Qubit routing using Graph neural network aided Monte Carlo tree search

Animesh Sinha,[*] Utkarsh Azad,[†] and Harjinder Singh[‡]
Center for Computational Natural Science, International Institute of Information Technology, Hyderabad.

(Dated: March 27, 2021)

In the near-term quantum hardware, qubit connectivity is limited to the nearest neighboring qubits. This means that they 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. This article aims to provide an efficient solution called QRoute for the aforementioned qubit routing problem. Our procedure aims to minimize the depth of the transformed quantum circuits by utilizing the Monte Carlo tree search to perform qubit routing, which is aided by a Graph neural network that evaluates the value function and action probabilities for each state. We show that our procedure is architecture agnostic and outperforms other available qubit routing implementations on various circuit benchmarks.

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 exists a few pervading problems 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 quantum instructions that can be executed on them to perform some useful quantum computation [1]. 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 [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. 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 [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*. In this procedure, we propose to use the Monte Carlo tree search (MCTS), which is a look-ahead search algorithm for finding optimal decisions in the decision space guided by some 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], which is a neural network architecture that is being 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. 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.

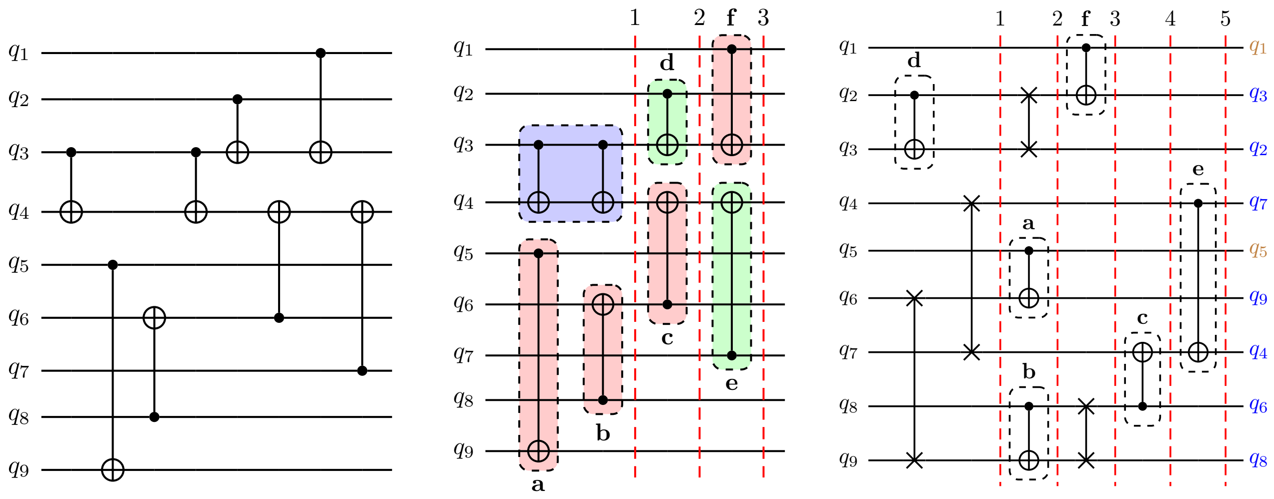
II. QUBIT ROUTING

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

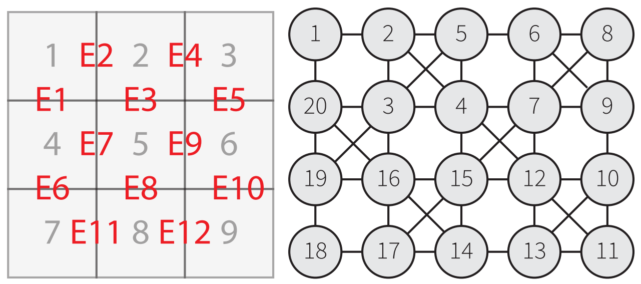
The topology of quantum hardware can be visualized as a qubit connectivity graph (Fig. 2). Each node in this graph would correspond to a physical qubit which in turn might correspond to a logical qubit depending on whether it is being used in the execution of quantum instruction set or not. The quantum 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. Amongst these two type of gate

* 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. 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



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

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

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 [9]. This edge could 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 qubits [14].

