“© 2020 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.”

Quantum Circuit Transformation Based on Simulated Annealing and Heuristic Search

Xiangzhen Zhou, Sanjiang Li, and Yuan Feng

Abstract—**Quantum algorithm design usually assumes access to a perfect quantum computer with ideal properties like full connectivity, noise-freedom and arbitrarily long coherence time. In Noisy Intermediate-Scale Quantum (NISQ) devices, however, the number of qubits is highly limited and quantum operation error and qubit coherence are not negligible. Besides, the connectivity of physical qubits in a quantum processing unit (QPU) is also strictly constrained. Thereby, additional operations like SWAP gates have to be inserted to satisfy this constraint while preserving the functionality of the original circuit. This process is known as quantum circuit transformation. Adding additional gates will increase both the size and depth of a quantum circuit and therefore cause further decay of the performance of a quantum circuit. Thus it is crucial to minimize the number of added gates. In this paper, we propose an efficient method to solve this problem. We first choose by using simulated annealing an initial mapping which fits well with the input circuit and then, with the help of a heuristic cost function, stepwise apply the best selected SWAP gates until all quantum gates in the circuit can be executed. Our algorithm runs in time polynomial in all parameters including the size and the qubit number of the input circuit, and the qubit number in the QPU. Its space complexity is quadratic to the number of edges in the QPU. Experimental results on extensive realistic circuits confirm that the proposed method is efficient and the number of added gates of our algorithm is, on average, only 57% of that of state-of-the-art algorithms on IBM Q20 (Tokyo), the most recent IBM quantum device.**

Index Terms—**NISQ, quantum circuit transformation, qubit mapping, qubit allocation, qubit routing, quantum processing unit**

I. Introduction

IN Noisy Intermediate-Scale Quantum (NISQ) era, it is unrealistic to implement quantum error correction due to the strictly limited number of qubits [1]. This drawback brings huge challenge to quantum program compilation because the noise will have large impact on final circuits and may often make the results meaningless. Besides, the connectivity of qubits in an NISQ device is also limited. Only those neighbouring qubits can be coupled and only between them can two-qubit operations be implemented [2]. As a result, a large number of modifications must be done to adapt a quantum circuit to the real quantum devices. This process is termed as quantum circuit transformation [3], qubit mapping [4], qubit allocation [5], qubit routing [6] or qubit movement [7] in the literature. We call it quantum circuit transformation in this paper.

Quantum circuit transformation is an essential part for quantum circuit compilation. The main idea behind is to convert an ideal quantum circuit, in which full connectivity among qubits is assumed and noise is ignored, to a quantum circuit respecting constraints imposed by the NISQ devices [3]. Usually this process will bring in a large number of auxiliary gates like SWAP gates and Hadamard gates which will in turn increase both the size and depth of the generated quantum circuit and sometimes make the error of the whole circuit unacceptable [2]. Hence, it is vital for the success of quantum computation to find an automated approach that can efficiently transform any input quantum circuit into one that respects the physical constraints imposed by the NISQ devices with a small overhead in terms of the size, depth or error. The aim of this paper is to provide an efficient method to reduce the number of added gates required for quantum circuit transformation. Interested readers are referred to [8] and [9], [7], [10] for works aiming to minimize depth and, respectively, error.

The quantum circuit transformation problem can be reduced to token swapping or template matching in graph theory [11], [12]. Unfortunately, both of these problems are NP-complete [3]. Hence, designing algorithms to solve the quantum circuit transformation problem while making trade off between time consuming and the quality of results has brought lots of interest in both the quantum computing community and the integrated circuits community [2].

There are currently three major approaches to the quantum circuit transformation problem. The first one is to use heuristic search to construct the output quantum circuit step by step from the original input quantum circuit [4], [6], [13], [5], [14]. Usually, these search algorithms need an initial mapping as the input, and it can be set arbitrarily or via some greedy methods [13], [6], [15]. Recently, a novel reverse traversal technique is proposed in [4] to choose the initial mapping with the consideration of the whole circuit. The second approach is to utilize unitary matrix decomposition algorithms to reconstruct a quantum circuit from scratch while preserving the functionality of the input circuit [16], [17]. The third one is to convert the quantum circuit transformation problem to some existing problems like AI planning [18], [19], Integer Linear Programming (ILP) [20], and Satisfiability Modulo Theory (SMT) [21], [7] and use ready-made tools for these problems to find acceptable results.

Xiangzhen Zhou is with State Key Lab of Millimeter Waves, Southeast University, Nanjing 211189, China and Centre for Quantum Software and Information, Faculty of Engineering and Information Technology, University of Technology Sydney, NSW 2007, Australia.

Sanjiang Li is with Centre for Quantum Software and Information, Faculty of Engineering and Information Technology, University of Technology Sydney, NSW 2007, Australia (e-mail: sanjiang.li@uts.edu.au).

Yuan Feng is with Centre for Quantum Software and Information, Faculty of Engineering and Information Technology, University of Technology Sydney, NSW 2007, Australia (e-mail: yuan.feng@uts.edu.au).

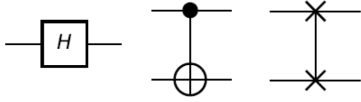

Fig. 1. Hadamard, CNOT and SWAP gate (from left to right).

In this paper, we follows the first approach. Our main contributions are listed as follows. First, we propose a simulated annealing based algorithm to find a near-optimal initial mapping for the input circuit. Second, we design a flexible heuristic cost function to evaluate the possible operations that may be applied to transform the current circuit. The heuristic function supports weight parameters to reflect the variable influence of gates in different layers. Third, a heuristic search algorithm with a novel selection mechanism is designed, where in each step of the search process, instead of selecting the operation with minimum cost to apply, we look one step ahead and select the operation which has the best consecutive operation to apply. In this way, the algorithm is able to avoid the local minimum effectively. Fourth, a pruning mechanism is introduced to reduce the size of search space and ensure the program terminates in reasonable time.

Note that the look-ahead mechanism has already been introduced in the heuristic cost function during the search process in existing works like [13], [4]. In [15], Cowtan et al. introduced another look-ahead mechanism for selecting the best SWAP for transforming CNOT gates in the current front layer. In this paper, we adopt a 'double' look-ahead mechanism: in addition to looking ahead at subsequent layers when defining the cost function, we also look ahead (at grandchild states) in finding the state with minimal cost in order to make the best transformation. Thanks to this novel idea, the proposed algorithm is able to find a better solution with less circuit size within acceptable running time. Experimental results on extensive realistic circuits show that our algorithm is efficient and the number of added gates of our algorithm on IBM Q20 (IBM QX5, resp.) is, on average, only 43% and 57% (87% and 90%, resp.) of that of the state-of-the-art algorithms in [4] and [15] ([13] and [15], resp.)

The remainder of this paper is organized as follows: some background knowledge about quantum computation is given in Section II, and the quantum circuit transformation problem is formally defined in Section III. Section IV is devoted to the detailed description of our proposed algorithm. We report experimental results in Section V and conclude the paper in Section VI.

II. Background

In classical computation, information is stored in memory in the form of binary digits, i.e., bits. The quantum counterpart of bit, called qubit, has two basis states denoted by $|0\rangle$ and $|1\rangle$, respectively. Different from a classical bit, a qubit $|\psi\rangle$ can be in a linear combination of basis states [22], i.e.,

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{1}$$

where $|\alpha|^2 + |\beta|^2 = 1$. Information processing or computation is realized by applying quantum gates on qubits. Typical gates

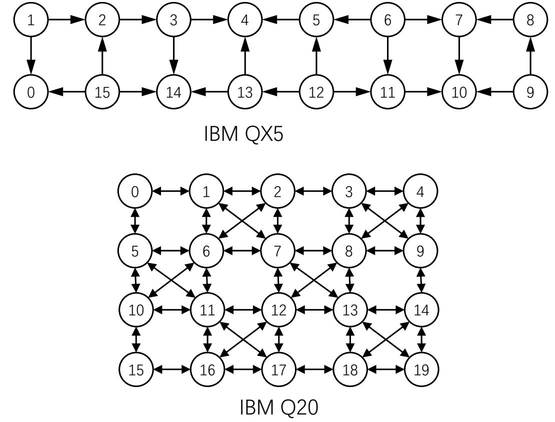

Fig. 2. Two architecture graphs for IBM QX architecture.

which we are concerned with in this paper are Hadamard gate H, CNOT gate and SWAP gate, depicted in Fig. 1. H is a single-qubit gate which can evenly mix the basis states to produce a superposed one. CNOT and SWAP are both two-qubit gates, i.e., they operate on two qubits. A CNOT gate flips the target qubit (indicated graphically with $\oplus$) if and only if the control qubit (indicated graphically with a black dot $\bullet$) is in state $|1\rangle$, while a SWAP gate exchanges the states of the two qubits operated.

Quantum circuits are the most commonly used model to describe quantum algorithms, which consist of input qubits, quantum gates, measurements and classical registers [23]. However, as far as quantum circuit transformation is concerned, only input qubits and quantum gates are relevant. Thus in this paper, a quantum circuit is simply represented as a pair (Q, C), where Q is the set of involved qubits and C a sequence of quantum gates. For a generic quantum circuit to be executed in a real quantum processing unit (QPU), two more steps have to be taken:

- *Compilation* process. As only limited quantum operations are available in a QPU, quantum gates in the circuit must be decomposed into elementary gates first [24], [25]. In this paper, we take single-qubit and CNOT gates as elementary gates as they are universal to implement any quantum circuit and supported by, say, IBM QX architectures.
- *Transformation* process. Qubits in a real QPU are typically laid out in a fixed topology and CNOT gates can only be applied on neighbouring qubits. Such a connectivity topology can be described by an *architecture graph* or *coupling graph* [3] which is a directed graph with each node representing a qubit in the QPU. A quantum circuit consisting of only single-qubit and CNOT gates is said to *respect* the QPU constraint if for every CNOT gate in the circuit, there is a directed edge in the architecture graph from the control qubit to the target qubit. The transformation process is then to convert a quantum circuit (say, those obtained from the above compilation process) into one that respects the QPU constraint so that it can be executed on the QPU.

In this paper, we only focus on the transformation process.

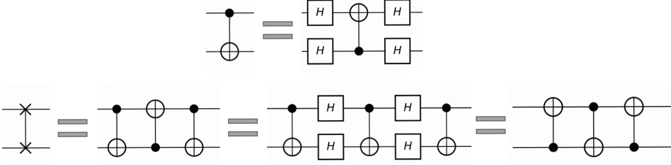

Fig. 3. Some gate decomposition and transformation rules.

