Nuwa: A Quantum Circuit Transpiler Based on a Finite-Horizon Heuristic for Placement and Routing

Shengru Ren,† 1 KaWai Chen,† 1 Navid Ghadermarzy,† 1 Brandon Nguyen,† Yanhao Huang,† and Pooya Ronagh† 1, 2, 3, 4

1IQB Information Technologies (1QBit), Vancouver, BC, Canada
2Institute for Quantum Computing, University of Waterloo, Waterloo, ON, Canada
3Department of Physics & Astronomy, University of Waterloo, Waterloo, ON, Canada
4Perimeter Institute for Theoretical Physics, Waterloo, ON, Canada

We introduce a novel transpiler for the placement and routing of quantum circuits on arbitrary target hardware architectures. We use finite-horizon, and optionally discounted, reward functions to heuristically find a suitable placement and routing policy. We employ a finite lookahead to refine the reward functions when breaking a tie between multiple policies. We benchmark our transpiler against multiple alternative solutions and on various test sets of quantum algorithms to demonstrate the benefits of our approach.

I. INTRODUCTION

Classical optimization plays an important role in the study and development of quantum computing software–hardware stacks. Various layers of compilers must be used to transform a quantum algorithm written in a quantum programming language [24] to form an executable algorithm on a quantum processor. Ideally, this process can be fully automated so that a quantum software developer can abstract her attention away from considerations of optimal commutations of quantum gates [18, 19], mitigating various sources of noise and errors on the executed instruction sets [29], and the specific architectures of backend quantum devices [24].

An optimization problem related to compiling for a specific architecture requires the construction of an efficient transformation of the quantum program into one executable on specific systems of physical or logical qubits with geometric locality constraints such as 2D or 3D nearest-neighbour connectivity or other limited sets of multi-qubit entangling gates. This compilation step is of interest for the near-term compilation of small- to moderate-sized quantum algorithms on noisy, intermediate-scale quantum (NISQ) systems that have between tens and thousands of qubits [25]. Moreover, similar compilation procedures are of interest for fault-tolerant quantum computation (FTQC) on the large-scale architectures anticipated for quantum computers, such as modular architectures comprising arrays of superconducting qubits [22] or quantum dots and donors [44], and modular trapped-ion architectures dependent on shuttling ions between different trapping zones [3].

The goal of the aforementioned optimization task is to compile an input quantum circuit, written as an abstract sequence of quantum gates, into another circuit comprising quantum gates physically available on the target quantum processor while incurring the least possible overhead. A compiler that performs this translation may instead be called a transpiler, especially when the circuit transformation is happening at the same level of software abstraction (e.g., when both the input quantum circuit and the compiled circuit are written in terms of the Clifford + T gates), as opposed to when the compilation is from a higher level of abstraction to a lower one. The transpiler may aim to reduce the total number of additional gates it introduces in order to minimize the number of erroneous operations performed. Alternatively, the transpiler may reduce the depth, or makespan, of the compiled circuit in order to reduce the overall runtime of the algorithm, assuming that simultaneous gates can be performed with a negligible amount of cross-talk.

In this paper, we focus on the former optimization objective of reducing the total number of additional gates the transpiler introduces.

This optimization problem is often modelled as a placement and routing problem of finding a mapping of the abstract qubits to the physical qubits (placement), followed by iterations of performing entangling gates between qubits that are far apart by moving them closer to each other (routing), for example, via SWAP gates. The placement and routing problem has been extensively studied in the context of NISQ algorithms [2, 4, 10, 16, 21, 22, 25, 28–31, 33, 38–40, 43, 46, 48, 51, 52], as well as in the fault-tolerant compilation of quantum algorithms using concatenated error-correcting codes [26], lattice surgery–based fault-tolerant quantum computing (FTQC) [22, 62], and defect-based FTQC [20].

Additionally, the placement problem has been specifically studied in [25, 41, 42, 42]. Our focus in this paper is on developing techniques for routing, although we use them to choose good placements as well as in cases where multiple options are available. We should also distinguish the literature cited herein and our paper from the works [13, 14, 17, 19, 50], wherein architectural considerations are incorporated with compilers operating at the circuit synthesis level.

The problem and several of its variants are known...
to be NP-hard \cite{0,10}. Therefore, several of the mentioned references rely on heuristic methods for finding a (sub-)optimal placement and routing policy. Some of the solutions relevant to our approach are those based on the observation that the placement and routing problem is an inherently temporal problem over the span of multiple decision epochs. This includes an exact, but exponentially expensive, dynamic programming solution introduced in \cite{40}, temporal planning and constraint programming \cite{2,45}, integer programming \cite{80}, and methods that use reinforcement learning \cite{10,59}. On one hand, these approaches are superior to the greedy techniques that neglect the future impact of decisions made at earlier decision epochs. On the other hand, the global nature of the optimization problem solved hinders the scalability of the algorithm, especially in instances where the compilation procedure is expected to include computationally intensive subroutines such as the training of neural networks.

In this paper, we propose a middle-ground solution to the problem that captures both the efficiency of heuristic and greedy approaches, and the optimality of dynamic programming solutions in temporal decision making problems. This results in an algorithm that scales favourably in the SWAP gate overhead compared to other algorithms. As shown in Section IV and Appendix B, our algorithm achieves superior scaling compared to the state-of-the-art transpilers for general quantum circuits \cite{4,11}. We note that while \cite{51} and \cite{28} show improved results over those of \cite{11} for small circuits, a scalable performance improvement is not evident from these papers. In addition, \cite{21} and \cite{46} customize the transpilation problem for 2-local Hamiltonian simulation circuits whereas our algorithm is not specific to the type of input quantum algorithm (see also Fig. 4 for the benchmarking results of our algorithm on different types of quantum circuits).

### A. An Overview of Our Algorithm

Although our algorithm is a heuristic search method, it is inspired by techniques in approximate dynamic programming \cite{34}, as will be apparent from the terminology we use. We introduce two reward functions, the placement score Eq. 5 and edge score Eq. 6, that resemble the (possibly discounted) cumulative reward structures in dynamic programming and reinforcement learning. However, instead of performing costly exact or approximate dynamic programming recursions on them, we restrict their definitions to a finite decision horizon in which only a finite number of subsequent two-qubit gates are taken into account. We then use heuristic search to decide on the immediate actions to take, that is, the placement and routing of a subsequent layer of gates. The full placement and routing policy is formed by iterating over this process. The reward functions introduce two hyperparameters, namely, a discount factor and the decision horizon. These hyperparameters are used to tune the significance of future gates within the decision horizon on the values of the reward functions and, consequently, on the routing policy.

Calculating the finite-horizon reward functions frequently results in a tie across multiple routing options. This result is more prominent in the case of more-structured quantum circuits, which constitute most of the practical input circuits of the transpiler. Our algorithm uses a tie-breaking subroutine (Section III B 1) to refine the routing decisions. We use a hyperparameter called the lookahead depth to determine how many decisions to take into account (i.e., the number of SWAP gates we intend to insert) in order to refine the evaluation of our immediate actions when a tie occurs between the reward function values.

Our transpiler is agnostic with respect to hardware architectures; it is capable of receiving a representation of the connectivity of any target architecture as an input. Moreover, the hyperparameters decision horizon, discount factor, and lookahead depth allow our algorithm to adapt to the structure and recurrent patterns of the input quantum circuits and the classical computing budget available to it in order to provide a suitable (sub-)optimal placement and routing policy.

### B. Related Literature

We provide a detailed description of our approach in Section IV and in Section IV we demonstrate the superior performance of our transpiler against other algorithms proposed in the recent literature that, to our knowledge, are representative of the state of the art. Having said that, our approach incorporates several ideas from these competitors, which in this section we summarize comparatively.

