therefore cannot be operated in the current timestep.

Definition III.2 (Action) *It is a set of SWAP gates (represented by the pair of qubits it acts on) such that all gates are local, and its union with the set of operations that were scheduled in the same timestep forms a parallelizable set.*

We are performing a tree search over state-action pairs. Since the number of actions that can be taken at any timestep is exponential in the number of connections on the hardware, we are forced to build a single action up, step-by-step.

Definition III.3 (Move) *It is a single step in a search procedure which either builds up the action or applies it to the current state. Moves are of the following two types:*

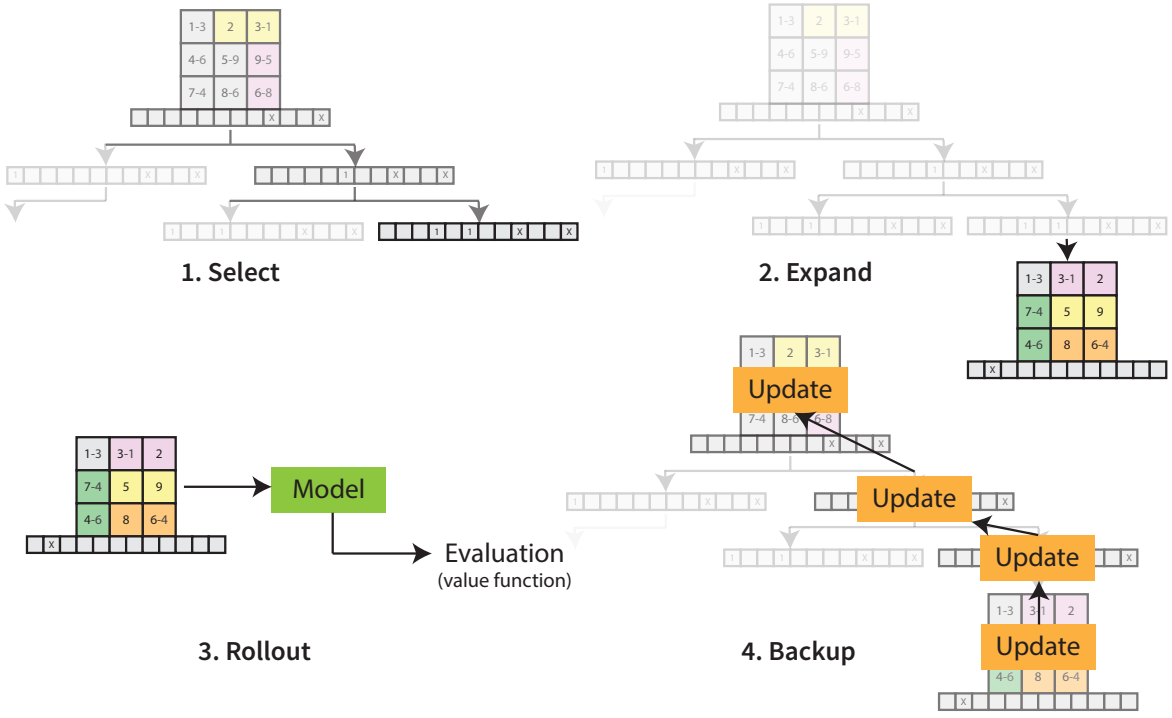


FIG. 3: Iteration of a Monte Carlo tree search: (i) select - recursively choosing a node in the search tree for exploration using a selection criteria, (ii) expand - expanding the tree using the available actions if an unexplored leaf node is selected, (iii) rollout - estimating long term reward using a neural network for the action-state pair for the new node added to search tree, and (iv) backup - propagating reward value from the leaf nodes upwards to update the evaluations of its ancestors.

