Yang et al. *EPJ Quantum Technology* (2025) 12:20
https://doi.org/10.1140/epjqt/s40507-025-00320-x

EPJ Quantum Technology
a SpringerOpen Journal

EPJ.org

RESEARCH Open Access

A scalable routing method for superconducting quantum processor

Tian Yang[1†], Chen Liang[1,2†], Weilong Wang[1*], Bo Zhao[1], Lixin Wang[1], Qibing Xiong[1], Xuefei Feng[1,3] and Zheng Shan[1*]

*Correspondence:
wangwl19888@163.com;
shanzhengzz@163.com
[1]Laboratory for Advanced Computing and Intelligence Engineering, Information Engineering University, Zhengzhou, 450001 Henan, China
Full list of author information is available at the end of the article
[†]Equal contributors

Abstract

Routing design is an important aspect in aiding the completion of the Quantum Processing Unit (QPU) layout design for large-scale superconducting quantum processors. One of the research focuses is how to generate reliable routing schemes within a short time. In this study, we propose a superconducting quantum processor auto-routing method for supporting scalable architecture, which is mainly implemented through the bidirectional A star algorithm, the backtracking algorithm, and the greedy strategy. By using this method, the number of crossovers and corners can be reduced while efficiently completing the processor routing. To verify the effectiveness of our method, we selected 5 types of qubit numbers for processor routing experiments. The experimental results show that compared to the improved A star algorithm of Qiskit Metal, our method reduces the average execution time by at least 43.61% and 41.68% in serial and parallel, respectively. Compared with four other routing algorithms, our method has a minimum average reduction of 10.63% and 1.21% in the number of crossovers and corners, respectively. In addition, our method supports the processor routing design of planar and flip-chip architectures, and can automatically process both airbridge and insulation types of crossovers. Therefore, our method can provide efficient and reliable automated routing design to assist the development of large-scale superconducting quantum processors.

Keywords: Superconducting quantum processor; Routing design; Bidirectional A star algorithm; Automation

1 Introduction

Due to high compatibility with existing integrated circuits in circuit design and fabrication, as well as high feasibility of qubit control and measurement, superconducting quantum circuits have become one of the preferred solutions for building scalable quantum processors [1, 2]. So far, the number of qubits in the superconducting quantum processor has exceeded 1000 [3], which strongly demonstrates the feasibility of superconducting quantum computing schemes in scalable architecture. This has also accelerated the progress of quantum computing technology towards commercialization and generalization.

Scaling up the number of qubits and ensuring the quality of the qubits on the superconducting quantum processor can provide a powerful computational resource for solv-

Springer

ing complex problems [4–7], but it also poses challenges to the design of superconducting quantum circuits. It includes processor topology design [8–10], circuit components design [11–14], hamiltonian parameters design [15–17], and processor routing design [18–20]. Among them, the processor routing design is an important part of aiding the completion of the QPU layout design for large-scale superconducting quantum processors. A well-designed routing scheme can optimize the processor routing spacing, reduce the number of crossovers and corners, lower the influence of crosstalk, and thus improve the performance and quality of scalable quantum processors. However, as the number of qubits grows, manually completing the processor routing design becomes incrementally difficult, leading to reduced processor design efficiency and increased design costs. Therefore, the experience of electronic design automation (EDA) in the integrated circuits should be taken into account to develop tools and methods related to quantum electronic design automation (QEDA) [21, 22] to assist superconducting quantum processor designers in efficiently and accurately completing routing design.

Routing design is a mature research field in Printed Circuit Board (PCB) layout, with many excellent algorithms proposed by researchers to address PCB routing problems [23, 24]. The PCB routing design methods can guide the routing design of superconducting quantum processors. However, due to differences between PCB and superconducting quantum processor in terms of operating environment, materials, fabrication technologies, and control signals, their routing rules and constraints (such as trace width, length, and spacing) also differ [2, 25–27]. Therefore, traditional PCB routing design methods could not be directly applied to the routing design of superconducting quantum processors. Based on these differences, there is still room for research and optimization in the routing design of superconducting quantum processors. Currently, scholars and commercial organizations have invested in the research of superconducting quantum processor routing design. IQM has developed an automated design tool, KQCircuits [28], for superconducting quantum circuits based on Python to help users avoid human errors in the design process. KQCircuits provides coplanar waveguide components in different shapes to support processor routing design for both planar and flip-chip processors on the graphical user interface (GUI). At the same time, it can also be used as a standalone Python package, using programming methods to complete the design. However, KQCircuits' routing design relies mainly on manual tuning, which will lengthen the processor design cycle. IBM has also developed its own superconducting quantum circuit design and simulation tool called Qiskit Metal [29], based on Python. It is used for the layout design and analysis of superconducting quantum processors. Researchers have designed a routing path search strategy based on an improved A star algorithm (using the heap data structure) on Qiskit Metal. This strategy can automatically complete the specified routing design from a given starting point to an ending point, taking into account obstacles and anchor points defined by users. However, it does not support global automatic routing of the processor. In the processor layout design of the large-scale integrated qubits, manually specifying the assignment scheme for qubits and pins will also reduce the design efficiency of the processor. At the same time, Qiskit Metal does not support automatic crossover processing for areas where routing paths overlap (see Fig. 1), requiring manual secondary modification of the generated layout. In addition, neither of them supports the optimization of the number of crossovers and corners in the routing design. An increase in the number of crossovers may result in the impedance mismatch and increased crosstalk between qubits [30], while

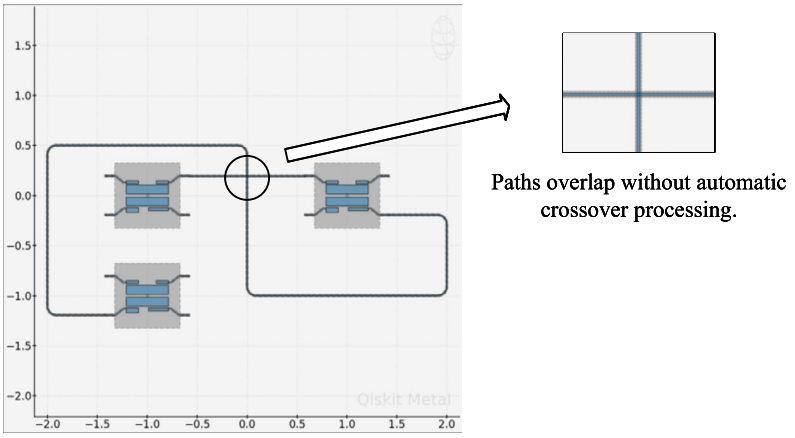


Figure 1 Example of Qiskit Metal routing overlap

the excessive corners may lead to the reflection of the signal, affecting the integrity and stability of the signal [31]. Therefore, there is still room for improvement in the routing design of superconducting quantum processors.