We use two distinct routines for placement and routing. Our algorithm first chooses a placement by generating a set of candidates and ranking them according to their placement scores (Section III A). In principle, any placement can be considered in this ranking, but in this paper we confine our routine to generating candidate placements using the approaches of \cite{6} and \cite{4}. \cite{6} uses a linear path built from the connectivity between qubits to find the initial placement, and in \cite{4} a matching-based method is used (see Section III A 2 and Section III A 1 for more-detailed descriptions of the two placement methods). Neither of these references provide or utilize a systematic metric for comparing multiple placements with each other, whereas our placement score incorporates the effect of future gates in the input circuit for the purpose of this comparison.

In contrast, \cite{25} iterates between a forward and backward traversal of the sequence of two-qubit gates in order to generate a placement that takes future gates into account. This procedure is used to update the initial placement while routing the qubits in the forward and
backward passes using a heuristic. As such, [25] does not use separate stages for placement and routing; instead, the two tasks are intertwined. Iterating over this process becomes more expensive as the depth of the circuit grows and there is no guarantee that a greater number of iterations will produce better transpilations of the input circuit. On the other hand, [25] uses two sets of future gates (in either the forward or reverse passes); the gates to be processed immediately in the execution order and a subsequent set of future gates, to rank possible routing options. The idea of using future gates in a heuristic search to make better routing decisions is systematically incorporated in our work using the (discounted and cumulative) edge scores with a tunable hyperparameter that captures the impact of gates in future layers (see Section II and Section III B for the definitions of the concepts of layers and edge scores, respectively).

Unlike the strategy taken in [25], [4] presents the finding that a routing scheme chosen in a greedy fashion performs better overall in numerical benchmarking. In [4], the routing policy is also chosen greedily and based on the two-qubit gates to be processed immediately in the execution order. A distance vector is incorporated in this process. The SWAP gate is chosen from a set of candidates that reduces the greatest distance between pairs of qubits that are involved in two-qubit gates in line to be executed. The process is continued until either there is only one candidate left or a predefined limit has been reached. If this method fails, in [4] the same strategy is repeated on pairs of SWAP gates instead of individual ones, and if this also fails it resorts to brute force by routing maximally distant qubits involved in a two-qubit gate towards each other in order to escape this situation.

We also use a similar myopic strategy for routing (see the definition of immediate edge scores in Section III B). However, according to our findings, symmetries of the input circuit can frequently cause degenerate decisions in a myopic routing strategy. This is why in our tie-breaking subroutine (Section III B 1) we use the more general edge scores (with discounted contributions from gates in the further layers included) to choose the edge that leads to the minimum number of future SWAP gates needed within the decision horizon.

Finally, it is worth noting that [25] and [4] consider the trade-off between optimizing the number of inserted SWAP gates and the depth of the circuit (assuming independent gates can be executed concurrently). However, in this paper, we focus on optimizing the total number of additional gates, assuming the near-term and intermediate-scale scenarios in which the quantum state can to some extent be protected against decoherence (using techniques such as dynamic decoupling, error mitigation, and error correction). However, higher SWAP overhead and the cross-talk caused by performing multiple gates concurrently are both more detrimental to the performance of a quantum device.

II. THE PLACEMENT AND ROUTING PROBLEM

The problem of interest in this paper is as follows. Given an input quantum circuit, or input circuit, we wish to find a second circuit, or compiled circuit, that executes the quantum algorithm but uses the quantum gates available on a physical device, called the target architecture, or architecture for short. We assume that the input circuit is determined by a sequence

$$\sigma = (g_1, \ldots, g_N)$$

of \(N\) single-qubit rotations and CNOT gates affecting a set

$$Q = \{q_1, \ldots, q_M\}$$

of \(M\) (logical) qubits. The target architecture of the physical device also allows it to perform similar gates, but CNOT gates are not available between all pairs of physical qubits. The goal of our algorithm is to generate a compiled circuit that can be executed on the target architecture by inserting a plurality of SWAP gates in the sequence. We note that every SWAP gate can be performed using three CNOT gates. As the addition of gates creates a deeper circuit, the compiled circuit is more prone to errors and takes a longer time to execute. Therefore, our compiler has to minimize the number of inserted SWAP gates, from here on called the SWAP count, \(N_S\).

Figure 1(a) shows a representation of an input circuit. The target architecture to which we intend to compile our quantum algorithm is represented using an undirected graph called a coupling graph, \(G = (V, E)\), with vertices \(V\) and edges \(E\). Each physical qubit is represented as a vertex \(v \in V\) and the two-qubit gates available between physical qubits are represented by the edges \(\{u, v\} \in E\) between the vertices \(u\) and \(v\). We do not consider direction for the edges in this paper; however, our algorithm can be adapted to the case of devices with a particular direction for two-qubit gates as in the case of [7]. Figure 1(b) shows the coupling graph of the IBM Q20 Tokyo chip as an example target architecture [17].

We construct a layering for the input circuit, which is
FIG. 2. Example circuit after gates are grouped into layers using (a) fine layering and (b) coarse layering. Each highlighted box represents a layer. Gates in a given highlighted box belong to the same layer.

FIG. 3. (a) $R_Z$–control, (b) Control–control, (c) $R_X$–target, and (d) Target–target.

III. DESCRIPTION OF OUR ALGORITHM

A. Placement

The first step of our algorithm is to assign each qubit in the circuit to a vertex. An injective mapping

$p: Q \rightarrow V$

of qubits involved in the input circuit $\sigma$ to vertices of the coupling graph is called a placement. To a placement $p$ we associate a placement score

$$\Sigma_{p,\ell}(\lambda, \delta) = \sum_{g \in Q} \sum_{g' \in G} d_G(p(g), p(g')),$$

where $\lambda$ is a positive integer called the decision horizon, or horizon for short. For each qubit in $Q$, the horizon represents the number of future gates in the circuit taken into account in order to evaluate the utility of a placement. For every qubit $q \in Q$, the set $\sigma_{q,\lambda} \subseteq \sigma$ is the subset of the first $\lambda$ gates $g$ in $\sigma$ acting on $q$ with a unique other qubit involved. Here, $g = (g_1, g_2)$ is a gate acting on qubits $g_1$ and $g_2$, and $d_G$ denotes the distance between vertices in the coupling graph $G$. The value $\delta \in [0,1]$ is called the discount factor, and represents the significance of the contribution of later gates to the placement score. Discount factors closer to 0 aggressively suppress the contribution of gates in future layers, sorting them in chronological order, roughly speaking, whereas discount factors closer to 1 assign larger, yet exponentially decaying, weights on the distances between qubits involved in gates in future layers, allowing the future gates to make greater contributions. The best choice of discount factors depends on the structure of the input circuit. The hyperparameters decision horizon and discount factor appear also in the definition of edge scores in Section III B. We study the effect of these hyperparameters on the performance of our algorithm in Appendix A.

A good placement is one that can lead to a significant reduction in the final SWAP gate count. We present two heuristics for generating placements: matching-based and linear placement. We generate a number of placements using the two heuristics. We then compare the placements with each other using the placement score Eq. (5). Afterwards, we use the placement with the lowest placement score to route the circuits.

Within the same layer can be executed in any order, thus reducing the potential SWAP count, $N_S$. 

III. DESCRIPTION OF OUR ALGORITHM

Our algorithm consists of two main subroutines: placement and routing. We describe our method for finding a good placement in Section III A and then present our routing subroutine in Section III B.
1. Matching-based placement

This method involves finding a maximal matching on the coupling graph, and constructing a placement that maps the gates of the first layer onto the edges of the matching. The idea of using a matching in the coupling graph was first introduced in \[1\].

We employ two methods for finding a matching. The first is a greedy approach, in which edges are added to the matching until no additional edges can be added. The second approach is to use Edmonds’ blossom algorithm \[8\] to construct a maximum matching. In general, a maximum matching covers a larger number of target qubits. However, the blossom algorithm takes more computation time than the greedy matching and its effect is not significant for highly structured coupling graphs such as regular lattices. We refer the reader to Table \[1\] for a comparison of the performance of the greedy and blossom matching methods.