1. *SWAP(n_1, n_2): Inserts a new SWAP on nodes n_1 and n_2 into the action set. Such an insertion is only possible if the operation is local and resulting set of operations for the timestep form a parallel set.*

2. *COMMIT: Finishes the construction of the action set for that timestep. It also uses the action formed until now to update the state s_t (schedules the SWAP gates on the hardware), and resets the action set for the next step.*

In reality, different gate operations take different counts of timesteps for execution. For example, if a hardware requires SWAP gate to be broken down into CNOT gates, then it would take three timesteps for complete execution [14]. This means, operations which are being scheduled must maintain mutual exclusivity with other other operations over the nodes which participates in them. This is essential to minimizing the depth of the circuit since it models paralleizability of operations.

However, constructing a parallelizable set and representing the state of parallelization to our heuristic evaluator is a challenge. But an analogy can be drawn here to the nodes being thought of as "resources" that cannot be shared, and the operations as "consumers" [23]. This motivates us to propose the use of Mutex Locks for this purpose. These will lock a node until a scheduled gate operation involving that node executes completely. Therefore, this allows our framework to naturally handle different types of operations which take different amounts of time to complete.

For every state-action pair, the application of a feasible move m on it will result in a new state-action pair: $(s, a) \xrightarrow{m} (s', a')$. Associated with each such state-action-move pair $((s, a), m)$, we maintain two additional values that are used by MCTS:

(i) *N-value* - The number of times we have taken the said move m from said state-action pair (s, a).
(ii) *Q-value* - Given a reward function $\mathcal{R}$, it is the average long-term reward expected after taking said move m over all iterations of the search. (Future rewards are discounted by a factor γ)

$$Q((s,a),m) = \mathcal{R}((s,a),m) + \gamma \frac{\sum_{m'} N((s',a'),m') \cdot Q((s',a'),m')}{\sum_{m'} N((s',a'),m')} \quad (1)$$

B. Monte Carlo Tree Search

Monte Carlo tree search progresses iteratively by executing its four phases: select, expand, rollout, and backup as illustrated in Fig. 3. In each iteration, it begins traversing down the existing search tree by selecting the node with the maximum UCT value (Eq. 2) at each level. During this traversal, whenever it encounters a leaf node, it expands the tree by choosing a move m from that leaf node. Then, it estimates the scalar evaluation for the new state-action pair and backpropagates it up the tree to update evaluations of its ancestors.

To build an optimal action set, we would want to select the move m with the maximum true Q-value. But since true Q-values are intractably expensive to compute, we can only approximate the Q-values through efficient exploration. We use the Upper Confidence Bound on Trees (UCT) objective [12] to balance exploration and exploitation as we traverse through the search tree. Moreover, as this problem results in a highly asymmetric tree, since some move block a lot of other moves, while others block fewer moves, we use the formulation of UCT adapted for asymmetric trees [24]:

$$\mathrm{UCT}((s,a),m) = Q((s,a),m) + c\frac{\sqrt{N((s,a))}}{N((s,a),m)} \times p(m|(s,a)) \tag{2}$$

Here, the value $p(m|(s,a))$ is the prior policy function, which is obtained by adding a Dirichlet noise to the policy output of the neural network [25]. As MCTS continues probing the action space, it gets a better estimate of the true values of the actions. This means that it acts as a policy enhancement function whose output policy (Eq. 3) can be used to train the neural network's prior, and the average Q-value computed can be used to train its scalar evaluation (Eq. 4).

$$\pi(m|(s,a)) \propto N((s,a),m)) \tag{3}$$

$$\mathcal{V}((s,a)) = \frac{\sum_m Q((s,a),m)}{\sum_m N((s,a),m)} \tag{4}$$

The details of how MCTS progresses have been elaborated in Algo. 1. Once it gets terminated, i.e., the search gets completed, we go down the tree selecting the child with the maximum Q-value at each step until a COMMIT action is found, we use the action set of the selected state-action pair to schedule SWAPs for the current timestep, and we re-root the tree at the child node of the COMMIT action to prepare for the next timestep.

C. Neural Network Architecture

Each iteration of the MCTS requires evaluation of Q-values for a newly encountered state-action pair. But these values are impossible to be computed exactly since it would involve an intractable number of iterations in exploring and expanding the complete search tree. Therefore, it is favorable to heuristically evaluate the expected long-term reward from the state-action pair using a Neural Network, as it acts as an excellent function approximator that can learn the symmetries and general rules inherent to the system.