The QPU topologies we are concerned with are IBM QX architectures QX5 and Q20 shown in Fig. 2, but our approach is applicable to any architecture graph, including for example Rigetti 16Q Aspen-4[1]. Notice that edges in IBM Q20 are bidirectional (or, undirected) and thus either node of each edge can be the control qubit of a CNOT gate. Depicted in Fig. 3 are several gate transformation rules which are quite useful in gate decomposition and circuit transformation. The top equivalence shows that we can exchange the control and target qubits of a CNOT by adding two Hadamard gates before and after it, while the bottom ones show different ways of implementing a SWAP gate in QX structures.

To simplify the presentation, we distinguish between two kinds of quantum circuits in this paper. *Logical* circuits are ideal and high-level gate descriptions of quantum algorithms without considering any physical constraints imposed by QPUs. In contrast, *physical* quantum circuits are low-level gate-model implementation which respect the QPU concerned. The purpose of the circuit transformation process mentioned above is then to convert a logical circuit to a physical one. Accordingly, qubits appearing in logical circuits are called logical qubits while those appearing in physical circuits are called physical ones. We note here that the terms logical circuits and logical qubits are also used in a different area, namely, quantum error correction [26].

III. Quantum Circuit Transformation

The main objective of quantum circuit transformation is to transform an input logical circuit to a physical one so that the constraints imposed by the QPU are satisfied. To simplify the problem, we only consider the connectivity constraints for CNOT gates as specified by the architecture graph (see Section II). This means that single-qubit gates have no effect in the circuit transformation process, and we assume without loss of generality that the input logical circuit consists only of CNOT gates. Furthermore, a CNOT gate is simply denoted as a pair $\langle q, q' \rangle$, where q is the control qubit and q' is the target qubit. We call the CNOT gate $\langle q', q \rangle$ the *inverse* of $\langle q, q' \rangle$.

Let $AG = (V, E)$ be the architecture graph of a QPU, where V is the set of physical qubits and E the set of directed edges along which CNOT gates can be performed. Given a logical circuit $LC = (Q, C^l)$ with $|Q| \leq |V|$, we need to construct a physical circuit $PC = (V, C^p)$ such that

- LC and PC are equivalent in functionality.
- C^p only contains CNOT gates and single qubit gates.
- For any CNOT gate $\langle q, q' \rangle$ in C^p, $(q, q') \in E$.

[1]https://www.rigetti.com/qpu

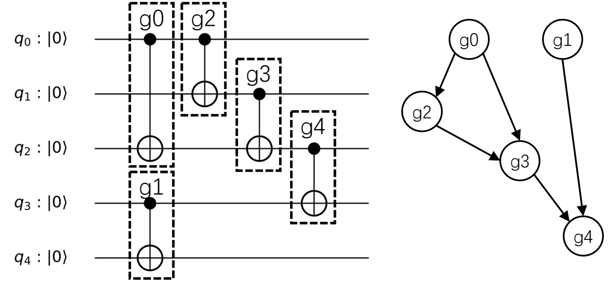

Fig. 4. On the left is an example for logical quantum circuit with only CNOT gates and right DAG representing dependency order of the left circuit.

It is easy to find a physical circuit that satisfies the above conditions, but the real challenge is to find one with *minimal* size or depth, which is NP-complete in general [27], [5]. In this paper, we modify the input logical circuit stepwise by inserting auxiliary gates like CNOT and H, as shown in Fig. 1, until the logical circuit is transformed into a physical circuit that can be executed on the QPU. To evaluate the effectiveness of quantum circuit transformation algorithms, we use the sizes of the output circuits, i.e., the total number of its elementary gates.

A. Dependency Graph

CNOT gates in a logic circuit $LC = (Q, C^l)$ are not independent. We say a CNOT gate $\langle q, q' \rangle$ *depends* on another $\langle p, p' \rangle$ if the latter must be executed before the former. This happens when $\langle p, p' \rangle$ is in front of $\langle q, q' \rangle$ in C^l and they share a common qubit, or when $\langle q, q' \rangle$ depends on a CNOT gate which depends on $\langle p, p' \rangle$.

In general, we can construct a directed acyclic graph (DAG), called the *dependency graph* [28], to characterize the dependency between gates in a logical circuit LC [4]. Each node of the dependency graph represents a gate and each directed edge the dependency relationship from one gate to another. The front layer of LC, denoted $\mathcal{F}(LC)$ or $\mathcal{L}_0(LC)$, consists of all gates in LC which have no parents in the dependency graph. The second layer $\mathcal{L}_1(LC)$ is then the front layer of the circuit obtained from LC by deleting all gates in $\mathcal{F}(LC)$. Analogously, we can define the k-th layer $\mathcal{L}_k(LC)$ of LC for all $k \geq 0$. Consider the circuit shown in Fig. 4 as an example. Initially, gates g_0 and g_1 can be applied in parallel because there are no gates before them and they are independent from each other. Thus $\mathcal{F}(LC) = \{g_0, g_1\}$. Then, gate g_2 can be executed after g_0, g_3 after g_2 and g_0, and g_4 after g_3 and g_1. Thus $\mathcal{L}_1(LC) = \{g_2\}$, $\mathcal{L}_2(LC) = \{g_3\}$, and $\mathcal{L}_3(LC) = \{g_4\}$.

B. Qubit Mapping

At each step of the circuit transformation, qubits in the logical circuit are mapped or allocated to physical qubits in the QPU [5]. Mathematically, a qubit mapping is a function τ from Q to V such that $\tau(q) = \tau(q')$ if and only if $q = q'$ [3]. The mapping may change at consecutive steps of the transformation which is determined by the inserted auxiliary gates.

Given a logical circuit LC and a mapping τ, a CNOT gate $g = \langle q, q' \rangle$ in LC is said to be *satisfied* by τ, or τ satisfies g, if $(\tau(q), \tau(q'))$ is a directed edge in AG. Furthermore, g is *executable* by τ if it appears in the front layer of LC and is satisfied by τ. In this case, we remove it from LC and append a CNOT gate $\tau(g) := \langle \tau(q), \tau(q') \rangle$ to the end of the physical circuit. This process is called the *execution* of g.

IV. The Proposed Algorithm

In this section, details of the proposed algorithm will be explained step by step. Let $AG = (V, E)$ be the architecture graph and $LC = (Q, C^l)$ the input logic circuit consisting only CNOT gates. The goal of the algorithm is to try to minimize the size, i.e., the total number of elementary gates, of the output physical circuit.

We first generate an initial mapping τ_{ini} by using simulated annealing (Algorithm 1), and then stepwise construct the output physical circuit by adding auxiliary CNOT or Hadamard gates while processing gates in the input logic circuit. The *state* of each step is described by a mapping τ' from logic qubits in Q to physical qubits in V, the currently constructed physical circuit PC' which obeys the constraints imposed by AG, and the logic circuit LC' with gates that have not been processed. A cost function which assigns decreasing weights to gates in later layers is used to select the state of the next step. Note that the above procedure is standard for circuit transformation, and has been adopted in [13]. Our algorithm distinguishes itself from the previous ones in the ways of choosing the initial mapping (Sec IV-B), the definition of the cost function, and the strategy of updating step states (Sec IV-C).

A. CNOT Distance

In graph theory, the distance from a source node v to a destination node v' in a directed graph G, written $\mathsf{dist}_G(v, v')$, is the minimal number of edges needed to traverse from v to v'. Suppose AG is the architecture graph of the QPU we consider. We define the *CNOT distance* from v to v' in AG, written $\mathsf{dist}_{cnot}(v, v')$, as the minimal number of auxiliary CNOT and Hadamard gates required to execute the CNOT gate $\langle v, v' \rangle$ in the QPU. Here 'execute' is in the same sense as we have described in Section III-B. To execute $\langle v, v' \rangle$, we need to bring the two qubits v and v' close to each other by swapping and then, when they are neighbours in AG, we further check if the direction is from v to v' or vice versa.

For bi-directed (or, undirected) architecture graph such as that of Q20, we need only to bring v close to v' or vice versa, and the CNOT distance is simply computed as $\mathsf{dist}_{cnot}(v, v') = 3 \times (\mathsf{dist}_{AG}(v, v') - 1)$. This is because only $\mathsf{dist}_{AG}(v, v') - 1$ swaps are required and each SWAP requires only 3 CNOT gates to implement (see Fig. 3 (top)). For directed architecture graph such as that of QX5, the situation is a little complicated, where we also need to consider the direction of the CNOT gates. We first compute all shortest paths from v to v' (ignoring the directions). Suppose $d = \mathsf{dist}_{AG}(v, v')$. If there is an undirected shortest path $\pi = \langle v_0 \equiv v, v_1, ..., v_d \equiv v' \rangle$ in which (v_i, v_{i+1}) is a directed edge in QX5 for some i, then the CNOT distance is computed as $\mathsf{dist}_{cnot}(v, v') = 7 \times (d - 1)$, because a SWAP gate is decomposed into 7 elementary gates (see Fig. 3 (bottom)). Otherwise, we have $\mathsf{dist}_{cnot}(v, v') = 7 \times (d - 1) + 4$, as we need to add 2 Hadamard gates before and after to change the direction of the target CNOT [13].

Take QX5 as an example. Suppose the logic qubits q and q' are mapped to v_3 and v_1, which correspond to nodes 3 and 1 in Fig. 2, respectively, and we want to implement the CNOT gate $\langle q, q' \rangle$, with q the control qubit and q' the target qubit. One solution is to add a SWAP gate between qubit v_1 and v_2 to bring q one step close to q', and a CNOT gate between v_2 and v_3 together with 4 additional Hadamard gates (cf. Figure 3) to change the direction of the CNOT gate. Because a SWAP gate can be decomposed into 7 elementary gates complying with the directions in QX5, the CNOT distance from v_3 to v_1 in QX5 is 11.

For simplicity, in out algorithm we precompute the CNOT distance for all node pairs in AG by using, say, a breadth-first search algorithm.

B. Initial Mapping

For several state-of-the-art algorithms [6], [4], [13], the selection of a good initial mapping has a significant impact on the quality of the final physical circuit. Motivated by this observation, our algorithm intends to find an initial mapping that 'fits' most gates such that fewer SWAP gates are required in the circuit transformation process.