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:

$$\mathrm{R}(circuit,\ topology) \rightarrow circuit' \tag{1}$$

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 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 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 also 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 **(author?)** [15], for developing a Computer Aided Design-based (CAD) routing strategy. Since then, this problem has been outlined and presented in many different ways. Some of them being graph-based architecture-agnostic solution by **(author?)** [7] and reinforcement learning in a combinatorial action space solution by **(author?)** [9]. 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 **(author?)** [8].

Recently, Monte Carlo tree search, a popular reinforcement learning algorithm [16] previously proven successful in a variety of domains like playing puzzle games such as Chess and Go [17] was used by **(author?)** [18] 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 **(author?)** [19], 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. [20]

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

$$\mathrm{s_t} = (\mathrm{circuit},\ \mathrm{device},\ \mathrm{s_t.map},\ \mathrm{s_t.progress},\ \mathrm{s_t.locks}) \quad (2)$$

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.

$$O = \{(q_x, q_y) \mid q_x, q_y \in Q\} \quad (3)$$

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 **(author?)** [8]), 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.

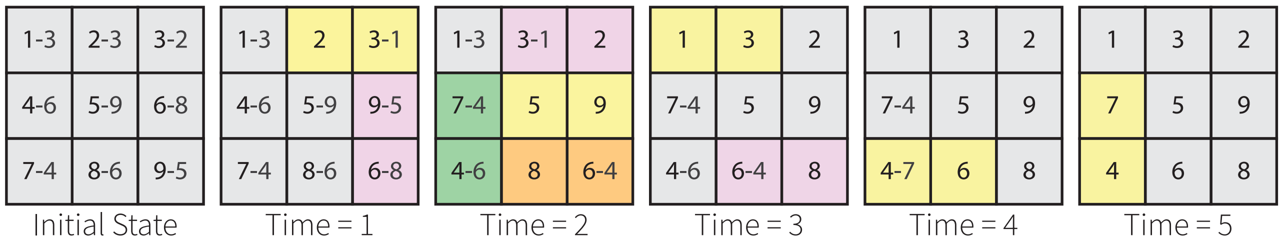


FIG. 3: Routing progress for 3×3 grid architecture while for the quantum circuit in Fig. 1a. Initial state shows the gate scheduled on each qubit which gets executed as control and target qubit adjacent to each other via SWAPs (shown in green and purple). The final state (at time=5) shows t he final 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 **(author?)** [21] 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