So, once the MCTS sends a state-action pair to the evaluator, it begins by committing the action to the state and getting the resultant state. We then generate the following featurized representation of this state and pass this representation through the neural-network architecture as shown in Fig. 4.

(i) *Node Targets* - It is a square boolean matrix whose rows and columns correspond to the nodes on a target device. An element (i, j) is true iff some logical qubits q_x and q_y are currently mapped to

Algorithm 1: Monte Carlo tree search

```
Data: state s_t
Initialize: root ← (s_t, empty action set)
loop n_mcts times
    (s, a) ← root
    repeat
        Compute UCT values using prior + noise
        Select move that maximizes UCT value
        if (s, a).child[move] ≠ null then
            (s, a) ← (s, a).child[move]
        else
            if move = COMMIT then
                s' ← step (s, a)
                a' ← empty set
            else
                s' ← s
                a' ← insert into a the qubit pair
                  corresponding to the move
            end
            state.child[move] ← (s', a')
            store reward[(s, a), move] ← R(s', a') -
              R(s, a)
        end
    until previous move was expand
    reward ← evaluation from model of (s, a)
    while (s, a) ≠ root do
        p-move ← move from parent of (s, a) to (s, a)
        (s, a) ← parent in tree of (s, a)
        reward ← reward[(s, a), p-move] + γ· reward
        update (s, a).Q-value[move] with reward
        increment (s, a).N-value[move] by 1
    end
end
memorize the Q-values and N-values at the root for
  training the model later
(s, a) ← root
repeat
    (s, a) ← child of (s, a) with maximum Q-value
until move ≠ COMMIT
return a
```

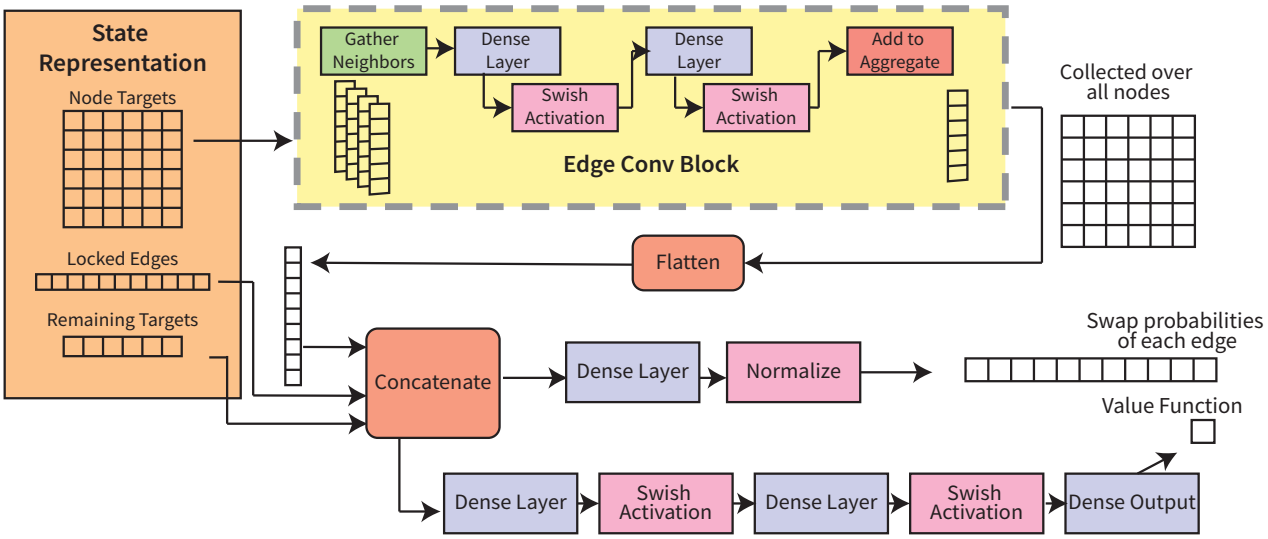


FIG. 4: Graph Neural Network Architecture that approximates the value function and the policy function. The State object, shown on the far left, is provided as an input. The edge convolution block extracts the features from the input object, collects them, flattens then and concatenates then with the rest of the input for further processing. Here the network diverges into two segments, one called the value head which gives a scalar output representing the quality of the state and another policy head which provides the probability distribution over the best action from this state.

nodes i and j respectively, such that (q_x, q_y) is the first unscheduled operation that q_x partakes in.

