Qubit routing using Graph neural network aided Monte Carlo tree search

Animesh Sinha,[*] Utkarsh Azad,[†] Harjinder Singh,[‡] Jai Bardhan, Kalp Shah, and Bhuvanesh Sridharan
Center for Computational Natural Science, International Institute of Information Technology, Hyderabad.
(Dated: March 20, 2021)

In near-term quantum hardware, the connectivity is limited to nearest neighboring qubits. This means that they support two-qubit operations only on the qubits that can interact with each other. To respect this constraint, compilers perform the task of qubit routing, i.e., to transforms quantum circuits either by including additional SWAP gates or by reversing existing CNOT gates to satisfy the connectivity constraints of the target topology. This article aims to provide an efficient solution for the qubit routing problem called QRoute, which minimizes the depth of these transformed circuits. We use the Monte Carlo tree search to perform qubit routing, which is aided by a Graph neural network that is used to evaluate the value function and action probabilities throughout the qubit routing procedure. Our procedure outperforms currently available qubit routing implementations on realistic circuit benchmarks while competing decently on random circuit benchmarks and is architecture agnostic.

I. INTRODUCTION

The present-day quantum computers, more generally known as Noisy Intermediate-Scale quantum (NISQ) devices [**?**] come in a variety of hardware architectures [**? ? ?**], but there exists a few pervading problems across them. These problems constitute the poor quality of qubits, limited connectivity between qubits, and the absence of error-correction for noise-induced errors in the execution of gate operations. Hence, this places a considerable limitation on the number of quantum instructions that can be executed on them to perform some useful quantum computation. These instructions together can be realized as a sequential series of one or two-qubit gates acting on the qubits which can be visualized more easily as a quantum circuit as shown in Fig. 1a.

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. These transformations usually include the addition of SWAP gates and the reversal of existing CNOT gates. Such transformation ensures that any non-local quantum operations are performed only between the qubits that nearest-neighbors and can interact with each other. This process of circuit transformation by a compiler routine for the target hardware is known as *qubit routing*. The output instructions in the transformed quantum circuit would be following the connectivity constraints and would essentially result in the same overall unitary evolution as the original circuit [1].

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 addition of extra SWAP gates. This overhead in the circuit depth becomes 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. In this article, we present a procedure that we refer to as *QRoute*. In this procedure, we propose to use the Monte Carlo tree search (MCTS) for the task of depth minimization, as an easier to train and better performing machine learning setting, which is aided by a Graph neural network (GNN) that is used to evaluate the value function and action probabilities throughout the compilation.

Structure: In Section II, we introduce the problem of qubit routing, the previous works that are done in the field, and show our work differs from them. Next, we represent the methodology of the QRoute algorithm in Section III. Then, 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 depending on whether they are being used in the execution of quantum instruction set or not. The quantum instruction set 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. Amongst these two type of gate operations, two-qubit qubit gates such as CNOT can only be performed between two logical qubits iff the there exist an edge between the nodes, i.e., the physical qubits, that corresponds to them respectively. This edge could

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

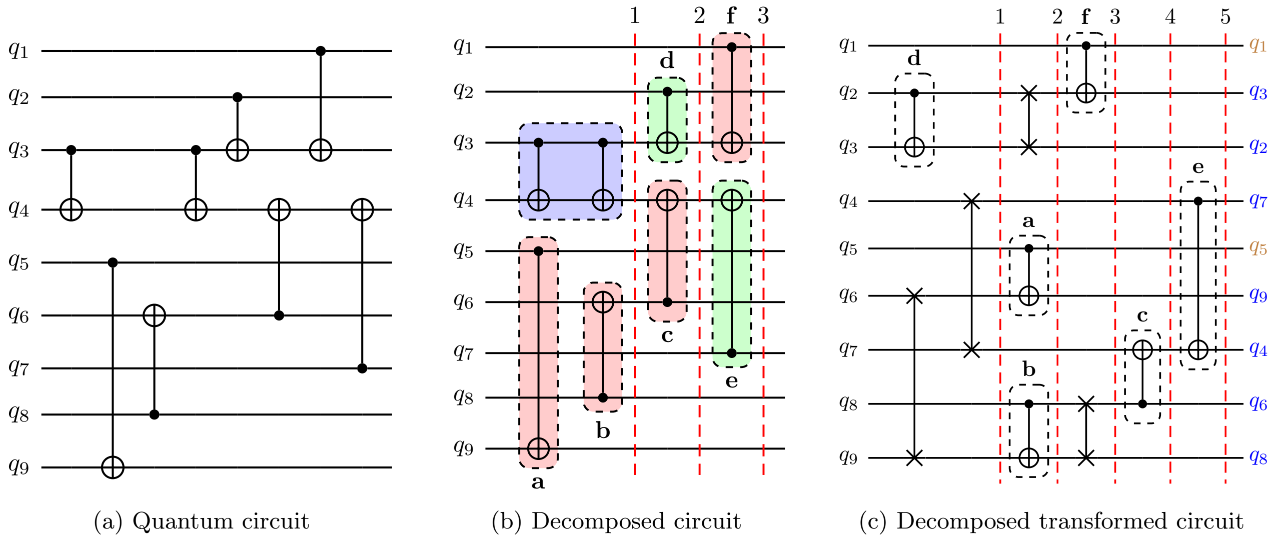

FIG. 1: An example of qubit routing on a quantum circuit for 3×3 grid architecture (Figure 2a). (a) For simplicity, the original quantum circuit consists only of two-qubit gate operations. (b) Decomposition of the original quantum circuit into series of slices such that all the instructions present in a slice can be executed in parallel. Here, the two-qubit gate operations: $\{d, e\}$ (green) comply with the topology of the grid architecture whereas the operations: $\{a, b, c, f\}$ (red) does 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. Here, 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