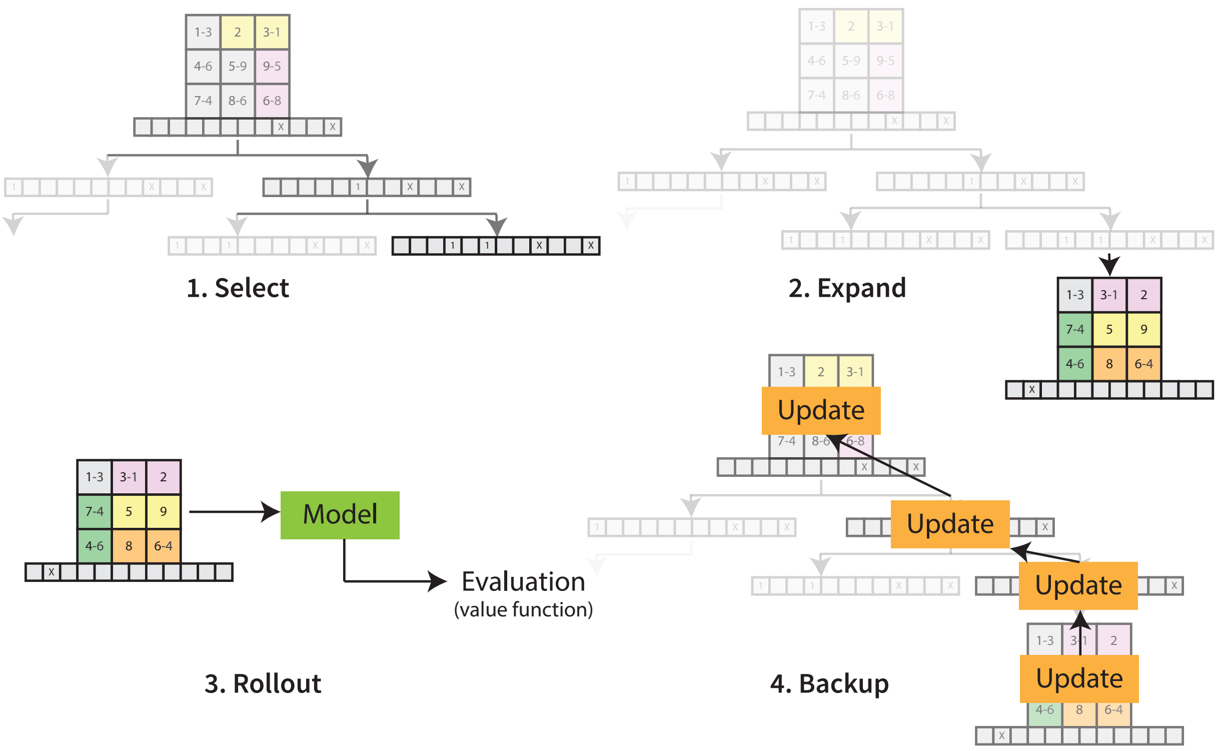


FIG. 4: Steps of Monte-Carlo tree search: (i) Select - Choose a mcts-state for exploration at current timestep, (ii) Expand - Calculating reward for newly encountered mcts-state and assign it as a child state, (iii) Rollout - Estimating long term value using a neural network for the state in the exploration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

of which break the mutual exclusivity constraint. We 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 Fig. 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 current state, and for each valid action from it, it maintains the statistics associated with that action. The following data is maintained in an mcts-state object:

- A reference to the corresponding state obect.
- A list of child nodes, each of which will be an mcts-state too.
- N-values corresponding to each child, which is number of iterations in which that child was explored.
- Q-values corresponding to each child, which is the mean estimated reward obtained when that child is explored.
- The solution array, that represents the action corresponding to the mcts-state (maintained as a list of swaps).

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

$$V(s) = \max_{s'} Q(s,a) \tag{4}$$

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

$$Q(s,a) = R(s,a,s') + \gamma V(s') \tag{5}$$

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).

$$\mathrm{UCT}(s,a) = Q(s,a) + c\frac{\sqrt{n(s)}}{n(s,a)} \times p(s,a) \tag{6}$$

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 **(author?)** [19]

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] ≠ 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 ← $\underbrace{\boxed{0}\boxed{0}\boxed{0}\boxed{\dots}\boxed{0}}_{\#\text{ edges on device}}$
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 ≠ 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.