To this end, we define the *gate cost* of a CNOT gate $g = \langle q, q' \rangle$ under a mapping $\tau : Q \to V$ as

$$\mathsf{cost}_{gate}(g, \tau) = \mathsf{dist}_{cnot}(\tau(q), \tau(q')). \tag{2}$$

Our ideal initial mapping τ^*_{ini} is then given by

$$\tau^*_{ini} = \arg\min_{\tau} \left\{ \sum_{g \in C^*} \mathsf{cost}_{gate}(g, \tau) \right\} \tag{3}$$

where C^* is a selected subset of the logical circuit LC. Here we use C^* instead of C^l to calculate the initial mapping. This is because taking into account all gates in C^l would bring further overhead and be unnecessary because gates in the tail of the circuit would have little impact on the initial mapping.

Simulated annealing (SA), inspired by the annealing process in metallurgy [29], is designed for approximating the global optimum of a given cost function. The algorithm tries to find the best state in the search space. In each trial for searching a better state, the algorithm generates a new state based on the previous one, calculates its cost and compares it with the previous one and decides whether this new state should be accepted. To escape from local optima, the algorithm accepts the new generated state with a certain probability even if its cost is worse than the previous one. The acceptance probability is decided by the current temperature which declines during the search process until the minimum value is reached.

We propose an efficient simulated annealing based algorithm (Algorithm 1) to find a good approximation of τ^*_{ini}, where $T_{\max}$, $T_{\min}$, Δ and R are, respectively, the starting

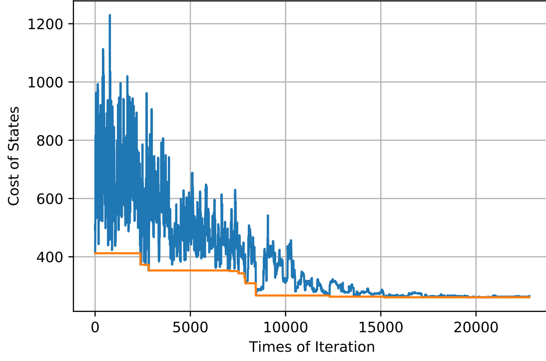

Fig. 5. Convergence of the simulated annealing algorithm on circuit adr4-197 and IBM Q20, where the blue and orange lines represent the cost of accepted states and existing best states, respectively. We set empirically $T_{\max} = 100$, $T_{\min} = 1$, $\Delta = 0.98$, $R = 100$ and C^* the first half of gates in the logical circuit.

temperature, the minimum temperature, the decline coefficient for the temperature and the repeated times for one temperature. Fig. 5 shows convergence of the simulated annealing process on a real quantum circuit adr4-197. Note that the cost of states converges after sufficient iterations, showing that the temperature is low enough. The fluctuation of the cost of accepted states is caused by the above mentioned acceptance probability for worse states.

In the following two subsections, we describe in detail how to generate all possible child and grandchild states and how the cost of a child or grandchild state is calculated. We will illustrate the construction by using the quantum circuit shown in Fig. 6 and the test architecture graph AG_{test} in Fig. 7.

Algorithm 1: Simulated annealing for computing the initial mapping

input : A set C^* for considered gates in a logical quantum circuit.
output: An approximation of the optimal initial mapping given in Eq.(3).

```
begin
    Initialize parameters T_max, T_min, Δ, R, and an arbitrary mapping τ;
    T ← T_max, bcost ← ∞, cost ← ∞;
    while T ≥ T_min do
        i ← 1;
        while i ≤ R do
            i ← i + 1;
            Change mapping τ randomly to generate a new mapping τ_new;
            ncost = Σ_{g∈C*} cost_gate(g, τ_new);
            if ncost < bcost then
                bcost ← ncost;
                τ_ini ← τ_new;
            end
            if ncost < cost then
                cost ← ncost;
                τ ← τ_new;
            else
                cost ← ncost and τ ← τ_new with prob. exp((cost−ncost)/T);
            end
        end
        T ← Δ × T;
    end
    return τ_ini
end
```

C. Heuristic search with look-ahead

We have shown in the previous section how to construct the initial mapping τ_{ini} for our circuit transformation algorithm, thus obtaining the state $s^0 := (\tau_{ini}, PC_0, LC_0)$ for the first step, where PC_0 and LC_0 are respectively the physical and logic circuits after executing all gates in LC which are executable in τ_{ini}. Suppose we are in state $s^i := (\tau_i, PC_i, LC_i)$ at the i-th step for $i \geq 0$. This section is devoted to the strategy of choosing s^{i+1} for the $i+1$-th step. Obviously, depending on the different ways of adding auxiliary CNOT and H gates, there are multiple child states of s^i to choose from. One natural way is to select the one with the minimal cost. This surely gives a fine method for extending s^i, but (as shown in Fig. 10) the sizes of the output physical circuits are not always desirable. In this paper, we propose a novel way to select the next state: we look one level ahead to calculate the costs of all *grandchild* states of s^i, and choose the child of s^i which has a child (thus a grandchild of s^i) with the minimum cost among all grandchildren of s^i.

To this end, we have to specify for a given state $s^i := (\tau_i, PC_i, LC_i)$, (1) how to extend s^i to get all its children and grandchildren, and (2) how to define the costs of its grandchildren. We are going to elaborate these two points one by one in the following.

Extend s^i. There are two natural ways to extend s^i.

- Way 1: Apply on τ_i a swap operation represented as an edge in AG one of whose end nodes is the image under τ_i of some qubit appearing in a gate in the front layer of LC_i, and obtain a new mapping τ_i'. Accordingly, we extend PC_i with the CNOT + H implementation of the SWAP gate corresponding to this swap operation. Then we execute recursively all gates in LC_i (not only those in the front layer, but also those executable when their precedents have already been executed by τ_i') which are executable in τ_i'. The resultant state is then a child of s^i.
- Way 2 only applies when AG is directed and there is a CNOT gate $\langle q, q' \rangle$ in the front layer of LC_i which is inversely executable, i.e. its inverse gate $\langle q', q \rangle$ is executable, in τ_i. In this case, we add 4 Hadamard gates to change the direction of $\langle q, q' \rangle$ (cf. Figure 3 (top)), extend PC_i with all these 5 gates, and delete $\langle q, q' \rangle$ from LC_i. Again, we execute recursively all gates in LC_i which are executable in τ_i to get a child of s^i.

Finally, for each child of s^i, we extend one level further to get its grandchildren. We denote by $\{s^i_j : j \in J\}$ and $\{s^i_{j,k} : j \in J, k \in K\}$ the set of children and grandchildren of

s^i, respectively.

Example 1. *We consider the quantum circuit shown in Fig. 6 and the test architecture graph AG_{test} in Fig. 7. Applying Alg. 1 we get the initial mapping $\tau_{ini} : Q \to V$ which maps q_i to v_i for each $0 \le i \le 4$. For convenience, we write such a mapping as a list of length 5. For example, $\tau_{ini} = [0, 1, 2, 3, 4]$. Note that the front layer contains two gates, viz., $\langle q_2, q_1 \rangle$ and $\langle q_3, q_4 \rangle$. As the latter is directly executable by τ_{ini}, the initial state $s = (\tau_{ini}, PC_0 := \{\langle q_3, q_4 \rangle\}, LC_0 := LC \backslash \{\langle q_3, q_4 \rangle\})$, where LC is the circuit shown in Fig. 6.*

We next show how to construct the child states of s. Note that there is only one gate, viz. $\langle q_2, q_1 \rangle$, in the first layer of LC_0, and τ_{ini} maps q_1 and q_2 to, respectively, v_1 and v_2. Only 4 edges, $(1, 0), (1, 2), (2, 3)$ and $(5, 2)$, in AG_{test} (see Fig. 7) are relevant. For each of them, we obtain a corresponding swap operation and a corresponding child state. Since AG_{test} is directed and τ_{ini} can execute $\langle q_1, q_2 \rangle$, another child state can be obtained by using Way 2. Therefore, s has in total 5 child states and Table I gives the mappings and physical circuits of these child states as well as the corresponding operations. Similarly, we also construct the grandchild states of s, also shown in Table I. Here in the 'Operation' column, we use an edge in AG_{test} to denote the corresponding swap operation on mappings, and a CNOT gate to denote the operation of changing its direction.

Evaluate the grandchildren of s^i. The cost of a grandchild $s^i_{j,k}$ of s^i consists of two parts: the first part, $\text{cost}_g(s^i_{j,k})$, is the number of auxiliary CNOT and Hadamard gates added during the evolution from s^i to $s^i_{j,k}$, and the second part, $\text{cost}_h(s^i_{j,k})$, is an estimated cost for completing the remaining gates in the logic circuit of $s^i_{j,k}$.

The first part depends on the different ways of extending s^i and s^i_j to obtain $s^i_{j,k}$. If AG is undirected, then only Way 1 is available for the extensions and 3 CNOT gates suffice to implement the required swap operation on the mapping. Thus $\text{cost}_g(s^i_{j,k}) = 6$. Otherwise, 7 gates (3 CNOTs and 4 Hadamard shown in Fig. 3) for Way 1 and 4 Hadamard for Way 2 are needed. Thus $\text{cost}_g(s^i_{j,k})$ can be 14, 11, or 8. Consider the state s in Example 1. From Table I we can see that each grandchild state of s has cost_g 14 or 11.

For the second part of the cost, we employ a look-ahead mechanism first demonstrated in [4]. Given a generic state $s = (\tau_s, PC_s, LC_s)$, we partition the gates in LC_s into different layers according to its dependency graph. Denote by L_k, $k \ge 0$, these layers such that L_0 is the front layer. Then the heuristically estimated cost of s is defined as

$$\text{cost}_h(s) = \sum_{k=0}^{\ell} w_k \left(\sum_{g \in L_k} \text{cost}_{gate}(g, \tau_s) \right), \qquad (4)$$
$$+ w_s \times (d-1) \times N_{swap} \times N_s$$

where d is the diameter of the architecture graph, N_{swap} the number of elementary gates needed to compose a SWAP gate, and N_s is the number of gates in LC_s. The parameters $\ell > 0$, w_k $(0 \le k \le \ell)$ and w_s are taken empirically but normally we assume $1 = w_0 \ge w_1 \ge \cdots \ge w_l \ge w_s \ge 0$. This reflects the intuition that the closer a gate is from the front layer of the circuit, the more it contributes to the total cost of executing the whole circuit, as subsequent dependent gates will not be executable unless it has been processed. Table I gives the heuristic costs for all child and grandchild states of s in Example 1, where we take $\ell = 3$, $w_1 = 1$, $w_2 = 0.8$, $w_3 = 0.6$, $w_s = 0.4$. Note that the diameter of QX5 is 8 and each SWAP gate is composed by 7 elementary gates in QX5. Thus $d = 8$ and $N_{swap} = 7$.