Using a matching in the coupling graph might not result in a complete placement. The remaining qubits without an assigned placement can be mapped to the remaining vertices at random. However, this may cause the qubits to be assigned to vertices that are far apart, especially when the number of vertices is much larger than the number of qubits. This would require us to introduce unnecessary SWAP gates in the compiled circuit in order to move the qubits closer to each other. To prevent this behaviour, we introduce another heuristic to keep the placed qubits closer to the more-connected regions in the architecture.

2. Linear placement

We construct maximal paths in the coupling graph using a greedy algorithm. This is done in two ways: the algorithm greedily favours either the high-degree or low-degree vertices. The latter method is used to account for architectures that have star-like subgraphs, for which starting from the highest-degree vertices might result in very short paths (see Fig. \[4\] for an example). For more-complex architectures, this greedy approach can be performed continually until a maximal linear forest has been found.

After the paths are constructed, the placement algorithm iterates through the vertices of any given longest path and moves the qubits to the earliest free vertices on the path.

As demonstrated in Section \[IVA\] the two heuristics introduced above can be used as stand-alone methods or in combination. Optionally, a plurality of placements can be generated and compared according to their placement scores.

B. Routing

The placement provided in Section \[IVA\] is used as input for our routing subroutine. Its pseudocode is shown in Algorithm \[1\]. The goal of the algorithm is to identify a set of edges in the coupling graph along which SWAP gates will be inserted.

![FIG. 4. Example architecture containing a star-like subgraph. Starting path construction from \(v_5\) might result in a short path, whereas starting from \(v_1\) will result in a longer path.](image-url)
At this point, if there are more vertices than qubits, some of the vertices will not be in the image of the placement function. We note that, if a qubit has not yet been used in any gate in \( \tau \), the qubit is free to be reassigned to an unoccupied vertex or exchange places with another unused qubit. We call such qubits free qubits.

After updating \( L_1 \), for every free qubit found in the input circuit, we ascertain whether a different placement for the free qubit improves its placement score, in which case we update the placement of the qubit. If all qubits in \( \sigma \) have already been included in \( \tau \), as there are no free qubits, this heuristic step will not be executed.

If no executable gates are found, the algorithm finds a number of edges to insert Swap gates along such that the qubits involved in non-executable gates are swapped to adjacent vertices, rendering them executable. Since Swap gates can be inserted on any edge in the coupling graph, we need to determine which edge would be the best next choice.

An edge \( e = \{q_1, q_2\} \in E \) in the coupling graph is a good choice if it results in fewer future Swap gates according to our routing algorithm. We determine such an edge using its edge score, which we define as a summation of two terms

\[
\Omega_{p,\ell}(\lambda, \delta, e) = \sum_{i=1,2} \Omega_{p,\ell}(\lambda, \delta, e, q_i).
\]

Here, each \( \Omega_{p,\ell}(\lambda, \delta, e, q_i) \) is zero if the first gate (not necessarily \( e \)) involving \( q_i \) is not in \( L_1 \), and otherwise

\[
\Omega_{p,\ell}(\lambda, \delta, e, q_i) = \sum_{g \in \sigma_{q_i,\lambda}} \delta^{|\ell(g)|-1} [d_G(p(g_1), p(g_2)) - d_G(p_e(g_1), p_e(g_2))].
\]

Here, \( \sigma_{q_i,\lambda} \subseteq \sigma \) is the set of the first \( \lambda \) gates in \( \sigma \) that act on \( q_i \) and one other qubit. The current placement \( p \) and the new placement \( p_e \) are associated with the edge \( e \) by switching the placement of the qubits mapped to \( q_1 \) and \( q_2 \), if any exist. That is, \( p_e(p^{-1}(q_1)) = q_2 \) if \( q_1 \) is in the image of \( p \), and, similarly, \( p_e(p^{-1}(q_2)) = q_1 \) if \( q_2 \) is in the image of \( p \).

Unlike in the definition of \( \sigma_{p,\lambda} \) in Section III A, we allow the repetition of gates in the definition of \( \sigma_{q,\lambda} \), as it is natural to expect that each gate between the same qubits should affect the edge score. We also empirically verify that not enforcing the condition \( \Omega_{p,\ell}(\lambda, \delta, e, q_i) = 0 \) when the first gate involving \( q_i \) is not in \( L_1 \) results in worse performance for our algorithm.

Figure 5 provides an example of the edge score calculation. Here, \( \lambda \) represents the decision horizon and \( \delta \) represents the discount factor. The extreme case of \( \delta = 0 \) is of particular interest, as it corresponds to the "myopic" decision horizon in which only the gates in the first layer are taken into account. The edge scores

\[
\Omega_{p,\ell}(e) := \Omega_{p,\ell}(\lambda \ge |L_1|, \delta = 0, e)
\]

are called immediate edge scores.

We use the immediate edge scores to heuristically find a convenient edge along which to insert a Swap gate. It is sufficient to calculate the edge scores only for the edges that are incident to the vertices corresponding to the gates in \( L_1 \). This is because inserting Swap gates along edges that are not incident to qubits in \( L_1 \) has no impact on \( L_1 \). We insert a Swap gate along the edge with the maximum immediate edge score. If multiple edges are found to have the maximum edge score, we invoke a tie-breaking heuristic, which we explain in Section III B 1.

The routing algorithm iterates over the above procedure until there are no gates left in \( L_1 \), at which point all gates have been routed and the algorithm terminates.

1. Tie-breaking subroutine

In many cases, multiple edges are found to have the maximum edge score. We call such edges degenerate. To break a tie between degenerate edges, the algorithm temporarily inserts a Swap gate for each such edge. It then continues the routing procedure, using an edge score.
with a decision horizon $\lambda > |L_1|$, until it generates a given number $D > 0$ of additional SWAP gates, or it terminates when the entire input circuit is compiled. The hyperparameter $D$ is called the lookahead depth. For each degenerate edge, the goal of this multi-step lookahead is to inspect the future state of the placement score after the insertion of $D$ SWAP gates within the decision horizon. We note that, compared to $\lambda$, $D$ has to be sufficiently small for this $D$-step lookahead to remain within the decision horizon.

If, after the addition of $D$ SWAP gates, the entire input circuit has not been routed, the placement score of the resulting placement is stored as a tie-breaking score for the degenerate edges.

However, if the entire input circuit has been routed with $D' \leq D$ additional SWAP gates, then the tie-breaking score of the degenerate edge is defined as $D' - D$. The tie-breaking procedure then selects the edge with the smallest tie-breaking score. If a degenerate case still occurs for the tie-breaking scores, then one of the degenerate edges is selected at random.

2. Complexity analysis

The time complexity of the routing procedure depends on the size of the input circuit, the number of steps required to calculate the insertion of a SWAP gate, and the number of tie-breaking subroutine calculations. To estimate the upper bound, we assume the worst case in our analysis. In this case, all qubits are involved in each step of the calculation. The routing procedure starts by performing an update on the layering, which has a complexity of $O(\lambda N)$, where $N$ is the number of qubits. The computation of the immediate edge score has a complexity of $O(N)$. Assuming all edge scores are degenerate, we call the tie-breaking subroutine for all candidate edges. The subroutine's complexity depends on the lookahead depth $D$. For each of the candidate edges, we need to insert at most $D$ SWAP gates. This leads to a time complexity of $O(DN)$. In total, the time complexity for routing a single gate is $O(\lambda N) + O(N) + O(DN)$. Using $N_G$ to denote the total number of gates, the time complexity of our routing procedure is $O(N_G \cdot (\lambda + D)N)$.