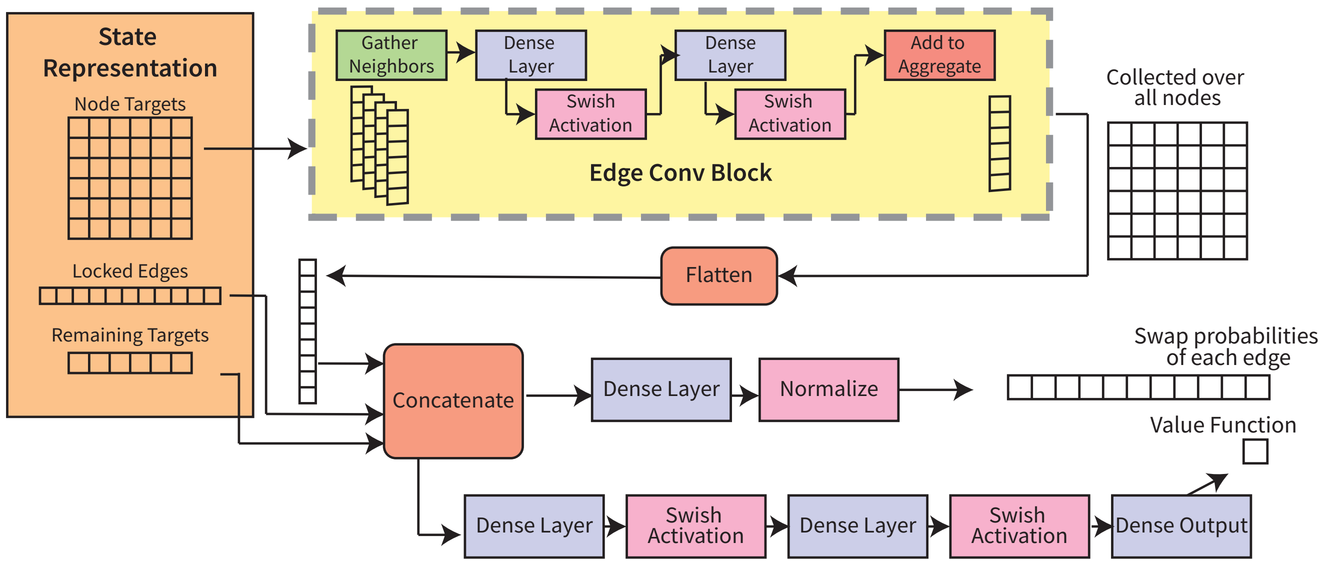


FIG. 5: Graph Neural Network Architecture with a value head and a policy head

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 [22] 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 input to the value and policy heads.

The value head is 2 fully connected layers with Swish

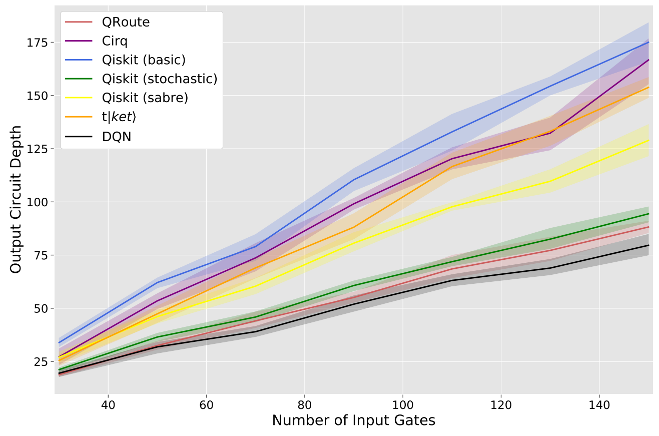


FIG. 6: Comparative performance of routing algorithms on random circuits as a function of the number of two-qubit operations in the circuit. Here, the results for DQN have been adapted from the CDR data (Eq. 7) received from the authors, and has not been replicated by us.

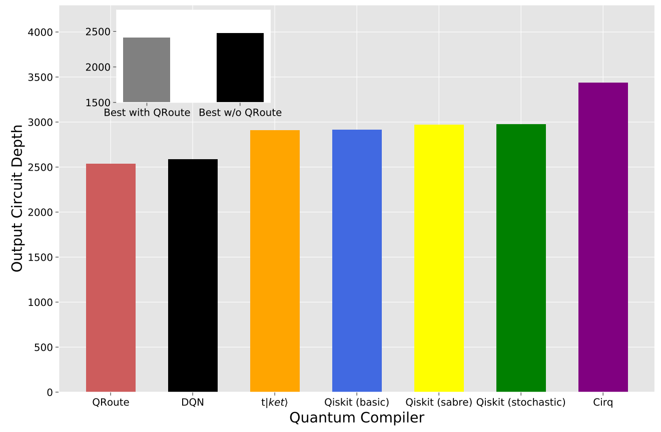


FIG. 7: Plots of output circuit depths of routing algorithms over small realistic circuits (≤ 100 gates), summed over the entire dataset. The inset shows the results on the same data comparing the best performant scheduler excluding and including QRoute on each circuit respectively. Here, the results for DQN have been extrapolated from the CDR data (Eq. 7) received from the authors, and hasn't been replicated by us.