Finally, the total cost of a grandchild $s^i_{j,k}$ of s^i is computed as

$$\text{cost}(s^i_{j,k}) = \text{cost}_g(s^i_{j,k}) + \text{cost}_h(s^i_{j,k}). \qquad (5)$$

Suppose $s^i_{j^*,k^*}$ is a grandchild state with the minimum cost. Then $s^i_{j^*}$ is selected as the state for the $i+1$-th step; that is, we let $s^{i+1} = s^i_{j^*}$. For the state s in Example 1, we can see from Table I that $s_{1,2}$ is the grandchild state with minimum cost. Thus we select its parent s_1 as our next state. Note that s_1 happens to be the child state which also has the minimum cost among all child states of s. In general, this coincidence does not hold. The whole algorithm for circuit transformation is shown in Algorithm 2.

D. Fallback via remote CNOT

During the search process, there is a small possibility that our algorithm does not halt. This happens when a child state with better cost may be good for gates in look-ahead layers but increases the distances of gates in the front layer. To address this problem, a fallback mechanism is introduced to ensure that the program terminates in reasonable time.

A direct way for fallback is to select a gate $\langle q, q' \rangle$ in the front layer and then choose a SWAP operation that will reduce the shortest path between the two corresponding nodes v, v' with $\tau_s(q) = v$ and $\tau_s(q') = v'$ in the architecture graph [3], where s denotes the current state. However, this method may change the mapping that the algorithm may want to keep as it is preferred by look-ahead layers. To protect the preferred mapping, remote CNOT operations [9], which are depicted in Fig. 8, are introduced in the fallback. After imposing remote CNOT gates, the circuit has the same functionality while preserving the current mapping. The fallback is activated when no gates are removed from LC_s after a certain prefixed number of rounds.

E. Complexity of the Search Process

In each layer, there are at most $|Q|/2$ gates, where Q is the set of qubits in the input logic circuit. Thus, the time complexity of computing the cost (cf. Eq.(5)) of any state is $O(\ell \cdot |Q|)$, where ℓ is the prefixed small number of layers we select for Eq.(4). For our evaluation (see Section V), we take $\ell = 3$ for all circuits.

By construction, each state s has at most $|E| + |Q|/2$ child states, where E is the set of edges in the architecture graph, or, equivalently, the number of possible SWAP operations that can be added to the circuit and $|Q|/2$ is the number of CNOT gates in the front layer of the current logic circuit that can be applied by adding four extra Hadamard gates to change the direction.

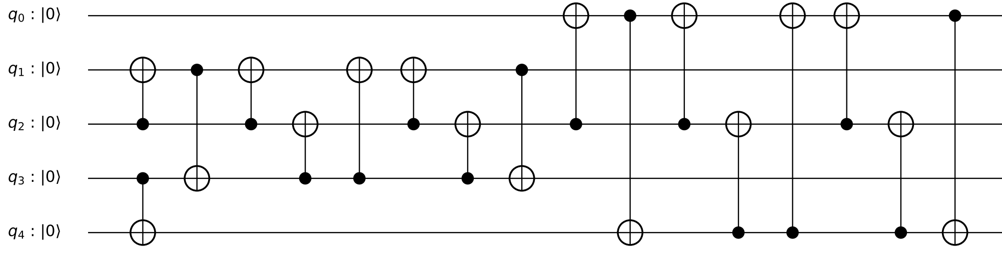

Fig. 6. The quantum circuit alu-v0_27 with all single qubit gates removed.

TABLE I
THE CHILD AND GRANDCHILD STATES OF A STATE IN EXAMPLE 1.

Child State	Mapping	Newly added gates in PC_s	Operation	cost_g	cost_h
s_0	[1,0,2,3,4]	$\{0 \leftrightarrows 1\}$	(1,0)	7	208.2
s_1	[1,0,2,3,4]	$\{1 \leftrightarrows 2, \langle 1,2\rangle, \langle 2,3\rangle, \langle 1,2\rangle\}$	(1,2)	7	167.2
s_2	[0,1,3,2,4]	$\{2 \leftrightarrows 3\}$	(2,3)	7	199.0
s_3	[0,1,5,3,4]	$\{5 \leftrightarrows 2\}$	(5,2)	7	203.0
s_4	[0,1,2,3,4]	$\{\overleftarrow{\langle 1,2\rangle}\}$	$\langle q_2, q_1\rangle$	4	188.8
$s_{0,0}$	[0,1,2,3,4]	$\{0 \leftrightarrows 1\}$	(0,1)	14	195.8
$s_{0,1}$	[1,5,2,3,4]	$\{0 \leftrightarrows 5\}$	(0,5)	14	195.8
$s_{0,2}$	[2,0,1,3,4]	$\{1 \leftrightarrows 2, \langle 1,0\rangle\}$	(1,2)	14	199.2
$s_{0,3}$	[1,0,3,2,4]	$\{2 \leftrightarrows 3\}$	(2,3)	14	211.4
$s_{0,4}$	[1,0,5,3,4]	$\{2 \leftrightarrows 5, \langle 5,0\rangle\}$	(2,5)	14	196.0
$s_{1,0}$	[1,2,0,3,4]	$\{0 \leftrightarrows 1\}$	(0,1)	14	177.6
$s_{1,1}$	[0,1,2,3,4]	$\{1 \leftrightarrows 2\}$	(1,2)	14	166.2
$s_{1,2}$	[0,3,1,2,4]	$\{2 \leftrightarrows 3\}$	(2,3)	14	157.6
$s_{1,3}$	[0,2,1,4,3]	$\{3 \leftrightarrows 4\}$	(3,4)	14	175.0
$s_{2,0}$	[1,0,3,2,4]	$\{0 \leftrightarrows 1\}$	(0,1)	14	211.4
$s_{2,1}$	[0,2,3,1,4]	$\{1 \leftrightarrows 2\}$	(1,2)	14	194.6
$s_{2,2}$	[0,1,2,3,4]	$\{2 \leftrightarrows 3\}$	(2,3)	14	195.8
$s_{2,3}$	[0,1,4,2,3]	$\{3 \leftrightarrows 4\}$	(3,4)	14	208.6
$s_{3,0}$	[1,0,5,3,4]	$\{0 \leftrightarrows 1, \langle 5,0\rangle\}$	(0,1)	14	196.0
$s_{3,1}$	[5,1,0,3,4]	$\{0 \leftrightarrows 5\}$	(0,5)	14	201.8
$s_{3,2}$	[0,2,5,3,4]	$\{1 \leftrightarrows 2, \langle 5,2\rangle, \langle 2,3\rangle, \langle 5,2\rangle\}$	(1,2)	14	160.8
$s_{3,3}$	[0,1,2,3,4]	$\{2 \leftrightarrows 5\}$	(2,5)	14	195.8
$s_{3,4}$	[0,1,4,3,5]	$\{4 \leftrightarrows 5\}$	(4,5)	14	211.4
$s_{4,0}$	[1,0,2,3,4]	$\{0 \leftrightarrows 1\}$	(0,1)	11	200.6
$s_{4,1}$	[0,2,1,3,4]	$\{1 \leftrightarrows 2, \langle 2,3\rangle, \langle 1,2\rangle\}$	(1,2)	11	167.2
$s_{4,2}$	[0,1,3,2,4]	$\{2 \leftrightarrows 3, \langle 1,2\rangle\}$	(2,3)	11	177.6
$s_{4,3}$	[0,1,2,4,3]	$\{3 \leftrightarrows 4\}$	(3,4)	11	200.0

Here $s = (\tau_{ini}, PC_0 := \{\langle q_3, q_4\rangle\}, LC_0 := LC \backslash \{\langle q_3, q_4\rangle\})$, $i \leftrightarrows j$ denotes the swap operation of i and j and $\overleftarrow{\langle i,j\rangle}$ denotes the operation that changes the direction of the CNOT gate $\langle i, j\rangle$.

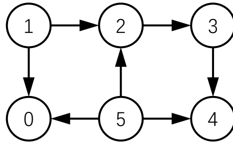

Fig. 7. A test architecture graph AG_{test}.

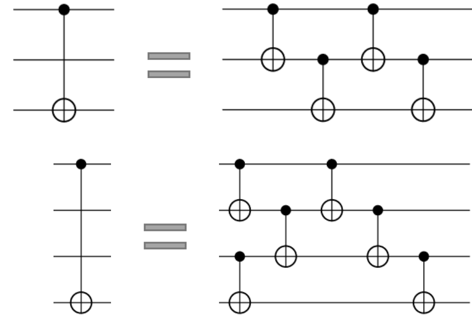

Fig. 8. Schematic for remote CNOT operations with 2 and 3 hops. Generalized form can be found in [16].

Suppose the input circuit contains m CNOT gates. If we activate the fallback when no gates are removed from LC_s after K rounds, then the search procedure has at most $K \times m$ states. This is because each activation of the fallback will execute a selected gate due to the use of remote CNOT. Therefore, the overall time complexity of the search is $O(\ell \cdot |Q| \cdot (|E| + |Q|/2)^2 \cdot m \cdot K)$. Because $|Q| \leq |V|$ and $|E| \leq |V| \cdot (|V|-1)/2$, it is bounded by $O(|V|^4 \cdot |Q| \cdot \ell \cdot m \cdot K)$.

For the space complexity, in each state s, we maintain a depth-2 search tree rooted at s. Thus the space complexity of the algorithm is bounded by $O((|E| + |Q|/2)^2)$, i.e., $O(|V|^4)$.

