Qubit Routing using Graph Neural Network aided Monte Carlo Tree Search

Animesh Sinha,* Utkarsh Azad,† Jai Bardhan, Kalp Shah, Bhuvanesh Sridharan, and Dr. Harjinder Singh‡
Center for Computational Natural Science, International Institute of Information Technology, Hyderabad.
(Dated: February 24, 2021)

Current day Quantum Computers offer limited connectivity, two qubit operations can only be applied on physically neighboring qubits. To respect this constraint when compiling quantum circuits to some hardware, we need to insert swap gates. It's desirable that the depth of the final circuit with the inserted gates is minimized, this article aims to provide an efficient solution to this problem.

We use Monte Carlo Tree Search to perform qubit routing, which is aided by a Graph Neural Network for evaluating the long term value of states as well as estimating prior probabilities of which actions are worth exploring. Minimizing circuit depth involves being able to perform operations in parallel, we using mutex locks to efficiently maintain this information and present it to the neural network. This should be of relevance to both the particular task of Qubit routing and as a general idea in several combinatorial optimizations.

I. INTRODUCTION

A. Describing the Problem

Present day quantum computer come in a variety of hardware architectures, but a pervading problem across them is limited qubit connectivity for two qubit operations and high error rates, which compound when increasing the depth of the circuit, due to which the execution of deep quantum circuits is an unfeasible task.

To make an arbitrary quantum circuit executable on a given target architecture, a quantum compiler has to insert SWAP gates so that gates in the original circuit only ever occur between qubits located at adjacent nodes, a process known as Qubit Routing. The process produce a new circuit, possibly with a greater depth, that implements the same unitary function as the original circuit while respecting the topological constraints [1].

Minimizing the depth of the quantum circuit resulting from the routing process is the objective we are trying to achieve.

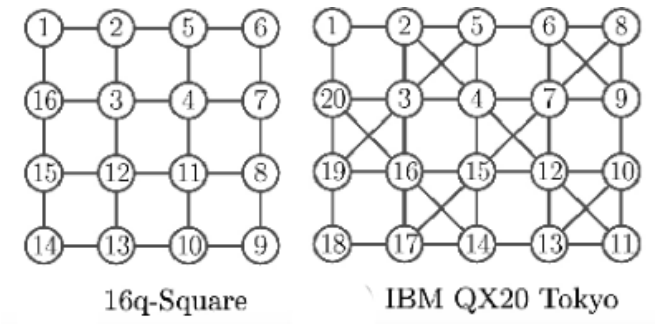

FIG. 1. Topology for Common Device Architectures [2]

B. Related Work

The problem of Qubit routing was outlined and presented with a graph based architecture-agnostic solution by Cowtan *et al.* [3]. The routing problem was phrased as a reinforcement learning problem in a combinatorial action space by Herbert and Sengupta [4]. 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 extended to use Double Deep Q-learning and Prioritized Experience Replay by Pozzi *et al.* [1].

Monte Carlo Tree Search (MCTS) has been a popular reinforcement learning algorithm [5–7] leading to great success in a variety of domains, playing puzzle games like Chess and Go [8] for instance. A version of MCTS, adapted for better exploration on Asymmetric Trees was proposed by Moerland *et al.* [9], which will be the formulation of choice for us. An unaided Monte Carlo Tree Search in the context of minimizing the total volume of quantum circuits (number of gates, ignoring the parallelization) was proposed by Zhou *et al.* [10], and has produced promising results.

C. Contribution of this work

This article proposes using Monte Carlo tree search for the task of total Depth minimization, as an easier to train and better performing machine learning setting.

- We use an array of mutex locks to represent the information the Agent needs to know to be able to make effective use of parallelization.
- We propose a graph neural network architecture to approximate the value function and the policy function for any given state of the system.
- We provide a simple python package to test and visualize routing algorithms with different neural net

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

architectures, combining algorithms, reward structures, etc.