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 QRoute against the routing algorithms from other state-of-the-art frameworks on various circuit benchmarks: (i) Qiskit and its three variants [23]: (a) basic, (b) stochastic, and (c) sabre, (ii) Deep-Q-Networks (DQN) from [8], (iii) Cirq [24], and (iv) t|ket⟩ from Cambridge Quantum Computing (CQC) [25]. Out of these, we note that 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 [26] but were disabled in our simulations to have a fair comparison. Moreover, the results for DQN given below are adapted from the data provided by the authors of **(author?)** [8], and couldn't be replicated by us.

A. Random Test Circuits

Our first benchmark for comparing our performance is the 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. 6. 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 (DQN) guided simulated annealer, after performing equivalently well to it on low layer density circuits. Nevertheless, QRoute still does consistently better than all the other major frameworks: Qiskit, Cirq and t|ket⟩, despite the increase in the average layer density and also scales reasonably well. This means, that even though the MCTS is guided reasonably well through the action space by our GNN, in few cases it might still be unable to distinguish between the optimality of two actions differing by few q-values due to limited number of iterations. This could lead to a few layer errors in the overall procedure which can be attributed as the reason to our close (on an average ≤ 4 layers) defeat to DQN on the random circuits with larger layer density.

B. Small Realistic Circuits

Next we test on the set of all circuits which 100 or less gates from the IBM-Q realistic quantum circuit dataset used by **(author?)** [27]. 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. The plot is in Fig. 7. Since the lack

