Quantum Circuit Transformation Based on GraphQL and Tabu Search⋆

First Aaaaaaauthor[1][0000−ereer1111−2222−3333], Second Author[2,3][1111−2222−3333−4444], and Third Author[3][2222−−3333−4444−5555]

[1] Princeton University, Princeton NJ 08544, USA
[2] Springer Heidelberg, Tiergartenstr. 17, 69121 Heidelberg, Germany
`lncs@springer.com`
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
`{abc,lncs}@uni-heidelberg.de`

Abstract. Since we are in the NISQ era, quantum systems are prone to interact with the surrounding environment to generate noise, which can overwhelm the signal in the circuit. One way to eliminate errors is to use quantum error correction. Due to the limitation of circuit size, the computing power of NISQ technology is limited. Quantum error correction brings heavy overhead in terms of the number of qubits and gates. Therefore, it is difficult to achieve scale expansion using quantum error correction. Currently, the quantum circuit we simulate ignores quantum noise. In addition, the operation between qubits of NISQ devices is limited, and only adjacent qubits can be operated. Therefore, a large number of modifications must be made to the logic circuit to adapt to physical limitations. It is vital to the success of quantum computing how to find an automated method to effectively transform any input quantum circuit into a quantum that satisfies the physical constraints imposed by the NISQ device and has a small overhead in terms of number of auxiliary gates, circuit depth or errors circuit. This paper mainly summarizes the advantages and disadvantages of current quantum algorithms, and adjusts the life cycle of qubits through preprocessing of quantum circuits to increase the parallelism of quantum circuits. Then, the GraphQL combined subgraph isomorphism algorithm is used to generate high-quality initial mapping, and finally a circuit transformation algorithm based on Tabu Search is proposed to obtain a multi-objective circuit that satisfies the constraints of logic circuits and physical circuits. For a benchmark composed of 159 circuits and IBM Q20, compared with the initial mapping based on the VF algorithm, our initial mapping has an average efficiency improvement of 22.26%. Our algorithm only needs 461 seconds to run all, and other state-of-the-art algorithms are difficult to handle large circuits.

Keywords: Quantum circuit transformation · Subgraph isomorphism · Initial mapping · Tabu Search

---

1   Introduction

From the discovery of quantum mechanics in the early 20th century to the present, quantum technology has been applied in practice, but large quantum computers have not yet been established, and most of the contributions of quantum information to computer science are still in the theoretical stage. In March 2017, IBM developed the first 5-qubits backend named IBM QX2. In June, it launched the second 16-qubits backend IBM QX3. It was launched in December. The revised versions of 5-qubits and 16-qubits are called IBM QX4 and IBM QX5 respectively. IBM Q provides the public with free quantum computer resources on the cloud. If we want to use these quantum computer resources, we must map quantum circuits to a given physical architecture and satisfy physical constraints. This requires a set of highly efficient and automatic mapping procedures. Quantum circuit transformation is an important part of quantum circuit compilation. The main idea is to convert the input logic circuit into a physical logic circuit and satisfy the constraints of the physical circuit.

There are currently five main methods for solving qubit allocation. The first method is to use the unit matrix factorization algorithm to rearrange the quantum circuit from the beginning while retaining the function of the input circuit[6, 13]. The second method is to convert the quantum circuit transformation problem into some existing problems, such as AI planning [21, 3], Integer Linear Programming (ILP)[1], Satisfiable Modulus Theory (SMT) [11], or using the ready-made tools for these problems to find acceptable results. But these methods may run for a long time and can only be applied to a small amount of qubit. And these tools cannot take advantage of some of the properties of quantum mapping. The third method is to use precise methods to construct output quantum circuits. This method is only suitable for simple quantum structures and cannot be extended to complex quantum structures[18]. The fourth method is to use the relevant conclusions of graph theory. [16] use the minimum linear arrangement problem in graph theory to model the problem of reducing the interaction distance. It divides a given circuit into several sub-circuits, and then applies the minimum linear arrangement problem respectively, and turns non-adjacent gates in the sub-circuits into adjacent circuits by adding auxiliary gates. Finally it uses the minimum linear permutation problem to find a permutation, and uses bubble sort to calculate the number of SWAP gates needed. In [5] and 2019 Matsuo A [10], a two-step approach is proposed to reformulate the subtasks of gate scheduling as a graph problem. According to the graph coloring problem and the maximum subgraph isomorphism, the SWAP operations are added to minimize its overhead. Both of them move a qubit from the initial position to the target position in the best possible path with minimal cost. The former defines a priority to get the initial mapping, and the latter is purely to solve the problem of position movement. They all divide the SWAP of qubits into three categories. The first is a movement that is beneficial to both qubits; the second is a qubit is advantageous, and the other qubit is not mapped; the third is that one qubit is advantageous and one qubit is harmful. Then they calculate the scores from the initial position to the target position according to the type, and

move. The fifth is to use heuristic search [24, 4, 7, 22, 18], the circuit mapping process hopes to find a minimum number of SWAPs, but it takes exponential time, so heuristic search is used to control the evaluation function to obtain an acceptable solution. [24] divides a given circuit into multiple layers, which can be implemented in a $CNOT$ constraint compatible manner. Then, for each of these layers, a respectively compatible mapping is determined, which requires as few additional gates as possible. The main idea is to determine the cheapest path from the root node to the target node (the path with the lowest cost). Since the search space is usually exponential, complex mechanisms are used to keep the paths considered as few as possible. [22] designs a heuristic search algorithm with a novel selection mechanism, in which in each step of the search process, we do not choose the lowest cost operation to be applied, but looks forward one step, and then chooses the best continuous operation operation. In this way, the algorithm can effectively avoid local minimum. And a pruning mechanism is introduced to reduce the size of the search space and ensure that the program terminates in a reasonable time. The time complexity of this algorithm is $O(|V|^4)$. [7] Proposes a SWAP-based search scheme SABRE. Comparing with previous search algorithms based on exhaustive mapping, this search scheme achieves an exponential acceleration of search complexity with the results of previous schemes. This fast search scheme ensures the scalability of SABRE to adapt to the large quantum equipment in the NISQ era. By introducing the attenuation effect in the heuristic cost function, different hardware compatible circuits are generated by switching the number of gates in the circuit according to the circuit depth. This makes SABRE suitable for NISQ devices with different characteristics and optimization goals. The routing algorithm implemented in $t\,|ket\rangle$[4] can ensure that any quantum circuit is compiled into any architecture. The algorithm is divided into four stages: decomposing the input circuit into time steps, determining the initial position, routing across time steps, and finally cleaning up. The heuristic method in $t\,|ket\rangle$ matches or is better than the results of other circuit mapping systems in terms of depth and total number of gates of the compiled circuit, and the running time is greatly reduced, allowing larger circuits to be routed. 2019 Tannu [20] proposed a Variation-aware Qubit Movement strategy, which takes advantage of the change in error rate and a change-aware qubit allocation strategy by trying to select the route with the lowest probability of failure. This strategy allocates program qubits to physical Qubits to take advantage of SWAPs in the error rate, thereby minimizing the use of links with high error rates.

In general, it can be used as an initial search algorithm to generate the initial solution. [14] uses a heuristic method to find the initial mapping, and uses IBM's compiler to benchmark. The preliminary results show that the cost can be reduced by up to 10% only by placing qubits that are different from the default position (trivial placement) only in the actual circuit instance on the actual NISQ architecture. Recently, a novel reverse traversal technique is proposed in [7], which selects the initial mapping method considering the whole circuit. In [22], an annealing algorithm is proposed to find favorable initial mapping. The

heuristic initial mapping generated by the scheme is unstable and can not be used in business. In [8], VF subgraph isomorphism algorithm is used to generate initial mapping. Compared with VF mapping, our algorithm based on GraphQL reduces the number of SWAP gates by 22.29% and the depth by 11.17%.

The biggest problem facing quantum information processing is the problem of quantum decoherence. The entanglement of the quantum system with the surrounding environment and quantum measurement will cause the disappearance of quantum coherence. Since it is now in the Noisyed Intermiat-Scale Quantum (NISQ) era, there are only dozens of qubits, and it is unrealistic to realize quantum error correction[15]. Quantum physical circuits are also limited. It is basically impossible to directly map logical circuit to physical architecture graph, and quantum gate operations can only be performed between adjacent qubits, so it is necessary to convert the circuit by adding auxiliary gates are used to satisfy logical and physical constraints, and this process may introduce a lot of errors, which brings a huge challenge to the program compilation, because noise will have a greater impact on the final circuit and may make the result meaningless.

The main contributions of this paper are as follows.

1. This paper summarizes the current status, problems, and breakthrough directions of the current quantum mapping work, and points out their advantages and disadvantages to existing solutions. On this basis, this paper proposes a combinated subgraph isomorphism initial mapping and Tabu Search heuristic SWAP search scheme GQLTS.
2. Considering that the quantum coherence time is very short, the longest coherence time of a superconducting quantum chip is still within 10us-100us, the time of a single quantum gate is about 20ns, the time of a 2-qubits gate is about 40ns, and the time of measurement operation is about 300ns-1us. In order to ensure that the quantum operation is completed in the coherent time, we preprocess the quantum circuit to increase the parallelism of the quantum and relatively reduce the depth of the output circuit.
3. It has been proved that the initial mapping has a great influence on the quantum circuit mapping, so we tried a variety of schemes, hoping to output the quantum circuit with the fewest auxiliary gates. In the end, we choose to use combined subgraph isomorphism to obtain a partial initial mapping set of qubits, and then use the vertex completion algorithm to map the unmapped qubits to the coupled graph. Compared with the VF2 subgraph isomorphism scheme, combined subgraph isomorphism algorithm performs better.
4. We propose a heuristic SWAP search scheme GQLTS based on Tabu Search, which can handle large circuits in a short time at a low cost. Compared with the previous precise search and heuristic schemes, this scheme can complete the circuit transformation in a shorter time, but GQLTS needs to be added compared to the wgtgraph algorithm Equal or more auxiliary gates. GQLTS can complete the search of the 159 circuits in this paper within a few minutes, but the heuristic search takes a few days to get all the results, and even the heuristic scheme may not get results when processing large circuits.