IV. NUMERICAL RESULTS

In this section, we present our results and compare them with those of other placement and routing algorithms. To the best of our knowledge, SABRE [24] is a commonly used quantum compiler for benchmarking, arct [4] scales well, and runs fast for large circuits, and tket [11] is the state-of-the-art library for general quantum circuits.

In Section IV A, we discuss the effects of different combinations of the placement schemes, introduced in Section III A on the SWAP count. As part of an ablation study, we benchmark the performance of our algorithm both with and without the application of the commutation rules presented in Fig. 3. Then, in Section IV B, we use a set of hyperparameters that provides a good balance between performance and runtime, and the best placement scheme found to compare our algorithm against the above-mentioned alternatives. We perform a hyperparameter study and find that while certain choices of hyperparameters are evidently inadequate (e.g., the discount factors $\delta = 0$ or 1), small circuits are not very sensitive to the values of the hyperparameters. The results of the study can be found in Appendix A.

A. Placement Heuristics

In our experiments, we benchmark three placement heuristics: (1) linear placement as a stand-alone placement heuristic; (2) the combination of matching-based placement and linear placement; and (3) a random mixture of the two methods to generate a plurality of placements, followed by performing a number of random SWAP gates before calculating the score of each placement. Methods (1) and (2) each generate a single placement, but method (3) generates multiple placements that are then compared to each other according to their placement scores, with the placement that has the best score being selected. We choose the hyperparameters $\delta = 0.1, D = 7$, and $\lambda = 40$ in our benchmarking, performing benchmarks on the test set of circuits provided in [25]. We select the hyperparameters here with the aim to strike a balance between performance and runtime for the circuits tested.

We report our results in Table 4.

As shown in Table 4, method (1) performs best for Ising model circuits, independent of whether we apply commutation rules. This is expected, as the two-qubit gates in the circuits representing one-dimensional Ising models are CNOT gates that are applied to successive qubits. Given the linear structure of Ising model circuits, they are not representative of general benchmarking circuits; therefore, we exclude them from our discussion and the results in Table IV.

We observe from Table 4 that method (3) produces the greatest number of best results for smaller circuits (i.e., circuits with fewer than 100 gates) among the benchmark tests we run, both with and without the application of commutation rules. However, for larger circuits (i.e., those having more than 400 gates), method (2B) produces the greatest number of best results, both with and without applying commutation, by taking advantage of the matching algorithms and randomness involved in producing the rest of the placements.

In Table 4, we use the best results as a baseline. We calculate the average percentage increase in the SWAP count for each method, as compared to the baseline. On average, method (3) outperforms the other methods, and the use of commutation rules improves the performance of the transpiler. As a result, we use method (3) and
apply commutation rules for the following benchmark, in which we compare our algorithm’s performance against that of other algorithms.

Table I. Swap counts with different placement methods performed on the test quantum circuits in [25]. Method (1): linear placement, method (2A): matching-based placement with greedy matching, method (2B): matching-based placement with blossom matching, and method (3): multiple placements. The lowest Swap count for each circuit is shown in bold text and with an asterisk.

| Circuit | No. of Qubits | No. of Gates | Avg. Swap Count – Without Commutation | Avg. Swap Count – With Commutation |
|---------|---------------|--------------|--------------------------------------|-----------------------------------|
|         |               |              | (1) (2A) (2B) (3)                    | (1) (2A) (2B) (3)                 |
| 4mod5-v1_22 | 5             | 21           | 3 3.33 2.33 2.33                    | 2* 3 2.33 2*                     |
| mod5mils_65  | 5             | 35           | 4 4.67 2.33* 4.33                   | 2* 3.33 4.33 2.33*               |
| alu-v0_27   | 5             | 36           | 3.33 5.67 3.33* 3.33                | 2* 3.33 5.33 3.33                |
| deco24-v2_43| 4             | 52           | 5.67 5.67 5.67 2.33*                | 9.33 7.33 6 3.33*               |
| 4gt13_92   | 5             | 66           | 8.67 7 5.67 2.33*                  | 9.67 4.67 4.67 4*               |
| ising_model_10 | 10            | 480          | 0* 7.33 9.67 5                      | 0* 5.67 8 5                      |
| ising_model_13 | 13            | 633          | 0* 12 11 6.33                      | 0* 10.67 11.67 6.67             |
| ising_model_16 | 16            | 786          | 1 10.33 14 7.33                    | 0* 14.67 14.33 6.33             |
| rd84_142   | 15            | 343          | 49.33 59.67 52.67 43.33            | 39.67* 47 47.67 44.33            |
| adr4_197   | 13            | 3439         | 325.67 302 277.33 318              | 293.67 263.67 248.33* 248.67    |
| radd_250   | 13            | 3213         | 294.33 307 311 288.33              | 227.33* 257.33 245 258          |
| z4_268     | 11            | 3073         | 277.67 260.67 253.33 270           | 210.33* 243.67 237.33 225.33    |
| sym6_145   | 7             | 3888         | 324.67 314 220 282.67              | 231 264.67 201* 205.67          |
| misex1_241 | 15            | 4813         | 345.33 339.67 351.33 311.33        | 282 275.33 326.33 268.67*       |
| rd73_252   | 10            | 5321         | 430.67 480.33 486.33 412           | 375.33* 416 402 396.67          |
| cycle10_2_110 | 12           | 6050         | 567.67 539.33 479.33 559           | 489.67 421.67* 468.67 456.33    |
| square_root_7 | 15           | 7630         | 558 416* 563.33 523                 | 431 519 501 420.33              |
| sqn_258    | 10            | 10223        | 937.67 838 919 776                 | 697.67 690.67* 747 693.33       |
| rd84_253   | 12            | 13658        | 1332.67 1202.33 1272 1170.33       | 1043.33 1011.67 1067.67 1010*   |
| col4_215   | 15            | 17938        | 1861.67 1682.33 1841 1850.67       | 1665.33 1630.67 1498* 1639.67   |
| sym9_193   | 10            | 34881        | 3305 3272 3089.33 3249.33          | 2878.67 2782.67 2453* 2696.67   |
| 9symml_195 | 11            | 34881        | 3113.33 2945 3153 3034             | 2459 2674 2147.33* 2817         |
| Number of Lowest Swap Counts | 2 | 1 | 0 | 2 | 9 | 2 | 5 | 6 |

Table II. Average relative increase in the Swap counts for each placement method shown in Table I. The relative increase percentages are with respect to the baseline, defined as the lowest Swap count observed between all four methods. Method (1): linear placement, method (2A): matching-based placement with greedy matching, method (2B): matching-based placement with blossom matching, and method (3): multiple placements.

| Without Commutation | (1) | (2A) | (2B) | (3) |
|---------------------|-----|------|------|-----|
| 44.4%               | 45.5% | 45.2% | 24.9% |
| 23.5%               | 24.1% | 25.3% | 9.1% |

B. Comparison with Other Algorithms

We first compare the results obtained from our traspilier, called Nuwa, with those produced by arct in [4] and tket in [1]. Several families of practically important quantum circuits have been contributed by [4, 87] for benchmarking purposes. We benchmark the three algorithms on these families, namely, quantum circuits pertaining to quantum signal processing (QSP) [24], quantum Fourier transformation (QFT), and the product formula expansion of exponentials of local Hamiltonians, which is of interest in Hamiltonian simulation applications [5]. We also benchmark the three algorithms on randomly generated circuits. Note that the initial placements for the four algorithms are not identical.

The results of our comparison are shown in Fig. 6. On average, Nuwa uses 9.0% fewer Swap gates than tket and 51.8% fewer Swap gates than arct for QSP circuits. For QFT circuits, the average improvement is 22.4% and 52.6%, respectively. For product formula circuits, Nuwa uses 13.1% more Swap gates than tket but 83.1% fewer than arct. For random circuits, the average improvement is 15.7% and 45.1%, respectively.