To address the shortcomings of existing studies, in this paper, we proposed a superconducting quantum processor automatic routing method for supporting scalable architecture. Since the fixed-frequency superconducting qubits have achieved remarkable success in the stability and scalability of superconducting quantum processors [3, 32–34], our method primarily focuses on the optimization of XY control line paths with the fixed-frequency superconducting qubits. Our method can complete the routing design of all qubits while reducing the overall number of crossovers and corners, support multi-process parallel acceleration, and automatically complete processor routing design for planar and flip-chip architectures. It consists of three modules: layout projection, grid map routing, and GDS routing. The flow of our method is shown in Fig. 2. Firstly, the layout projection is performed according to the input QPU layout (completed components placement). Then, assign appropriate pins to the qubits to complete the grid map routing while reducing the number of crossovers and corners. Here, we construct the routing optimization problem as a mathematical model to design a method for solving pins assignment scheme (primarily implemented using the bidirectional A star algorithm, the backtracking algorithm, and the greedy strategy). Meanwhile, by dividing different regions on the grid map, our pins assignment strategy can be executed in parallel by multiple processes. Finally, perform GDS routing. Convert grid paths into GDS paths, correct and draw the GDS paths (including automatic processing of crossovers), and output the completed routing design in QPU layout.

In order to verify the effectiveness of our method, we selected five types of qubit numbers (16∼128) for processor routing experiments. In terms of execution efficiency, compared to the improved A star algorithm of Qiskit Metal, our method reduces the average execution time by a minimum of 43.61% and 41.68% in serial mode and parallel mode, respectively. In terms of crossovers and corners optimization, we compare four routing methods (standard bidirectional A star algorithm, no second optimization method, Ref. [35], and random pins assignment algorithm) with our method. Meanwhile, we randomly generate three topologies for routing experiments on different types of qubit numbers.

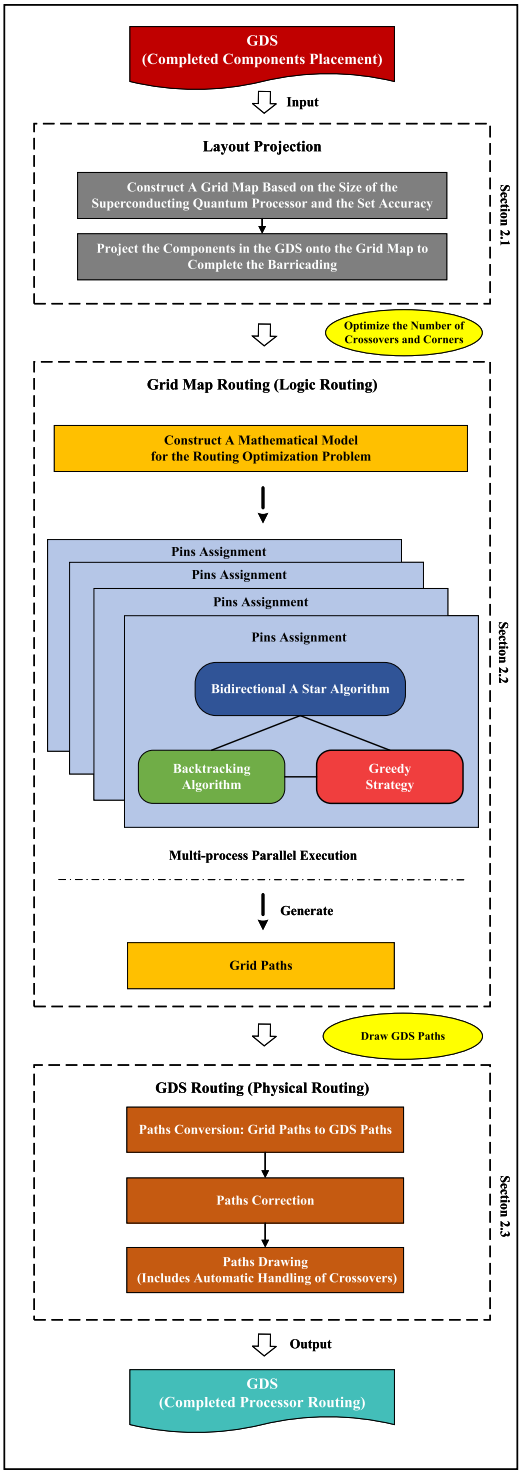


Figure 2 Overview of basic ideas of the method

Through experimental comparison with four routing methods, our method has a minimum average reduction of 10.63% in the number of crossovers and 5.42% in the number of corners under the planar architecture. For the flip-chip architecture, our method has a

minimum average reduction of 1.21% in the number of corners. In addition, our method has an automatic crossover processing function, which can flexibly meet the requirements for two types of crossovers: airbridge and insulation. These properties make our method more convenient in the superconducting quantum processor routing design and can provide designers with efficient and reliable routing solutions. Meanwhile, our method can be applied in the automation of superconducting quantum processor design to promote the development of QEDA.

2 Methods

In this section, we will provide a detailed description of the three modules for superconducting quantum processor routing method.

2.1 Layout projection

To automatically optimize the number of crossovers and corners, it is essential to ensure that the starting and ending positions of routing, routing obstacles, feasible areas for routing, and crossover areas are identified on the processor layout, in order to design reasonable routing paths. However, as the number of qubits increases, component structures need to be adjusted, and component placement schemes would be changed. It will become exceptionally complex to directly describe the above information through the GDS file. Meanwhile, it may be necessary to repeatedly search and undo routing paths during the optimization process. If we operate directly on the GDS file, it will lead to a cumbersome routing design process and reduce the design efficiency. Here, we choose to project the processor layout onto a grid map (represented by a two-dimensional array) to store and describe the layout information. By utilizing different numerical values for the grid, it is possible to simplify the representation of layout information, facilitating the rapid differentiation of various regions along the routing paths. Additionally, grid maps enable easy searching, retracting of routing paths, and the counting of crossover and corner numbers, thereby efficiently seeking better routing solutions.

The layout projection includes two steps: grid map initialization and obstacle placement. The specific details are as follows:

2.1.1 Grid map initialization

Based on the horizontal and vertical ranges of the QPU layout ($S_{\text{processor\_GDS\_X}}$ and $S_{\text{processor\_GDS\_Y}}$), distance from pin ports to the boundaries of the processor, and the set precision Δ_{set}, the initial grid map is constructed. Figure 3 shows an example of grid map initialization. Detailed explanation can be found in Appendix A.