Algorithm 2: Circuit transformation with look-ahead

```
input  : A logic circuit LC = (Q, C^l), an initial
         mapping τ constructed by Algorithm 1 and an
         architecture graph AG = (V, E) with
         |Q| ≤ |V|.
output: A physical circuit (V, C^p) which satisfies AG
         and is equivalent to LC.
begin
    (PC, LC) ← Execute(τ, PC, LC);
    while LC ≠ ∅ do
        L ← F(LC), the first layer of LC;
        Cld ← ∅;
        for e ∈ E which touches some gate in L under
          τ do
            τ' ← swap_e ∘ τ;
            PC' ← PC' by adding (the CNOT + H
              implementation of) a SWAP gate
              corresponding to swap_e;
            (PC', LC') ← Execute(τ', PC', LC);
            gcost ← 3 if e^{-1} ∈ E and 7 otherwise;
            Cld ← Cld ∪ {(τ', PC', LC', gcost)};
        end
        for g ∈ L which is inversely executable by τ do
            PC' ← PC' by adding τ(g) complemented
              by four H gates before and after it;
            LC' ← LC' by deleting g;
            (PC', LC') ← Execute(τ, PC', LC');
            Cld ← Cld ∪ {(τ, PC', LC', 4)};
        end
        mCost ← ∞;
        for (τ', PC', LC', gcost) ∈ Cld do
            cost ← minChildHcost(τ', LC');
            if cost + gcost < mCost then
                mCost ← cost + gcost;
                (τ, PC, LC) ← (τ', PC', LC');
            end
        end
    end
    return PC
end
```

F. Optimization

In Algorithm 2, the search space grows exponentially if the depth of look-ahead is increased. Therefore, a pruning mechanism is introduced to reduce the size of the search space while preserving the quality of the output physical circuit. More specifically, a child state s^i_j of s^i will be removed if both $\mathsf{cost}'_h(s^i_j) > \mathsf{cost}'_h(s^i)$ and $\mathsf{cost}_h(s^i_j) - \mathsf{cost}'_h(s^i_j) > \mathsf{cost}_h(s^i) - \mathsf{cost}'_h(s^i)$, where $\mathsf{cost}'_h(s) = w \times (d-1) \times N_{swap} \times N_s s$ as defined in Eq.(4). In Example. 1, states $s_{1,0}$ will be pruned. This is because $\mathsf{cost}'_h(s) = 145.6$, $\mathsf{cost}_h(s) = 167.2$, $\mathsf{cost}'_h(s_0) = 145.6$, $\mathsf{cost}_h(s_0) = 177.6$ and $\mathsf{cost}_h(s_0) > \mathsf{cost}_h(s)$, $\mathsf{cost}_h(s_0) - \mathsf{cost}'_h(s_0) > \mathsf{cost}_h(s) - \mathsf{cost}'_h(s)$.

From Fig. 9 we can see that the pruning mechanism has limited influence on the sizes of the output circuits while the time consumption is reduced by a large amount.

Procedure Execute(τ, PC, LC)

```
input  : A mapping τ : Q → V, a physical circuit PC,
         and a logic circuit LC.
output: A pair (PC', LC') obtained by executing as
         many as possible gates which satisfy τ.
begin
    PC' ← PC; LC' ← LC;
    do
        EL ← {g ∈ F(LC') : g is executable by τ};
        for g ∈ EL do
            PC' ← PC' by adding τ(g);
            LC' ← LC' by deleting g;
        end
    while EL ≠ ∅;
    return (PC', LC')
end
```

Procedure minChildHcost(τ, LC)

```
input  : A mapping τ : Q → V and a logic circuit LC.
output: The minimal cost of all children of τ.
begin
    L ← F(LC);
    mCost ← ∞;
    for e ∈ E which touches some gate in L under τ
      do
        τ' ← swap_e ∘ τ;
        (PC', LC') ← Execute(τ', ∅, LC);
        gcost ← 3 if e^{-1} ∈ E and 7 otherwise;
        hcost ← hcost(τ', LC') according to Eq.(4);
        if hcost + gcost < mCost then
            mCost ← hcost + gcost;
        end
    end
    for g ∈ L which is inversely executable by τ do
        LC' ← LC by deleting g;
        (PC', LC') ← Execute(τ, ∅, LC);
        hcost ← hcost(τ, LC') according to Eq.(4);
        if hcost + 4 < mCost then
            mCost ← hcost + 4;
        end
    end
    return mCost;
end
```

V. Programming and Benchmarks

To evaluate our approach, we compare it with previous algorithms proposed for the same purpose in the literature [13], [4], [15]. We use Python as our programming language and IBM Qiskit [30] as auxiliary environment. The code can be found in GitHub. All experiments are conducted in a laptop with i7-8750H CPU and 16GB memory. The results are reported in Tables II, III and IV, in which column 'Comparison' shows the percentage of the number of added gates in our algorithm to the compared one. Specifically, let n_{comp} and n_{ours} be the numbers of gates added by the compared algorithm and

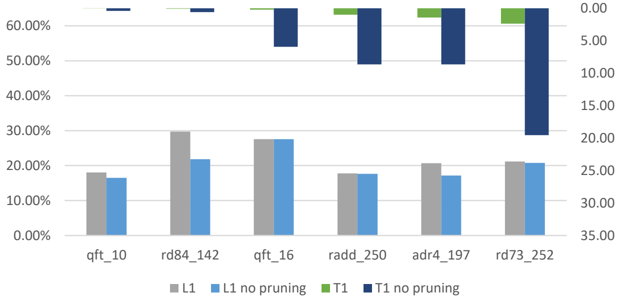

Fig. 9. Effectiveness of the pruning mechanism obtained by running exemplary circuits on IBM Q20. The gray and blue bars on the bottom are *ratios* (corresponding to the left vertical axis) of the number of added gates to that of original gates for the proposed algorithm with and without pruning. The bars on the top correspond to the time consumption (*seconds*, corresponding to the right vertical axis) for the search process.

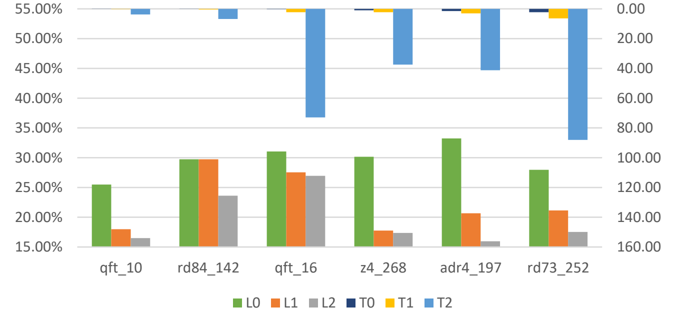

Fig. 10. Experiments on IBM Q20 for different look-ahead depths. The bars on the bottom and top represent respectively the *ratio* (corresponding to the left vertical axis) of the number of added gates to that of the original gates and the consumed time (*seconds*, corresponding to the right vertical axis) for different circuits and different look-ahead depths.

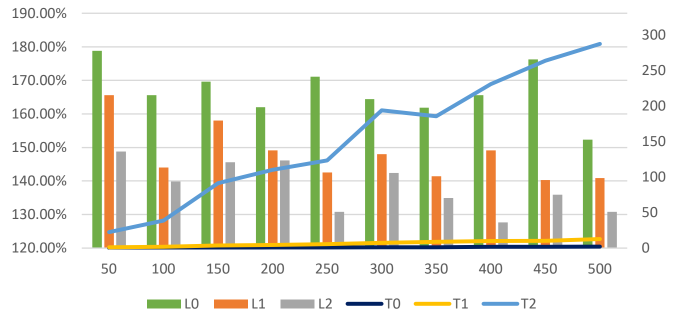

Fig. 11. Experiments for random circuits on IBM Q20 for different look-ahead depths. The horizontal axis denotes the number of gates of input circuits which are generated randomly. The bars and lines represent respectively the *ratio* (corresponding to the left vertical axis) of the average number of added gates to that of the original gates and the average consumed time (*seconds*, corresponding to the right vertical axis) for different random circuits and different look-ahead depths.

by ours respectively, then the percentages in the above column are obtained by n_{ours}/n_{comp}. Note that a symbol $0/0$ appears when both n_{comp} and n_{ours} are zero.

Table II demonstrates the superiority of the initial mapping output by simulated annealing (Alg. 1) (SA-based for short) compared to the naive initial mapping[2] (naive for short) and the initial mapping generated in the A^* algorithm [13] (A^*-based for short) on IBM Q20, where the 'Comparison' column shows the comparison between the SA-based initial mapping and the A^*-based one. From the last row of Table II we can see that, on average, the number of added gates by our algorithm with SA-based initial mappings is only 62.95% of the number of added gates by our algorithm with A^*-based initial mappings.

For the heuristic search, we compare our algorithm to the ones introduced in [13] and [4], which are, respectively, the state-of-the-art algorithms for IBM QX5 and Q20. We set the number of look-ahead layers as $\ell = 3$, and the weight parameters $w_1 = 1$, $w_2 = 0.8$, $w_3 = 0.6$, $w_4 = 0.4 \times (D_{AG} - 1) \times N_{swap}$ in Eq.(4), where D_{AG} is the diameter of the architecture graph and N_{swap} the number of elementary gates needed to compose a SWAP gate. The threshold number K for activating fallback is set to be $0.5 \times D_{AG}$.

The algorithm proposed in [13] utilizes A^* to find the best solution of each layer. It has exponential time complexity and only considers one layer for look-ahead when designing the heuristic cost function. Like ours, their A^*-based algorithm works for both directed and undirected architecture graphs. As confirmed in [4], it is comparable with the algorithm in [4] when Q20 is used as the QPU. So we only make the comparison on QX5. From the experimental results reported in Table III, we can see that our algorithm has a conspicuous improvement over the algorithm in [13]. Moreover, it is very efficient: for input circuits with up to 10,000 elementary gates, our algorithm finds the solution within one minute.

The algorithm proposed in [4] uses reverse traversal technique to search for a good initial mapping and has polynomial complexity. Although it considers multiple levels in its heuristic function, this algorithm does not consider the weights for gates in different layers in the heuristic function. Unlike our algorithm, the algorithm in [4] can only be applied to undirected architecture graphs. Therefore, we only compared it with ours on Q20. From the experimental results reported in Table IV, we see that, for small circuits, both algorithms find the optimal output circuits; but for circuits with large size, our algorithm again has a conspicuous improvement. As for QX5, our algorithm is able to find within two minutes the solution to input circuits with up to 30,000 elementary gates.

We also compared our algorithm with the algorithm proposed in [15], which also works for both directed and undirected architecture graph and its performance is comparable with the one in [13]. In Appendix, from the experimental results reported in Table 5 and 6, we can see that our algorithm also has a better performance.

It is worth mentioning that, if the depth for look-ahead in the selection process is increased, the quality of the output circuits could be further improved. However, the time consumption will be increased dramatically. See Fig. 10 for the experiment on a few examples, which indicates that 1-depth look-ahead reaches the best trade off of time and performance. Fig. 11 exhibits the time consumption and quality for some random circuits. It further confirms that increasing the search depth

[2]The mapping which maps the i-th logical qubit q_i to the i-th physical qubit v_i for all i.