From these changes, we hope to gain an agent which is not immediately greedy since it can only see the current state and not effectively plan for the future, and to stabilize the decision process which suffered from the randomness of introduced by simulated annealing.

II. METHOD

The solution to this problem involves maintaining a convenient representation of the current state of the total system, that is the remainder of the problem and the current state of the solution, and evolving it using actions chosen by our model. In this section we describe both these steps. (Code for the complete simulation and visualization is available on `https://github.com/AnimeshSinha1309/quantum-rl`)

A. State representation

1. Circuit Representation

Each circuit is input as a set of qubit $\mathbb{Q}$ and an partially ordered set of operations $\mathbb{O} = \{(q_x, q_y) | q_x, q_y \in \mathbb{Q}\}$. We convert this to a lists of lists representation as in Pozzi *et al.* [1], where for each logical qubit, q_i, we store the list of all other qubits that participate in 2 qubit operations with q_i, in the order they appear in the logical circuit.

So a gate (q_x, q_y) is executable if and only if the first $i-1$ operations for q_x and $j-1$ operations for q_y have been executed and the i^{th} operation for q_x is q_y and the j^{th} operation for q_y is q_x.

2. Mutex Locks

The core of the routing problem lies in the parallelization of the actions under the constraints that two operations in the same timestep cannot operate on the same qubit. A way to think about this is to call the qubits "resources" and the operations that are trying to use the "consumers" of this resource. This is reminiscent of several parallel processes competing for the same memory. We try to adapt the same solution of mutual exclusions locks [11] in this version of the problem.

Our output circuit will consist of 2 types of operations, CNOTs and SWAPs. All single qubit operations are much faster than any 2 qubit operations and therefore ignored. Depending on the primitive gates available on the hardware, a SWAP can either be an elementary gate and equivalent to CNOT, or be decomposed into 3 CNOT gates. So the locks array, which is a part of the state, should be a vector $l_t \in Z_+^{n_nodes}$. For each of the nodes, it states the number of timesteps it will stay locked for. This vector is also presented to the Neural Network so help it learn to avoid attempting to use the same node and several operations.

3. The State Object

Our simulation will proceed in timesteps. The entire specification of our system will be done by the state $s_t = (C, D, q_t, p_t, l_t)$.

- $q_t \in \mathbb{Z}_{n\_nodes}^{n\_nodes}$ is a vector denoting the qubit locations, the i$^{\text{th}}$ element in the array denotes the index of the logical qubit which is present on the i$^{\text{th}}$ element.
- $p_t \in \mathbb{Z}_+^{n\_nodes}$ is a vector which represents the progress of each qubit on the circuit.
- $l_t \in \mathbb{Z}_+^{n\_nodes}$ is a vector which stores the number of operations this qubit will stay locked for. It's a mutex lock of sorts, ensuring that a CNOT operation blocks a qubit for the next timestep, or a swap gate blocks it for the next 3.
- C - the circuit object. Common across all states, contains the list of lists representation of the circuit.
- D - the device object. Allows access to the shortest distance between any two nodes on the device graph, thereby also checking if a two qubit gate is admissible on the hardware given the nodes it has to operate on.

4. Action Space

Through our search, we evolve the state using Actions. These actions are one of two types: $SWAP(a, b)$ or $CNOT(a, b)$. A $SWAP$ action evolves the mapping by swapping elements in the qubit locations array q_t. A $CNOT$ action is only performed when a gate in the logical circuit is executable, as described here (II A 1), and satisfies the topological constraints of the circuit.

B. Reinforcement Learning Setting

Given a 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. This is updated in each step of our simulation for all the states that were visited.

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

We want to find the a, for each s which maximizes $Q(s,a)$, however, since we have to approximate $Q(s,a)$ by explicitly trying out all A, we employ a tree search to maximize this function.