2.1.2 Obstacle placement

After initializing the grid map, we project the components on the processor and complete the obstacle placement. We set the components that have been placed to include pins, qubits (Transmon or Xmon), coupling components (Direct Coupler or Bus Resonator Coupler), $\lambda/4$ readout resonators, and transmission lines. The shapes of the components are based on Qiskit Metal's components design. Here, since the transmission lines are typically coupled with the readout resonators and follow regular routing paths, we consider the transmission lines as completed placed components, optimizing the design of

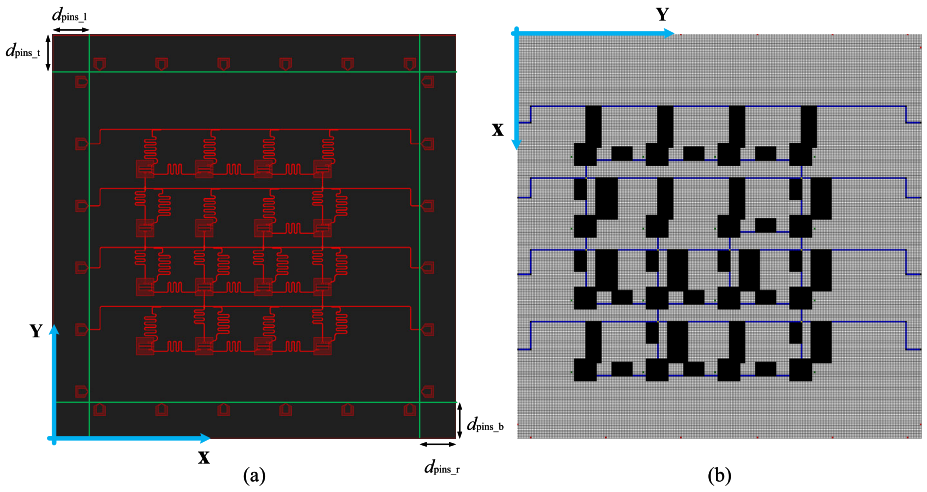


Figure 3 Example of the processor projection initialisation boundaries

the paths for qubit control lines. Figure 4 shows the projections of planar and flip-chip architectures on grid maps. The meanings of different grids are shown in Table 1. Detailed explanation can be found in Appendix B.

2.2 Grid map routing (logic routing)

After getting the Grid Map with completed obstacles, we will perform the Grid Map Routing. To optimize the number of crossovers and corners, we not only need to find a feasible routing path for each qubit, but also need to avoid crossover areas as much as possible and reduce the modification of the path direction during the path planning process. To this end, we first construct a mathematical model for the routing optimization problem to clarify the relationships and constraints of the variables, and define the objective function (Sect. 2.2.1). Then, by analyzing the mathematical model, we propose a two-step method to solve the problem: 1). Optimize the routing path of the qubit drivelines with the determined pins. By changing the design of the bidirectional A star algorithm and using a corner removal step, the number of crossovers and corners on the path can be reduced while efficiently completing the path search (Sect. 2.2.2); 2). Optimize the pins assignment scheme for qubits. By combining the backtracking algorithm and the greedy strategy, a pins assignment strategy is proposed to reduce the search space and find a reasonable pins assignment scheme. The assignment scheme allows all qubits to complete the routing design while reducing the overall number of crossovers and corners (Sect. 2.2.3). The detailed explanations are as follows.

2.2.1 Mathematical model

Firstly, we construct the routing optimization problem as a mathematical model to clarify routing constraints and objective function. Assuming the number of qubits is N, the number of pins is M, and $M \geq N$. The path from the starting point of control line for qubit i (Q_i) to pin port j is denoted by $path_{ij}$. $cross(path_{ij})$ and $corner(path_{ij})$ represent the number of crossovers and corners on $path_{ij}$, respectively. To optimize the number of

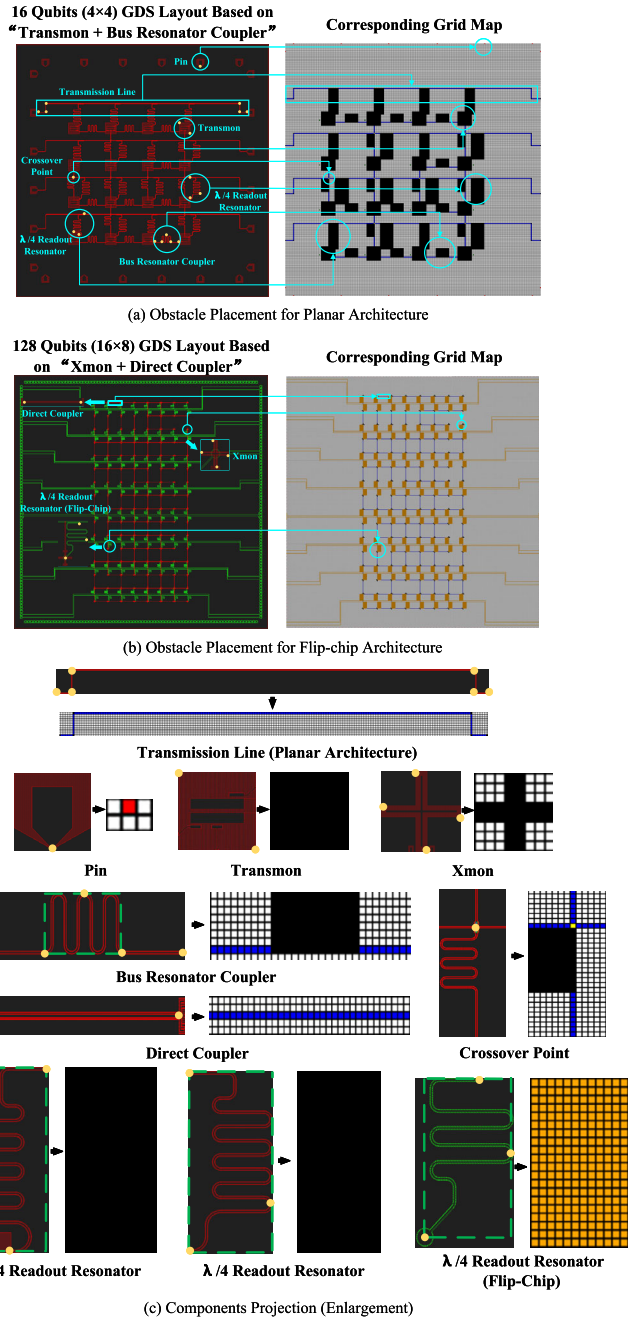


Figure 4 Example of the projections of planar and flip-chip architectures. The golden dots (mark points) are used to record the corresponding coordinates of the components in the QPU layout, in order to define the projected area of the components on the grid map. The green dashed lines indicate the areas where the resonators need to be projected as rectangles. In the planar architecture, transmission lines are considered as crossover areas (represented in blue). In the flip-chip architecture, the projections of the readout resonators, transmission lines, and control lines on the control chip (C-Chip) are represented in orange, and transmission lines are no longer regarded as crossover areas