The rest of this paper is arranged as follows. Section 2 introduces the background of quantum computing and quantum information. Section 3 analyzes the problems that need to be dealt with the transformation of quantum circuits. Section 4 introduces our algorithm in detail. Section 5 introduces the experiment and results, and the last section summarizes the paper and discusses future research.

2   Background

This section introduces some basic knowledge of quantum computing and quantum information, and related symbols

2.1   Software conditions

Qubits. Classical information is stored in bit, quantum information is stored in qubit, a qubit has two states, marked as $|0\rangle$ or $|1\rangle$, qubit also can be in any linear superposition state $|\phi\rangle = a|0\rangle + b|1\rangle$, where $a^2 + b^2 = 1$, qubit is in the state $|0\rangle$ with the probability of a^2 and in the state $|1\rangle$ with the probability of b^2.

Quantum Gate. Any quantum gates acting on a qubit will change the qubit from a ground state to a superposition state. The measurement operation causes the superposition state qubit to collapse to the ground state, and any unitary operator on a single qubit can be written as a combination of the global phase and rotation operations on the qubit. Suppose U is a unitary operator on a single qubit, then there are real numbers $\alpha, \beta, \delta, \gamma$, such that U satisfies the following equation.

$$U = e^{i\alpha} R_z(\beta) R_y(\gamma) R_z(\delta) \tag{1}$$

Common quantum gate symbols and their matrices are shown in the Fig. 1. Physical qubit and logical qubit are represented by Q, q, respectively.

Quantum Circuit. The number of qubits used in quantum circuit is called circuit width w. The logic dependence graph (see Fig. 3) of a circuit is obtained by parallelizing and layering the circuit by topological sorting. The number of layers that can be executed in parallel is called the depth d of the circuit. As shown in the Fig. 2 is a logic circuit (LC) generated according to the *openQASM* program. Each line represents a qubit, and the gate operation on the line acts on the corresponding qubit. In this paper, circuits with a depth less than 100 are called small circuits, circuits with a depth greater than 1000 are called large circuits, and the rest are medium circuits. The execution order of the circuit is from left to right. The depth of the circuit (see Fig. 2) is 6 and the width is 5. Since the single quantum gate is *local* [17], it is not necessary to consider the single quantum gate in circuit transformation. In this paper, the initial mapping is generated by regarding qubit in circuit as node and 2-qubits gate as edge. The generated logical circuit architecture graph $\mathcal{AG}_L = (V_L, E_L)$ and physical
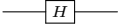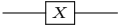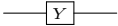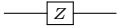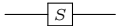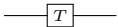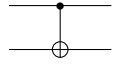
Hadamard gate	H	$\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Pauli-X gate	X	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y gate	Y	$\begin{bmatrix} 1 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z gate	Z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
phase gate	S	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
$\frac{\pi}{8}$ gate	T	$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}$
$CNOT$ gate		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

Fig. 1. The symbols of common quantum gates and their matrices

architecture graph $\mathcal{AG}_P = (V_P, E_P)$ are matched by subgraphs. As shown in Fig. 4(a) is the logical connection graph of the original circuit(see Fig. 2), and Fig. 4(b) is the partial architecture graph of IBM Q20.
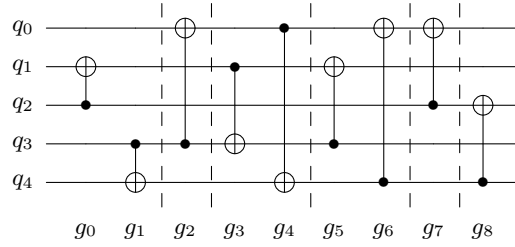
Fig. 2. Original circuit

2.2 Hardware Condition

Architectures This paper mainly discusses the physical circuit of IBM. Let $\mathcal{AG}_\mathcal{P} = (V_P, E_P)$ denote the architecture graph of the physical circuit, V_P denotes the physical qubit set, and E_P represents the directed edge that the $CNOT$
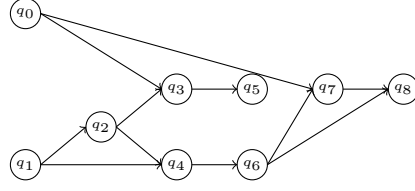
Fig. 3. The directed acyclic graph(DAG) of original circuit in Fig. 2
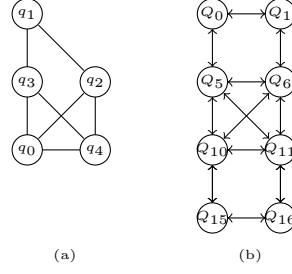
Fig. 4. (a)The architecture graph of original circuit in Fig. 2. (b) The partial architecture graph of IBM Q20
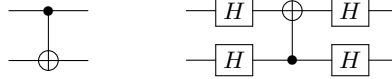
Fig. 5. Transformation of gate direction
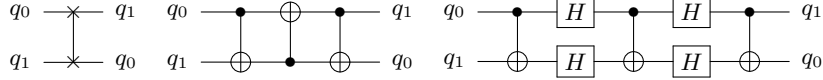
Fig. 6. Decomposition of a SWAP gate

gate can execute. As shown in the Fig. 7(a) and (b) are the physical architecture graph of the 5-qubits of IBM QX2, (c) and (d) are the physical architecture graph of 16-qubits of IBM QX3, and (e) are the physical architecture graph of IBM Q20 series. The arrow in the figure indicates that the qubit at the beginning of the arrow can control the qubit at the end of the arrow, and the 2-qubits gate operation can only be performed between qubit with edges connected. Since the quantum logic circuit does not consider physical constraints, any gate operation can be performed between two non-adjacent logic qubits, but IBM physical circuit only supports single quantum gate and $CNOT$ gate between two adjacent qubits. Therefore, before the circuit transformation, the circuit is simplified to

a circuit with only single quantum gate and $CNOT$ gate. This part of the work has been implemented in [12, 2]. [9] is proved that almost all gates of two qubits can be represented by general quantum gates. Generally, mapping logic qubit directly to physical qubit can not satisfy the limit of logic circuit, and circuit transformation is needed to make quantum gate can be executed on physical devices. This process is called circuit transformation in this paper. In this paper, we insert the auxiliary gate(SWAP)as shown in the Fig. 6, so that two non-adjacent qubits can be logically swap to adjacent positions, or change direction between two adjacent qubits (see Fig. 5). The introduction of auxiliary gate may lead to errors, which may lead to large deviation between the final results and the actual situation. The quantum system is easy to interact with the surrounding environment, resulting in errors. In the NISQ era, quantum error correction is difficult to achieve. Due to the decoherence problem of quantum, the quantum operation needs to be completed in the coherent period, and the time of quantum in the coherent state is very short. Therefore, it is necessary to improve the parallelism of qubits as much as possible to minimize the depth of quantum circuit. This is the focus of this paper. We hope to find a circuit mapping scheme with the minimum number of auxiliary gates and the circuit depth in the acceptable time.
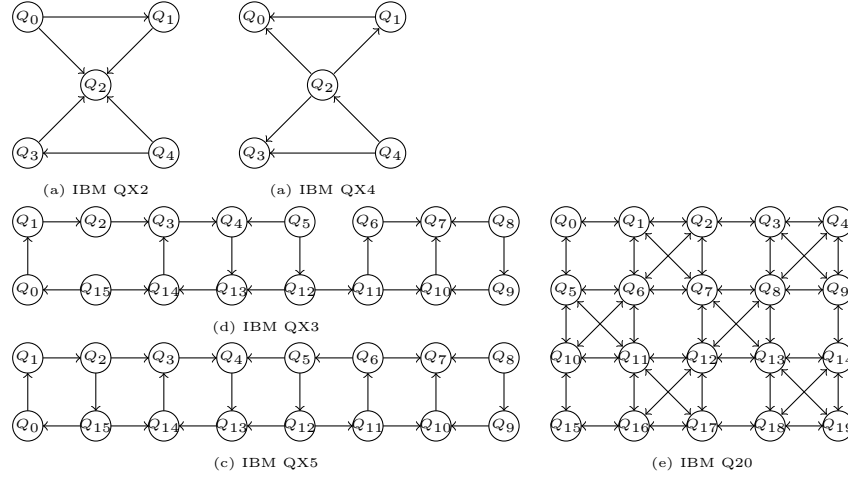
Fig. 7. IBM QX architectures

3   Problem Analysis

The goal of this paper is to find a circuit mapping scheme with the minimum number of auxiliary gates and the minimum circuit depth in the acceptable time, so that the input circuit can satisfy the physical circuit constraints.