TABLE II
THE PERFORMANCE IMPROVEMENT BROUGHT BY SIMULATED ANNEALING ON IBM Q20.

Circuit Name	Original #Gates	Original #CNOT	#Added Naive	#Added A^*-based	#Added SA-based	Comparison
4mod5-v1_22	21	11	9	3	0	0.00%
mod5mils_65	35	16	21	9	0	0.00%
alu-v0_27	36	17	15	21	6	28.57%
decod24-v2 43	52	38	33	12	0	0.00%
4gt13_92	66	30	54	33	0	0.00%
ising_model_10	480	90	48	36	0	0.00%
ising_model_13	633	120	54	51	0	0.00%
ising_model_16	786	150	96	60	0	0.00%
qft_10	200	90	90	45	36	80.00%
qft_16	512	240	228	198	135	68.18%
rd84_142	343	154	147	120	102	85.00%
adr4_197	3439	1498	1044	1056	711	67.33%
radd_250	3213	1405	969	789	729	92.40%
z4_268	3073	1343	852	879	546	62.12%
sym6_145	3888	1701	846	1425	744	52.21%
misex1_241	4813	2100	1017	1560	921	59.04%
rd73_252	5321	2319	1803	1494	1125	75.30%
cycle10_2_110	6050	2648	1383	1599	1038	64.92%
square_root_7	7630	3089	1620	1989	1353	68.02%
sqn_258	10223	4459	3105	2766	1953	70.61%
rd84_253	13658	5960	4140	3642	3198	87.81%
co14_215	17936	7840	4878	6348	4356	68.62%
sym9_193	34881	15232	9663	11055	6123	55.39%
9symml_195	34881	15232	9663	11055	6036	54.60%
Summary	152170	65782	41778	46245	29112	62.95%

TABLE III
COMPARISON OF OUR ALGORITHM WITH THE A^*-BASED ALGORITHM IN [13] ON IBM QX5.

Circuit Name	Original #Gates	Original #CNOT	#Added A^*	#Added Ours	Running Time (s)	Comparison
mini_alu_305	173	77	561	372	0.27	66.31%
qft_10	200	90	437	273	0.30	62.47%
sys6-v0_111	215	98	725	486	0.32	67.03%
rd73_140	230	104	704	504	0.42	71.59%
sym6_316	270	123	875	655	0.43	74.86%
rd53_311	275	124	817	710	0.44	86.90%
sym9_146	328	148	989	777	0.53	78.56%
rd84_142	343	154	1038	782	0.59	75.34%
ising_model_10	480	90	200	142	0.65	71.00%
cnt3-5_180	485	215	1218	1068	0.83	87.68%
qft_16	512	240	1264	890	0.91	70.41%
ising_model_13	633	120	280	199	1.36	71.07%
ising_model_16	786	150	320	263	1.64	82.19%
wim_266	986	427	2881	2071	1.86	71.88%
cm152a_212	1221	532	3307	2613	1.95	79.01%
cm42a_207	1776	771	4433	3836	3.39	86.53%
pm1_249	1776	771	4433	3800	3.42	85.72%
dc1_220	1914	833	5095	4043	3.44	79.35%
squar5_261	1993	869	5355	4648	4.60	86.80%
sqrt8_260	3009	1314	8331	7172	8.05	86.09%
z4_268	3073	1343	8120	6922	8.40	85.25%
adr4_197	3439	1498	9273	8084	8.70	87.18%
sym6_145	3888	1701	9538	7906	10.95	82.89%
misex1_241	4813	2100	12620	10901	14.74	86.38%
ham15_107	8763	3858	22980	20066	44.66	87.32%
dc2_222	9462	4131	26441	22955	58.38	86.82%
sqn_258	10223	4459	26734	22851	58.91	85.48%
inc_237	10619	4636	28532	24896	59.28	87.26%
co14_215	17936	7840	51894	43424	320.81	83.68%
life_238	22445	9800	59672	52827	265.09	88.53%
max46_240	27126	11844	69726	61829	395.91	88.67%
9symml_195	34881	15232	95272	82310	667.18	86.39%
dist_223	38046	16624	103683	92045	829.44	88.78%
sao2_257	38577	16864	108419	93533	1059.06	86.27%
Summary	250896	109180	676167	585853	#	86.64%

TABLE IV
COMPARISON OF OUR ALGORITHM WITH THE ALGORITHM IN [4] ON IBM Q20.

Circuit Name	Original #Gates	Original #CNOT	#Added Alg. in [4]	#Added Ours	Running Time (s)	Comparison
4mod5-v1_22	21	11	0	0	0.00	0/0
mod5mils_65	35	16	0	0	0.00	0/0
alu-v0_27	36	17	3	6	0.00	200.00%
decod24-v2 43	52	38	0	0	0.00	0/0
4gt13_92	66	30	0	0	0.00	0/0
ising_model_10	480	90	0	0	0.00	0/0
ising_model_13	633	120	0	0	0.01	0/0
ising_model_16	786	150	0	0	0.01	0/0
qft_10	200	90	54	36	0.16	66.67%
qft_16	512	240	186	135	0.38	72.58%
rd84_142	343	154	105	102	1.50	97.14%
adr4_197	3439	1498	1614	711	2.08	44.05%
radd_250	3213	1405	1275	729	2.13	57.18%
z4_268	3073	1343	1365	546	2.65	40.00%
sym6_145	3888	1701	1272	744	2.82	58.49%
misex1_241	4813	2100	1521	921	3.44	60.55%
rd73_252	5321	2319	2133	1125	5.21	52.74%
cycle10_2_110	6050	2648	2622	1038	5.46	39.59%
square_root_7	7630	3089	2598	1353	12.24	52.08%
sqn_258	10223	4459	4344	1953	13.91	44.96%
rd84_253	13658	5960	6147	3198	34.75	52.03%
co14_215	17936	7840	8982	4356	88.90	48.50%
sym9_193	34881	15232	16653	6123	126.68	36.77%
9symml_195	34881	15232	17268	6036	137.47	34.95%
Summary	152170	65782	68142	29112	#	42.72%

will produce a significant improvement on the size of output circuit, sometimes more than 20%. However, the running time will increase hugely and become unacceptable when the depth becomes 2. This suggests that the benefit brought by increasing the depth seems to be uneconomical. Nevertheless, it may still be acceptable in some application scenarios if high-performance computing devices are available and smaller size of the output circuit is desired. If this is the case, the algorithm can be easily adjusted by modifying the relevant parameters of our algorithm. Besides, the weight parameters in the heuristic function are also adjustable when different architecture graphs and circuits are considered.

VI. CONCLUSION AND DISCUSSION

In this paper, we propose an algorithm to solve the quantum circuit transformation problem by using simulated annealing and heuristic search. A double look-ahead mechanism is novelly adopted in the algorithm. We look ahead at subsequent layers when defining a flexible heuristic cost function which also supports weight parameters to reflect the variable influence of gates in different layers. Moreover, we look ahead at grandchild states with minimal cost in selecting the best state for the next step of the circuit transformation. Detailed evaluation on extensive realistic circuits shows that our algorithm has consistent and significant improvement when compared with the two state-of-the-art algorithms proposed in the literature for IBM QX5 and Q20.

Although our algorithm can produce significantly better results than the state-of-the-art algorithms, it is not optimal. As discussed in Sec V, even better results could be obtained through increasing the search depth in our algorithm. Apparently, if exhaustive search is employed, in principle we can even generate the optimal results. Indeed, it was shown in [31] that, for circuits with up to 5 qubits and 100 gates and IBM QX4, the exact solution can be computed within an acceptable time. Similar sub-optimal results were obtained in [32] by using exhaustive search on IBM QX5 for circuits with up to 6 qubits and 800 gates. These approaches are unpractical for circuits with more qubits and/or a large number of CNOT gates. In [20], the permutation problem is formulated as an ILP problem and the same results as in [32] are obtained with smaller time overhead. This ILP approach is still not scalable, especially for circuits with ≥ 10 qubits and ≥ 500 gates. For examples, consider the circuits 'cm82a_208' with 8 qubits and 650 gates and 'sys6-v0_111' with 10 qubits and 215 gates. The ILP-based algorithm, also implemented in Python, needs 414 seconds and, respectively, 1 hour while our algorithm only needs 0.93 and, respectively, 0.32 seconds.

While our aim in this paper is to show that the number of SWAPs required for quantum circuit transformation could be significantly reduced, minimizing depth or latency and circuit error is also important. Note that in most cases, the number of CNOT gates in our output circuit is already smaller than the depth of the output circuit (only CNOT gates are counted) of the compared algorithm. Consider all circuits in Table IV. The sum of the depths of the output circuits (CNOT only) obtained by the algorithm in [15] is 394192 (not shown in the table), while the sum of the numbers of CNOT gates in the output circuits obtained by our algorithm is 365040 = 248553 (original #CNOT) + 116487 (added #CNOT) (see the last row of the table).

For future studies, we propose the following problems to solve. First, our program still runs slowly for circuits with large sizes. Thus it is necessary to optimize the code to reduce the running time. Second, the quality of the initial mappings obtained from the simulated annealing algorithm

(Alg. 1) is not stable, which is not acceptable for commercial use. Third, we only considered connectivity in the architecture graphs; other constraints like *cross talk*, which will invalidate some concurrent gate operations [33], [27], *gate error* caused by various noise effects [7], [10], [34], *classical control* led by shared channels among physical qubits [8] and *qubits decoherence* [7], [9], [35] should be included to make the algorithm more practical. Fourth, only using the sizes of circuits as the criterion for evaluation is not enough. Criteria like circuit error and running time should also be considered in future work.

References

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

[2] R. Wille, A. Fowler, and Y. Naveh, "Computer-aided design for quantum computation," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–6.

[3] A. M. Childs, E. Schoute, and C. M. Unsal, "Circuit transformations for quantum architectures," *arXiv preprint arXiv:1902.09102*, 2019.

[4] G. Li, Y. Ding, and Y. Xie, "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*. ACM, 2019, pp. 1001–1014.

[5] M. Y. Siraichi, V. F. d. Santos, S. Collange, and F. M. Q. Pereira, "Qubit allocation," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. ACM, 2018, pp. 113–125.

[6] A. Paler, "On the influence of initial qubit placement during NISQ circuit compilation," in *International Workshop on Quantum Technology and Optimization Problems*. Springer, 2019, pp. 207–217.