Table 1 The meanings of different grids

Value	Color	Meaning
Grid[*x*,*y*] = 0	white	the feasible routing areas
Grid[*x*,*y*] = 1	black	obstacles
Grid[*x*,*y*] = 2	red	pin port
Grid[*x*,*y*] = 3	green	the starting point of the control line
Grid[*x*,*y*] = 4	blue	the crossover areas
Grid[*x*,*y*] = 5	yellow	the crossover point
Grid[*x*,*y*] = 6	orange	readout resonators, transmission lines, and control lines (Flip-Chip Architecture)

crossovers and corners, the mathematical model is constructed as follows:

$$
\begin{aligned}
&\min(f(path_{ij}))\\
&f(path_{ij}) = \alpha \times \lambda_1 \times cross(path_{ij}) + \beta \times \lambda_2 \times corner(path_{ij}) + \gamma \times length(path_{ij})\\
&s.t.\\
&C_1 : \alpha + \beta + \gamma = 1\\
&C_2 : 0 \le \gamma < \beta < \alpha \le 1\\
&C_3 : N \le M\\
&C_4 : 1 \le i \le N\\
&C_5 : 1 \le j \le M\\
&C_6 : path_{ij} \notin \emptyset\\
&C_7 : Spacing(path_{ij}, path_{hl}) \ge d_1, i \ne h \wedge j \ne l \wedge 1 \le h \le N \wedge 1 \le l \le M\\
&C_8 : \begin{array}{l} Spacing(path_{ij}, \mathrm{Grid}[x,y]) \ge d_2,\\ \mathrm{Grid}[x,y] = 1 \wedge x \in \left[0, S_{\mathrm{grid\_map\_X}} - 1\right] \wedge y \in \left[0, S_{\mathrm{grid\_map\_Y}} - 1\right] \end{array}\\
&C_9 : \begin{array}{l} Spacing(path_{ij}, \mathrm{Grid}[x,y]) \ge d_3,\\ \mathrm{Grid}[x,y] = 4 \wedge \mathrm{Grid}[x,y] \notin path_{ij} \end{array}\\
&C_{10} : \begin{array}{l} \mathrm{Grid}[x,y] = 5 \wedge \mathrm{Grid}[x+1,y] \ne 5 \wedge \mathrm{Grid}[x-1,y] \ne 5 \wedge \mathrm{Grid}[x,y+1] \ne 5\\ \wedge \mathrm{Grid}[x,y-1] \ne 5, \mathrm{Grid}[x,y] \in path_{ij} \end{array}
\end{aligned}
\tag{1}
$$

where $f(path_{ij})$ is the objective function used to evaluate the actual effectiveness of the path scheme $path_{ij}$. In $f(path_{ij})$, we have added path length $length(path_{ij})$ as an influence factor for evaluation. The rationale for this is that when the number of crossovers and corners in the routing path is the same for different pin ports, priority is given to the pin port closest to the qubit. This can increase the search space for the routing paths of the remaining qubits. The weight factors α, β, and γ represent the emphasis on crossover, corner, and length in path evaluation. In this paper, we set the highest priority on crossover, followed by corner, and lastly on length. λ_1, λ_2 are the orders of magnitude adjustment factors to prevent $cross(path_{ij})$ and $corner(path_{ij})$ from being ignored if they take a small range of values. The values of λ_1 and λ_2 are adjusted according to the grid map, with a range of 5 to $10 \times \varepsilon(\kappa)$ in this paper (ε, κ see Sect. 2.2.2). C_7-C_{10} represents the routing constraints that need to be satisfied. Specifically, C_7 indicates that the spacing between different routing paths should be at least d_1. C_8 indicates that the spacing between the routing path and ob-

stacles should be at least d_2. C_9 indicates that when there are no intersections between the routing path and the crossover areas (*i.e.*, no crossover points exist), the spacing should be at least d_3. C_{10} indicates that two crossover points cannot be adjacent. By assigning different pin ports, we aim to minimize the objective function value under the routing constraints in order to find the optimal routing path.

According to the mathematical model, in order to minimize the objective function value, on the one hand, it is necessary to design an efficient path search algorithm to optimize the routing path under the determined pins, avoiding choosing crossover areas and changing path directions. On the other hand, it is essential to design a reasonable pins assignment strategy to generate appropriate assignment schemes for different qubits and pin ports, further reducing the number of crossovers and corners. The specific designs are detailed in Sect. 2.2.2 and Sect. 2.2.3.

2.2.2 Optimize the routing path of the qubit drivelines with the determined pins

In this section, we use the bidirectional A star algorithm and a corner removal step to optimize the routing path when the start node and end node are determined.

The implementation of the bidirectional A star algorithm mainly includes three items: the open list, the closed list, and the heuristic evaluation function [36]. The open list is used to store nodes to be visited, while the closed list is used for storing nodes that have been visited. The heuristic evaluation function is used to evaluate the cost of nodes to be visited, which is usually calculated as the sum of the actual steps from the start node to the node being evaluated, and the distance from that node to the end node. It guides the algorithm to select the node with the lowest cost for path expansion. During each iteration, the algorithm first selects the node with the minimum evaluation function value from the open list as the current node. Then, each neighboring node in the up, down, left, and right directions of the current node is checked against the constraints, the closed list, and the open list. If the neighboring node violates the constraints or is present in the closed list, it is ignored. Else, if the neighboring node is not in the open list, it is added to the open list, the current node is set as its parent, and its evaluation function value is recorded. Else, if the neighboring node exists in the open list, then check whether the path to this neighboring node through the current node is better (generally evaluated by the smaller actual number of steps from the start node to the neighboring node). If the new path is better, set the parent of the neighboring node to the current node and recalculate the evaluation function value for the neighboring node. Finally, remove the current node from the open list and add it to the closed list. Here, if the algorithm does not meet the termination condition, the above process will be repeated. Also, if the order of visiting neighboring nodes is different each time, the final generated path may vary when the evaluation function values of different neighboring nodes are the same. Additionally, the bidirectional A star algorithm performs simultaneous searches from the start point to the end point and from the end point to the start point. Each search process maintains its own open list and closed list. When the current node in one search direction belongs to a closed list node in another search direction, the search path is determined. Compared to unidirectional path search, the bidirectional A star algorithm reduces unnecessary search space and decreases time costs. Therefore, it is more suitable for cases where the search space is large, and the end node is far from the start node.

Based on the above description of the bidirectional A star algorithm, one can see that a large number of node additions, deletions, queries, and comparisons (or sorting) are

required in the open list. If one can optimize the time complexity of these operations, it will further reduce the algorithm's runtime. Meanwhile, during the path search process, the heuristic evaluation function does not account for the presence of crossovers and corners in the path. Therefore, we chose a data structure with better operational efficiency to implement the open list, added penalty terms to the heuristic evaluation function, and designed a corner removal step to further improve the efficiency of the routing path search while reducing the number of crossovers and corners in the routing path. The details are as follows:

A. Open list data structure In this paper, we use the binary tree-based Min-heap as the data structure for the open list. In the Min-heap, the node with the smallest evaluation function value can be queried with O(1) time complexity, which helps bidirectional A star algorithm to quickly determine the node to be expanded next. Additionally, the time complexity of the Min-heap on insertion, deletion, and sorting operations are all O(logn), which can achieve rapid updates to the open list and accelerate the path search process.