Problem in qubit Mapping A quantum circuit transformation problem mainly includes the following four steps, among which the third step of *qubits assignment* and the fourth step of qubit routing problem are both NP-comple[18]

1. Convert programming language into logical quantum circuit.
2. Decomposes the circuit into basic gates. Single qubit gates and $CNOT$ gates are used as basic gates, because they are commonly used to implement any quantum circuit and are supported by the IBM QX architecture.
3. The initial mapping τ has an important impact on the subsequent addition of auxiliary gates. In this paper, we use the subgraph isomorphism algorithm to get a logical architecture graph that satisfies the quantum physical circuit as much as possible. Since most quantum logic architecture graph can not find a perfect mapping on the physical architecture graph, we try to satisfy as many nodes as possible.

   It will change the mapping relations τ that the original mapping inserts SWAP gates. Mapping is a injective function $\tau : q \to Q$. If two mappings are equal $\tau(q_i) = \tau(q_j)$, if and only if $i = j$.
4. The task of modifying the circuit to match the memory layout of a particular quantum computer is called the qubit routing problem [4]. Since quantum algorithms are usually designed without referring to the connectivity constraints of any specific hardware, routing problems need to be solved before the implementation of quantum circuits. Therefore, qubit routing forms a necessary stage of any compiler for quantum software.

   Given the logic circuit LC, physical structure $\mathcal{AG}_P$, and an initial mapping τ, $CNOT$ gate $g = \langle q_i, q_j \rangle$, if gate G is executable, then $\langle \tau(q_i), \tau(q_j) \rangle$ is a directed edge on $\mathcal{AG}_P$.

   Example 1. Fig. 4(a) is the logical structure of Fig. 2, Fig. 4(b) is the partial architecture graph of IBM Q20, and initial mapping is $\tau = \{q_0 \to Q_{10}, q_1 \to Q_0, q_2 \to Q_6, q_3 \to Q_5, q_4 \to Q_{11}\}$. $g_0 = \langle q_2, q_1 \rangle$ is executable, since $\langle \tau(q_2), \tau(q_2) \rangle = \langle Q_5, Q_0 \rangle$ exists in $\mathcal{AG}_P$. But $g_3 = \langle q_1, q_3 \rangle$ is not executable, since $\langle \tau(q_1), \tau(q_3) \rangle = \langle Q_0, Q_6 \rangle$ does not exist in $\mathcal{AG}_P$.

4 Solution

The program proposed in this paper mainly includes preprocessing, initial mapping, and minimum SWAP algorithm based on Tabu Search and A^* algorithm.

4.1 Preprocessing

Before the transformation of the SWAP circuit based on Tabu Search, we need to preprocess it to get more convenient data to shorten our search time and space. In the preprocessing stage, we first adjust the circuit of the input openQASM program to shorten the life cycle of qubits, then convert the openQASM code into a layered form, and generate a logical dependency graph(DAG) and logical architecture graph(LCG), and then read the physical architecture graph into the memory in the specified format, then use BFS to calculate the shortest distance between each node on the architecture graph.

Circuit Adjustment In order to shorten the life cycle of qubits and improve the parallelism of qubits, we use a layered method [23] to analyze the life cycle of qubits, and pack the operations that can be executed in parallel into a *bundle*, forming a layered bundle format. A conversion method is designed to use the layered bundle format to determine which operations can be moved, which reduces the life cycle of these qubits. The algorithm reduces the error rate of quantum programs by 11%. On most quantum workloads, the longest qubit lifetime and the average qubit lifetime can be reduced by more than 20%, and the execution time of some quantum programs can also be reduced.

Shortest Distance As long as the physical architecture graph is determined, the shortest distance between two qubits can be calculated. In this paper, the shortest distance matrix $dist[i][j]$ is calculated by $Floyd-Warshall$ algorithm, which represents the shortest distance from Q_i to Q_j, and the distance of each edge is 1. For IBM QX2, QX3, QX4, QX5, the SWAP operation needs 7 gates (3 $CNOT$ gates and 4 H gates). Only 4 H gates are needed to change direction between two adjacent qubits. For a $CNOT$ gate $\langle q_i, q_j \rangle$, and two qubits are mapped to Q_i and Q_j respectively, $\tau(q_i) = Q_i$, $\tau(q_j) = Q_j$, then the cost of executing g under the shortest distance path is $cost_{cnot}(q_i, q_j) = 7 \times (dist[i][j] - 1)$. If they move to adjacent positions, but there is no edge from Q_i to Q_j, they need to add 4 H gates to adjust their directions. Then the cost between them is $cost_{cnot}(q_i, q_j) = 3 \times (dist[i][j] - 1)$. In IBM Q20 structure, all the edges are bidirectional. The SWAP operation requires 3 gates (3 $CNOT$ gates), and there is no need to change the direction. Then the calculation formula for IBM Q20 structure is $cost_{cnot}(q_i, q_j) = 3 \times (dist[i][j] - 1)$ The time complexity of this step is $O(N^3)$.

Example 2. Taking the QX5 structure as an example, suppose there is a $CNOT$ gate $g = \langle q_i, q_j \rangle$, q_i is mapped to Q_1, q_j is mapped to Q_{14}, and the shortest distance between them is $dist[1][14] = 3$. There are three shortest paths to move Q_1 to the adjacent position of Q_{14}: $\Pi = \{\pi_0, \pi_1, \pi_2\}$ $\pi_0 = Q_1 \to Q_2 \to Q_3 \leftarrow Q_{14}$, $\pi_1 = Q_1 \to Q_2 \to Q_{15} \to Q_{14}$, $\pi_2 = Q_1 \to Q_0 \to Q_{15} \to Q_{14}$. Their costs are $cost_{\pi_0} = 18$, $cost_{\pi_1} = 14$, $cost_{\pi_2} = 14$, respectively.

Circuit Layering The SWAP minimization algorithm based on Tabu Search or the A^* algorithm traverses each level of quantum gate search that can be executed in parallel. Thus, we layer the adjusted circuit, traverse the entire program sequentially, and add gates that can be executed in parallel to one layer, otherwise a new layer is added. The $CNOT$ gate is represented by $\langle q, q' \rangle$, q is the control qubit, and q' is the target qubit. $L(LC) = \{\mathcal{L}_0, \mathcal{L}_1, ..., \mathcal{L}_n\}$ represents the layered circuit, $\mathcal{L}_i$, $(0 \leq i \leq n)$ represents a quantum gates set that can be executed in parallel. The quantum gates set separated by the dotted line in the Fig. 2. $\mathcal{L}_0 = g_0, g_1, \mathcal{L}_1 = g_2, \mathcal{L}_2 = g_3, g_4, \mathcal{L}_3 = g_5, g_6, \mathcal{L}_4 = g_7, \mathcal{L}_5 = g_8$.

At the same time, we generate circuit architecture graph $\mathcal{AG}_{\mathcal{L}} = (V_L, E_L)$, which is an undirected graph, V_L contains the vertex and the degree of the vertex, and E_L represents the undirected edge that the $CNOT$ gate can execute.

4.2   Initial Mapping

It has been proved that the initial mapping has an important influence on *qubits assignment*, and the subgraph isomorphism can be reduced to *qubits assignment*, so we want to use the subgraph isomorphism algorithm to find an initial mapping which is closer to the optimal. In the physical architecture graph, it is almost impossible to find a subgraph that exactly matches the logical architecture graph, so we hope to find a partial mapping that can maximize the match. $SubgraphCompare$[19] compares several current better subgraph isomorphism algorithm combinations, it shows that using the filtering and sorting ideas of GraphQL algorithm to process candidate nodes, the local candidate calculation method based on set intersection to enumerate the results is the best. Since $SubgraphCompare$ used in this paper is only suitable for fully connected subgraph isomorphism, there may be no 2-qubit gate operation between one qubit and other qubits in our circuit. The architecture graph formed in this way cannot use $SubgraphCompare$ of this paper to generate part of the initial mapping, because the subgraph matching will first match the node with the largest degree, and we hope to minimize the impact on the logical dependency graph. Therefore, we artificially connect the qubit with a degree of 0 in the logic architecture graph to the qubit with the largest degree.

We use the algorithm (GraphQL) recommended in [19] for partial subgraph isomorphism. The logical architecture graph $\mathcal{AG}_{\mathcal{L}}$ and the physical architecture graph $\mathcal{AG}_{\mathcal{P}}$ generated by the preprocessing process are regarded as an undirected graph as input, and then the combined subgraph isomorphism algorithm $SubgraphCompare$ is executed. The output of the improved combined subgraph isomorphism algorithm is a file containing all the isomorphism processes. Since only a small number of nodes may be matched during the isomorphism process, we finally select only the case with the most isomorphism nodes as the result of the combined subgraph isomorphism. Then we complete the unmapped nodes in the partial mapping based on the connectivity of the nodes or the degree of the nodes. The mapping completion algorithm based on node connectivity is shown in Algorithm 1.

The input of Algorithm 1 is a target graph ($\mathcal{AG}_P$), query graph ($\mathcal{AG}_L$), and the current mapping relations T. First initialize an empty queue Q, which stores unmatched nodes in the map $\tau \in T$. Then it traverses τ and adds the unmatched nodes to the queue. The remaining unmatched points we want to try to map them with the nodes that are not matched in the more concentrated area of $\mathcal{AG}_P$. That is, the final mapping relations can be a dense in the target graph, which can reduce subsequent SWAP operations. At the beginning, we also tried to randomly match the remaining unmatched nodes, but this may lead to isomorphism to a position far away from other nodes, adding subsequent auxiliary gate operations. In the query graph, if the unmatched point has an edge

adjacent to the matched point, it will be matched to its adjacent position first, and if the adjacent position has been matched, it will be matched to the adjacent unmatched node. Finally it gets all the processed candidate initial mappings and outputs them to a file.