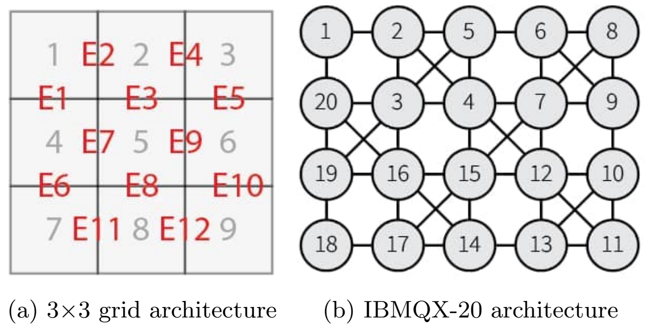

FIG. 2: Examples of qubit connectivity graphs for some common quantum architectures

be either be unidirectional or bidirectional, i.e., CNOT can be performed either in one direction or 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 pair Hadamard gates acting on both control and target qubit [].

Now, given a target hardware topology and a quantum circuit, the task of qubit routing refers to transforming this quantum circuit by adding 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:

$$\mathrm{R}(circuit,\ topology) \rightarrow circuit'$$

Depth of *circuit'* (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 circuit 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 in w.r.t the original circuit.

We can breakdown this goal into 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 []. In this work, we only focus on the latter problem of qubit movement and refer to it as qubit routing. However, we do note that qubit allocation is an important problem and can play important role in minimizing the effort needed to perform the latter.

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.* [2], for developing a Computer Aided Design-based (CAD) routing strategy. In subsequent years, this problem has been outlined and presented in many different ways. Some of them being graph-based architecture-agnostic solution by Cowtan *et al.* [3] and reinforcement learning in a combinatorial action space solution by Herbert and Sengupta [4]. In the latter, they also proposed the use of simulated annealing to search through the combinatorial action space, aided by a Feed-Forward neural network to judge the long-term expected compilation time. This was further extended to use Double Deep Q-learning and prioritized experience replay by Pozzi *et al.* [1].

Recently, Monte Carlo tree search, a popular reinforcement learning algorithm [5–7] previously proven successful in a variety of domains like playing puzzle games such as Chess and Go [8] was used by Zhou *et al.* [9] 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). In our work, we focus on a formulation of MCTS given by Moerland *et al.* [10], that is adapted for better exploration of Asymmetric Trees.

C. Our Contributions

As mentioned before, in this work 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 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 test and visualize routing algorithms with different neural net architectures, combining algorithms, reward structures, etc.

III. METHOD

The QRoute algorithm has four major segments:

1. Maintaining and evolving an efficient representation of the current state of the entire system, which includes both the operations from the input circuit which are yet to be scheduled, and the current state of the solution.

2. A value function and policy function approximator, here are graph neural network, which will evaluate the long term reward expected from the current state of the simulation.

3. A Monte Carlo tree search agent, which will take the current state, and search over the set of actions available trying to find the optimal action based on immediate rewards and the long term evaluation offered by the value function approximator.

In this section, we describe all these elements of our solution. [11]

A. State representation

The QRoute algorithm takes an input circuit and makes progress through it iteratively, scheduling a list of parallelizable SWAP gates and CNOT gates subjected to the topology of the input hardware architecture. We refer to each such iteration as a timestep. At any given timestep t, the entire state of the compilation is captured by the state object s_t at that timestep

$$\mathrm{s_t} = (\mathrm{circuit},\ \mathrm{device},\ \mathrm{s_t.map},\ \mathrm{s_t.progress},\ \mathrm{s_t.locks})$$

This is necessary and sufficient to capture the entire state of the system, in addition to this we also cache some alternate but useful representations of the state as listed in section III C. We will describe the constituents of the state tuple in this subsection.

1. *Circuit Representation*

A circuit is a set of qubit $\mathbb{Q}$ together with an partially ordered set of operations on those qubits.

$$O = \{(q_x, q_y) \mid q_x, q_y \in Q\}$$

The order in which two operations appear in the circuit is relevant to the overall unitary function implemented by it iff they share one or both of the qubits participating in them.

We convert the circuit to a lists of lists representation (as used by Pozzi *et al.* [1]), where for each logical qubit, q_i, we store the list of all other qubits that participate in two-qubit operations with q_i, in the order in which they appear in the logical circuit. This *circuit* object is common over all the states in the simulation and does not evolve.

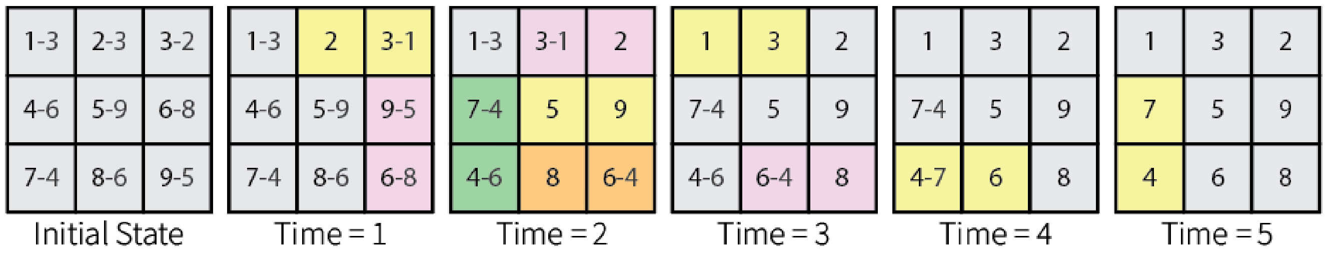

FIG. 3: Routing progress for 3×3 grid architecture while for the quantum circuit in Fig. 1a. Initial state shows the gate scheduled on each qubit which gets executed as control and target qubit adjacent to each other via SWAPs. The final state (at time=5) shows t he final evolved qubit mapping. Here, each state (at time step=t) corresponds to the t^{th} time slice in Fig. 1c

2. *Device Topology*

We need to have access to the topology of the target quantum devices to check whether a gate operation being performed in the input quantum circuit is feasible or not. This is stored as a graph in the *device* object, all the nodes on the target hardware are nodes in the graph, and all pairs of adjacent qubits have an edge between them, as illustrated in figure 2.