C. Monte Carlo Tree Search

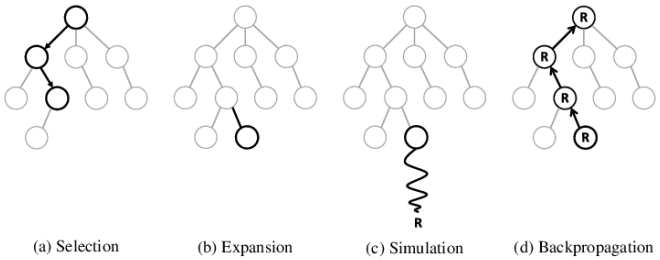

FIG. 2. Scaling of Initial Depth overhead with size of hardware

To search over the action space, we employ a tree search which consists of 4 steps. The first is a selection step which chooses the optimal node to explore. This choice is made by maximizing the Upper Confidence Bound on Trees value, which is a sum of the expected reward Q and expected standard deviation multiplied by a constant c. This constant weights the exploration tendency of the agent against the exploitation (choosing the best action based on current knowledge). The next step, called Expansion, explores the futures of the selected state. The third step is rollout, where we use our Neural Network to approximate the value function of the state. The fourth and final step is using the explicitly obtained rewards and the value function at the expanded state to update the Q functions of all states in it's ancestory.

$$UCT_s(a) = Q(s,a) + \sqrt{\frac{n(s,a)}{1+n(a)}} \times p(s,a) \qquad (3)$$

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 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.* [9]

D. Neural Network Architecture

Our tree search needs a value function approximator and a policy function approximator. The Value function will help us evaluate the total expected rewards from a state. The policy function will help choose which actions are with exploring. It's seems easier to learn a good policy function, since it's heavily dependent only on the current state, as opposed to a value function which depends a lot on the expected rewards in the future.

is a neural network which helps approximate the value function, which is an estimate of the number of additional timesteps that the circuit might take to compile, and the policy function, which is a vector of size n_edges, with real valued elements between 0 and 1.

III. RESULTS

A. Full Layer Circuits

A full layer in a quantum circuit refers to a logical circuit that has operations on all qubits such that each qubit is involved in exactly one operation. Such a circuit is perfectly parallelizable and can be executed in a single step. Therefore is serves as a test for the initial mapping as well as a sanity check for the parallelization step.

Given a good initial mapping, we manage to transform the circuit with no depth overhead. The results for our default algorithm (Not made specifically for this task, are shown in the graph 4)

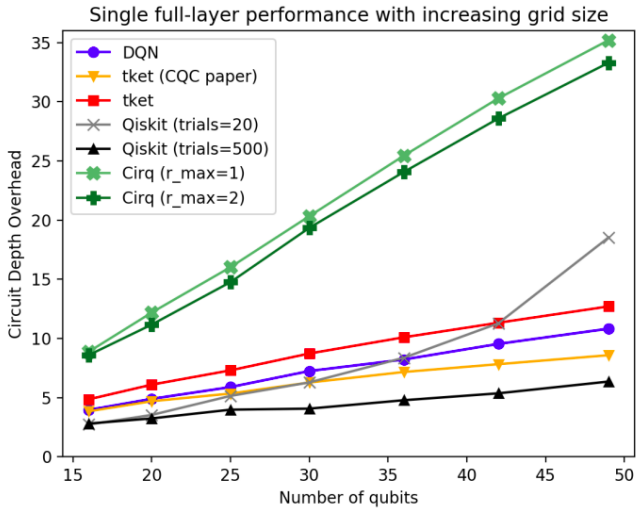

FIG. 4. Scaling of Initial Depth overhead with size of hardware

B. Random Test Circuits

We test the scaling of our algorithms on randomized circuits of varying number of gates to schedule and the number of qubits on target hardware.

We note that our method is relatively unphased by the growth in the size of the hardware, and scales better than other methods as the number of operations to schedule grow.