Lines 2-7 are to calculate the maximum number of qubits ml that can be matched in the mapping relations between logical qubits and physical Qubits obtained by the $SubgraphCompare$ algorithm. Lines 8-49 are to complete the logical qubit unmapped nodes in the mapping scheme with the number of matches equal to ml in the mapping relations, and we use the greedy strategy to allocate. Line 11, we initialized an empty queue Q, which stores unmapped logical qubits. Lines 12-18, we will traverse the map and add the unmapped qubit to Q. Line 20, loop until Q becomes empty, and all logical qubits are mapped to physical Qubits. We take out the first element in Q. Line 21 and 22 are respectively to get the adjacency matrix of $\mathcal{AG}_P$ and $\mathcal{AG}_L$. Line23 is to initializes an empty map tm, and the keys are sorted in descending order. The key consists of the number of nodes that are connected to qId in the adjacency matrix and have been mapped in the current mapping scheme and the nodes constitute a unique key. Lines 25-31 are to traverse the point m connected to qId in the adjacency matrix. If the node m has not been mapped in the map mapping, the node is stored in the tm. Line 32-47 is to traverse the tm, select the node with the largest number of connections to qId in the tm, and it has been mapped to the node ($tm.firstValue$) on the physical architecture graph. The tId in line 33 is the node with the largest number of qId connections corresponding to the node on the physical architecture graph. Line 35 is to remove the object to be matched in the tm from the tm. Lines 36-43 are to select the node adjacent to the tId in the adjacency matrix of the tId, and map the qId to the node.

Example 3. Following the previous example, we first use the combined subgraph isomorphism algorithm for the logical architecture graph (see Fig. 4(a)) and physical architecture graph (see Fig. 7(e)) to obtain the partial mapping result set $T = \{\tau_0, \tau_1, ..., \tau_n\}$, and then we use one of the partial mapping as an example $\tau_i = \{q_0 \rightarrow Q_{10}, q_1 \rightarrow -1, q_2 \rightarrow Q_6, q_3 \rightarrow Q_5, q_4 \rightarrow Q_{11}\}$, $0 \leq i{<}n$. $q_1 \rightarrow -1$ means that q_1 is not mapped to the physical structure in the subgraph isomorphism stage, so we need to perform mapping completion. The algorithm 1 completion strategy is to find the part of the mapping with the maximum mapped nodes in T and perform the mapping completion as the initial mapping. In this example, the maximum number of mapped nodes is four. Next, we demonstrate that τ_i is mapped and completed, and the unmapped nodes in τ_i are added to the queue Q, $Q = \{q_1\}$, and the loop ends until Q is empty. We Take the first element of Q and put it in qId, then get the adjacency matrix of the query graph and the target graph, traverse the node q_m connected to qId in the adjacency matrix, if the node q_m is matched, then we put q_m into the map, The key of the map is the number of connections between q_m and qId + "-" + $\tau_i(q_m)$, the value is q_m, and the keys are sorted in descending order. Then we get $tm = [\{'0 - 5', q_3\}, \{'-1 - 6', q_2\}, \{'-1 - 11', q_4\}, \{'-1 - 10', q_0\}]$, Then traverse the tm and take out the $value = q_3$ of the elements in the tm, calculate the

---

Algorithm 1: initial mapping algorithm based on GraphQL

Input: $\mathcal{AG_L}$: The architecture of logical circuit
$\mathcal{AG_P}$: The architecture of physical circuit
T: A partial mapping set obtained by $SubgraphCompare$
Output: result: A collection of mapping relations between $\mathcal{AG_L}$ and $\mathcal{AG_P}$

1 **Initialize** result= $\emptyset$;
2 $ml \leftarrow 0$; The most mapped length in mappings
3 **for** $\tau \in T$ **do**
4 | **if** $ml < \tau.length$ **then**
5 | | $ml \leftarrow \tau.\text{length}$;
6 | **end**
7 **end**
8 **for** $\tau \in T$ **do**
9 | **if** $ml{=}\tau.length$ **then**
10 | | result.add(τ);
11 | | $Q \leftarrow$ initial an empty unmapped node Queue
12 | | $i \leftarrow 1$;
13 | | **while** $i{\leq}\tau.length$ **do**
14 | | | **if** $\tau[i]{=}{-}1$ **then**
15 | | | | $Q \leftarrow i$;
16 | | | **end**
17 | | | $i \leftarrow i + 1$;
18 | | **end**
19 | | **while** Q *is not empty* **do**
20 | | | int $qId \leftarrow$ Q.poll();
21 | | | $targetAdj \leftarrow \mathcal{AG_P}.$adjacencyMatrix();
22 | | | $queryAdj \leftarrow \mathcal{AG_L}.$adjacencyMatrix();
23 | | | $tm \leftarrow$ initial an empty map
24 | | | $m \leftarrow 1$;
25 | | | **while** $m \leq queryAdj[qId].length$ **do**
26 | | | | **if** $\tau[m]{\neq}{-}1$ **then**
27 | | | | | $tm \leftarrow tm\cup$
28 | | | | | $\{queryAdj[qId][m] + " - " + \tau[m], m\}$;
29 | | | | **end**
30 | | | | $m \leftarrow m + 1$;
31 | | | **end**
32 | | | **while** tm *is not empty* **do**
33 | | | | $tId \leftarrow \tau[$tm.firstValue$]$;
34 | | | | $k \leftarrow 0$;
35 | | | | $tm \leftarrow tm\backslash tm.first()$;
36 | | | | **while** $k < targetAdj[tId].length$ **do**
37 | | | | | **if** $(targetAdj[tId][k] \neq {-}1$ or $targetAdj[k][tId] \neq {-}1)$
38 | | | | | *and not $\tau.contains(k)$* **then**
39 | | | | | | $\tau[qId] \leftarrow k$;
40 | | | | | | break;
41 | | | | | **end**
42 | | | | | $k \leftarrow k + 1$;
43 | | | | **end**
44 | | | | **if** $k \neq targetAdj[tId].length$ **then**
45 | | | | | break;
46 | | | | **end**
47 | | | **end**
48 | | **end**
49 | **end**
50 **end**

value $tId = Q_5$ of q_m in the current mapping $\tau_i(q_3)$, and map qId to the node connected to tId and not yet mapped. If the nodes connected to tId have been mapped, the loop continues. In this example, it can be directly mapped to Q_0. In the end, we get $\tau_i = \{q_0 \to Q_{10}, q_1 \to 0, q_2 \to Q_6, q_3 \to Q_5, q_4 \to Q_{11}\}$.

4.3   Swap Minimization

Tabu Search Tabu Search algorithm is a type of heuristic algorithm. The disadvantage of heuristic algorithm is that it thinks that the local optimal is the global optimal. There is a Tabu table in the Tabu Search to avoid falling into the local optimal. The circuit transformation operation in this paper mainly relies on the idea of tabu search algorithm, aiming to solve the large-scale circuit that the current algorithm is difficult to handle, and hope to get a result closer to the optimal solution in a short time. There are mainly the following objects in Tabu Search: neighborhood fields, neighborhood action, Tabu table, candidate set sum, Tabu object, evaluation function, and amnesty rules. All the edges that can be swapped in the current map are the neighborhood fields in Tabu thinking. The recently reached state is added to the Tabu table, and objects in the Tabu table will not be searched as much as possible. The Tabu table in this paper is just in line with our parallel thinking. In order to increase the parallelism of quantum circuits, we will give priority to inserting auxiliary operations that can be executed in parallel. We try not to use the recently operated qubits as much as possible, which are added to the Tabu table, in the same time. The candidate set is to select several neighborhood objects with the best target value or evaluation value from the neighborhood fields to join the candidate set. It can be obtained by observation that many SWAP operations are meaningless, so in order to save search space, we should perform pruning. Only the swap of edges adjacent to the gate node with at least one edge is meaningful, so our neighborhood fields is the shortest path on the physical architecture graph of the gates. Their qubits are not adjacent to the current layer, and the edges on these shortest paths are all part of the neighborhood fields. The Tabu object is the object in the Tabu table. The evaluation function is to select an SWAP evaluation formula from the candidate set, generally taking the objective function as the evaluation function. The evaluation function is to satisfy some gate mapping operations, and the number of SWAP gates added should be small, and the depth of the entire circuit should be small. The amnesty rule is that when all objects in the candidate set are Tabu, or after one object is Tabu, the target value will be greatly reduced. In order to achieve the global optimal, the Tabu object can be added to the candidate set. As shown in Algorithm 3.

The calculation of the neighborhood fields is shown in Algorithm 2. The input are the current circuit mapping parent τ_p, physical Qubits to logical qubits mapping *qubits*, logical qubits to physical Qubits mapping *locations*, the current layer list of all gates currentLayers cl, all the gates nextLayer of the next layer nl, and the output is a candidate set of the current mapping, The mapping generated by a transformation, where the physical mapping and the logical mapping are changed synchronously. E_w is the edge of all the shortest paths in the physical

architecture graph of all gates in the current layer. The weight of the edge is the number of times each edge appears in the path. If there are multiple edges reachable in the same shortest distance, we will choose the path with the largest total weight in all the shortest paths, in the lines 8-18. Lines 22-37 is to swap all the edges of this path and add them to the candidate set, and calculate the cost of each candidate.