(ii) *Locked Edges* - It is a set of edges (pairs of connected nodes) that are still locked due to either of its qubits being involved in an operation in the current timestep or another longer operation that hasn't yet terminated from the previous timesteps.

(iii) *Remaining targets* - It is a list of the number of gate-operations that are yet to be scheduled for each logical qubit.

The SWAP operations each qubit would partake in depends primarily on its target node, and on those of the nodes in its neighborhood that might be competing for the same resources. It seems reasonable that we can use a Graph Neural Network with the device topology graph for its connectivity since the decision of the optimal SWAP action for some node is largely affected by other nodes in its physical neighborhood. Therefore, our architecture includes an edge-convolution block [13], followed by some fully-connected layers with Swish [26] activations for the policy and value heads. The value function and the policy function computed from this neural network are returned back to the MCTS.

IV. RESULTS

We compare QRoute against the routing algorithms from other state-of-the-art frameworks on various circuit benchmarks: (i) Qiskit and its three variants [27]: (a) basic, (b) stochastic, and (c) sabre, (ii) Deep-Q-Networks (DQN) from [8], (iii) Cirq [28], and (iv) t|ket⟩ from Cambridge Quantum Computing (CQC) [29]. The algorithm t|ket⟩ uses BRIDGE gates along with SWAP gates and Qiskit uses gate commutation rules while perform qubit routing. These strategies are shown to be advantageous in achieving lower circuit depths [30] but were disabled in our simulations to have a fair comparison. The results for DQN shown are adapted from the data provided by the authors Pozzi *et al.* [8].

A. Random Test Circuits

The first benchmark for comparing our performance comprises of random circuits. These circuits are generated on the fly and initialized with the same number of qubits as there are nodes on the device. Then two-qubit gates are put between any pair of qubits chosen at random. In our simulations, the number of such gates is varied from 30 to 150 and the results for assessing performance of different frameworks are given in Fig. 5. The experiments were repeated 10 times on each circuit size, and final results were aggregated over this repetition.

Amongst the frameworks compared, QRoute ranks a very close second only to Deep-Q-Network guided simulated annealer (DQN). Nevertheless, QRoute still does consistently better than all the other major frameworks: Qiskit, Cirq and t|ket⟩, and it scales well when we in-

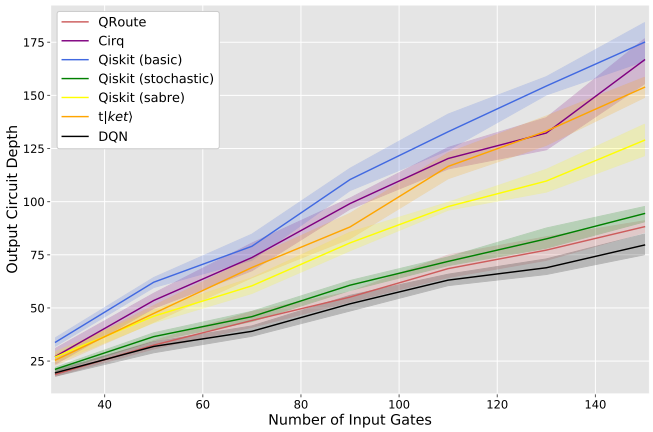


FIG. 5: Comparative performance of routing algorithms on random circuits as a function of the number of two-qubit operations in the circuit.