We next compare Nuwa’s results with those of Qiskit implementation [15] of SABRE. We use the same circuits as used in [25]. Results produced by tket are also included, and are shown in Table III. We note that we were not able to include SABRE in the study whose results are shown in Fig. 6 on test instances of [27], as the Qiskit implementation of SABRE requires the number of vertices of the coupling graph to be the same as the number of qubits in the input quantum circuit.

Nuwa outperforms other algorithms by a significant margin, both with and without applying commutation.

We also benchmark the three algorithms on QSP circuits as used in [25]. Results produced by tket are also included, and are shown in Table III. We note that we were not able to include SABRE in the study whose results are shown in Fig. 6 on test instances of [27], as the Qiskit implementation of SABRE requires the number of vertices of the coupling graph to be the same as the number of qubits in the input quantum circuit.

The results of our comparison are shown in Fig. 6. On average, Nuwa uses 9.0% fewer Swap gates than tket and 51.8% fewer Swap gates than arct for QSP circuits. For QFT circuits, the average improvement is 22.4% and 52.6%, respectively. For product formula circuits, Nuwa uses 13.1% more Swap gates than tket but 83.1% fewer than arct. For random circuits, the average improvement is 15.7% and 45.1%, respectively.

We next compare Nuwa’s results with those of Qiskit implementation [15] of SABRE. We use the same circuits as used in [25]. Results produced by tket are also included, and are shown in Table III. We note that we were not able to include SABRE in the study whose results are shown in Fig. 6 on test instances of [27], as the Qiskit implementation of SABRE requires the number of vertices of the coupling graph to be the same as the number of qubits in the input quantum circuit.

Nuwa outperforms other algorithms by a significant margin, both with and without applying commutation.
FIG. 6. (a) Swap counts of the QSP circuits. (b) Average Swap counts of randomly generated circuits. (c) Swap counts of QFT circuits. (d) Average Swap counts of the product formula circuits. For (b) and (d), we use the median and interquartile range for each average data point.

| Circuit          | No. of Qubits | No. of Gates | tket | SABRE | arct  | Nuwa |
|------------------|---------------|--------------|------|-------|-------|------|
|                  |               |              |      |       | Without Commutation | With Commutation |
| 4mod5-v1,22      | 20            | 20           | 20   | 20    | 20    | 20   |
| mod5mib,65       | 20            | 20           | 20   | 20    | 20    | 20   |
| alu-v0,27        | 20            | 20           | 20   | 20    | 20    | 20   |
| decod24-v2,43    | 20            | 20           | 20   | 20    | 20    | 20   |
| 4gt13,92         | 20            | 20           | 20   | 20    | 20    | 20   |
| ising_model,10   | 20            | 20           | 20   | 20    | 20    | 20   |
| ising_model,13   | 20            | 20           | 20   | 20    | 20    | 20   |
rules, except for the circuit $\text{rdS4.142}$. Without applying commutation rules, Nuwa uses approximately 34% fewer SWAP gates than $\text{tket}$ on average, and 66% fewer than SABRE. With commutation rules applied, the average improvement percentages are 40% and 71%, respectively.

Finally, we note that the benchmarking instances in [25] are extended to a larger set of input circuits in [53]. We refer the reader to Appendix B for the results of this extended benchmarking performed using two target architectures.

V. CONCLUSION

The development of quantum circuit compilers and transpilers requires exploration and incorporation of classical optimization and operations research techniques. Not only are these compilers and transpilers expected to fulfill critical roles in future quantum computing software stacks, but to serve as important research tools for the production of architecture-aware resource estimates for quantum algorithms. Transpilers could, in turn, be used to find improved architectures for quantum devices, and to provide more-accurate estimates of the number and fidelity of gates required for practical applications [11, 13, 36].

In this paper, we have introduced a novel transpiler for the placement and routing of qubits on a target architecture. Our transpiler uses a finite decision horizon to evaluate two (possibly discounted) rewards: the placement and edge score functions. It also incorporates a finite lookahead tie-breaking subroutine. We have benchmarked our transpiler against multiple alternative solutions and on various test sets. We showed that our algorithm performs significantly better compared to other algorithms when the circuits contain a large number of gates.

Quantum resources are much more scarce than classical computing resources, which justifies transpilers spending substantial classical computing resources in the optimization of placement and routing policies. It is, however, important to have scalable solutions that can be employed on larger quantum circuits or adapted to fault-tolerant circuit compilation for much larger quantum algorithms. In addition, it is important for the transpiler to be flexible with respect to target architectures. This flexibility allows us to use transpilers as research tools for quantitative analysis of the overhead of compiling quantum algorithms on various target architectures. Our transpiler provides these critical advantages while improving the state of the art.

ACKNOWLEDGMENTS

This project was funded by 1QBit. P. R. additionally acknowledges the financial support of Mike and Ophelia Lazaridis, and Innovation, Science and Economic Development Canada. All authors thank our editor, Marko Bucyk, for his careful review and editing of the manuscript.

[1] Ryan Babbush, Jarrod R McClean, Michael Newman, Craig Gidney, Sergio Boixo, and Hartmut Neven. 2021. Focus beyond quadratic speedups for error-corrected quantum advantage. *PRX Quantum* 2, 1 (2021), 010103.

[2] Kyle Booth, Minh Do, J Beck, Eleanor Rieffel, Davide Venturelli, and Jeremy Frank. 2018. Comparing and integrating constraint programming and temporal planning for quantum circuit compilation. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 28.

[3] Kenneth R Brown, Jungsang Kim, and Christopher Monroe. 2016. Co-designing a scalable quantum computer with trapped atomic ions. *npj Quantum Information* 2, 1 (2016), 1–10.

[4] Andrew M Childs, Eddie Schoute, and Cem M Unsal. 2019. Circuit transformations for quantum architectures. *arXiv preprint arXiv:1902.09102* (2019).

[5] Andrew M Childs, Yuan Su, Minh C Tran, Nathan Wiebe, and Shuchen Zhu. 2021. Theory of trotter error with commutator scaling. *Physical Review X* 11, 1 (2021), 011020.

[6] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. 2019. On the qubit routing problem. *arXiv preprint arXiv:1902.08091* (2019).

[7] Gerhard W Dueck, Anirban Pathak, Md Mazder Rahman, Abhishek Shukla, and Anindita Banerjee. 2018. Optimization of Circuits for IBM’s five-qubit Quantum Computers. In *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE, 680–684.

[8] Jack Edmonds. 1965. Paths, trees, and flowers. *Canadian Journal of Mathematics* 17 (1965), 449–467.

[9] Hongxiang Fan, Ce Guo, and Wayne Luk. 2022. Optimizing quantum circuit placement via machine learning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 19–24.

[10] Azim Farghadan and Naser Mohammadzadeh. 2017. Quantum circuit physical design flow for 2D nearest-neighbor architectures. *International Journal of Circuit Theory and Applications* 45, 7 (2017), 989–1000.

[11] Blake Gerard and Martin Kong. 2021. Exploring Affine Abstractions for Qubit Mapping. In *2021 IEEE/ACM Second International Workshop on Quantum Computing Software (QCS)*. IEEE, 43–54.

[12] Blake Gerard and Martin Kong. 2021. String Abstractions for Qubit Mapping. *arXiv preprint arXiv:2111.03716* (2021).

[13] Vlad Gheorghiu, Jiaxin Huang, Sarah Meng Li, Michele Mosca, and Priyanka Mukhopadhyay. 2022. Reducing the CNOT count for Clifford+ T circuits on NISQ archi-
tectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022).

[14] Craig Gidney and Martin Ekerä. 2021. How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits. *Quantum* 5 (2021), 433.

[15] Abraham Héctor et al. 2019. Qiskit: An Open-source Framework for Quantum Computing. *Retrieved from DOI: DOI: https://doi.org/10.5281/zenodo.2562110* (2019).