B. Heuristic evaluation function To reduce the number of crossovers and corners in the path search, we added corresponding penalty terms to the heuristic evaluation function $F(Node_i)$, as follows:

$$F(Node_i) = G(Node_i) + H(Node_i) + U(Node_i) + T(Node_i) \tag{2}$$

where $G(Node_i)$ represents the cost from the start node to the current node $Node_i$ (represented by the depth of the path search), and $H(Node_i)$ represents the estimated cost from the current node $Node_i$ to the goal node (represented by the Manhattan distance in the grid map). $G(Node_i)$ and $H(Node_i)$ are common components in the heuristic evaluation function design of the bidirectional A star algorithm. Their combination helps improve search efficiency and optimize the distance cost of the search path. In addition, we have added crossover cost $U(Node_i)$ and corner cost $T(Node_i)$ to prevent the algorithm from choosing crossover areas and changing directions during the path search. The calculations for these can be seen in Eqs. (3) and (4). Here, the direction of a node is related to the exploration direction of its parent node. The node obtained by exploring from the parent node to the left or right is set to have the direction 'west' or 'east'. The node obtained by exploring from the parent node to the up or down is set to have the direction 'north' or 'south'.

$$U(Node_i) = \begin{cases} \varepsilon, Grid[x_{Node_i}, y_{Node_i}] = 4 \\ 0, \text{otherwise} \end{cases} \tag{3}$$

$$T(Node_i) = \begin{cases} \kappa, Node_i.\text{direction} \neq Node_i.\text{parent.direction} \\ 0, \text{otherwise} \end{cases} \tag{4}$$

From the equations above, it can be seen that when $Node_i$ is in the crossover areas, $F(Node_i)$ increases, indicating an increase in the cost of selecting $Node_i$ and reducing the priority of searching for $Node_i$. Similarly, if $Node_i$ has a different direction from its parent node, the selection cost of $Node_i$ is increased, reducing the likelihood of being chosen during the search. Therefore, by adding additional crossover cost and corner cost, the number of crossovers and corners on the routing path can be reduced.

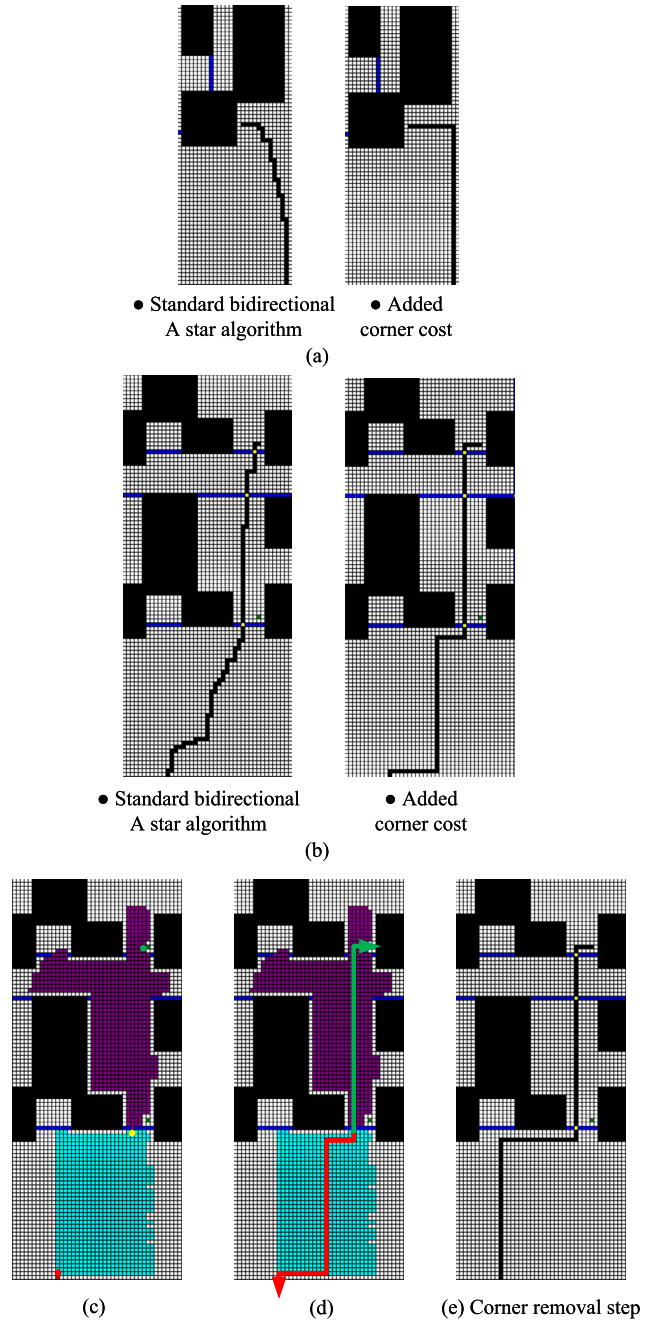


Figure 5 Example of the corner removal step. Purple and cyan nodes represent the nodes in the closed lists for forward and reverse searches, respectively. Subject to the direction constraints of the connection ports, the nodes marked with green and red circles are indicated as the mandatory points of the routing path, *i.e.* the actual start and end points. The node marked with the yellow circle is the current node of the forward search

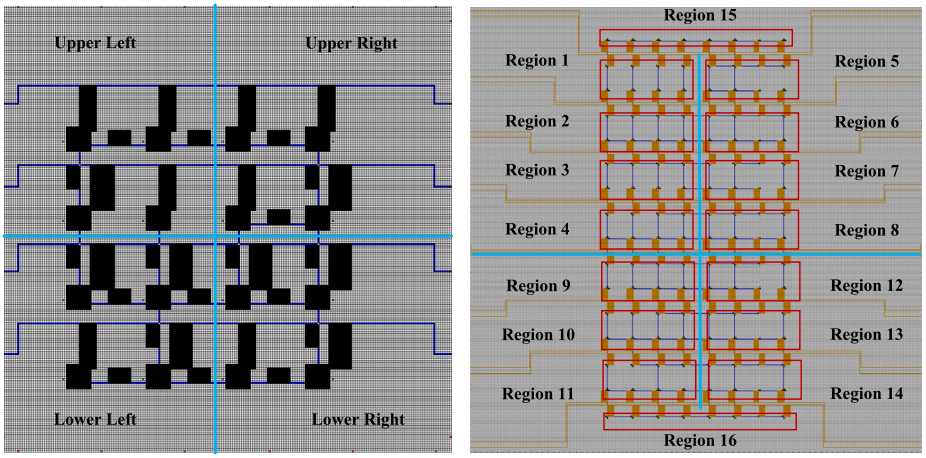
C. Corner removal step Adding corner cost to the heuristic evaluation function can reduce the number of direction changes in the path search and optimize the routing path (see Fig. 5(a)). However, during path searching, there may be cases where even with the addition of corner cost, the number of corners in the path may not be the minimum (see Fig. 5(b)). This is mainly caused by the execution principle of the bidirectional A star algo-