We also maintain the shortest distances between all pairs of nodes, by running the Floyd-Warshall algorithm on the device graph, so that we can offer an intermediate reward function for bringing two qubits which participate in an operations that is waiting to be scheduled closer.

3. *Node to Qubit Mapping*

Each node (or physical qubit) is mapped to either a logical qubit, or left empty. We maintain this mapping $Q \rightarrow N$, and evolve it using SWAP operations as shown in Fig. 3. This mapping from the logical qubits to nodes is injective but not surjective. The initial value of this mapping is either obtained through a qubit allocation algorithm, or assigned randomly.

4. *Qubit Progress*

For each qubit, we maintain the progress it has made through the circuit. If $\mathrm{s_t.progress[q]} = k$, then the first k operations in the logical circuit which q participates in should have been scheduled in the output circuit before the current timestep t.

So a gate (q_x, q_y) is waiting to be scheduled if and only progress of q_x is i and that for q_y and j, and in the circuit object the $(\mathrm{i}+1)^{\mathrm{th}}$ operation of q_x is with q_y and the $(\mathrm{j}+1)^{\mathrm{th}}$ operation of q_y is with q_x.

5. *Mutual Exclusion Locks*

Operations which are being scheduled must maintain mutual exclusivity with other operations over the nodes which participate in them. This is essential to minimizing the depth of the circuit since it models parallelizability of operations.

Since we can think of the nodes as “resources” and the operations that are trying to use them as “consumers”, we note that our method is identical to the parallelization problems in classical computers, and we can adapt mutex locks Dijkstra [12] to ensure mutual exclusion.

Different gates take different amounts of time to execute, for example if the primitive gates are CNOT on some hardware, then a SWAP gate will decompose to 3 CNOT gates and take 3 time-steps to complete execution. This is why we need to maintain the locks in the state object, since an operation from a previous timestep could still be holding locks on some nodes in the current timestep.

B. Monte Carlo Tree Search

Each timesteps starts off by greedily scheduling any CNOT gates that are waiting to be scheduled and are local on the hardware given the current qubit-to-node mapping. The CNOT gates, when scheduled, update the locks to ensure that the SWAP operations which will be generated for the same timestep can be executed in parallel with the CNOTs. To find the optimal action, a parallelizable set of swaps for the state at any timestep, the QRoute algorithm employs a Monte Carlo tree search to search over the action space.

1. *In the Combinatorial Action Space*

There are exponentially many sets of SWAPs that together constitute an action. If there are e edges on the target hardware, then we have 2^e possible actions, many of which break the mutual exclusivity constraint. We