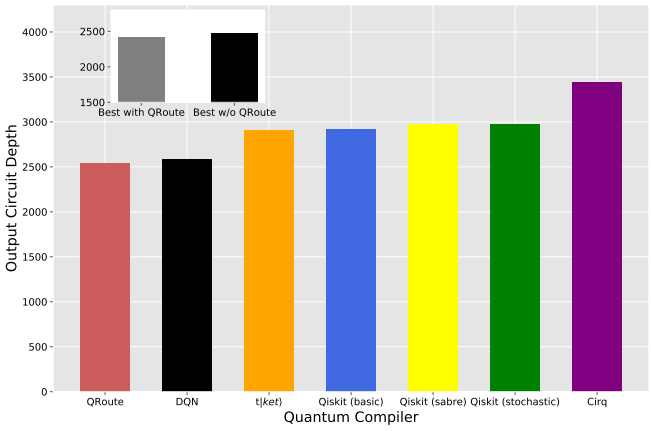


FIG. 6: Plots of output circuit depths of routing algorithms over small realistic circuits (≤ 100 gates), summed over the entire dataset. The inset shows the results on the same data comparing the best performant scheduler excluding and including QRoute on each circuit respectively.

crease the number of layers and the layer density in the input circuit. QRoute shows equivalent performance to DQN on smaller circuits, and on the larger circuits it outputs depths which are on average ≤ 4 layers more than those of DQN. Some part of this can be attributed to MCTS, in it's limited depth search, choosing the worse of two moves with very close Q-values, resulting in the scheduling of some unnecessary SWAP operations.

B. Small Realistic Circuits

Next we test on the set of all circuits which use 100 or less gates from the IBM-Q realistic quantum circuit dataset used by Zulehner *et al.* [31]. The comparative performance of all routing frameworks has been shown by plotting the depths of the output circuits summed over all the circuits in the test set in Fig. 6. Since the lack of a good initial qubit allocation becomes a significant problem for all pure routing algorithms on small circuits, we have benchmarked QRoute on this dataset from three trials with different initial allocations.

The model presented herein has the best performance on this dataset. We also compare the best result from a pool of all routers including QRoute against that of another pool of the same routers but excluding QRoute. The pool including QRoute gives on average 2.5% lower circuit depth, indicating that there is a significant number of circuits where QRoute is the best routing method available.

On this dataset also, closest to QRoute performance is shown by Deep-Q-Network guided simulated annealer. To compare performances, we look at the average circuit depth ratio (CDR), which is defined by [8]:

$$\text{CDR} = \frac{1}{\#\text{circuits}} \sum_{\text{circuits}} \frac{\text{Output Circuit Depth}}{\text{Input Circuit Depth}} \qquad (5)$$

The resultant CDR for QRoute is 1.178, where as the reported CDR for the DQN is 1.19. In fact, QRoute outperforms DQN on at least 80% of the circuits. This is significant because in contrast to the random circuit benchmark, the realistic benchmarks consist of the circuits that are closer to the circuits used in useful computation.

C. Large Realistic Circuit

For final benchmark, we take eight large circuits ranging from 154 gates to 5960 gates in its input from the IBM-Quantum realistic test dataset [31]. The results are plotted in Fig. 7. QRoute has the best performance of all available routing methods: Qiskit and t|ket⟩, on every one of these sampled circuits with on an average 13.6% lower circuit depth, and notable increase in winning difference on the larger circuits.

The results from DQN and Cirq are not available for these benchmarks as they are not designed to scale to such huge circuits. In case of DQN, the CDR data results were not provided for the circuits over 200 gates, mainly because simulated annealing used in it is computationally expensive. Similarly, for Cirq, it takes several days to compile each of the near 5000 qubit circuits. In contrast, QRoute is able to compile these circuits in at most 4 hours, and its compilation process can be sped up by reducing the depth of the search. Spending more time, however, helps MCTS to better approximate the Q-values leading to circuits with lower resulting depth.