rithm. The bidirectional A star algorithm performs simultaneous searches from the start point to the end point (forward search) and from the end point to the start point (reverse search). The algorithm constructs an open list and a closed list for each search separately. When the current node in the forward search belongs to the closed list of the reverse search, the iteration stops, and vice versa. In Fig. 5(c), we show the nodes in the closed lists for forward and reverse searches in the example of Fig. 5(b), represented in purple and cyan, respectively. Since the pin is located on the lower side of the processor (with its connection port facing upward) and the control line start from the left side of the qubit (with its connection port facing left), the nodes marked with green and red circles in Fig. 5(c) are the mandatory nodes the routing path must pass through, which are the actual start point and end point. The node marked with yellow circle in Fig. 5(c) represent the current node of the forward search. Here, the iteration stops because the current node of the forward search belongs to the closed list of the reverse search. After the iteration stops, the algorithm constructs the search path. When the iteration ends at the current node of the forward search, the algorithm merges the path of the forward search (found by iterating from the current node to the start point via parent nodes) with the path of the reverse search (found by iterating from the corresponding node in the closed list to the end point via parent nodes) for output, and vice versa. In Fig. 5(d), we use the green arrow and the red arrow to indicate the paths of the forward and reverse searches, respectively. According to the algorithm's definition, the path iterated back from the current node to the start point is the optimal path. However, the path iterated back from the nodes in the closed list may not necessarily be the optimal path, as the closed list stores all the visited nodes. Therefore, there may still be corners in the path that need to be optimized. To address this, we propose a corner removal step, which further reduces the number of corners in the path generated by the bidirectional A star algorithm. This algorithm merges segments with corners into "L"-shaped paths as much as possible (see Fig. 5(e)), while satisfying the routing constraints and without increasing the number of crossovers. The detailed pseudocode is shown in Appendix C.

2.2.3 Optimize the pins assignment scheme for qubits

In this section, we propose a pins assignment strategy that combines the backtracking algorithm and the greedy strategy with the method from the previous section. By generating a reasonable pins assignment scheme, optimize the global number of crossovers and corners on the processor. The specifics are as follows:

A. Assignment by dividing regions As the processor size and the number of qubits increase, attempting to assign global pin ports for each qubit will expand the search scope for pins assignment, reduce the efficiency of the assignment strategy, and may potentially lead to global routing conflicts. For this purpose, we divide the processor's global grid map into regions. Local pin assignments are made to qubits in different regions to reduce the search scope and the overall assignment complexity. Specifically, for the planar architecture, we divide the grid map into four regions: upper left, upper right, lower left, and lower right, as shown in Fig. 6(a). The number of pins in each region is greater than or equal to the number of qubits, *i.e.*, $M_k \geq N_k (k = 0, 1, 2, 3)$. For the flip-chip architecture, since the transmission line paths are no longer considered as crossover areas, it can be further divided into regions to reduce the search space, as shown in Fig. 6(b) (all regions



(a) Dividing Regions for Planar Architecture

(b) Dividing Regions for Flip-chip Architecture

Figure 6 Example of dividing regions

are represented by dark red boxes). In our flip-chip architecture, we consider the first and last rows of qubits in the processor topology as a separate region, respectively. For the remaining qubits, based on the four regions of upper left, upper right, lower left, and lower right, we group every two rows together to form new regions. Meanwhile, since the pins assignment of each region is relatively independent, the pins assignment strategy can be processed in parallel using a multi-process scheme under the divide and conquer strategy. This helps to reduce the running time of the pins assignment strategy and improve routing efficiency.

B. Order of assignment After completing the division of regions, the assignment order of qubits needs to be considered. The qubits that complete pins assignment first will occupy the search area of subsequent routing paths in the path search, so different assignment orders may affect the search direction of the routing path and generate different routing results. Here, we assign in ascending order within each region based on the distance between qubits and the boundary of the grid map (*i.e.*, the assignment order is from outside to inside), as shown in Fig. 7. Figure 7(a) shows a quarter region of a 64 (8×8) qubits QPU layout with 30×4 pins. Figure 7(b) is the corresponding grid map. Figure 7(c) shows the assignment order used in our method (from outside to inside). Qubits with shorter distances to the boundary indicate a smaller search scope for their routing paths. Assigning these qubits first allows for quicker determination of the corresponding pins assignment scheme. At the same time, it also helps to eliminate infeasible assignment schemes for qubits that are far from the boundary, avoiding the situation of backtracking and rerouting, as shown in Fig. 7(d). It can be seen that when the assignment order is from the inside out, the outermost qubits marked by the red circles cannot be assigned pin and require backtracking.

In the routing design based on the flip-chip architecture, since the routing area does not contain qubits and coupling components, it has fewer constraints, resulting in a higher degree of freedom for the path search in the pins assignment. The routing paths may not be sufficiently regular, which can affect the pins assignment of subsequent qubits. For example, in Fig. 8(a), the four qubits within the red circles are unable to complete the routing

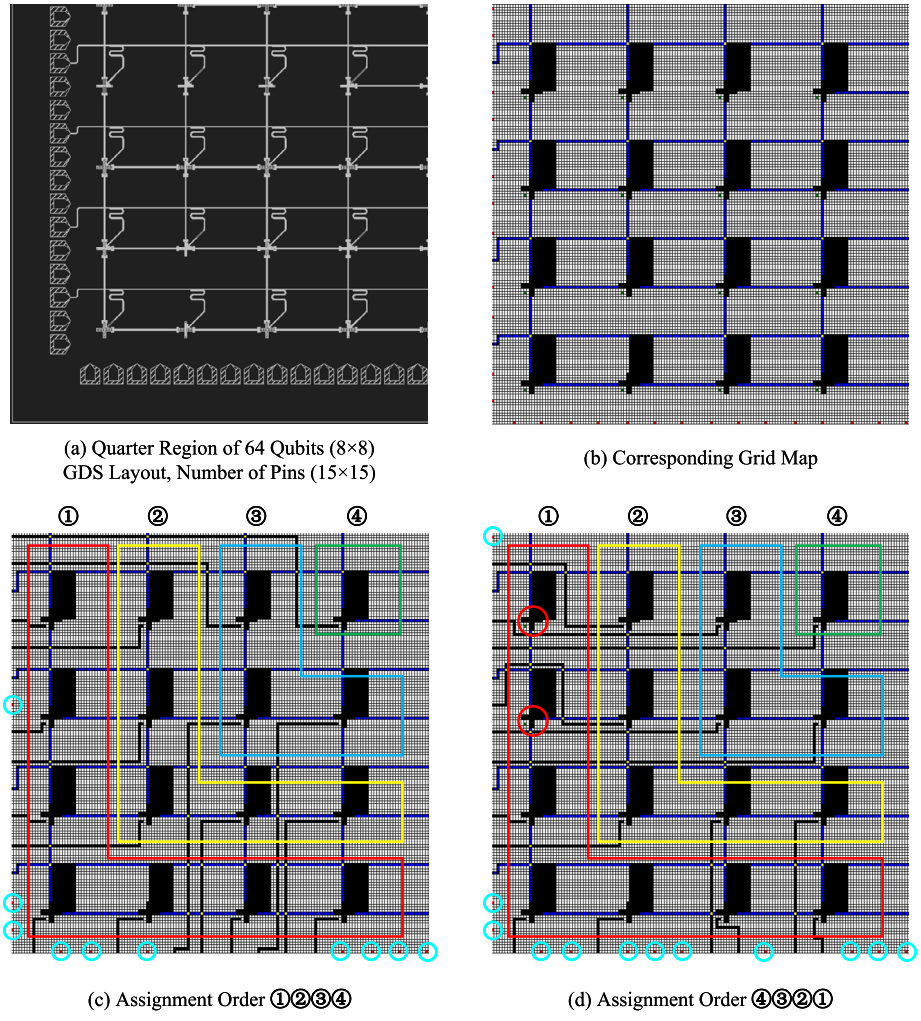
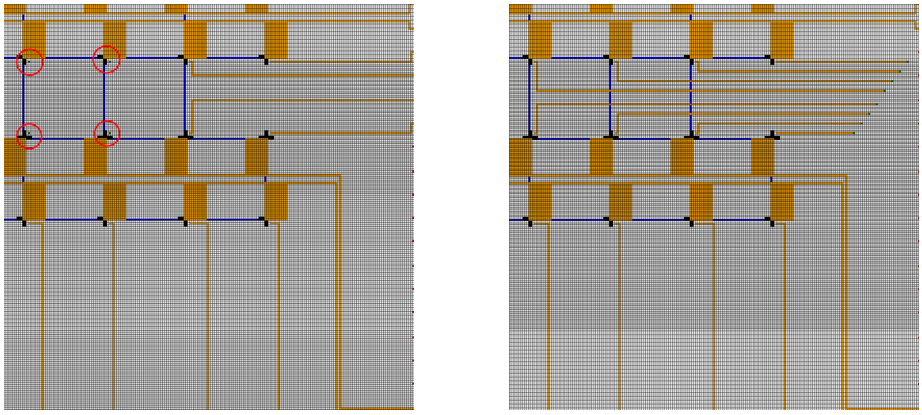