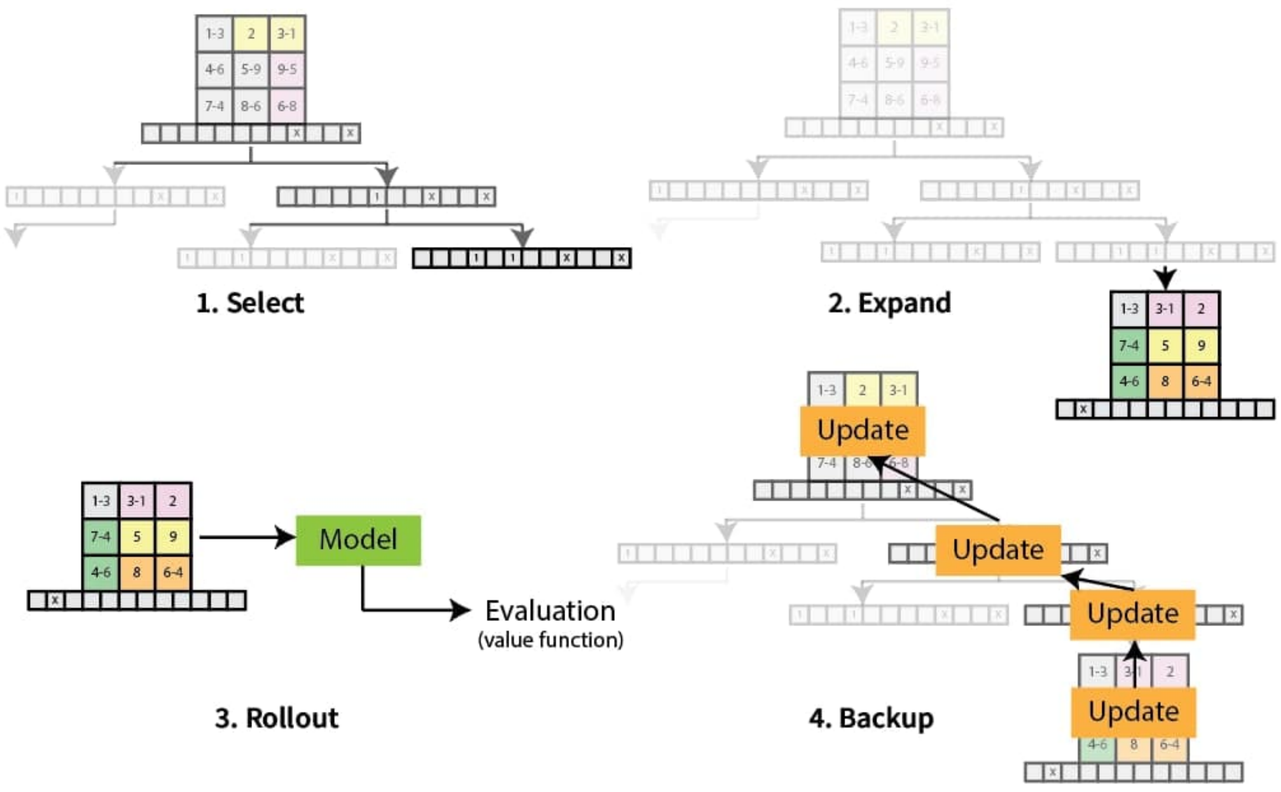

FIG. 4: Steps of Monte-Carlo tree search: (i) Select - Choose a mcts-state for exploration at current timestep, (ii) Expand - Calculating reward for newly encountered mcts-state and assign it as a child state, (iii) Rollout - Estimating long term value using a neural network for the state in the exploration tree, and (iv) Backup - Propagating value at leaf nodes upwards to update reward values of its ancestors.

need to search the optimal action from the ones that don't. To search in this massive action space, we will build up our action one by one, starting off with an empty list, computing the set of SWAPs which do not violate the mutual exclusivity constraint, and choosing one of them, and iteratively continuing to build up our actions with more such gates. But since we may not always want to use the entirety of our swapping ability, we also add a "commit" action, which breaks out of this iterative process. The commit action is trivially chosen if there are no more SWAPs that are possible in the current timestep.

We represent this "action", as a boolean vector of length equal to the number of edges. This building up of actions shown in each individual rendering of the tree in figure 4.

2. *The State of the Search*

The tree search needs to maintain an augmented form of the state, we shall call this mcts-state. This holds the state and a valid action from the state together with statistics on which state to go

- A reference to the corresponding State
- A list of child mcts-states.
- N-values corresponding to each child mcts-state, number of iterations in which that child was explored.
- Q-values corresponding to each child mcts-state, mean estimated reward obtained when that child is explored.
- solution, a list of SWAPs that represents the action that corresponds to the mcts-state in search.

Given an mcts-state s, the value function of that state is the expected total reward in the future.

$$V(s) = \max_{s'} Q(s,a) \tag{1}$$

The Q-function is the total expected reward in the future after we took an action a from state s_t.

$$Q(s,a) = R(s,a,s') + \gamma V(s') \tag{2}$$

3. How the Search Proceeds

Monte Carlo Tree Search iteratively executes it's four phases of select, expand, rollout and backup.

a. The first is a selection step which chooses the mcts-state we wish to explore in the current timestep. This choice is choosing the child mcts-state with the maximum Upper Confidence Bound on Trees (UCT) value for each mcts-state, and making our way down the tree till we reach a leaf node which has no explored children. UCT is the sum of the expected reward Q and uncertainty in reward by a ratio c. This ratio trades off the exploration of the agent against the exploitation (choosing the best action based on current knowledge).

$$\mathrm{UCT}(s,a) = Q(s,a) + c\frac{\sqrt{n(s)}}{n(s,a)} \times p(s,a) \qquad (3)$$

We recognize that running MCTS on this problem results in a highly assymetric tree, since some actions block a lot of other actions, and some actions don't interfere with any other action at all. So we add a Dirichlet noise term in the prior probabilities generated by our neural network to prevent our tree search from getting stuck down a single path. The formulation of MCTS for asymmetric trees is adapted from Moerland *et al.* [10]

b. The next step, Expansion, is triggered when we have taken an action from a mcts-state which has never been explored before. In this case, we create a new mcts-state, assign it to be the child state in our tree, and we compute the increase/decrease in reward due to that move and store it on the edge. The environment gives us a reward for bringing gates waiting to be scheduled closer together, actually scheduling gates, and finishing the entire compilation process. On the edge from the parent mcts-state to it's child, we assign a value equal to the increase in the expected reward from the parent to the child state. This value is 0 if the chosen action is commit.

c. Once we are at a leaf node on the tree, we get our neural network to estimate the long term value of the state, in what is called the rollout stage. MCTS can also be performed without a neural network, in which case we randomly sample a bunch of paths to the leaf nodes and estimate the value of the current state based on the average reward received from those random runs. However, neural networks are much better at generalizing the features learnt from one state to another. This is particularly useful since very similar representations are encountered several times in the simulation.

d. Finally, we have the evaluation of a node, and we need to update this information in it's parents. We take the value at the leaf nodes, and propagate it up the tree to it's ancestors, updating the q-values, which is the average expected reward from the parent, and the n-values, which is the number of times the particular action was taken.