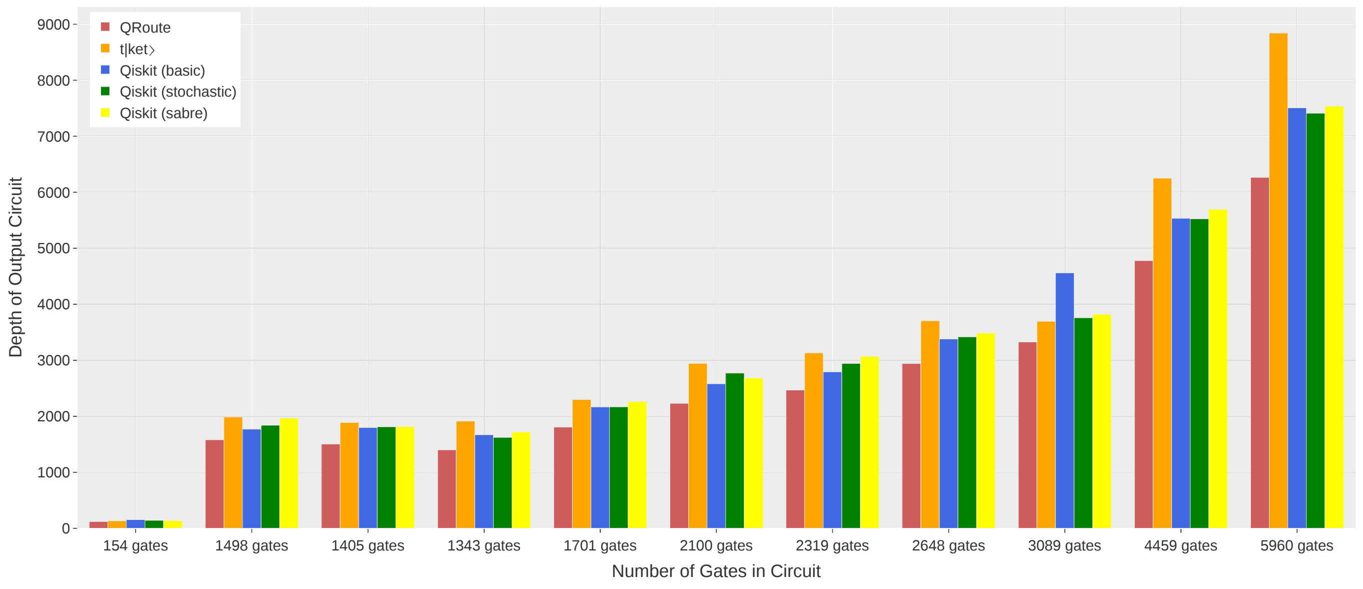


FIG. 8: The results over eight circuits sampled from the large realistic dataset benchmark, the outputs of each routing algorithm are shown for every circuit.

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.

Our model 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. We note that the pool including QRoute gives on an average 2.5% lower circuit depth, indicating that there is a significant number of circuits where QRoute is the best routing method available.

Our closest competitor on this dataset also is the Deep-Q-Network guided simulated annealer. To compare our performances, we look at the average circuit depth ratio (CDR), which is defined by:

$$\text{CDR} = \frac{1}{\#\text{circuits}} \sum_{\text{circuits}} \frac{\text{Output Circuit Depth}}{\text{Input Circuit Depth}} \tag{7}$$

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

C. Large Realistic Circuit

For our final benchmark, we took eight large circuits ranging from 154 gates to 5960 gates in its input from the IBM-Quantum realistic test dataset [27]. The results are detailed in the Table I and plotted in Fig. 8. Our model 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.

Here, we note that the results from DQN and Cirq are not available for this 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, majorly because simulated annealing used in it is an computationally expensive process which would require extensive runtime. Similarly, for Cirq, it takes several days to compile each of the near 5000 qubit circuits. In contrast to them, in our case, MCTS has to return a recommended optimal action at the end of the search iteration. These recommended actions gradually improve as the statistics about the simulated routing-tree gets updated.

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 a combinatorial action spaces and enforcing constraints using mutex locks can be adapted for several other combinatorial optimization problems. [28–32]

Our approach is also flexible enough that it allows us to compile circuits of any size onto any device, from small ones like IBMQX20 with 20 qubits, to much larger hard-

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

Input Circuit		Output Circuit Depth				
Name	Number of Gates	QRoute	t|ket⟩	Qiskit (basic)	Qiskit (stochastic)	Qiskit (sabre)
rd84_142	154	120	154	142	138	133
adr4_197	1498	1580	1770	1840	1968	1988
radd_250	1405	1504	1799	1812	1815	1888
z4_268	1343	1400	1670	1623	1718	1914
sym6_145	1701	1806	2167	2168	2261	2299
misex1_241	2100	2231	2580	2770	2681	2944
rd73_252	2319	2468	2793	2943	3071	3132
cycle10_2_110	2648	2941	3380	3418	3485	3705
square_root_7	3089	3327	4560	3759	3822	3695
sqn_258	4459	4779	5535	5526	5696	6252
rd84_253	5960	6264	7507	7411	7537	8843

ware like Google Sycamore with 53 qubits (The Circuit Depth Ratio on small circuits on Google Sycamore was 1.64,). Our model can also intrinsically deal with hardware with different primitive instruction set, for example on hardware where SWAP gates are not a primitive and they get decomposed to 3 operations. Our approach also enjoys significant tunability, hyperparameters can be changed easily to alter the tradeoff between time 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 personal computer 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's hard to achieve that without reducing the number of search iterations and trading off a bit of performance. Better, more predictive neural networks can help squeeze in slightly better speeds here.

One of the challenges of prior methods like DQN, that used Simulated Annealing to build up their actions was that the algorithm couldn't 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 our future explorations.