Example 4. Under the mapping $\tau_i = \{q_0 \rightarrow Q_{10}, q_1 \rightarrow 0, q_2 \rightarrow Q_6, q_3 \rightarrow Q_5, q_4 \rightarrow Q_{11}\}$, for $L_0 = \{g_0, g_1\}$, $dist_{cnot}(g_0) = 7$, $dist_{cnot}(g_1) = 7$. Gate g_1 can be executed directly under the τ_i mapping, so it is directly deleted from L_0. But g_0 cannot be executed under the mapping τ_i, so circuit transformation is required. First, we need to calculate the field, and add the edge of the shortest path of the gate that cannot be executed in L_0 to the number of occurrences of the same edge of E_w as the weight of the edge. Now the gate that cannot be executed in L_0 is g_0. Its shortest path is $T = \{\{Q_6 \rightarrow Q_1 \rightarrow Q_0\}, \{Q_6 \rightarrow Q_5 \rightarrow Q_0\}\}$ $E_w = \{\langle Q_6, Q_1 \rangle, \langle Q_1, Q_0 \rangle, \langle Q_6, Q_5 \rangle, \langle Q_5, Q_0 \rangle, \}$, and then traverse the shortest path to calculate the path with the highest weight, $w_{\pi_0} = 2$, $w_{\pi_0} = 2$. The two weights are equal to take the former. Then the edges on the swap path are added to the candidate set, so the current candidate set is $\{SWAP(Q_6, Q_1), SWAP(Q_1, Q_0)\}$.

The circuit mapping algorithm based on Tabu Search takes a layered circuit and an initial mapping as input, and outputs a circuit that can be executed on the specified architecture graph. Algorithm 3 performs a Tabu Search on the gates of each layer of the layered circuit, and obtains the transformed circuit of each layer. The transformed circuit mapping of each layer is used as the initial mapping of the circuit of the next layer. Lines 2 and 3 regard the initial mapping as the best mapping and the current mapping. Lines 5 to 17 loop to check whether the current mapping satisfies the execution of all gates of the current layer. If it is not satisfied or the number of iterations has not reached the set maximum number, the loop will continue, otherwise the program will terminate and the circuit is not found in the physical device on the way of execution. The sixth line is to get the neighborhood fields of the current mapping, and the seventh line is to find the best mapping in the candidate set. The mapping will first remove the overlapping elements of the candidate set and the Tabu table, and then in the remaining candidates choose the mapping with the lowest cost. Lines 8 to 13 are the amnesty rules. When the best candidate is not found, the candidate set elements are all the same as the Tabu table elements, then the amnesty rule needs to be executed to find the candidate elements. An optimal mapping, as the next initial mapping, is to select the candidate with the lowest cost in the candidate set. Lines 14-16 are to update the best mapping and the current mapping with the selected mapping, and add the SWAPs operation performed by the best mapping to the Tabu table, indicating that this SWAPs has just been performed, and the algorithm should try to avoid re-swap the just swapped qubits. Then it will judge whether the program stop condition is reached. The program stopping condition is to determine whether the number of iterations has

---

Algorithm 2: Calculate the neighborhood fields

Input: p: The current circuit's solution
$dist$: The shortest paths of physical architecture
$qubits$: The mapping from physical Qubits to logic qubits
$locations$: The mapping from logic qubits to physical Qubits
cl: Gates included in the current layer of circuits
nl: Gates included in the next layer of circuits
Output: $results$: The set of candidate solution

1 **Initialize** $results \leftarrow \emptyset$;
2 $E_w \leftarrow$ Calculate the weight of each edge
3 **for** $l \in cl$ **do**
4 $l_1 \leftarrow locations[l.control]$;
5 $l_2 \leftarrow locations[l.target]$;
6 $distance \leftarrow dist[l_1][l_2]$;
7 $ml \leftarrow 0; sum \leftarrow 0 ; path\_index \leftarrow -1$;
8   **for** $path \in distance.paths$ **do**
9   **for** $edge \in path$ **do**
10   **if** $E_w.contains(edge)$ **then**
11 $e \leftarrow E_w.findEqualWithEdge(edge)$;
12 $sum \leftarrow sum + e.weight$;
13   **end**
14   **end**
15   **if** $ml< sum$ **then**
16 $ml \leftarrow sum$;
17 $path\_index \leftarrow l.index$;
18   **end**
19   **end**
20   **if** $path\_index¿=0$ **then**
21 $j \leftarrow 1$;
22   **while** $j <distance.paths[path\_index].length$ **do**
23 $newQubits \leftarrow qubits$;
24 $newLocations \leftarrow locations$;
25 $q_1 \leftarrow newQubits[distance.paths[path\_index]][j].source$;
26 $q_2 \leftarrow newQubits[distance.paths[path\_index]][j].target$;
27   **if** $q_1 \neq$ -1 **then**
28 $newLocations[q_1] \leftarrow q_2$;
29   **end**
30   **if** $q_2 \neq$-1 **then**
31 $newLocations[q_2] \leftarrow q_1$;
32   **end**
33 $s \leftarrow \emptyset$;
34 $s.swaps \leftarrow p.swaps \cup \{distance.paths[path\_index][j]\}$;
35 $s.value \leftarrow computeEvaluateValue(dist, newLocations, cl, p)$;
36 $results \leftarrow results \cup s$;
37 $j \leftarrow j + 1$;
38   **end**
39   **end**
40   return $results$;
41 **end**

---

reached the maximum number, or the current mapping satisfies the execution of all gates in the current layer. If the stop condition not satisfied, continue to loop.

Example 5. We continue the previous example. Tabu Search requires an initial solution, and then searches based on this solution. Before searching, we need to get a series of initial candidate SWAP sets and select the one with the lower evaluation score as the initial solution. For $L_0 = \{g_0, g_1\}$, the initial candidate set is $\{SWAP(Q_6, Q_1), \ SWAP(Q_1, Q_0)\}$. $cost(SWAP(Q_6, Q_1)) = 3.0$, $cost(SWAP(Q_1, Q_0)) = 3.0$. The algorithm will choose to swap Q_6 and Q_1, at this time the mapping becomes $\tau_i = \{q_0 \rightarrow Q_{10}, q_1 \rightarrow Q_0, q_2 \rightarrow Q_1, q_3 \rightarrow Q_5, q_4 \rightarrow Q_{11}\}$. The Tabu Search loops to determine whether it is reached stop condition, when the current iteration number reaches the limits, or the current mapping satisfies the execution of all gates in L_0. It can be seen that the current mapping has satisfied the execution of g_0, thus the search of the current layer is over, and the Tabu Search of the next layer is continued.

Algorithm 3: Tabu Search

Input: $iniS$: The initial state
tl: Tabu list
Output: $bestS$: The final state and SWAPs

1   **Initialize** $bestS \leftarrow iniS$;
2   $currS \leftarrow iniS$;
3   $iter \leftarrow 1$;
4   **while** *not mustStop(++iter, bestS)* **do**
5      $C \leftarrow currS.neighbors$;   /* candidate set */
6      $bestN \leftarrow findBestNeighbor(C, tl)$;
7   **if** *bestN is empty* **then**
8   **if** $C = NULL$ **then**
9   *break*;
10   **end**
11        $bestN \leftarrow findAmnestyNeighbor(C, tl)$;
12   **end**
13      $bestS \leftarrow bestN$;
14      $currS \leftarrow bestN$;
15      $tl \leftarrow tl \cup \{currS\}$
16   **end**
17   return $bestS$;

Evaluation function design   The goal of this paper is mainly to consider the depth of or the size of the generation circuit, because the current quantum error correction cannot be applied in practice. The longer the quantum execution time, the greater the error introduced, so we want to shorten the life cycle

of qubits as much as possible. And in the algorithm based on Tabu Search, there is a Tabu list naturally, which just satisfies our needs. This paper tested two evaluation functions, one using the depth of the generating circuit as the evaluation criterion 5, and the other using the number of gates in the generating circuit as the evaluation criterion 4.

$$cost_{L_i}(SWAP(Q_i, Q_j)) = \sum_{g \in L}(dist[g.control][g.target]) \qquad (2)$$

$$cost_{L_i}(SWAP(Q_i, Q_j)) = Depth(L_i) \qquad (3)$$

Among them, $cost_{L_i}(SWAP(Q_i, Q_j))$ means that the evaluation value of Q_i, Q_j is being swap, We only calculate the depth of the unmapped gates of the after the SWAP as in the equation 5 or the distance between the unmapped gates as in the equation 4 .

Look ahead   Since the number of qubits used in current quantum circuits is small, the number of gates in each layer after layering is small. If we only consider the gates of one layer when choosing the swap scheme, the swap scheme selected by the algorithm only satisfies the gate execution of the i-th layer, but the output of the i-th layer is used as the input of the (i+1)-th (less than the total number of layers of the circuit) layer. We consider that the swap scheme of the i-th layer will affect the mapping of the (i+1)-th layer, so we add the circuit of the (i+d)th layer into the consideration. However, in terms of priority, it is necessary to give priority to the execution of the gate set of the i-th layer, so we introduce an attenuation factor δ, which controls the influence of the (i+d)th layer gate set on the circuit swap of the i-th layer. Each time the algorithm is swapped, the gates of the latter layer or several layers are considered together. Experiments show that $d = 2$, $\delta = 0.9$, the final effect is the best. Our evaluation function can be rewritten as

$$cost_h(SWAP(Q_i, Q_j)) = \sum_{g \in L_i}(dist[g.control][g.target])+$$
$$\delta \times \sum_{j=i}^{i+d} \sum_{g \in Lj}(dist[g.control][g.target]) \qquad (4)$$

$$cost_h(SWAP(Q_i, Q_j)) = Depth(L_i) + \delta \times Depth(\sum_{j=i}^{i+d} L_j). \qquad (5)$$

Complexity Suppose the given logic circuit structure diagram is $\mathcal{AG}_{\mathcal{L}} = (V_L, E_L)$, the physical circuit structure diagram is $\mathcal{AG}_{\mathcal{P}} = (V_P, E_P)$, the initial mapping is τ, the depth of the circuit is d, the number of qubits is V_L, TS searches each time Search the 2-qubits gate on the layer, and perform TS searches at most d times. Starting from the initial mapping, first delete the executable gates of the first layer under the initial mapping, and then add all the shortest

paths of the remaining k gates to the candidate set. The shortest path length is $(|E_P| - 1)$, the candidate set size is $(|E_P| - 1) * k$. Each SWAP will make the total distance between the doors smaller. In the worst case, the number of SWAPs is $((|E_P| - 1) * k)^{(|E_P|-2)*k}$, but our selection strategy will make this The number of SWAPs is significantly reduced. Our time complexity is $d * ((|E_P| - 1) * k)^{(|E_P|-2)*k}$, and the space complexity is the size of our candidate set $(E_P - 1) * k$.