Algorithm 1: Monte Carlo tree search

```
   Data: state $s_t$
1  Initialize: root ← new node($s_t$, solution)
2  loop
3      node ← root
4      repeat
5          Compute UCT values using prior from model
             + noise
6          action ← action from node with maximum
             UCT value
7          if state.child[action] $\neq$ null then
8              node ← node.child[action]
9          else
10             if action is commit then
11                 next-state ← step (node.state,
                     node.solution)
12                 next-solution ← $\underbrace{\boxed{0}\boxed{0}\boxed{0}\boxed{\dots}\boxed{0}}_{\text{—device.edges—}}$
13             else
14                 next-state ← node.state
15                 next-solution ← node.solution with a
                     set bit at action index
16             end
17             state.child[action] ← new node
                 (node.state, next-solution)
18             store reward[node, action] ←
                 reward(node.state, node.solution) -
                 reward(next-state, next-solution)
19         end
20     until previous action was expand
21     final-state ← step (node.state, node.solution)
22     reward ← evaluation from model of final-state
       while state $\neq$ root do
23         parent-action ← action which led to node from
             its parent node
24         node ← node.parent
25         reward ← reward[node, parent-action] +
             reward × discount-factor
26         update node.q-value[action] with reward
27         increment node.n-value[action] by 1
28     end
29 end
30 memorize the q-values and n-values at the root node
     for training the model later
```

4. The final action choice

Once the tree search is done, we use the tree to take our final action. We move down the tree again from the root, this time guided by the q-values and not by UCT, and move down till we find a commit action. We return the action, and move our root to the child state result from the commit action.

In addition, the n-values are representative of the probabilities with which each action was chose, and the q-value is the search enhanced estimate of the value function, we store both of these for the root node to train the policy and value function heads of our neural network.

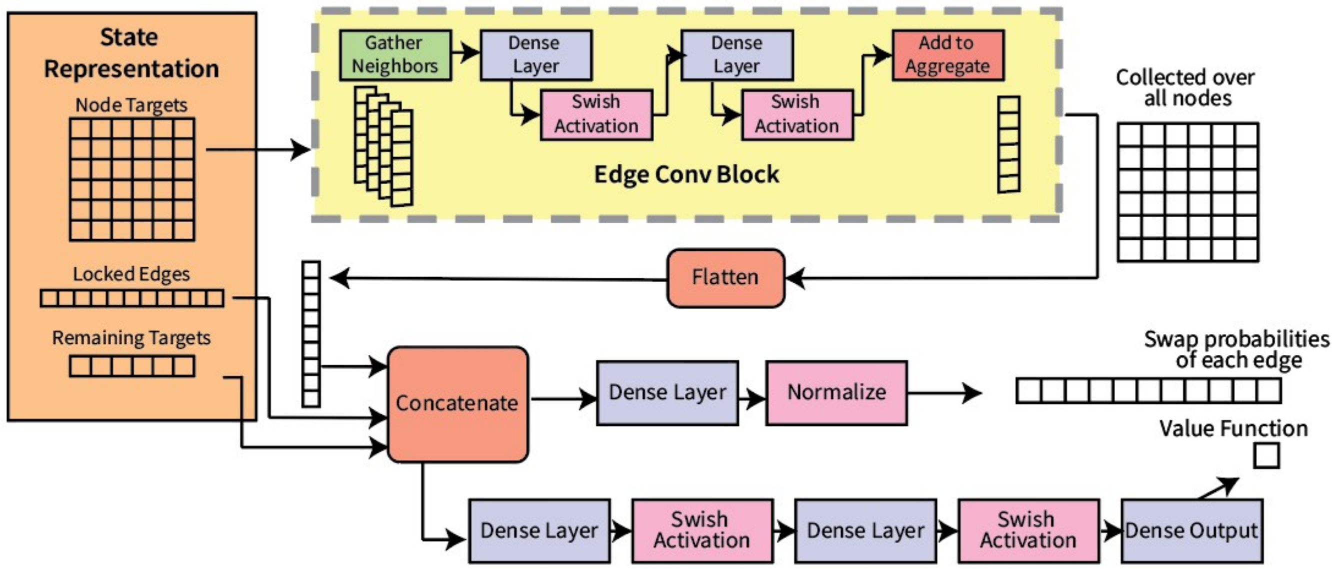

FIG. 5: Graph Neural Network Architecture with a Value head and a Policy head

C. Neural Network Architecture

Using the monte carlo tree search, we want to compute the q-values, which are the expected long term rewards, for each action from the current state. But these values are impossible to compute exactly, since it would involve visiting all leaf nodes in the subtree of the current state, and the number of these nodes grows exponentially with each new layer (timestep) in the tree. Therefore, it's necessary for us to stop after computing explicitly a few layers ahead in the future, and to heuristically evaluate the expected long term reward from the state.

Neural Nets are extremely good as function approximators. They are able to use the information gained from experiences from different runs, timesteps, and nodes. In this particular problem, very long term planning is not of particularly great use, optimally scheduling of the current and next few gates is more important. This leads us to feeding the following state representation to our neural network:

1. Node Targets: If there are n nodes on the device, it's a matrix of $n \times n$ boolean values. Let the mapped to node i be q_i and that to node j be q_j. The element at (i,j) is true if and only if $(q_i,\ q_j)$ is a gate waiting to be scheduled. So each row at atmost one value that is true, but a column has one or more than one.

2. Locked Edges: An equivalent representation of the locks in our state. It's a boolean vector of length equal to the number of edges, each element of the vector corresponds to an edge and is true if either of the nodes that the edge connects are locked (either from the previous state or due to the SWAPs we have already selected for the current action).

3. Remaining targets: An integer vector of length equal to number of nodes. For each logical qubit, the number of operations that it will participate in which yet to be scheduled on the hardware.

First, we need to get a feature vector for the gates which are currently waiting to be scheduled on the hardware, which is completely represented by the node targets matrix. This would be useful for both predicting the estimated value of the state as well as prior probabilities of which SWAP action should be chosen.

For each qubit, the SWAP operation it would ideally partake in depends primarily on it's target node, and on the SWAP operations that other nodes in it's neighborhood might be trying to partake in, competing for the same resources. It seems reasonable, therefore, that we can use a Graph Neural Network with the device topology graph for it's structure. Each node in the graph bears one row of the node-targets matrix. It is an assumption that the feature vector should only be dependent on some representation of the nodes and targets in it's local neighborhood on the device graph, but it's a reasonable one at that. And it promotes the use of message passing to convey this information about any node across it's neighborhood.