Figure 7 Example of the assignment order. Qubits marked with red circles indicate that the pins assignment cannot be completed and need to be backtracked and rerouted. The cyan circles mark the remaining pins

design. This is because the qubits at the front of the assignment order have fewer constraints and higher degree of freedom in searching for routing paths, occupying the routing areas for subsequent qubits and causing routing conflicts for them. To address this, we propose a starting node transformation strategy. Firstly, this strategy sets an intermediate node for each qubit based on the number of qubits, pins, and boundary in the assignment area. Then, the method of Sect. 2.2.2 is used to complete the path search from the starting node to the intermediate node, ensuring more regular routing paths. Finally, the intermediate node is set as the new starting node for pins assignment. Here, the setting of intermediate nodes considers node spacing to avoid routing congestion, and controls the distance between each intermediate node and the grid map boundary to ensure pins assignment order, as shown in Fig. 8(b). The detailed pseudocode is shown in Appendix D.

C. Backtracking algorithm and greedy strategy According to the determined assignment order, we assign corresponding pin ports to the qubits within the region. Although the assignment order is determined, minimizing the number of crossovers and corners within



(a) Routing Problem for Flip-chip Architecture

(b) Starting Node Conversion for Flip-chip Architecture

Figure 8 Example of the starting node conversion for the flip-chip architecture. In (a), the qubits within the red circles indicate that they cannot find the routing path (*i.e.*, they cannot be assigned to valid pins), requiring backtracking and adjustment of the earlier routing scheme according to the assignment order

each region still requires evaluating the routing results of at most $A_{M_k}^{N_k}$ different assignment schemes (of course, many acceptable solutions and simple heuristic methods can be used to avoid looking through clearly suboptimal and duplicate solutions to reduce the number of comparisons). Here, M_k and N_k are the number of pins and the number of qubits in the assignment region k, respectively. $A_{M_k}^{N_k}$ is the number of combinations of all pins assignment schemes. As the number of qubits and pins increases, this process becomes time-consuming and challenging. Therefore, to reduce the search cost in the solution space while obtaining a reasonable pins assignment scheme, we introduced the backtracking algorithm and greedy strategy into the pins assignment process.

Specifically, based on the assignment order and the available pins, we use the method from the previous subsection to select the pin port j for the current qubit i (Q_i) that minimizes $f(path_{ij})$ (where $f(path_{ij})$ is used to evaluate the number of crossovers and corners in the routing, as described in Sect. 2.2.1), assigning it as the pin for Q_i. By making a locally optimal pin choice for each qubit, we progressively build the pins assignment scheme. Making a locally optimal choice at each step is called the greedy strategy [37]. The decision is made based on the currently available information, without considering the future consequences of these decisions. Although the greedy strategy may not always find the optimal solution, it can reduce the search space (not requiring consideration of all assignment combinations) and help find a reasonable pins assignment scheme.

Additionally, to prevent the situation where no feasible routing path can be found for the pins assignment of Q_i, we adopt the backtracking algorithm. If a suitable path cannot be found when a qubit is assigned to different pins, we will backtrack to the pin assignment of the previous qubit according to the qubit assignment order and make adjustments (such as selecting a suboptimal pin assignment), allowing the subsequent qubits to complete the routing design. During the search process, if the system discovers at a certain step that the originally selected option does not achieve the goal, it will go back to the previous step and re-select another option, which is the idea of backtracking algorithm [38]. In practical implementation, this involves rolling back the assignment process to the pin assignment of

Q_{i-1} (following the assignment order of the qubits), putting the original pins assignment scheme for Q_{i-1} into a taboo list to prohibit the search, and then reselecting the pin port j that minimizes $f(path_{i-1j})$ as the new assignment scheme for Q_{i-1}. Afterward, we proceed with the pins assignment for Q_i. If replacing all assignable pins of Q_{i-1} fails to find a feasible routing path for the pins assignment of Q_i, then backtrack to the pins assignment of Q_{i-2}. At the same time, release all the pins assignment schemes related to Q_{i-1} from the taboo list and put the original pins assignment scheme for Q_{i-2} into the taboo list. Then restart the pins assignment process for Q_{i-2} and continue this process iteratively. According to our strategy, in the best case the number of evaluation is reduced to $N_k(2M_k - N_k + 1)/2$. The detailed pseudocode is shown in Appendix E.

2.3 GDS routing (physical routing)

After completing the grid map routing, we perform QPU layout routing based on the grid paths. This process is mainly divided into three steps: paths conversion, paths correction, and paths drawing, as detailed below:

2.3.1 Paths conversion

In the grid path, we have saved each grid coordinate from the starting point of the control line to the ending point of the pin port. However, different coordinate systems prevent direct application to the QPU layout. To address this, we designed a conversion algorithm to transform grid paths into GDS paths. The main idea is to determine the direction, distance, and corner processing of the routing path in the QPU layout based on the numerical changes in grid coordinates. The detailed pseudocode is shown in Appendix F.