5   Experiment

The experiment in this paper is performed on a Linux machine with a clocked frequency of 2.3GHz and a memory of 64G. This paper compares the combined subgraph isomorphism algorithm (GraphQL) and circuit transformation algorithm based on Tabu Search with the wghtgr in [8] and the heuristic algorithm A^* in [24].

First, we compared the efficiency of initial mapping on τ_{optm} [24], $\tau_{wghtgraph}$ [8] and τ_{GQLTS}. In order to observe the results of these two initial mapping algorithms intuitively, we used the same circuit transformation [24]A^* algorithm to compare the initial mapping algorithms. We tested 159 circuits. Experiments show that within five minutes τ_{optm} has 121 circuits get results, $\tau_{wghtgraph}$ has 106 circuits to get results, τ_{GQLTS} has 131 circuits to get results. Among them, there are 103 circuits the results are obtained. Comparison of $\tau_{wghtgraph}$ algorithm and τ_{GQLTS} algorithm, the $\tau_{wghtgraph}$ algorithm has 21 circuits with fewer SWAPs and 19 circuits with small depth, and the τ_{GQLTS} algorithm has 54 circuits with fewer SWAPs and 60 circuits with small depth, and they have 25 circuits with equal depth and 29 circuits with equal SWAPs. The SWAPs of the τ_{GQLTS} algorithm is relatively reduced by 22.4418%, and the depth is reduced by 11.2482%. Comparison of τ_{optm} algorithm and τ_{GQLTS} algorithm, the τ_{optm} algorithm has 1 circuits with fewer SWAPs and 2 circuits with small depth, and the τ_{GQLTS} algorithm has 99 circuits with fewer SWAPs and 98 circuits with small depth, and they have 4 circuits with equal depth and 4 circuits with equal SWAPs. The SWAPs of the τ_{GQLTS} algorithm is relatively reduced by 27.0219%, and the depth is reduced by 14.1242%. As shown in Table 1, there are 104 circuits. Three initial mappings are compared with the depth of the generated circuits under the A^* algorithm and the number of SWAP gates added.τ_{GQLTS}/τ_{optm} is to calculate the efficiency improvement of the former and the latter, the formula is $(n_{optm} - n_{GQLTS})/n_{optm}$.

	τ_{optm}	$\tau_{wghtgraph}$	τ_{GQLTS}	τ_{GQLTS}/τ_{optm}	$\tau_{GQLTS}/\tau_{wghtgraph}$
depth	168895	163422	145040	14.1241%	11.2482%
added	20439	19232	14916	27.0219%	22.4418%

Table 1. Compare τ_{optm}, $\tau_{wghtgraph}$, and τ_{GQLTS}

We compared the use of small depth priority(GQLTS_{dep}) and the added auxiliary gate less priority(GQLTS_{num}). The two indicators were used as objective functions, and 159 circuits were tested. The depth of the final circuit obtained by GQLTS_{num} is 1.93% smaller than GQLTS_{dep} on average, and the number of auxiliary gates added is 4.53% smaller on average. Adding a SWAP gate, the circuit need to add three CNOT gates, and the depth will be increased by 3. While the number of SWAP gates added is small, the circuit depth is also reduced accordingly. Thus we use SWAP quantity first to give better results.

Finally, we compared the use of GQLTS Search and wgtgraph algorithm. Since the wgtgraph algorithm only uses 2-qubits gates, it is impossible to compare the depth of the generated circuit, So we compared the number of SWAP gates added and compared with time. Since large circuits may not get results for a long time, we consider it meaningless. This paper sets a five-minute time-out period and tested 159 circuits. GQLTS_{num} Search only takes 461 seconds, GQLTS_{dep} takes 485 seconds, and wgtgraph run 159 circuits in 1908 seconds, but only 98 files get results, 64 of them there are 66 circuits for small circuits to get results, 49 medium circuits only have 35 circuits for results, and no circuit output in 44 large circuits. Although Tabu Search can quickly produce results on large circuits, in contrast, more auxiliary gates are added. In 98 small and medium-sized circuits with the results obtained by wgtgraph, the number of SWAP gates added by wgtgraph is 26.87% less than GQLTS_{num} on average, and the number of SWAP gates added by wgtgraph is 24.89% less than GQLTS_{dep} on average. Tabu Search can quickly output converted circuits on large circuits, but wgtgraph cannot get results in a short time. The detailed results of the circuit comparisons are in the appendix.

6   Conclusion

This paper proposes a heuristic SWAP method GQLTS based on Tabu Search to overcome the shortcomings of previous works, and proposes a combined subgraph isomorphism algorithm to generate high-quality initial mapping. Experimental results show that the initial mapping generated by GQLTS greatly reduces the number of SWAP gates inserted, and achieves multiple optimization goals in the search phase, and results can be obtained in a short time. Most small and medium-sized circuits can be obtained in a few seconds. The result can be obtained within a few minutes even for a large circuit, but the cost of insertion may be equal to or more than wghtgr. We introduce a lookahead plan to make each selected SWAP more in line with the constraints of the back gates. In future work, we will focus on reducing the number of auxiliary gates inserted as much as possible on the basis of increasing speed. In theory, our method is applicable to all NISQ devices, but further experiments are needed. Since our analog circuit ignores the noise generated by the circuit, we may introduce quantum noise for testing next to improve the fidelity of the circuit.

A   Experimental details of the SWAP gates added by the output circuit

Circuit name	qubit no.	CNOT no.	GQLTS$_{num}$ added	GQLTS$_{dep}$ added	optm added	wghtgr added
decod24-enable_126	6	149	28	42	60	16
4mod5-v0_19	5	16	0	0	0	0
4mod5-v0_18	5	31	2	5	4	4
mod5d2_64	5	25	5	6	8	3
4gt4-v0_72	6	113	14	10	33	14
alu-v3_35	5	18	2	4	8	2
4gt4-v0_73	6	179	27	34	76	12
alu-v3_34	5	24	2	3	7	2
3_17_13	3	17	0	0	6	0
4gt4-v0_78	6	109	12	8	48	4
4gt4-v0_79	6	105	17	17	48	3
4mod7-v1_96	5	72	16	19	27	7
mod10_171	5	108	17	20	39	9
ex2_227	7	275	48	59	121	33
mod10_176	5	78	14	14	38	8
0410184_169	5	9	2	2	49	3
4mod5-v0_20	5	10	0	0	4	0
aj-e11_165	5	69	8	8	33	7
alu-v1_28	5	18	2	4	11	2
4gt12-v0_86	6	116	28	33	48	3
4gt12-v0_87	6	112	27	32	45	2
4gt12-v0_88	6	86	5	5	25	4
alu-v1_29	5	17	4	4	11	2
ham7_104	7	149	28	34	68	12
C17_204	7	205	26	53	99	22
xor5_254	6	5	0	0	1	0
hwb4_49	5	107	14	15	38	11
rd73_140	10	104	23	26	35	20
decod24-v0_38	4	23	0	0	6	0
rd53_131	7	200	39	39	98	24
rd53_133	7	256	37	47	102	27
rd53_135	7	134	28	29	38	23
decod24-v2_43	4	22	0	0	9	0
rd53_138	8	60	14	16	23	9
rd32-v0_66	4	16	0	0	6	0
4gt13-v1_93	5	30	0	0	13	0
graycode6_47	6	5	0	0	0	0
4mod5-bdd_287	7	31	3	6	8	6
ham3_102	3	11	0	0	3	0
4gt4-v0_80	6	79	5	5	22	5
ex-1_166	3	9	0	0	3	0
mod5mils_65	5	16	0	0	6	0
0example	5	9	1	2	3	3
alu-v4_36	5	51	12	8	22	4
alu-v4_37	5	18	2	4	8	2
ex1_226	6	5	0	0	1	0
one-two-three-v0_98	5	65	11	13	32	10
one-two-three-v0_97	5	128	23	23	64	16
one-two-three-v3_101	5	32	3	4	14	3
rd32_270	5	36	3	3	6	6

Table 2. Compare the number of SWAP gates added by the output circuit on the IBM Q20