Formally, we take the node-targets vector and pass it through an edge convolution block [13] which uses two dense layers with swish activations [14] and finally adds to aggregate all messages. These output feature vectors from the edge convolution block are collected over a bunch of nodes and flattened to form the total feature vector. The concatenation of this with the locked-edges vector and the remaining-targets vector will form the in-

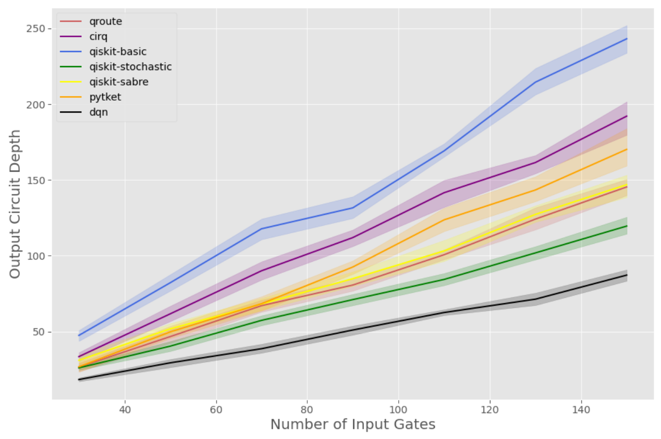

FIG. 6: Comparative performance against other methods across real world hardware

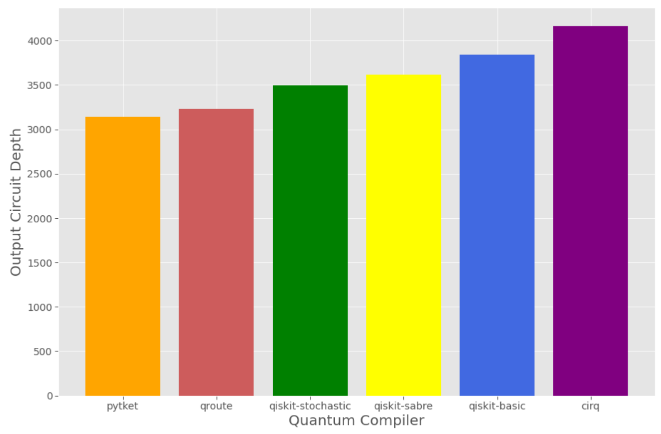

FIG. 7: Scaling of Initial Depth overhead with size of hardware

put to the value and policy heads.

The value head is 2 fully connected layers with Swish activations, and a final fully connected single neuron which has no activation.

The policy head is also just a single fully connected layer which has a output vector of size equal to the number of edges on device. We take the output of this layer, square it and then normalize it to convert the outputs of the neural network into a valid probability mass function. The outputs of this layer are interpreted as the probability of swapping the qubits which are on the ends of the corresponding edge.

IV. RESULTS

Here, we compare our algorithm QRoute against the other state-of-the-art algorirthms on various circuit benchmarks: Qiskit (and it's different variants) [15], Deep-Q-Networs (DQN) form [1], Cirq [16] and CQC Pytket [17].

A. Random Test Circuits

Our first benchmark is random circuits. These circuits are initialized with the same number of qubits as there are nodes on the device, and then two-qubit gates are randomly put between any pair of qubits, the number of such gates is varied from 30 to 150. We show the results of various frameworks in Fig. 6.

In the frameworks compared, QRoute ranks third, performing worse than Qiskit Stochastic and Deep-Q-Networks. (The results for the DQN were obtained from Pozzi *et al.* [1] and were not replicated.)

Random circuits are a particularly difficult task for our method, since the neural network cannot generalize what it learnt on different random circuits in the training run to what will be useful for evaluation in the test run. Random circuits are also particularly difficult to train on. Nevertheless, our method is still competitive against all other frameworks and consistently performs better than most.

B. Small Realistic Circuits

Next we test on the set of all circuits which have 100 gates or less from the IBM-Q realistic quantum circuit dataset []. The comparative performance of all frameworks has been shown by plotting the depths of the output circuits summed over all the circuits in the test set. The plot is in figure 7.

Here, once again QRoute outperforms most of the other frameworks, but does slightly worse than CQC Pytket. We attribute some part of our worse performance to the lack of a qubit allocation algorithm, which is a relevant overhead on small circuits.

C. Large Realistic Circuit

For our final benchmark, we took 8 large circuits ranging from 154 gates to 5960 gates in its input from the IBM-Quantum realistic test dataset. We can clearly see that as the circuits get larger, our QRoute algorithm scales better than any other method and consistently produces the best results on almost all the test circuits. The results are detailed in table I and plotted in figure 8.

The large circuits dataset was the largest dataset to train on and affords the QRoute the best opportunity to train on while being a harder task for all our compilers. Here the issue of qubit allocation is not a substantial part of the problem.