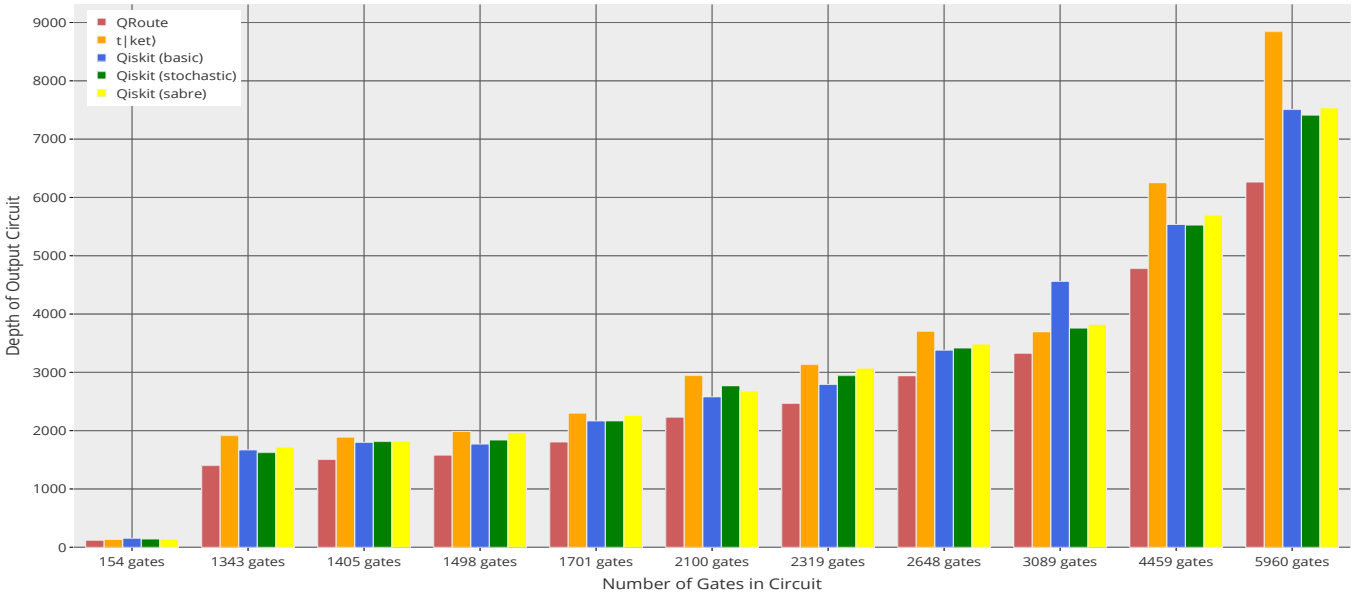


FIG. 7: The results over eight circuits sampled from the large realistic dataset benchmark, the outputs of each routing algorithm are shown for every circuit.

V. DISCUSSION AND CONCLUSION

In this article, we have shown that the problem of qubit routing has a very powerful and elegant formulation in Reinforcement Learning (RL) which can surpass the results of any classical heuristic algorithm across all sizes of circuits and types of architectures. Furthermore, the central idea of building up solutions step-by-step when searching in combinatorial action spaces and enforcing constraints using mutex locks, can be adapted for several other combinatorial optimization problems [32–36].

Our approach is flexible enough to compile circuits of any size onto any device, from small ones like IBMQX20 with 20 qubits, to much larger hardware like Google Sycamore with 53 qubits (the Circuit Depth Ratio for small realistic circuits on Google Sycamore was 1.64). Also, it intrinsically deals with hardware having different primitive instruction set, for example on hardware where SWAP gates are not a primitive and they get decomposed to 3 operations. QRoute enjoys significant tunability; hyperparameters can be changed easily to alter the tradeoff between time taken and optimality of decisions, exploration and exploitation, etc.

QRoute is a reasonably fast method, taking well under 10 minutes to route a circuit with under 100 operations, and at most 4 hours for those with upto 5000 operations, when tested on a personal machine with an i3 processor (3.7 GHz) and no GPU acceleration. Yet more can be desired in terms of speed, given that Qiskit and t|ket⟩ are a lot faster than QRoute, but it is hard to achieve that without reducing the number of search iterations and trading off a bit of performance. More predictive neural networks can help squeeze in slightly better speeds.