[7] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. Martonosi, "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*. ACM, 2019, pp. 1015–1029.

[8] L. Lao, D. M. Manzano, H. van Someren, I. Ashraf, and C. G. Almudever, "Mapping of quantum circuits onto NISQ superconducting processors," *arXiv preprint arXiv:1908.04226*, 2019.

[9] S. Nishio, Y. Pan, T. Satoh, H. Amano, and R. Van Meter, "Extracting success from IBM's 20-qubit machines using error-aware compilation," *arXiv preprint arXiv:1903.10963*, 2019.

[10] S. S. Tannu and M. K. Qureshi, "Not all qubits are created equal: a case for variability-aware policies for NISQ-era quantum computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 987–999.

[11] T. Miltzow, L. Narins, Y. Okamoto, G. Rote, A. Thomas, and T. Uno, "Approximation and hardness for token swapping," *arXiv preprint arXiv:1602.05150*, 2016.

[12] N. Alon, F. R. Chung, and R. L. Graham, "Routing permutations on graphs via matchings," *SIAM Journal on Discrete Mathematics*, vol. 7, no. 3, pp. 513–530, 1994.

[13] A. Zulehner, A. Paler, and R. Wille, "An efficient methodology for mapping quantum circuits to the IBM qx architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

[14] W. Finigan, M. Cubeddu, T. Lively, J. Flick, and P. Narang, "Qubit allocation for noisy intermediate-scale quantum computers," *arXiv preprint arXiv:1810.08291*, 2018.

[15] A. Cowtan, S. Dilkes, R. Duncan, A. Krajenbrink, W. Simmons, and S. Sivarajah, "On the qubit routing problem," *arXiv preprint arXiv:1902.08091*, 2019.

[16] B. Nash, V. Gheorghiu, and M. Mosca, "Quantum circuit optimizations for NISQ architectures," *arXiv preprint arXiv:1904.01972*, 2019.

[17] A. Kissinger and A. M.-v. de Griend, "Cnot circuit extraction for topologically-constrained quantum memories," *arXiv preprint arXiv:1904.00633*, 2019.

[18] K. E. Booth, M. Do, J. C. Beck, E. Rieffel, D. Venturelli, and J. Frank, "Comparing and integrating constraint programming and temporal planning for quantum circuit compilation," in *Twenty-Eighth International Conference on Automated Planning and Scheduling*, 2018.

[19] D. Venturelli, M. Do, E. G. Rieffel, and J. Frank, "Temporal planning for compilation of quantum approximate optimization circuits." in *IJCAI*, 2017, pp. 4440–4446.

[20] A. A. de Almeida, G. W. Dueck, and A. C. da Silva, "Finding optimal qubit permutations for IBM's quantum computer architectures," in *Proceedings of the 32nd Symposium on Integrated Circuits and Systems Design*. ACM, 2019, p. 13.

[21] P. Murali, N. M. Linke, M. Martonosi, A. J. Abhari, N. H. Nguyen, and C. H. Alderete, "Full-stack, real-system quantum computer studies: Architectural comparisons and design insights," *arXiv preprint arXiv:1905.11349*, 2019.

[22] M. A. Nielsen and I. L. Chuang, "Quantum information and quantum computation," *Cambridge: Cambridge University Press*, vol. 2, no. 8, p. 23, 2000.

[23] R. Van Meter, *Quantum networking*. John Wiley & Sons, 2014.

[24] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, "Elementary gates for quantum computation," *Physical Review A*, vol. 52, no. 5, p. 3457, 1995.

[25] T. Häner, D. S. Steiger, K. Svore, and M. Troyer, "A software methodology for compiling quantum programs," *Quantum Science and Technology*, vol. 3, no. 2, p. 020501, 2018.

[26] J. M. Gambetta, J. M. Chow, and M. Steffen, "Building logical qubits in a superconducting quantum computing system," *npj Quantum Information*, vol. 3, no. 1, pp. 1–7, 2017.

[27] A. Botea, A. Kishimoto, and R. Marinescu, "On the complexity of quantum circuit compilation," in *Eleventh Annual Symposium on Combinatorial Search*, 2018.

[28] T. Itoko, R. Raymond, T. Imamichi, A. Matsuo, and A. W. Cross, "Quantum circuit compilers using gate commutation rules," in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*. ACM, 2019, pp. 191–196.

[29] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[30] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C. Chen *et al.*, "Qiskit: An open-source framework for quantum computing," *Accessed on: Mar*, vol. 16, 2019.

[31] R. Wille, L. Burgholzer, and A. Zulehner, "Mapping quantum circuits to IBM qx architectures using the minimal number of swap and h operations," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 142.

[32] A. A. De Almeida, G. W. Dueck, and A. C. Da Silva, "Cnot gate optimizations via qubit permutations for IBM's quantum architectures," *Journal of Low Power Electronics*, vol. 15, no. 2, pp. 182–192, 2019.

[33] D. Venturelli, M. Do, B. O'Gorman, J. Frank, E. Rieffel, K. E. Booth, T. Nguyen, P. Narayan, and S. Nanda, "Quantum circuit compilation: An emerging application for automated reasoning," in *Proceedings of the Scheduling and Planning Applications Workshop*, 2019.

[34] G. Li, Y. Ding, and Y. Xie, "SANQ: A simulation framework for architecting noisy intermediate-scale quantum computing system," *arXiv preprint arXiv:1904.11590*, 2019.

[35] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver, "A quantum engineer's guide to superconducting qubits," *Applied Physics Reviews*, vol. 6, no. 2, p. 021318, 2019.

APPENDIX

Circuit Name	Original #Gates	Original #CNOT	#Added Alg. In [15]	#Added Ours	Running Time(s)	Comparison
graycode6_47	5	5	8	8	0.00	100.00%
xor5_254	7	5	23	7	0.00	30.43%
ex1_226	7	5	23	14	0.00	60.87%
4gt11_84	18	9	37	29	0.01	78.38%
ex-1_166	19	9	49	29	0.01	59.18%
ham3_102	20	11	56	32	0.01	57.14%
4mod5-v0_20	20	10	41	29	0.01	70.73%
4mod5-v1_22	21	11	41	32	0.01	78.05%
mod5d1_63	22	13	56	36	0.02	64.29%
4gt11_83	23	14	59	40	0.02	67.80%
4gt11_82	27	18	82	51	0.03	62.20%
rd32-v0_66	34	16	74	57	0.02	77.03%
mod5mils_65	35	16	97	53	0.02	54.64%
4mod5-v0_19	35	16	93	60	0.02	64.52%
rd32-v1_68	36	16	74	57	0.02	77.03%
alu-v0_27	36	17	82	61	0.02	74.39%
3_17_13	36	17	66	57	0.02	86.36%
4mod5-v1_24	36	16	77	60	0.02	77.92%
alu-v1_29	37	17	82	61	0.02	74.39%
alu-v1_28	37	18	94	68	0.02	72.34%
alu-v3_35	37	18	82	69	0.02	84.15%
alu-v2_33	37	17	82	54	0.02	65.85%
alu-v4_37	37	18	82	61	0.02	74.39%
miller_11	50	23	119	89	0.03	74.79%
decod24-v0_38	51	23	103	75	0.03	72.82%
alu-v3_34	52	24	129	108	0.03	83.72%
decod24-v2_43	52	22	103	89	0.02	86.41%
mod5d2_64	53	25	109	110	0.03	100.92%
4gt13_92	66	30	152	121	0.04	79.61%
4gt13-v1_93	68	30	144	118	0.05	81.94%
one-two-three-v2_100	69	32	148	130	0.04	87.84%
4mod5-v1_23	69	32	165	145	0.05	87.88%
4mod5-v0_18	69	31	146	143	0.05	97.95%
one-two-three-v3_101	70	32	174	129	0.06	74.14%
4mod5-bdd_287	70	31	149	130	0.04	87.25%
decod24-bdd_294	73	32	143	133	0.04	93.01%
4gt5_75	83	38	180	143	0.05	79.44%
alu-v0_26	84	38	197	164	0.06	83.25%
rd32_270	84	36	180	161	0.05	89.44%
alu-bdd_288	84	38	161	163	0.07	101.24%
decod24-v1_41	85	38	196	160	0.05	81.63%
4gt5_76	91	46	211	180	0.08	85.31%
4gt13_91	103	49	245	190	0.08	77.55%
4gt13_90	107	53	264	212	0.09	80.30%
alu-v4_36	115	51	260	211	0.08	81.15%
4gt5_77	131	58	278	232	0.10	83.45%
one-two-three-v1_99	132	59	315	260	0.09	82.54%
rd53_138	132	60	280	249	0.22	88.93%
one-two-three-v0_98	146	65	299	282	0.12	94.31%
4gt10-v1_81	148	66	358	274	0.10	76.54%
decod24-v3_45	150	64	346	257	0.11	74.28%
aj-e11_165	151	69	321	258	0.11	80.37%
4mod7-v0_94	162	72	378	294	0.14	77.78%
alu-v2_32	163	72	354	309	0.13	87.29%