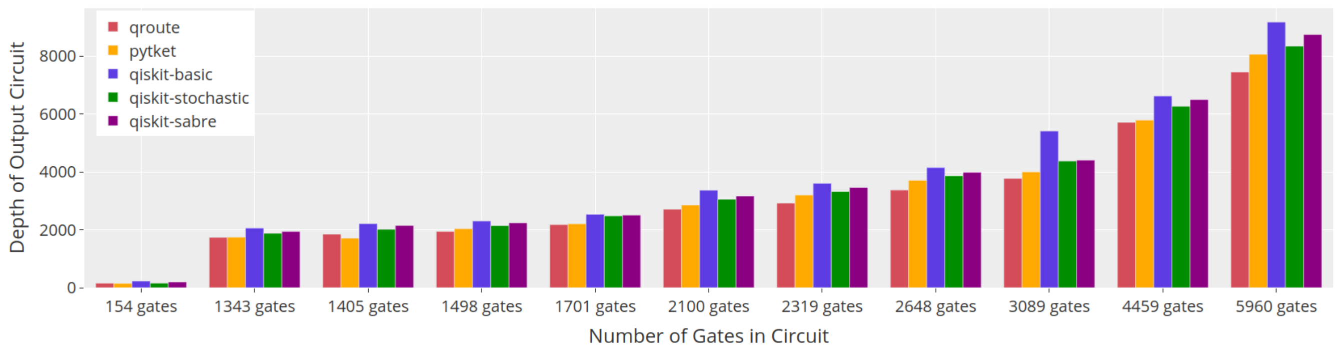

FIG. 8: Performance on some large realistic test circuits

TABLE I: Results of our algorithm on a set of realistic test circuits

Input Circuit		Output Circuit Depth				
Name	Number of Gates	Qroute	PyTket Depth	Qiskit-Basic	Qiskit-Stochastic	Qiskit-Sabre
rd84_142	154	154	145	227	156	196
adr4_197	1343	1738	1743	2057	1879	1940
radd_250	1405	1845	1707	2210	2013	2146
z4_268	1498	1942	2036	2300	2145	2238
sym6_145	1701	2180	2204	2537	2475	2509
misex1_241	2100	2712	2854	3365	3051	3163
rd73_252	2319	2922	3199	3602	3320	3461
cycle10_2_110	2648	3372	3706	4150	3862	3983
square_root_7	3089	3776	3997	5409	4372	4404
sqn_258	4459	5712	5788	6622	6269	6499
rd84_253	5960	7450	8063	9178	8345	8746

V. DISCUSSION AND FUTURE SCOPE

This method of Quantum Circuit Transformation provides competitive and many a times superior performance to other methods. However, this is my no means a definitive solution to the task of Quantum Circuit Transformations, and offers great scope of improvement. We believe that the following are avenues in which the algorithm can be significantly improved:

- Augmenting the Neural Network to take the future gates as an input. We propose to use a Transformer architecture for this, to learn a mapping from arbitrary-length list of gates to a feature vector which can be used in the Graph Neural Network to estimate the value function better.
- Finding more systematic approaches to manage the hyperparameters, e.g. Exploration-Exploitation tradeoff constant, Reward Parameters, etc.

Our software library allows easy access to implementing these neural networks by changing just a single file. We hope that this will aid future improvements and research in the task of quantum circuit transformations.

[1] M. G. Pozzi, S. J. Herbert, A. Sengupta, and R. D. Mullins, "Using reinforcement learning to perform qubit routing in quantum compilers," (2020), arXiv:2007.15957 [quant-ph].
[2] A. Zulehner, A. Paler, and R. Wille, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems (TCAD) (2018).
[3] 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, edited by W. van Dam and L. Mancinska (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019) pp. 5:1–5:32.
[4] S. Herbert and A. Sengupta, "Using reinforcement learning to find efficient qubit routing policies for deployment in near-term quantum computers," (2019), arXiv:1812.11619 [quant-ph].
[5] L. Kocsis and C. Szepesvari, in *ECML* (2006).
[6] R. Munos, Found. Trends Mach. Learn. **7**, 1 (2014).
[7] T. Cazenave and N. Jouandeau (2007).

[8] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, Nature **529**, 484 (2016).

[9] 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).

[10] T. M. Moerland, J. Broekens, A. Plaat, and C. M. Jonker, "Monte carlo tree search for asymmetric trees," (2018), arXiv:1805.09218 [stat.ML].

[11] Code for the complete simulation and visualization is available on `https://github.com/AnimeshSinha1309/quantum-rl`.

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

[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," (2019), arXiv:1801.07829 [cs.CV].

[14] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," (2017), arXiv:1710.05941 [cs.NE].