Circuit name	qubit no.	CNOT no.	GQLTS$_{num}$ added	GQLTS$_{dep}$ added	optm added	wghtgr added
rd53_130	7	448	89	100	190	49
rd53_251	8	564	104	131	230	45
4mod5-v1_24	5	16	0	0	3	0
mod5adder_127	6	239	21	56	111	20
4_49_16	5	99	20	17	40	10
hwb5_53	6	598	141	168	173	59
ex3_229	6	175	10	9	50	11
4gt10-v1_81	5	66	14	15	28	6
alu-v2_32	5	72	15	17	27	7
alu-v2_31	5	198	42	54	85	13
alu-v2_30	6	223	41	45	96	20
sf_276	6	336	12	52	138	12
decod24-v1_41	5	38	4	4	14	3
sf_274	6	336	34	21	82	12
4gt4-v1_74	6	119	17	24	37	9
alu-v2_33	5	17	4	4	8	2
cnt3-5_179	16	85	6	6	35	4
4mod5-v1_22	5	11	0	0	5	0
4mod5-v1_23	5	32	5	5	4	3
mini_alu_305	10	77	10	20	28	8
alu-v0_26	5	38	7	10	13	3
alu-bdd_288	7	38	4	12	16	6
alu-v0_27	5	17	2	4	11	2
4gt13_91	5	49	7	7	10	2
4gt5_77	5	58	12	12	20	6
4gt13_92	5	30	0	0	14	0
4gt5_76	5	46	7	10	24	5
4gt5_75	5	38	5	12	16	4
4gt12-v1_89	6	100	11	21	38	4
one-two-three-v1_99	5	59	12	10	26	7
4gt13_90	5	53	7	7	13	3
ising_model_10	10	90	0	0	5	0
4gt11_84	5	9	0	0	3	0
4gt11_83	5	14	0	0	0	0
mod5d1_63	5	13	0	0	1	0
4gt11_82	5	18	1	1	1	1
decod24-v3_45	5	64	15	15	32	8
rd32-v1_68	4	16	0	0	6	0
mini-alu_167	5	126	27	27	49	11
one-two-three-v2_100	5	32	3	4	8	3
4mod7-v0_94	5	72	8	13	36	9
cm82a_208	8	283	41	69	84	33
mod8-10_178	6	152	5	20	13	7
mod8-10_177	6	196	14	33	58	13
majority_239	7	267	39	43	105	33
miller_11	3	23	0	0	9	0
decod24-bdd_294	6	32	4	4	9	4
total	551	9244	1372	1738	3481	800

Table 3. Compare the number of SWAP gates added by the output circuit on the IBM Q20

Circuit name	qubit no.	CNOT no.	GQLTS$_{num}$ added	GQLTS$_{dep}$ added	optm added	wghtgr added
max46_240	10	11844	3473	4545	-	-
rd73_252	10	2319	586	761	-	-
cycle10_2_110	12	2648	919	1216	961	-
sqrt8_260	12	1314	379	492	457	-
urf4_187	11	224028	54785	60140	-	-
sqn_258	10	4459	1199	1420	-	-
f2_232	8	525	87	124	218	-
radd_250	13	1405	386	489	511	-
ham15_107	15	3858	1326	1689	-	-
sao2_257	14	16864	5346	7178	-	-
sym9_148	10	9408	1865	2432	-	-
urf5_280	9	23764	6989	8730	-	-
square_root_7	15	3089	812	2150	-	-
sys6-v0_111	10	98	23	26	38	-
hwb7_59	8	10681	2687	3551	3722	-
sym9_146	12	148	38	55	54	-
wim_266	11	427	93	120	147	-
urf2_152	8	35210	9181	11921	10577	-
urf5_159	9	71932	20258	25505	-	-
urf2_277	8	10066	2807	3798	3782	-
life_238	11	9800	2762	3576	-	-
root_255	13	7493	2128	3035	-	-
9symml_195	11	15232	4553	5986	-	-
sym10_262	12	28084	8534	11033	-	-
dc1_220	11	833	226	207	371	-
cm42a_207	14	771	182	229	294	-
rd53_311	13	124	26	48	47	-
dc2_222	15	4131	1383	1773	-	-
rd84_142	15	154	49	58	50	-
sym6_145	7	1701	317	449	750	-
co14_215	15	7840	3078	3819	-	-
cnt3-5_180	16	215	59	74	79	-
cm152a_212	12	532	103	129	168	-
sym6_316	14	123	30	39	56	-
mlp4_245	16	8232	2780	3490	-	-
hwb8_113	9	30372	10749	16489	-	-
qft_16	16	240	90	147	-	-
plus63mod4096_163	13	56329	19759	24273	-	-
urf1_149	9	80878	22551	28516	-	-
urf3_155	10	185276	50842	62903	-	-
urf3_279	10	60380	17999	23318	-	-
hwb9_119	10	90955	22946	30031	-	-
plus63mod8192_164	14	81865	28022	36207	-	-
pm1_249	14	771	182	229	294	-
sym9_193	11	15232	4382	5518	-	-
misex1_241	15	2100	480	754	600	-
urf1_278	9	26692	8010	10217	-	-
squar5_261	13	869	219	313	290	-
ground_state_estimation_10	13	154209	11671	22886	-	-
adr4_197	13	1498	516	670	-	-

Table 4. Compare the number of SWAP gates added by the output circuit on the IBM Q20

Circuit name	qubit no.	CNOT no.	$GQLTS_{num}$ added	$GQLTS_{dep}$ added	optm added	wghtgr added
hwb6_56	7	2952	698	933	909	-
clip_206	14	14772	5430	6865	-	-
cm85a_209	14	4986	2088	2225	-	-
rd84_253	12	5960	1849	2333	-	-
dist_223	13	16624	5623	7431	-	-
inc_237	16	4636	1193	1667	-	-
qft_10	10	90	23	34	30	-
urf6_160	15	75180	27524	32452	-	-
con1_216	9	415	86	118	177	-

Table 5. Compare the number of SWAP gates added by the output circuit on the IBM Q20

B Experimental details of the depth of the output circuit

Circuit name	qubit no.	CNOT no.	depths no.	GQLTS$_{num}$ depths	GQLTS$_{dep}$ depths	optm depths
decod24-enable_126	6	149	190	233	275	470
4mod5-v0_19	5	16	21	16	16	21
4mod5-v0_18	5	31	40	37	46	54
mod5d2_64	5	25	32	40	43	67
4gt4-v0_72	6	113	137	155	143	297
alu-v3_35	5	18	22	24	30	60
4gt4-v0_73	6	179	227	260	281	586
alu-v3_34	5	24	30	30	33	63
3_17_13	3	17	22	17	17	52
4gt4-v0_78	6	109	137	145	133	352
4gt4-v0_79	6	105	132	156	156	345
4mod7-v1_96	5	72	94	120	129	218
mod10_171	5	108	139	159	168	335
ex2_227	7	275	355	419	452	899
mod10_176	5	78	101	120	120	274
cycle10_2_110	12	2648	3386	5405	6296	7467
0410184_169	5	9	6	15	15	253
4mod5-v0_20	5	10	12	10	10	32
sqrt8_260	12	1314	1661	2451	2790	3561
aj-e11_165	5	69	86	93	93	250
alu-v1_28	5	18	22	24	30	70
f2_232	8	525	668	786	897	1672
radd_250	13	1405	1781	2563	2872	3985
4gt12-v0_86	6	116	135	200	215	334
4gt12-v0_87	6	112	131	193	208	324
4gt12-v0_88	6	86	108	101	101	222
alu-v1_29	5	17	22	29	29	64
ham7_104	7	149	185	233	251	491
C17_204	7	205	253	283	364	688
xor5_254	6	5	5	5	5	10
hwb4_49	5	107	134	149	152	308
rd73_140	10	104	92	173	182	185
decod24-v0_38	4	23	30	23	23	61
rd53_131	7	200	261	317	317	677
rd53_133	7	256	327	367	397	777
rd53_135	7	134	159	218	221	331
sys6-v0_111	10	98	75	167	176	188
decod24-v2_43	4	22	30	22	22	75
hwb7_59	8	10681	13437	18742	21334	29601
rd53_138	8	60	56	102	108	114
rd32-v0_66	4	16	20	16	16	51
sym9_146	12	148	127	262	313	309
4gt13-v1_93	5	30	39	30	30	102
graycode6_47	6	5	5	5	5	5
wim_266	11	427	514	706	787	1180
urf2_152	8	35210	44100	62753	70973	90299
urf2_277	8	10066	11390	18487	21460	26548
4mod5-bdd_287	7	31	41	40	49	71
ham3_102	3	11	13	11	11	28
4gt4-v0_80	6	79	101	94	94	206