[16] Steven Herbert and Akash Sengupta. 2018. Using reinforcement learning to find efficient qubit routing policies for deployment in near-term quantum computers. *arXiv preprint arXiv:1812.11619* (2018).

[17] Stefan Hillmich, Alwin Zulehner, and Robert Wille. 2021. Exploiting Quantum Teleportation in Quantum Circuit Mapping. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 792–797.

[18] Toshinari Itoko, Rudy Raymond, Takashi Imamichi, and Atsushi Matsuo. 2020. Optimization of quantum circuit mapping using gate transformation and commutation. *Integration* 70 (2020), 43–50.

[19] Toshinari Itoko, Rudy Raymond, Takashi Imamichi, Atsushi Matsuo, and Andrew W. Cross. 2019. Quantum circuit compilers using gate commutation rules. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. 191–196.

[20] Ali Javadi-Abhari, Pranav Gokhale, Adam Holmes, Diana Franklin, Kenneth R Brown, Margaret Martonosi, and Frederic T Chong. 2017. Optimized surface code communication in superconducting quantum computers. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 692–705.

[21] Lingling Lao and Dan E Browne. 2022. qpan: A quantum compiler for 2-local qubit hamiltonian simulation algorithms. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*. 351–365.

[22] Lingling Lao, Bas van Wee, Imran Ashraf, J van Someren, Nader Khammassi, Koen Bertels, and Carmen G Almudever. 2018. Mapping of lattice surgery-based quantum circuits on surface code architectures. *Quantum Science and Technology* 4, 1 (2018), 015005.

[23] Ryan LaRose. 2019. Overview and comparison of gate level quantum software platforms. *Quantum* 3 (2019), 130.

[24] Frank Leymann and Johanna Barzen. 2020. The bitter truth about gate-based quantum algorithms in the NISQ era. *Quantum Science and Technology* 5, 4 (2020), 044007.

[25] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the qubit mapping problem for NISQ-era quantum devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1001–1014.

[26] Chia-Chun Lin, Susmita Sur-Kolay, and Niraj K Jha. 2014. PAQCS: Physical design-aware fault-tolerant quantum circuit synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23, 7 (2014), 1221–1234.

[27] Guang Hao Low and Isaac L Chuang. 2017. Optimal Hamiltonian simulation by quantum signal processing. *Physical review letters* 118, 1 (2017), 010501.

[28] Abtin Molavi, Amanda Xu, Martin Diges, Lauren Pick, Swamit Tannu, and Aws Albaughthi. 2022. Qubit Mapping and Routing via MaxSAT. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, IEEE, 1078–1091.

[29] Prakash Murali, Jonathan M Baker, Ali Javadi-Abhari, Frederic T Chong, and Margaret Martonosi. 2019. Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1015–1029.

[30] Giacomo Naminicini, Lev S Bishop, Oktay Gündük, and Petar Jurcevic. 2022. Optimal qubit assignment and routing via integer programming. *ACM Transactions on Quantum Computing* 4, 1 (2022), 1–31.

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

[32] Alexandru Paler, Austin G Fowler, and Robert Wille. 2017. Online scheduled execution of quantum circuits protected by surface codes. *arXiv preprint arXiv:1711.01385* (2017).

[33] Alexandru Paler, Alwin Zulehner, and Robert Wille. 2021. NISQ circuit compilation is the travelling salesman problem on a torus. *Quantum Science and Technology* 6, 2 (2021), 025016.

[34] Warren B Powell. 2009. What you should know about approximate dynamic programming. *Naval Research Logistics (NRL)* 56, 3 (2009), 239–249.

[35] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (2018), 79.

[36] Krishanu Sankar, Artur Scherer, Satoshi Kako, Sam Reifenstein, Navid Ghadermarzy, Willem B. Krayenhoff, Yoshitaka Inui, Edwin Ng, Tatsuhiro Onodera, Pooyu Ronagh, and Yoshihisa Yamamoto. 2021. Benchmark Study of Quantum Algorithms for Combinatorial Optimization: Unitary versus Dissipative. *arXiv:2105.03528* [quant-ph].

[37] Eddie Schoute. 2020. *Circuit Transformations for Quantum Architectures*. Retrieved May 18, 2021 from *https://gitlab.umiacs.umd.edu/amchilds/arct*.

[38] Alireza Shaafaei, Melodi Saeedi, and Massoud Pedram. 2014. Qubit placement to minimize communication overhead in 2D quantum architectures. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 495–500.

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

[40] Marcos Yukio Siraichi, Vinícios Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintão Pereira. 2018. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 113–125.

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

[42] Bochen Tan and Jason Cong. 2021. Optimal qubit mapping with simultaneous gate absorption. *arXiv preprint arXiv:2109.06445* (2021).

[43] Arianne Meijer van de Griend and Sarah Meng Li. 2022. Dynamic qubit allocation and routing for constrained topologies by CNOT circuit re-synthesis. *arXiv preprint*
[44] LMK Vandersypen, H Bluhm, JS Clarke, AS Dzurak, R Ishihara, A Morello, DJ Reilly, LR Schreiber, and M Veldhorst. 2017. Interfacing spin qubits in quantum dots and donors—hot, dense, and coherent. *npj Quantum Information* 3, 1 (2017), 1–10.

[45] Davide Venturelli, Minh Do, Bryan O’Gorman, Jeremy Frank, Eleanor Rieffel, Kyle EC Booth, Thanh Nguyen, Parvathi Narayan, and Sasha Nanda. 2019. Quantum circuit compilation: An emerging application for automated reasoning. In *Proc. Schedul. Plan. Appl. Workshop*.

[46] Friedrich Wagner, Andreas Bärmann, Frauke LIers, and Markus Weissenbäck. 2022. Improving Quantum Computation by Optimized Qubit Routing. *arXiv preprint arXiv:2206.01294* (2022).

[47] Mathias Weiden, Justin Kalloor, John Kubiatowicz, Ed Younis, and Costin Iancu. 2022. Wide Quantum Circuit Optimization with Topology Aware Synthesis. *arXiv preprint arXiv:2206.13645* (2022).

[48] Robert Wille, Aaron Lye, and Rolf Drechsler. 2014. Optimal SWAP gate insertion for nearest neighbor quantum circuits. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 489–494.

[49] Xin-Chuan Wu, Marc Grau Davis, Frederic T Chong, and Costin Iancu. 2020. QGo: Scalable quantum circuit optimization using automated synthesis. *arXiv preprint arXiv:2012.09835* (2020).

[50] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. 2022. Synthesizing Quantum-Circuit Optimizers. *arXiv preprint arXiv:2211.09691* (2022).

[51] Xiangzhen Zhou, Yuan Feng, and Sanjiang Li. 2020. A Monte carlo tree search framework for quantum circuit transformation. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–7.

[52] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2018. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 7 (2018), 1226–1236.

[53] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2019. Mapping of OPENQASM programs to IBM QX satisfying the architectural constraints. Retrieved March 08, 2021 from [http://github.com/iic-jku/ibm_qx_mapping/tree/master/examples](http://github.com/iic-jku/ibm_qx_mapping/tree/master/examples).
APPENDIX

Appendix A: Hyperparameter Selection

We have introduced several hyperparameters in the placement and routing algorithm, and have studied their effects on the performance of our transpiler. The hyperparameters we have used are as follows:

- **Discount factor** $\delta$: This discount factor is used in the definitions of the edge scores $\Omega$ and placement scores $\Sigma$. The closer the discount factor is to 1, the more significant is the contribution of future gates in calculating the scores.

- **Decision horizon** $\lambda$: This hyperparameter determines the number of gates that are to be included in the calculations of the edge scores $\Omega$ and the placement scores $\Sigma$. A higher value of $\lambda$ indicates that a greater number of gates will be included in the score calculations.

- **Lookahead depth** $D$: This hyperparameter determines how many future actions (i.e., Swap gate insertions) the tie-breaking subroutine looks ahead from, within the decision horizon.