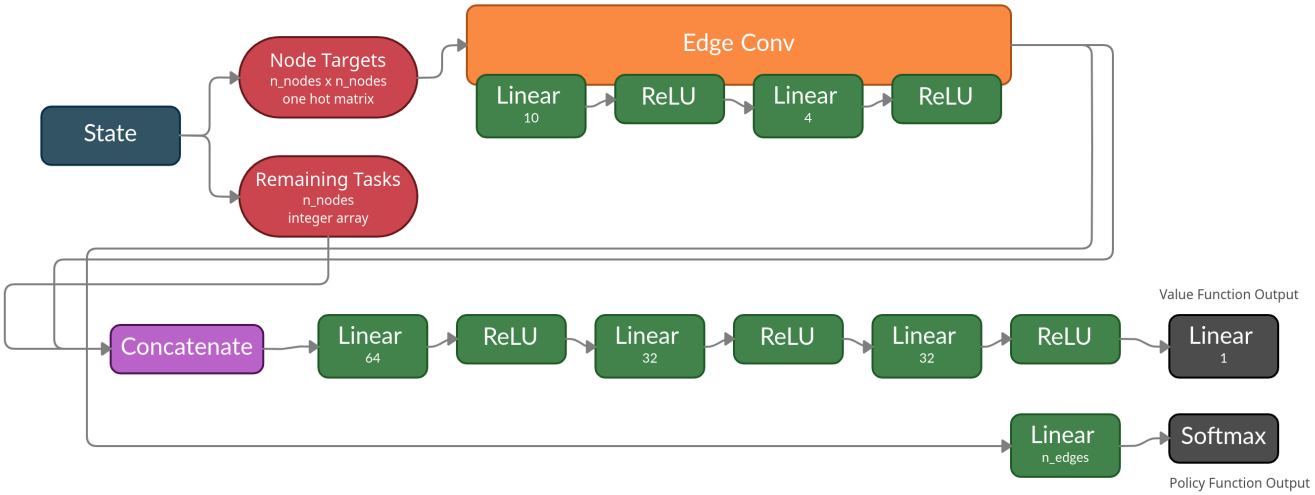

FIG. 3. Graph Neural Network Architecture with a Value head and a Policy head

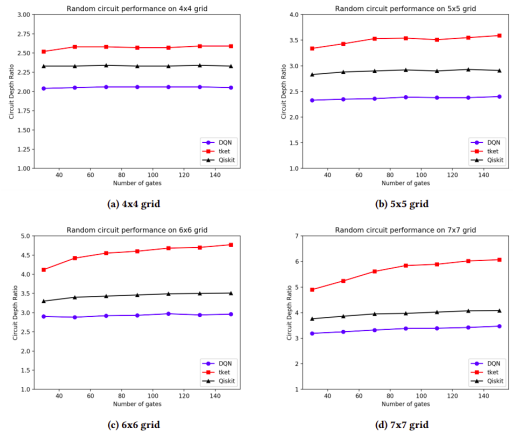

FIG. 5. Depth overhead on random circuits against number of gates in input circuit and number of qubits in the hardware.

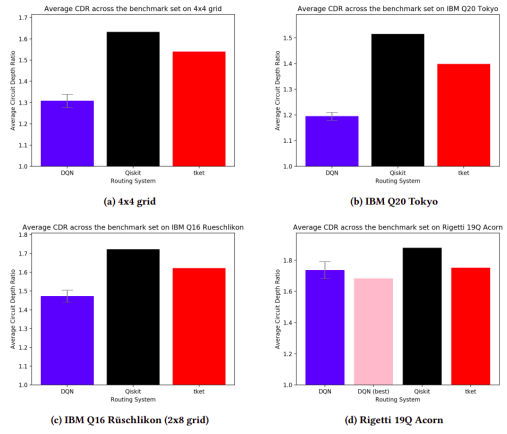

FIG. 6. Comparative performance against other methods across real world hardware

C. Realistic Test Set

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

Circuit	Gate Count	Cirq Depth	MCTS Depth
alu-v0_27			
ex1_226			
graycode6_47			
ham3_102			
mod5d1_63			
mod5mils_65			
xor5_254			