One of the challenges of methods like DQN, that use Simulated Annealing to build up their actions is that the algorithm cannot plan for the gates which are not yet waiting to be scheduled, those which will come to the head of the list once the gates which are currently waiting are executed [8]. QRoute also shares this deficiency, but the effect of this issue is mitigated by the explicit tree search which takes into account the rewards that will be accrued in the longer-term future. There is scope to further improve this by feeding the entire list of future targets directly into our neural network by using transformer encoders to handle the arbitrary length sequence data. This and other aspects of neural network design will be a primary facet of future explorations.

Another means of improving the performance would be to introduce new actions by incorporating use of BRIDGE gates [30] and gate commutation rules [14] alongside currently used SWAP gates. The advantage of former is that it allows running CNOT gates on non-adjacent qubit without permuting the ordering of the logical qubits; whereas, the latter would allow MCTS to recognize the redundancy in action space, making its exploration and selection more efficient.

Finally, we provide an open-sourced access to our software library [22]. It will allow researchers and developers to implement variants of our methods with minimal effort. We hope that this will aid future research in quantum circuit transformations.

On the whole, the Monte Carlo Tree Search for building up solutions in combinatorial action spaces has exceeded the current state of art methods that perform qubit routing. Despite its success, we note that QRoute is a primitive implementation of our ideas, and there is great scope of improvement in future.

ACKNOWLEDGEMENTS

A. S. and U. A. have contributed equally to the manuscript. A. S. and U. A. would like to acknowledge the assistance from their colleague Bhuvanesh Sridharan for his help in implementing and refining MCTS. Additionally, they would also like to thank their colleagues Jai Bardhan, Kalp Shah for their suggestions.

[1] J. Preskill, *"Quantum computing in the NISQ era and beyond,"* Quantum **2**, 79 (2018).

[2] "IBM Quantum," https://quantum-computing.ibm.com/ (2021).

[3] F. Arute, K. Arya, R. Babbush, and et al., *"Quantum supremacy using a programmable superconducting processor,"* Nature **574**, 505–510 (2019).

[4] P. J. Karalekas, N. A. Tezak, E. C. Peterson, C. A. Ryan, M. P. da Silva, and R. S. Smith, *"A quantum-classical cloud platform optimized for variational hybrid algorithms,"* Quantum Sci. Technol. **5**, 024003 (2020), arXiv:2001.04449.

[5] J. E. Bourassa, R. N. Alexander, M. Vasmer, A. Patil, I. Tzitrin, T. Matsuura, D. Su, B. Q. Baragiola, S. Guha, G. Dauphinais, K. K. Sabapathy, N. C. Menicucci, and I. Dhand, *"Blueprint for a scalable photonic fault-tolerant quantum computer,"* Quantum **5**, 392 (2021), arXiv:2010.02905.

[6] A. M. Childs, E. Schoute, and C. M. Unsal, in *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 135 (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019) pp. 3:1–3:24.

[7] A. Cowtan, S. Dilkes, R. Duncan, A. Krajenbrink, W. Simmons, and S. Sivarajah, in *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*, Leibniz International Proceedings in Informatics (LIPIcs), Vol. 135 (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019) pp. 5:1–5:32.

[8] M. G. Pozzi, S. J. Herbert, A. Sengupta, and R. D. Mullins, *"Using Reinforcement Learning to Perform Qubit Routing in Quantum Compilers,"* arXiv e-prints (2020), arXiv:2007.15957 [quant-ph].

[9] S. Herbert and A. Sengupta, *"Using Reinforcement Learning to find Efficient Qubit Routing Policies for Deployment in Near-term Quantum Computers,"* arXiv e-prints (2018), arXiv:1812.11619 [quant-ph].

[10] L. Kocsis and C. Szepesvari, in *ECML* (2006).

[11] R. Munos, *"From bandits to monte-carlo tree search: The optimistic principle applied to optimization and planning,"* Found. Trends Mach. Learn. **7**, 1 (2014).

[12] L. Kocsis and C. Szepesvári, in *European conference on machine learning* (Springer, 2006) pp. 282–293.

[13] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, *"Dynamic Graph CNN for Learning on Point Clouds,"* arXiv e-prints (2018), arXiv:1801.07829 [cs.CV].