Finally, we provide an open-sourced access to our software library [20]. It will allow researchers and developers to implement variants of our methods with minimal effort. We hope that this will aid the future research work in the task of quantum circuit transformations.

We conclude that on the whole, our idea of the Monte Carlo Tree Search for building up solutions in combinatorial actions spaces has shown to equalize and exceed 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 implementing 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," Aug. 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*, vol. 574, no. 7779, p. 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," 2020.

[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," 2020.

[6] A. M. Childs, E. Schoute, and C. M. Unsal, "Circuit Transformations for Quantum Architectures," in *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019)*, vol. 135 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 3:1–3:24, Schloss

Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019.

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

[8] M. G. Pozzi, S. J. Herbert, A. Sengupta, and R. D. Mullins, "Using reinforcement learning to perform qubit routing in quantum compilers," *arXiv:2007.15957 [quant-ph]*, 2020.

[9] S. Herbert and A. Sengupta, "Using reinforcement learning to find efficient qubit routing policies for deployment in near-term quantum computers," *arXiv:1812.11619 [quant-ph]*, 2019.

[10] L. Kocsis and C. Szepesvari, "Bandit based monte-carlo planning," 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.*, vol. 7, pp. 1–129, 2014.

[12] T. Cazenave and N. Jouandeau, "On the parallelization of uct," 2007.

[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.

[14] J. C. Garcia-Escartin and P. Chamorro-Posada, "Equivalent quantum circuits," 2011.

[15] 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 (TCAD)*, 2018.

[16] 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*, vol. 4, no. 1, pp. 1–43, 2012.

[17] 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*, vol. 529, pp. 484–489, Jan 2016.

[18] X. Zhou, Y. Feng, and S. Li, "A monte carlo tree search framework for quantum circuit transformation," in *Proceedings of the 39th International Conference on Computer-Aided Design*, ICCAD '20, (New York, NY, USA), Association for Computing Machinery, 2020.

[19] T. M. Moerland, J. Broekens, A. Plaat, and C. M. Jonker, "Monte carlo tree search for asymmetric trees," 2018.

[20] Code for the complete simulation and visualization is available on `https://github.com/AnimeshSinha1309/quantum-rl`, 2021.

[21] E. W. Dijkstra, *Cooperating Sequential Processes*, pp. 65–138. New York, NY: Springer New York, 2002.

[22] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2017.

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

[24] C. Developers, "Cirq," Mar. 2021.

[25] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, "t$|$ket$\rangle$: a retargetable compiler for NISQ devices," *Quantum Science and Technology*, vol. 6, p. 014003, nov 2020.

[26] T. Itoko, R. Raymond, T. Imamichi, and A. Matsuo, "Optimization of quantum circuit mapping using gate transformation and commutation," 2019.

[27] A. Zulehner, A. Paler, and R. Wille, "IBM Qiskit developer challenge." `https://www.ibm.com/blogs/research/2018/08/winners-qiskit-developer-challenge/`. Accessed: 2021-03-23.

[28] N. Mazyavkina, S. Sviridov, S. Ivanov, and E. Burnaev, "Reinforcement learning for combinatorial optimization: A survey," 2020.

[29] Z. Xing and S. Tu, "A graph neural network assisted monte carlo tree search approach to traveling salesman problem," *IEEE Access*, vol. 8, pp. 108418–108428, 2020.

[30] R. Xu and K. J. Lieberherr, "Learning self-game-play agents for combinatorial optimization problems," *arXiv:1903.03674 [cs.LG]*, 2019.

[31] K. Abe, Z. Xu, I. Sato, and M. Sugiyama, "Solving np-hard problems on graphs by reinforcement learning without domain knowledge," *arXiv:1905.11623 [cs.LG]*, 2019.

[32] A. Laterre, Y. Fu, M. K. Jabri, A. Cohen, D. Kas, K. Hajjar, T. S. Dahl, A. Kerkeni, and K. Beguir, "Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization," *arXiv:1807.01672 [cs.LG]*, 2018.