To understand how the hyperparameters affect Swap count, we generate six families of circuits: adder, amplitude estimation, Grover’s search, phase estimation, quantum Fourier transform (QFT), and random. All circuits follow Qiskit’s implementation \[15\].

We first vary the value of $\delta$ and fix the decision horizon at $\lambda = 100$ and lookahead depth at $D = 9$. From Fig. 7, higher values of $\delta$ in general result in lower Swap counts. We refer to the $\delta$ values of 0.7, 0.9, and 1 as large $\delta$ values. For adder circuits with 4, 8, 12, and 16 qubits, we attain the lowest Swap counts using large $\delta$, while for the adder circuit with 20 qubits, a value of $\delta = 0.1$ results in the lowest Swap count, followed closely by $\delta = 0.9$. In other words, using a large $\delta$ value results in the lowest Swap count in 80% of the test cases for adder circuits. For the amplitude estimation, Grover’s search, phase estimation, QFT, and random circuits, using a large $\delta$ value, we attain a lowest Swap count of $81.25\%$, $63.63\%$, $80\%$, $70.59\%$, and $55.56\%$ respectively.

Figure 8 shows that the value of $\lambda$ does not affect the Swap count in the case of the adder, amplitude estimation, QFT, and random circuits. As for the Grover’s search and phase estimation circuits, we observe that a higher value of $\lambda$ usually results in a lower Swap count, except for the Grover’s search circuit with 14 qubits, and the phase estimation circuits with 7, 8, 10, 14, and 15 qubits. However, even for circuits that do not result in the lowest Swap count using high values of $\lambda$, we observe that the difference in Swap count is not significant. Therefore, we conclude that having a higher value of $\lambda$ is in general beneficial.
FIG. 7. Varying the discount factor $\delta$ with the depth $D = 9$ and the decision horizon $\lambda = 100$. The SWAP count is normalized by the number of gates in the circuits. We offset the points representing the normalized SWAP values in the $x$ direction for visual clarity.
FIG. 8. Varying the decision horizon $\lambda$ with the depth $D = 9$ and the discount factor $\delta = 0.9$. The SWAP count is normalized by the number of gates in the circuits. We offset the points representing the normalized SWAP values in the $x$ direction for visual clarity.
FIG. 9. Varying the decision horizon $\lambda$ with the discount factor $\delta = 0.9$ and the lookahead depth $D = 0.5\lambda$. The total number of gates in a test circuit is denoted by $N_G$. The Swap count is normalized by the number of gates in the circuits. We offset the points representing the normalized Swap values in the x direction for visual clarity.
Appendix B: Detailed Benchmarking Results

Table IV shows the detailed benchmarking results for tket, SABRE, and Nuwa transpiling input circuits onto the coupling graphs of two target architectures: the IBM Q20 Tokyo chip and a 4 x 5 grid. In order to study the routing algorithms in isolation, we use a naïve fixed placement (i.e., matching the vertex and qubit numbers) across all experiments. We provide a visualization of these results in Fig. 10. The benchmarking circuit instances may be found in [53].