[14] J. C. Garcia-Escartin and P. Chamorro-Posada, *"Equivalent Quantum Circuits,"* arXiv e-prints (2011), arXiv:1110.2998 [quant-ph].

[15] S. S. Tannu and M. K. Qureshi, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19 (Association for Computing Machinery, New York, NY, USA, 2019) p. 987–999.

[16] A. Zulehner, A. Paler, and R. Wille, *"An Efficient Methodology for Mapping Quantum Circuits to the IBM QX Architectures,"* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **38**, 1226 (2019).

[17] A. Paler, A. Zulehner, and R. Wille, *"NISQ circuit compilation is the travelling salesman problem on a torus,"* Quantum Science and Technology **6**, 025016 (2021).

[18] A. Paler, L. M. Sasu, A. Florea, and R. Andonie, "Machine learning optimization of quantum circuit layouts," (2020), arXiv:2007.14608 [quant-ph].

[19] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, *"A survey of monte carlo tree search methods,"* IEEE Transactions on Computational Intelligence and AI in Games **4**, 1 (2012).

[20] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, and et al., *"Mastering the game of go with deep neural networks and tree search,"* Nature **529**, 484 (2016).

[21] X. Zhou, Y. Feng, and S. Li, in *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20 (Association for Computing Machinery, New York, NY, USA, 2020).

[22] Code for the complete simulation and visualization is available on https://github.com/AnimeshSinha1309/quantum-rl (2021).

[23] E. W. Dijkstra, "Cooperating sequential processes," in *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls* (Springer New York, New York, NY, 2002) pp. 65–138.

[24] T. M. Moerland, J. Broekens, A. Plaat, and C. M. Jonker, *"Monte Carlo Tree Search for Asymmetric Trees,"* arXiv e-prints (2018), arXiv:1805.09218 [stat.ML].

[25] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, *"Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm,"* arXiv e-prints (2017), arXiv:1712.01815 [CoRR].

[26] P. Ramachandran, B. Zoph, and Q. V. Le, *"Searching for Activation Functions,"* arXiv e-prints (2017), arXiv:1710.05941 [cs.NE].

[27] H. Abraham and et al., "Qiskit: An Open-source Framework for Quantum Computing," (2019).

[28] Cirq Developers, "Cirq," (2021).

[29] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, *"t|ket⟩: A retargetable compiler for NISQ devices,"* Quantum Science and Technology **6**, 014003 (2020).

[30] T. Itoko, R. Raymond, T. Imamichi, and A. Matsuo, *"Optimization of Quantum Circuit Mapping using*

Gate Transformation and Commutation," arXiv e-prints (2019), arXiv:1907.02686 [quant-ph].

[31] A. Zulehner, A. Paler, and R. Wille, "IBM Qiskit developer challenge," https://www.ibm.com/blogs/research/2018/08/winners-qiskit-developer-challenge/ (2018), accessed on: 2021-03-23.

[32] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev, *"Reinforcement Learning for Combinatorial Optimization: A Survey,"* arXiv e-prints (2020), arXiv:2003.03600 [cs.LG].

[33] Z. Xing and S. Tu, *"A graph neural network assisted monte carlo tree search approach to traveling salesman problem,"* IEEE Access **8**, 108418 (2020).

[34] R. Xu and K. Lieberherr, *"Learning Self-Game-Play Agents for Combinatorial Optimization Problems,"* arXiv e-prints (2019), arXiv:1903.03674 [cs.AI].

[35] K. Abe, Z. Xu, I. Sato, and M. Sugiyama, *"Solving NP-Hard Problems on Graphs with Extended AlphaGo Zero,"* arXiv e-prints (2019), arXiv:1905.11623 [cs.LG].

[36] A. Laterre, Y. Fu, M. Khalil Jabri, A.-S. Cohen, D. Kas, K. Hajjar, T. S. Dahl, A. Kerkeni, and K. Beguir, *"Ranked Reward: Enabling Self-Play Reinforcement Learning for Combinatorial Optimization,"* arXiv e-prints (2018), arXiv:1807.01672 [cs.LG].