Table 6. Compare the depth of the output circuit on the IBM Q20

Circuit name	qubit no.	CNOT no.	depths no.	GQLTS$_{num}$ depths	GQLTS$_{dep}$ depths	optm depths
ex-1_166	3	9	12	9	9	28
mod5mils_65	5	16	21	16	16	52
0example	5	9	6	12	15	15
alu-v4_36	5	51	66	87	75	170
alu-v4_37	5	18	22	24	30	60
ex1_226	6	5	5	5	5	10
one-two-three-v0_98	5	65	82	98	104	234
one-two-three-v0_97	5	128	163	197	197	443
one-two-three-v3_101	5	32	40	41	44	95
rd32_270	5	36	47	45	45	76
dc1_220	11	833	1041	1511	1454	2711
rd53_130	7	448	569	715	748	1417
rd53_251	8	564	712	876	957	1767
cm42a_207	14	771	940	1317	1458	2279
rd53_311	13	124	130	202	268	300
4mod5-v1_24	5	16	21	16	16	36
mod5adder_127	6	239	302	302	407	817
4_49_16	5	99	125	159	150	320
hwb5_53	6	598	758	1021	1102	1560
ex3_229	6	175	226	205	202	462
rd84_142	15	154	110	301	328	253
4gt10-v1_81	5	66	84	108	111	210
alu-v2_32	5	72	92	117	123	215
alu-v2_31	5	198	255	324	360	650
alu-v2_30	6	223	285	346	358	734
sym6_145	7	1701	2187	2652	3048	5716
sf_276	6	336	435	372	492	1096
decod24-v1_41	5	38	50	50	50	120
sf_274	6	336	436	438	399	822
4gt4-v1_74	6	119	154	170	191	329
alu-v2_33	5	17	22	29	29	59
cnt3-5_180	16	215	209	392	437	482
cm152a_212	12	532	684	841	919	1423
cnt3-5_179	16	85	61	103	103	166
sym6_316	14	123	135	213	240	378
4mod5-v1_22	5	11	12	11	11	37
4mod5-v1_23	5	32	41	47	47	55
mini_alu_305	10	77	71	107	137	187
alu-v0_26	5	38	49	59	68	108
alu-bdd_288	7	38	48	50	74	112
alu-v0_27	5	17	21	23	29	63
4gt13_91	5	49	61	70	70	108
4gt5_77	5	58	74	94	94	170
4gt13_92	5	30	38	30	30	103
4gt5_76	5	46	56	67	76	171
4gt5_75	5	38	47	53	74	127
4gt12-v1_89	6	100	130	133	163	313
one-two-three-v1_99	5	59	76	95	89	194
4gt13_90	5	53	65	74	74	124
pm1_249	14	771	940	1317	1458	2279

Table 7. Compare the depth of the output circuit on the IBM Q20

Circuit name	qubit no.	CNOT no.	depths no.	GQLTS$_{num}$ depths	GQLTS$_{dep}$ depths	optm depths
ising_model_10	10	90	52	90	90	107
misex1_241	15	2100	2676	3540	4362	5326
4gt11_84	5	9	11	9	9	25
4gt11_83	5	14	16	14	14	16
mod5d1_63	5	13	13	13	13	17
4gt11_82	5	18	20	21	21	25
squar5_261	13	869	1051	1526	1808	2309
decod24-v3_45	5	64	84	109	109	244
rd32-v1_68	4	16	21	16	16	52
hwb6_56	7	2952	3736	5046	5751	7773
mini-alu_167	5	126	162	207	207	400
one-two-three-v2_100	5	32	40	41	44	80
4mod7-v0_94	5	72	92	96	111	270
cm82a_208	8	283	340	406	490	699
mod8-10_178	6	152	193	167	212	243
mod8-10_177	6	196	251	238	295	525
majority_239	7	267	344	384	396	839
qft_10	10	90	37	159	192	135
miller_11	3	23	29	23	23	75
decod24-bdd_294	6	32	40	44	44	86
con1_216	9	415	508	673	769	1197
total	823	83416	103023	145372	164848	224731

Table 8. Compare the depth of the output circuit on the IBM Q20

Circuit name	qubit no.	CNOT no.	depths no.	$GQLTS_{num}$ depths	$GQLTS_{dep}$ depths	optm depths
max46_240	10	11844	14257	22263	25479	-
rd73_252	10	2319	2867	4077	4602	-
urf4_187	11	224028	264330	388383	404448	-
sqn_258	10	4459	5458	8056	8719	-
ham15_107	15	3858	4819	7836	8925	-
sao2_257	14	16864	19563	32902	38398	-
sym9_148	10	9408	12087	15003	16704	-
urf5_280	9	23764	27822	44731	49954	-
square_root_7	15	3089	3847	5525	9539	-
urf5_159	9	71932	89148	132706	148447	-
life_238	11	9800	12511	18086	20528	-
root_255	13	7493	8839	13877	16598	-
9symml_195	11	15232	19235	28891	33190	-
sym10_262	12	28084	35572	53686	61183	-
dc2_222	15	4131	5242	8280	9450	-
co14_215	15	7840	8570	17074	19297	-
mlp4_245	16	8232	10328	16572	18702	-
hwb8_113	9	30372	38717	62619	79839	-
qft_16	16	240	61	510	681	-
plus63mod4096_163	13	56329	72246	115606	129148	-
urf1_149	9	80878	99586	148531	166426	-
urf3_155	10	185276	229365	337802	373985	-
urf3_279	10	60380	70702	114377	130334	-
hwb9_119	10	90955	116199	159793	181048	-
plus63mod8192_164	14	81865	105142	165931	190486	-
sym9_193	11	15232	19235	28378	31786	-
ising_model_13	13	120	46	120	120	-
urf1_278	9	26692	30955	50722	57343	-
ising_model_16	16	150	57	150	150	-
ground_state_estimation_10	13	154209	217236	189222	222867	-
adr4_197	13	1498	1839	3046	3508	-
clip_206	14	14772	17879	31062	35367	-
cm85a_209	14	4986	6374	11250	11661	-
rd84_253	12	5960	7261	11507	12959	-
dist_223	13	16624	19694	33493	38917	-
inc_237	16	4636	5864	8215	9637	-
urf6_160	15	75180	93645	157752	172536	-

Table 9. Compare the depth of the output circuit on the IBM Q20

References

1. Almeida, A., Dueck, G., Silva, A.: Finding optimal qubit permutations for ibm's quantum computer architectures pp. 1–6 (08 2019). https://doi.org/10.1145/3338852.3339829

2. Barenco, A., Bennett, C., Cleve, R., DiVincenzo, D., Margolus, N., Shor, P., Sleator, T., Smolin, J., Weinfurter, H.: Elementary gates for quantum computation. Physical Review A **52** (03 1995). https://doi.org/10.1103/PhysRevA.52.3457
3. Bernal, D., Booth, K., Dridi, R., Alghassi, H., Tayur, S., Venturelli, D.: Integer programming techniques for minor-embedding in quantum annealers (12 2019)
4. Cowtan, A., Dilkes, S., Duncan, R., Krajenbrink, A., Simmons, W., Sivarajah, S.: On the qubit routing problem (02 2019)
5. Guerreschi, G.G., Park, J.: Two-step approach to scheduling quantum circuits. Quantum Science and Technology **3** (06 2018). https://doi.org/10.1088/2058-9565/aacf0b
6. Kissinger, A., Meijer, A.: Cnot circuit extraction for topologically-constrained quantum memories (04 2019)
7. Li, G., Ding, Y., Xie, Y.: Tackling the qubit mapping problem for nisq-era quantum devices (09 2018)
8. Li, S., Zhou, X., Feng, Y.: Qubit mapping based on subgraph isomorphism and filtered depth-limited search (2020)
9. Lloyd, Seth: Almost any quantum logic gate is universal. Physical Review Letters **75**(2), 346 (1995)
10. Matsuo, A., Yamashita, S.: An efficient method for quantum circuit placement problem on a 2-d grid pp. 162–168 (05 2019). https://doi.org/10.1007/978-3-030-21500-2_10
11. Murali, P., Linke, N., Martonosi, M., Abhari, A., Nguyen, N., Huerta Alderete, C.: Full-stack, real-system quantum computer studies: architectural comparisons and design insights pp. 527–540 (06 2019). https://doi.org/10.1145/3307650.3322273
12. Möttönen, M., Vartiainen, J.: Decompositions of general quantum gates. Frontiers in Artificial Intelligence and Applications (05 2005)
13. Nash, B., Gheorghiu, V., Mosca, M.: Quantum circuit optimizations for nisq architectures. Quantum Science and Technology **5** (02 2020). https://doi.org/10.1088/2058-9565/ab79b1
14. Paler, A.: On the influence of initial qubit placement during nisq circuit compilation (11 2018)
15. Preskill, J.: Quantum computing in the nisq era and beyond. Quantum **2** (2018)
16. Shafaei, A., Saeedi, M., Pedram, M.: Optimization of quantum circuits for interaction distance in linear nearest neighbor architectures. Proceedings - Design Automation Conference pp. 1–6 (05 2013). https://doi.org/10.1145/2463209.2488785
17. Shafaei, A., Saeedi, M., Pedram, M.: Optimization of quantum circuits for interaction distance in linear nearest neighbor architectures (2013)
18. Siraichi, M.Y., dos Santos, V.F., Collange, S., Pereira, F.M.Q.: Qubit allocation (2018)
19. Sun, S., Luo, Q.: In-memory subgraph matching: An in-depth study pp. 1083–1098 (06 2020). https://doi.org/10.1145/3318464.3380581
20. Tannu, S., Qureshi, M.: Not all qubits are created equal: A case for variability-aware policies for nisq-era quantum computers pp. 987–999 (04 2019). https://doi.org/10.1145/3297858.3304007
21. Venturelli, D., do, M., Rieffel, E., Frank, J.: Temporal planning for compilation of quantum approximate optimization circuits pp. 4440–4446 (08 2017). https://doi.org/10.24963/ijcai.2017/620
22. Xiangzhen, Z., Li, S., Feng, Y.: Quantum circuit transformation based on simulated annealing and heuristic search. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **PP**, 1–1 (01 2020). https://doi.org/10.1109/TCAD.2020.2969647

23. Zhang, Y., Deng, H., Li, Q., Haoze, S., Nie, L.: Optimizing quantum programs against decoherence: Delaying qubits into quantum superposition pp. 184–191 (07 2019). https://doi.org/10.1109/TASE.2019.000-2
24. Zulehner, A., Paler, A., Wille, R.: Efficient mapping of quantum circuits to the ibm qx architectures. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (12 2017). https://doi.org/10.1109/TCAD.2018.2846658