TABLE II. Results of our algorithm on a set of realistic test circuits

Circuit	Gate Count	Cirq Depth	MCTS Depth
rd84_142	154	154	
adr4_197	1498	2016	
radd_250	1405	1864	
z4_268	1343	1833	
sym6_145	1701	2357	
misex1_241	2100	2883	
rd73_252	2319	3143	
cycle10_2_110	2648	3595	
square_root_7	3089	3967	
sqn_258	4459	6067	

D. Speed of Transformation Process

Our Method is significantly faster than all other classical methods while still producing decent results. All our large circuit benchmarks are with 100 MCTS runs per step. The Following Image shows the time taken by our

algorithm and Cirq. We do not have this for over 2000 gates since Cirq didn't manage to output a result in over 2 days.

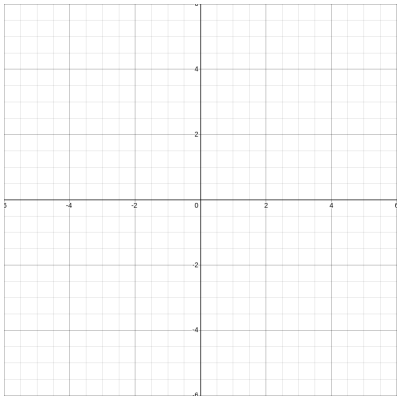

FIG. 7. Time Taken as a function of Circuit Size

IV. 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. Monte Carlo Tree Search

This is the basic structure of a generic Monte Carlo Tree Search algorithm aided by a value network and a policy network [8].

Algorithm 1 Monte Carlo Tree Search

Input: state s_i, model m
answer ← $\underbrace{\boxed{0}\boxed{\ldots}\boxed{0}}_{\text{—device.edges—}}$
Initialize $root \leftarrow$ Node(s_i, answer)
for i **in range** n_mcts **do**
  set $state \leftarrow$ **root**
  repeat
    select($state$**,** $model$**)** $\rightarrow action$
    if $state.child[action] \neq null$ **then**
      $action\_type \leftarrow selection$
      $state \leftarrow state.child[action]$
    else
      $action\_type \leftarrow expansion$
      expand ($state$**,** $action$**)**
    end if
  until $action\_type = expansion$
  rollout($state$**)** $\rightarrow reward$
  backup($state$**,** $reward$**)** $\rightarrow m$
end for

Algorithm 2 Selection Algorithm

Input: MCTS tree node $state$, model m
$prior \leftarrow m(state) + dirichlet$
Compute $UCT_i \leftarrow q_i + c \times \sqrt{\frac{n_i}{1+\sum n_i}} \times prior \ \forall i$
Choose node i with max UCT_i, randomly breaking tie

Algorithm 3 Expansion Algorithm

Input: MCTS tree node $state$, action a
if a = commit **then**
  $new\_action \leftarrow$ vector with all 0s.
  $step\_reward \leftarrow 0$
else
  $new\_action \leftarrow$ copy of $state.action$ with a^{th} element set to 1.
  $step\_reward \leftarrow 0$
end if
$state.child[a] \rightarrow$ new node ($action$, $step\_reward$)

Algorithm 4 Rollout Algorithm

Input: MCTS tree node $state$, model m
$next\_state \leftarrow$ environment step ($state$)
return $m(next\_state)$

Algorithm 5 Backup Process

Input: MCTS tree node $state$, model m
$prior \leftarrow m(state) + dirichlet$
Compute $UCT_i \leftarrow q_i + c \times \sqrt{\frac{n_i}{1+\sum n_i}} \times prior \ \forall i$
Choose node i with max UCT_i, randomly breaking tie

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

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

[10] 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).

[11] 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.

[12] 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.