[15] H. Abraham, AduOffei, R. Agarwal, I. Y. Akhalwaya, G. Aleksandrowicz, T. Alexander, M. Amy, E. Arbel, Arijit02, A. Asfaw, A. Avkhadiev, C. Azaustre, AzizNgoueya, A. Banerjee, A. Bansal, P. Barkoutsos, A. Barnawal, G. Barron, G. S. Barron, L. Bello, Y. Ben-Haim, D. Bevenius, A. Bhobe, L. S. Bishop, C. Blank, S. Bolos, S. Bosch, Brandon, S. Bravyi, Bryce-Fuller, D. Bucher, A. Burov, F. Cabrera, P. Calpin, L. Capelluto, J. Carballo, G. Carrascal, A. Chen, C.-F. Chen, E. Chen, J. C. Chen, R. Chen, J. M. Chow, S. Churchill, C. Claus, C. Clauss, R. Cocking, F. Correa, A. J. Cross, A. W. Cross, S. Cross, J. Cruz-Benito, C. Culver, A. D. Córcoles-Gonzales, S. Dague, T. E. Dandachi, M. Daniels, M. Dartiailh, DavideFrr, A. R. Davila, A. Dekusar, D. Ding, J. Doi, E. Drechsler, Drew, E. Dumitrescu, K. Dumon, I. Duran, K. EL-Safty, E. Eastman, G. Eberle, P. Eendebak, D. Egger, M. Everitt, P. M. Fernández, A. H. Ferrera, R. Fouilland, FranckChevallier, A. Frisch, A. Fuhrer, B. Fuller, M. GEORGE, J. Gacon, B. G. Gago, C. Gambella, J. M. Gambetta, A. Gammanpila, L. Garcia, T. Garg, S. Garion, A. Gilliam, A. Giridharan, J. Gomez-Mosquera, Gonzalo, S. de la Puente González, J. Gorzinski, I. Gould, D. Greenberg, D. Grinko, W. Guan, J. A. Gunnels, M. Haglund, I. Haide, I. Hamamura, O. C. Hamido, F. Harkins, V. Havlicek, J. Hellmers, Ł. Herok, S. Hillmich, H. Horii, C. Howington, S. Hu, W. Hu, J. Huang, R. Huisman, H. Imai, T. Imamichi, K. Ishizaki, R. Iten, T. Itoko, JamesSeaward, A. Javadi, A. Javadi-Abhari, W. Javed, Jessica, M. Jivrajani, K. Johns, S. Johnstun, Jonathan-Shoemaker, V. K, T. Kachmann, A. Kale, N. Kanazawa, Kang-Bae, A. Karazeev, P. Kassebaum, J. Kelso, S. King, Knabberjoe, Y. Kobayashi, A. Kovyrshin, R. Krishnakumar, V. Krishnan, K. Krsulich, P. Kumkar, G. Kus, R. LaRose, E. Lacal, R. Lambert, J. Lapeyre, J. Latone, S. Lawrence, C. Lee, G. Li, D. Liu, P. Liu, Y. Maeng, K. Majmudar, A. Malyshev, J. Manela, J. Marecek, M. Marques, D. Maslov, D. Mathews, A. Matsuo, D. T. McClure, C. McGarry, D. McKay, D. McPherson, S. Meesala, T. Metcalfe, M. Mevissen, A. Meyer, A. Mezzacapo, R. Midha, Z. Minev, A. Mitchell, N. Moll, J. Montanez, G. Monteiro, M. D. Mooring, R. Morales, N. Moran, M. Motta, MrF, P. Murali, J. Müggenburg, D. Nadlinger, K. Nakanishi, G. Nannicini, P. Nation, E. Navarro, Y. Naveh, S. W. Neagle, P. Neuweiler, J. Nicander, P. Niroula, H. Norlen, NuoWenLei, L. J. O'Riordan, O. Ogunbayo, P. Ollitrault, R. Otaolea, S. Oud, D. Padilha, H. Paik, S. Pal, Y. Pang, V. R. Pascuzzi, S. Perriello, A. Phan, F. Piro, M. Pistoia, C. Piveteau, P. Pocreau, A. Pozas-Kerstjens, M. Prokop, V. Prutyanov, D. Puzzuoli, J. Pérez, Quintiii, R. I. Rahman, A. Raja, N. Ramagiri, A. Rao, R. Raymond, R. M.-C. Redondo, M. Reuter, J. Rice, M. Riedemann, M. L. Rocca, D. M. Rodríguez, RohithKarur, M. Rossmannek, M. Ryu, T. SAPV, SamFerracin, M. Sandberg, H. Sandesara, R. Sapra, H. Sargsyan, A. Sarkar, N. Sathaye, B. Schmitt, C. Schnabel, Z. Schoenfeld, T. L. Scholten, E. Schoute, J. Schwarm, I. F. Sertage, K. Setia, N. Shammah, Y. Shi, A. Silva, A. Simonetto, N. Singstock, Y. Siraichi, I. Sitdikov, S. Sivarajah, M. B. Sletfjerding, J. A. Smolin, M. Soeken, I. O. Sokolov, I. Sokolov, SooluThomas, Starfish, D. Steenken, M. Stypulkoski, S. Sun, K. J. Sung, H. Takahashi, T. Takawale, I. Tavernelli, C. Taylor, P. Taylour, S. Thomas, M. Tillet, M. Tod, M. Tomasik, E. de la Torre, K. Trabing, M. Treinish, TrishaPe, D. Tulsi, W. Turner, Y. Vaknin, C. R. Valcarce, F. Varchon, A. C. Vazquez, V. Villar, D. Vogt-Lee, C. Vuillot, J. Weaver, J. Weidenfeller, R. Wieczorek, J. A. Wildstrom, E. Winston, J. J. Woehr, S. Woerner, R. Woo, C. J. Wood, R. Wood, S. Wood, S. Wood, J. Wootton, D. Yeralin, D. Yonge-Mallo, R. Young, J. Yu, C. Zachow, L. Zdanski, H. Zhang, C. Zoufal, Zoufalc, a kapila, a matsuo, bcamorrison, brandhsn, nick bronn, brosand, chlorophyll zz, csseifms, dekel.meirom, dekelmeirom, dekool, dime10, drholmie, dtrenev, ehchen, elfrocampeador, faisaldebouni, fanizzamarco, gabrieleagl, gadial, galeinston, georgios ts, gruu, hhorii, hykavitha, jagunther, jliu45, jscott2, kanejess, klinvill, krutik2966, kurarrr, lerongil, ma5x, merav aharoni, michelle4654, ordmoj, sagar pahwa, rmoyard, saswati qiskit, scottkelso, sethmerkel, shaashwat, sternparky, strickroman, sumitpuri, tigerjack, toural, tsura crisaldo, vvilpas, welien, willhbang, yang.luh, yotamvakninibm, and M. Čepulkovskis, "Qiskit: An open-source framework for quantum computing," (2019).

[16] C. Developers, "Cirq," (2021), See full list of authors on Github: https://github .com/quantumlib/Cirq/graphs/contributors.

[17] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, Quantum Science and Technology **6**, 014003 (2020).

[18] V. Gheorghiu, S. Li, M. Mosca, and P. Mukhopadhyay, arXiv: Quantum Physics (2020).

[19] 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, edited by W. van Dam and L. Mancinska (Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019) pp. 3:1–3:24.