| Input Circuit Name | No. of Qubits | No. of Gates | IBM Q20 Tokyo Graph | 4 x 5 Grid |
|--------------------|---------------|--------------|---------------------|------------|
| xor5_254           | 9             | 7            | 1*                  | 5          | 3          | 2*        | 5          | 3          |
| graycode6_47       | 6             | 5            | 1*                  | 2          | 6          | 2*        | 3          | 5          |
| ex1_226            | 6             | 7            | 1*                  | 2          | 3          | 2*        | 4          | 3          |
| 4gt11_84           | 4             | 18           | 3                    | 7          | 2*        | 3          | 7          | 2*        |
| 4mod5-v0_20        | 5             | 20           | 3*                  | 4          | 3*        | 3*        | 4          | 5          |
| ex1_166            | 3             | 19           | 3*                  | 6          | 3*        | 3*        | 6          | 3*        |
| 4mod5-v1_22        | 5             | 21           | 4                    | 7          | 1*        | 4          | 7          | 3*        |
| mod5d1_63          | 5             | 22           | 1*                  | 12         | 4          | 5          | 13         | 4*        |
| ham3_102           | 5             | 30           | 3                    | 6          | 2*        | 3          | 6          | 2*        |
| 4gt11_83           | 5             | 23           | 4*                  | 10         | 4*        | 4*        | 7          | 4*        |
| 4gt11_82           | 5             | 27           | 5*                  | 11         | 5*        | 6          | 10         | 4*        |
| rd32-v0_66         | 4             | 34           | 4                    | 6          | 1*        | 6*        | 14         | 6*        |
| alu-v0_27          | 5             | 36           | 3*                  | 13         | 4          | 7          | 12         | 6*        |
| 4mod5-v1_24        | 5             | 36           | 8                    | 12         | 6*        | 8          | 14         | 7*        |
| 4mod5-v0_19        | 5             | 35           | 3*                  | 16         | 5          | 7*        | 16         | 10        |
| mod5mils_65        | 5             | 35           | 3*                  | 17         | 5          | 6*        | 17         | 10        |
| rd32-v1_68         | 4             | 36           | 6                    | 14         | 1*        | 6*        | 14         | 6*        |
| alu-v1_28          | 5             | 37           | 3*                  | 10         | 3*        | 6*        | 10         | 8          |
| alu-v2_33          | 5             | 37           | 3*                  | 14         | 4          | 7          | 17         | 6*        |
| alu-v4_37          | 5             | 37           | 3*                  | 10         | 4          | 7          | 12         | 6*        |
| 3_17_13            | 3             | 36           | 3*                  | 10         | 4          | 6*        | 11         | 5*        |
| alu-v1_29          | 5             | 37           | 4*                  | 10         | 4*        | 8          | 10         | 6*        |
| miller_11          | 3             | 50           | 9                    | 19         | 0*        | 9*        | 19         | 9*        |
| alu-v3_34          | 5             | 52           | 6                    | 20         | 3*        | 10*       | 25         | 12         |
| decod24-v2_43      | 4             | 52           | 9                    | 20         | 2*        | 9          | 20         | 8*        |
| decod24-v0_38      | 4             | 51           | 6*                  | 17         | 7          | 9          | 18         | 8*        |
| mod5d2_64          | 5             | 53           | 6*                  | 26         | 8          | 11*       | 25         | 12         |
| 4gt13_92           | 5             | 66           | 9*                  | 21         | 9*        | 12         | 29         | 11*       |
| 4gt13-v1_93        | 5             | 68           | 6*                  | 24         | 7          | 12         | 27         | 9*        |
| 4mod5-v0_18        | 5             | 69           | 4*                  | 31         | 9          | 12*       | 31         | 14        |
| decod24-bdd_294    | 6             | 73           | 9*                  | 20         | 13         | 13*       | 23         | 14        |
| one-two_-v2_100    | 5             | 69           | 7                    | 22         | 4*        | 14         | 23         | 11*       |
| one-two_-v3_101    | 5             | 70           | 7*                  | 21         | 8          | 13*       | 21         | 14        |
| 4mod5-v1_23        | 5             | 69           | 4*                  | 35         | 9          | 14         | 33         | 12*       |
| 4mod5-bdd_287      | 7             | 70           | 7*                  | 13         | 8          | 14*       | 24         | 15        |
| rd32_270           | 5             | 84           | 6*                  | 34         | 11         | 16*       | 34         | 16*       |
| 4gt5_75            | 5             | 83           | 9*                  | 34         | 11         | 16         | 34         | 12*       |
| alu-bdd_288        | 7             | 84           | 15                   | 11         | 9*        | 16*       | 17         | 17        |
| alu-v0_26          | 5             | 84           | 10                   | 34         | 9*        | 16         | 34         | 14*       |
| decod24-v1_41      | 5             | 85           | 14                   | 26         | 6*        | 17         | 34         | 16*       |
| rd53_138           | 8             | 132          | 11*                 | 23         | 12         | 25*       | 38         | 31        |
| 4gt5_76            | 5             | 91           | 14                   | 28         | 12*       | 17*       | 40         | 17*       |
| 4gt13_91           | 5             | 103          | 13                   | 39         | 12*       | 19*       | 38         | 21        |
| cnt3-3_179         | 16            | 175          | 31                   | 26         | 24*       | 52         | 38*        | 40        |
| qft_10             | 10            | 200          | 16*                 | 57         | 22         | 24*       | 60         | 25        |
| 4gt13_90           | 5             | 107          | 15                   | 43         | 12*       | 20*       | 40         | 21        |
| alu-v4_36          | 5             | 115          | 13*                 | 41         | 13*       | 20         | 43         | 18*       |
| mini_alu_305       | 10            | 173          | 27                   | 41         | 22*       | 39         | 46         | 34*       |
| ising_model_10     | 10            | 480          | 9                    | 66         | 4*        | 15         | 30         | 11*       |
| ising_model_16     | 16            | 786          | 11*                 | 76         | 17        | 46         | 75         | 16*       |
| ising_model_13     | 13            | 633          | 9*                   | 43         | 15         | 26*       | 82         | 27        |
| 4gt5_77            | 5             | 131          | 18                   | 51         | 13*       | 22*       | 51         | 24        |
| sys6-v0_111        | 10            | 215          | 22                   | 44         | 21*       | 50         | 54         | 45*       |
5 132 15 59 11* 27* 58 28
5 146 16 47 11* 25 64 21*
5 150 19 42 15* 25* 57 27
5 148 24 40 11* 27* 62 29
5 151 27 45 14* 28 55 26*
5 162 45 19 30 63 27*
5 163 23 57 14* 29* 64 30
10 230 19* 57 24 51 57 49*
5 164 24 45 10* 31 64 28*
6 179 11* 36 17 34 42 33*
5 178 26 58 18* 33 75 28*
14 211 33* 50 36 58 55* 56
16 512 55 137 40* 79 142 55*
6 194 31 41 22* 45 46 33*
15 343 37 57 36* 74 62* 72
13 275 51 56 26* 64 94 62*
5 217 33 76 21* 44 87 36*
12 328 30* 59 31 70 100 60*
6 228 26 76 14* 41 64 40*
6 247 21* 31 28 45 73 41*
6 231 15* 35 30 43 67 39*
5 233 22* 94 24 41 94 37*
14 270 43 62 41* 59* 79 67
6 251 22* 31 30 46 76 41*
6 258 28 45 27* 53 89 44*
6 235 17* 36 32 44 68 40*
5 244 33 81 25* 44 89 43*
6 273 37 61 27* 64 87 45*
7 296 23* 76 23* 59 88 56*
5 288 38 96 35* 53 115 49*
5 290 41 122 36* 56 121 48*
7 320 43 61 17* 62 82 58*
6 338 35 75 20* 62 124 58*
6 342 29 71 23* 78 105 59*
16 485 83 127 32* 107 130 104*
6 403 52 81 39* 92 115 64*
6 395 43 89 25* 79 145 72*
6 440 44 97 35* 86 115 72*
7 467 70 134 25* 89* 167 91
5 451 42* 141 42* 90 196 76*
7 469 46 94 33* 94 147 88*
7 504 59 104 39* 96 181 93*
6 555 83 102 49* 121 184 96*
7 580 108 195 26* 121 156 98*
8 650 69 160 29* 120* 180 125
7 612 75 176 42* 121 192 105*
7 631 89 194 31* 119 220 101*
6 778 76 127 54* 147 262 127*
6 781 84 209 76* 172 201 137*
9 954 178 227 40* 193 307 170*
11 986 125 224 78* 227 307 184*
7 1043 106 219 72* 239 290 184*
8 1206 194 343 73* 252 343 217*
12 1221 202 260 79* 273 354 213*
8 1291 172 205 100* 278 406 233*
6 1336 153 328 110* 254 397 245*
14 1776 236 412 92* 374 489 305*
14 1776 236 285 92* 374 558 305*
11 1914 278 309 92* 396 610 338*
13 1993 339 392 128* 445 546 355*
11 3073 369 668 216* 750 874 537*
12 3009 476 552 252* 725 881 575*
13 3213 468 635 205* 708 939 608*
13 3439 495 641 224* 719 936 638*
| Circuit       | Swap Count | IBM Q20 Tokyo Coupling Graph | 4×5 Target Architecture |
|--------------|------------|------------------------------|-------------------------|
| sym6_145     | 7          | 3888                         | 1043                    |
| misex1_241   | 15         | 4813                         | 938                     |
| rd73_252     | 10         | 5321                         | 1043                    |
| cycle10_2_110| 12         | 6050                         | 910                     |
| hwb6_56      | 7          | 6723                         | 956                     |
| square_root_7| 15         | 7650                         | 1001                    |
| ham15_107    | 15         | 8763                         | 1163                    |
| dc2_222      | 15         | 9462                         | 1200                    |
| sqc_258      | 10         | 10223                        | 1100                    |
| inc_237      | 16         | 10619                        | 1232                    |
| cm5a_209     | 14         | 11414                        | 1720                    |
| rd84_235     | 12         | 13658                        | 1914                    |
| co14_215     | 15         | 17936                        | 2583                    |
| root_255     | 13         | 17159                        | 2547                    |
| mlp4_245     | 16         | 18852                        | 2168                    |
| urf2_277     | 8          | 20112                        | 3212                    |
| sym9_148     | 10         | 21504                        | 2852                    |
| life_238     | 11         | 22445                        | 3266                    |
| hwb7_59      | 8          | 24379                        | 2956                    |
| max46_240    | 10         | 27126                        | 3697                    |
| clip_206     | 14         | 33827                        | 4505                    |
| 9symm1_105   | 11         | 34881                        | 5232                    |
| sym9_133     | 11         | 34881                        | 5232                    |
| sao2_257     | 14         | 38577                        | 4312                    |
| dist_223     | 13         | 39806                        | 4707                    |
| urf5_280     | 9          | 40829                        | 6520                    |
| urf1_278     | 9          | 54766                        | 10646                   |
| sym10_262    | 12         | 64283                        | 9810                    |
| hwb8_113     | 9          | 69380                        | 11874                   |
| urf5_152     | 8          | 80480                        | 7748                    |
| urf3_279     | 10         | 125382                       | 19498                   |
| plus63mod...163| 13   | 128744                       | 20470                   |
| urf5_158     | 9          | 164416                       | 22682                   |
| urf6_160     | 15         | 171840                       | 22359                   |
| urf1_149     | 9          | 184864                       | 21828                   |
| plus63mod...164| 14   | 187112                       | 24956                   |
| hwb9_119     | 10         | 207775                       | 27687                   |
| urf3_155     | 10         | 423488                       | 51978                   |
| ground...10  | 13         | 390180                       | 15612                   |
| urf4_187     | 11         | 512064                       | 86174                   |

Number of Lowest Swap Counts: 44, 123

**TABLE IV:** Swap counts for tket, SABRE, and Nuwa. The lowest swap count achieved for each circuit is shown in bold text with an asterisk. Columns 4 to 6 show the results for the IBM Q20 Tokyo coupling graph represented in Fig. 1(b) and columns 7 to 9 show tests performed on a 4×5 target architecture.
FIG. 10. A plot corresponding to Table IV. (a) Swap counts of Nuwa vs. SABRE on the IBM Q20. (b) Swap counts of Nuwa vs. tket on the IBM Q20. (c) Swap counts of Nuwa vs. SABRE on a 4 x 5 grid. (d) Swap counts of Nuwa vs. tket on a 4 x 5 grid. The colours and sizes of the dots correspond to the number of gates in the circuit.