Circuit Name	Original #Gates	Original #CNOT	#Added Alg. In [15]	#Added Ours	Running Time(s)	Comparison
4mod7-v1_96	164	72	332	321	0.12	96.69%
cnt3-5_179	175	85	667	342	0.59	51.27%
mod10_176	178	78	388	320	0.15	82.47%
4gt4-v0_80	179	79	417	325	0.11	77.94%
4gt12-v0_88	194	86	450	373	0.14	82.89%
0410184_169	211	104	666	453	0.61	68.02%
4_49_16	217	99	508	395	0.16	77.76%
4gt12-v1_89	228	100	523	441	0.19	84.32%
4gt4-v0_79	231	105	489	437	0.22	89.37%
hwb4_49	233	107	512	451	0.20	88.09%
4gt4-v0_78	235	109	504	448	0.19	88.89%
mod10_171	244	108	535	452	0.23	84.49%
4gt12-v0_87	247	112	539	467	0.21	86.64%
4gt12-v0_86	251	116	554	478	0.26	86.28%
4gt4-v0_72	258	113	560	483	0.27	86.25%
4gt4-v1_74	273	119	627	528	0.22	84.21%
mini-alu_167	288	126	631	548	0.24	86.85%
one-two-three-v0_97	290	128	571	563	0.23	98.60%
rd53_135	296	134	733	623	0.36	84.99%
ham7_104	320	149	803	659	0.30	82.07%
decod24-enable_126	338	149	753	667	0.30	88.58%
mod8-10_178	342	152	886	685	0.32	77.31%
4gt4-v0_73	395	179	894	758	0.43	84.79%
ex3_229	403	175	844	763	0.37	90.40%
mod8-10_177	440	196	961	859	0.46	89.39%
alu-v2_31	451	198	1007	845	0.42	83.91%
C17_204	467	205	1121	972	0.54	86.71%
rd53_131	469	200	1145	925	0.52	80.79%
alu-v2_30	504	223	1123	1011	0.50	90.03%
mod5adder_127	555	239	1203	1021	0.60	84.87%
rd53_133	580	256	1374	1163	0.84	84.64%
majority_239	612	267	1461	1179	0.64	80.70%
ex2_227	631	275	1499	1264	0.73	84.32%
cm82a_208	650	283	1443	1370	0.93	94.94%
sf_276	778	336	1703	1463	0.74	85.91%
sf_274	781	336	1727	1494	0.87	86.51%
con1_216	954	415	2278	2086	1.50	91.57%
rd53_130	1043	448	2375	2067	1.39	87.03%
f2_232	1206	525	2681	2421	1.65	90.30%
rd53_251	1291	564	3144	2576	1.77	81.93%
hwb5_53	1336	598	3126	2606	1.52	83.37%
radd_250	3213	1405	8117	7227	8.81	89.04%
rd73_252	5321	2319	13349	12014	18.87	90.00%
cycle10_2_110	6050	2648	15654	13649	25.44	87.19%
hwb6_56	6723	2952	15779	13848	22.13	87.76%
cm85a_209	11414	4986	30371	27202	72.29	89.57%
rd84_253	13658	5960	36445	32626	115.99	89.52%
root_255	17159	7493	44265	41138	204.43	92.94%
mlp4_245	18852	8232	52128	44865	205.33	86.07%
urf2_277	20112	10066	58598	50031	241.43	85.38%
sym9_148	21504	9408	51730	45903	211.39	88.74%
hwb7_59	24379	10681	57679	51226	240.25	88.81%
clip_206	33827	14772	91616	81390	663.92	88.84%
sym9_193	34881	15232	91036	82224	651.67	90.32%
dist_223	38046	16624	99497	92225	829.44	92.69%
sao2_257	38577	16864	107369	93956	989.69	87.51%

Circuit Name	Original #Gates	Original #CNOT	#Added Alg. In [15]	#Added Ours	Running Time(s)	Comparison
urf5_280	49829	23764	133827	122929	1216.33	91.86%
urf1_278	54766	26692	153709	141286	1666.36	91.92%
sym10_262	64283	28084	171519	153981	2467.67	89.77%
Summary	485117	218181	1284512	1151566	#	89.65%

TABLE V: Comparison between our algorithm and the algorithm in [15] on IBM QX5.

Circuit Name	Original #Gates	Original #CNOT	#Added Alg. In [15]	#Added Ours	Running Time(s)	Comparison
graycode6_47	5	5	0	0	0.00	0/0
xor5_254	7	5	0	0	0.00	0/0
ex1_226	7	5	0	0	0.00	0/0
4gt11_84	18	9	0	0	0.00	0/0
ex-1_166	19	9	0	0	0.00	0/0
ham3_102	20	11	0	0	0.00	0/0
4mod5-v0_20	20	10	9	0	0.00	0.00%
4mod5-v1_22	21	11	9	0	0.00	0.00%
mod5d1_63	22	13	0	0	0.00	0/0
4gt11_83	23	14	12	0	0.00	0.00%
4gt11_82	27	18	12	3	0.00	25.00%
rd32-v0_66	34	16	0	0	0.00	0/0
mod5mils_65	35	16	9	0	0.00	0.00%
4mod5-v0_19	35	16	9	0	0.00	0.00%
rd32-v1_68	36	16	0	0	0.00	0/0
alu-v0_27	36	17	3	6	0.01	200.00%
3_17_13	36	17	0	0	0.00	0/0
4mod5-v1_24	36	16	12	0	0.00	0.00%
alu-v1_29	37	17	3	6	0.01	200.00%
alu-v1_28	37	18	3	6	0.01	200.00%
alu-v3_35	37	18	3	6	0.01	200.00%
alu-v2_33	37	17	9	6	0.01	66.67%
alu-v4_37	37	18	3	6	0.01	200.00%
miller_11	50	23	0	0	0.00	0/0
decod24-v0_38	51	23	0	0	0.00	0/0
alu-v3_34	52	24	3	6	0.01	200.00%
decod24-v2_43	52	22	0	0	0.00	0/0
mod5d2_64	53	25	12	12	0.01	100.00%
4gt13_92	66	30	18	0	0.00	0.00%
4gt13-v1_93	68	30	18	0	0.00	0.00%
one-two-three-v2_100	69	32	9	9	0.01	100.00%
4mod5-v1_23	69	32	12	9	0.02	75.00%
4mod5-v0_18	69	31	9	9	0.01	100.00%
one-two-three-v3_101	70	32	15	6	0.01	40.00%
4mod5-bdd_287	70	31	15	6	0.01	40.00%
decod24-bdd_294	73	32	21	15	0.02	71.43%
4gt5_75	83	38	15	15	0.02	100.00%
alu-v0_26	84	38	21	9	0.01	42.86%
rd32_270	84	36	18	12	0.01	66.67%
alu-bdd_288	84	38	45	24	0.03	53.33%
decod24-v1_41	85	38	18	15	0.02	83.33%
4gt5_76	91	46	27	15	0.02	55.56%
4gt13_91	103	49	6	15	0.02	250.00%
4gt13_90	107	53	9	27	0.02	300.00%
alu-v4_36	115	51	36	15	0.02	41.67%

Circuit Name	Original #Gates	Original #CNOT	#Added Alg. In [15]	#Added Ours	Running Time(s)	Comparison
4gt5_77	131	58	36	9	0.04	25.00%
one-two-three-v1_99	132	59	39	12	0.02	30.77%
rd53_138	132	60	39	27	0.05	69.23%
one-two-three-v0_98	146	65	27	24	0.04	88.89%
4gt10-v1_81	148	66	33	27	0.04	81.82%
decod24-v3_45	150	64	39	15	0.04	38.46%
aj-e11_165	151	69	24	18	0.02	75.00%
4mod7-v0_94	162	72	39	12	0.04	30.77%
alu-v2_32	163	72	39	15	0.02	38.46%
4mod7-v1_96	164	72	42	18	0.03	42.86%
cnt3-5_179	175	85	87	15	0.04	17.24%
mod10_176	178	78	36	24	0.03	66.67%
4gt4-v0_80	179	79	78	24	0.05	30.77%
4gt12-v0_88	194	86	21	21	0.05	100.00%
0410184_169	211	104	75	12	0.02	16.00%
4_49_16	217	99	69	36	0.04	52.17%
4gt12-v1_89	228	100	93	24	0.03	25.81%
4gt4-v0_79	231	105	96	12	0.04	12.50%
hwb4_49	233	107	45	33	0.04	73.33%
4gt4-v0_78	235	109	99	15	0.05	15.15%
mod10_171	244	108	60	24	0.04	40.00%
4gt12-v0_87	247	112	123	6	0.01	4.88%
4gt12-v0_86	251	116	123	9	0.02	7.32%
4gt4-v0_72	258	113	90	42	0.05	46.67%
4gt4-v1_74	273	119	114	78	0.10	68.42%
mini-alu_167	288	126	75	33	0.06	44.00%
one-two-three-v0_97	290	128	66	66	0.09	100.00%
rd53_135	296	134	48	54	0.11	112.50%
ham7_104	320	149	102	81	0.08	79.41%
decod24-enable_126	338	149	81	87	0.14	107.41%
mod8-10_178	342	152	162	21	0.04	12.96%
4gt4-v0_73	395	179	177	42	0.05	23.73%
ex3_229	403	175	174	18	0.05	10.34%
mod8-10_177	440	196	135	39	0.05	28.89%
alu-v2_31	451	198	63	54	0.07	85.71%
C17_204	467	205	114	96	0.14	84.21%
rd53_131	469	200	87	90	0.17	103.45%
alu-v2_30	504	223	105	45	0.07	42.86%
mod5adder_127	555	239	87	51	0.11	58.62%
rd53_133	580	256	159	105	0.16	66.04%
majority_239	612	267	123	84	0.13	68.29%
ex2_227	631	275	270	96	0.22	35.56%
cm82a_208	650	283	222	84	0.14	37.84%
sf_276	778	336	384	24	0.05	6.25%
sf_274	781	336	381	24	0.04	6.30%
con1_216	954	415	375	192	0.39	51.20%
rd53_130	1043	448	390	171	0.31	43.85%
f2_232	1206	525	225	213	0.43	94.67%
rd53_251	1291	564	309	204	0.34	66.02%
hwb5_53	1336	598	210	174	0.30	82.86%
radd_250	3213	1405	1647	669	2.34	40.62%
rd73_252	5321	2319	2115	1065	5.31	50.35%
cycle10_2_110	6050	2648	2424	1296	6.25	53.47%
hwb6_56	6723	2952	1719	1104	3.88	64.22%
cm85a_209	11414	4986	4173	2337	16.67	56.00%
rd84_253	13658	5960	5286	3246	37.18	61.41%

Circuit Name	Original #Gates	Original #CNOT	#Added Alg. In [15]	#Added Ours	Running Time(s)	Comparison
root_255	17159	7493	5601	3525	48.44	62.94%
mlp4_245	18852	8232	6462	4116	66.49	63.70%
urf2_277	20112	10066	8205	5934	77.04	72.32%
sym9_148	21504	9408	6438	2172	31.55	33.74%
hwb7_59	24379	10681	6378	4602	63.92	72.15%
clip_206	33827	14772	12624	6843	173.59	54.21%
sym9_193	34881	15232	11454	6441	144.55	56.23%
dist_223	38046	16624	12834	6936	187.64	54.04%
sao2_257	38577	16864	11742	7827	233.36	66.66%
urf5_280	49829	23764	20436	13065	374.76	63.93%
urf1_278	54766	26692	24600	15678	649.03	63.73%
sym10_262	64283	28084	20115	11697	518.75	58.15%
hwb8_113	69380	30372	35376	14976	646.41	42.33%
Summary	554497	248553	206142	116487	#	56.51%

TABLE VI: Comparison between our algorithm and the algorithm in [15] on IBM Q20.