2.3.2 Paths correction

Since the grid coordinates are obtained by rounding the numerical values on the GDS coordinates, the GDS path converted from the grid path may have deviations on the QPU layout (see Fig. 9). To ensure the reliability of the GDS path, we design a correction algorithm for the GDS path. The correction strategy is as follows:

I. Calculate the deviation value of the ending point of the GDS_path on the horizontal and vertical coordinates;
II. Reverse access the segmented paths on the GDS_Path (starting the correction from the segment closest to the starting point of the qubit control line);
III. Perform distance correction based on the direction of the segmented paths;
IV. If the distance of the corrected path is positive, end the correction;

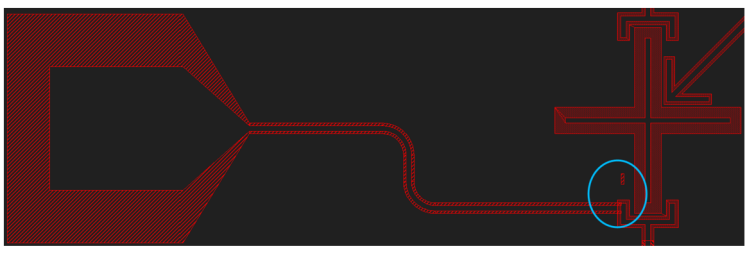


Figure 9 Example of the routing deviation on the QPU layout

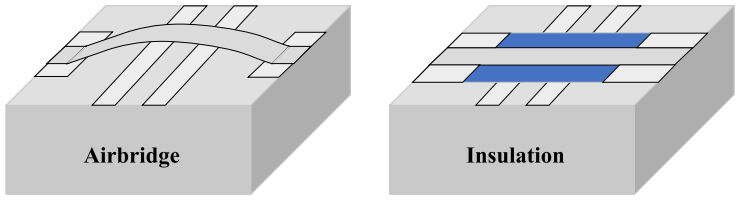


Figure 10 Example of two types of crossovers

V. Otherwise, keep the original distance, move down to the next segmented path, and proceed to step III.

The detailed pseudocode is shown in Appendix G.

2.3.3 Paths drawing

After completing the paths correction, the paths will be drawn on the QPU layout. By traversing the straight path and corner information stored in the GDS_Path, call the Path function in the gdspy [39] (a Python package) to complete the automatic routing. For automatic crossover processing, we support two common types of crossovers: airbridge and insulation (as shown in Fig. 10), with the design shapes referring to the Ref. [40, 41]. The processing methods for different crossover types are different. When the crossover type is airbridge, the routing path is first drawn, then the routing path is cut based on the crossover coordinates (Boolean operation), and finally, the airbridge is drawn at the cutting position. When the crossover type is insulation, we first draw a rectangular shape of the insulation with crossover coordinates as the center, and then draw the routing path. Since the initial crossover coordinates are converted based on the grid coordinates, it is also necessary to correct the crossover coordinates to accurately complete the path drawing on the QPU layout. Here, the crossover coordinates are corrected according to the revised GDS path. The detailed pseudocode is shown in Appendix H.

3 Experimental simulation and result analysis

To validate the effectiveness of our method, we conducted routing experiments using five types of qubit numbers. Specifically, qubit numbers of 16 (4×4), 24 (4×6), 32 (4×8), and 64 (8×8) were used for planar architecture routing design, while a scale of 128 (16×8) was used for flip-chip routing design. In the components placement of the processor layout, the "Transmon + Bus Resonator Coupler" design is used for qubit numbers of 16 and 24, while the for "Xmon + Direct Coupler" design is used for all qubit numbers. Here, we did not choose to conduct larger-scale routing design experiments using the "Transmon + Bus Resonator Coupler" design, primarily because this design produces more crossovers and corners in the routing path compared to the for "Xmon + Direct Coupler" design (see Sect. 3.2). The topology structures of the processors with different qubit numbers are randomly generated by the program. Detailed topology information (adjacency matrices) can be found in the supplementary materials. The shape of all components in the processor layout refer to Qiskit Metal [29] and Refs. [41, 42]. Here, for the convenience of component projections, the dimensions of each type of component are kept consistent, without considering the specific Hamiltonian parameters design. In the "Transmon + Bus Resonator Coupler" design, the size range of the Transmon is 650 μm $\times$ 650 μm, and the length of

the readout resonator is 4369.7 μm. In the "Xmon + Direct Coupler" design, the size range of the Xmon is 350 μm × 350 μm, and the length of the readout resonator is 3000 μm. The remaining information on the QPU layout for different qubit scales can be found in the table in Appendix I.

The control lines in superconducting quantum processors are implemented through co-planar waveguides (CPW). In the "Transmon + Bus Resonator Coupler" design, the control line starting point for the rightmost qubits is on the right side of the qubits, while for the others, it is on the left side. In the "Xmon + Direct Coupler" design based on the planar architecture, the control line starting point for the qubits in the lower right area is on the right side of the qubits, while for the others, it is on the left side. In the "Xmon + Direct Coupler" design based on the flip-chip architecture, the control line starting point for the qubits in the first $col/2$ columns (where col is the number of columns of qubits in the topology) is on the left side of the qubits, while for the others, it is on the right side. The parameter settings of CPW vary depending on the substrate material. In general, when the substrate material is sapphire, the width of the centre conductor s is 10 μm, the width of the gap between the centre conductor and the ground planes is 5 μm, and the line width is 20 μm; when the substrate material is high-resistance silicon, s is 10 μm, w is 6 μm, and the line width is 22 μm. The radius of the routing corners and the minimum spacing of the routing paths d_1 are set to 50 μm and 30 μm, respectively with reference to previous works [43, 44] and design experience. Meanwhile, the minimum spacing between the routing paths and obstacles d_2, and the minimum spacing between the routing paths and the crossover area without crossover points d_3 are also set to 30 μm as well. Set weight factors α, β, and γ to 0.5, 0.3, and 0.2, respectively. The hardware platform is: Intel(R) Core(TM) i9-13900KF 3.0 GHz.

3.1 Comparison of execution time

To compare the processor routing efficiency of our method on different qubit scales, we chose to compare the time of serial and parallel execution with the improved A star algorithm adopted by Qiskit Metal. Specifically, we replaced the search algorithm with the improved A star algorithm in the pins assignment strategy and compared it with our method. For each qubit scale, we ran each method five times separately on a single topology, recording the runtime for each processor routing completion and calculating the average time. In parallel execution, we create the corresponding number of processes based on the divided areas to complete the overall routing design. Table 2 shows the time results (rounded to two decimal places) of two methods for serial and parallel execution at different qubit scales. The number of pins in Table 2 does not include the pins occupied by transmission lines.

As can be seen from Table 2, the execution time of both methods in serial and parallel modes grows with the increase in routing complexity (the number of qubits, pins and the size of the grid map). This is due to the larger number of qubits, pins, and larger grid maps, which increase the possibility of routing schemes (*i.e.*, increase the search space for routing paths), resulting in increased runtime for both when searching for better routing schemes. In the 64-qubit case, we reduced the processor size and the corresponding number of grids, aiming to decrease the search range of routing paths and thus reduce search time. However, the time overhead of executing the routing algorithm remains high. This is due to the fact that there are more qubits and pins in the 64-qubit case, which increases the

Table 2 Comparison of execution time

Number of pins 16	Serial execution					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	179.84 s	180.44 s	180.29 s	179.66 s	179.82 s	180.01 s
Our Method	**60.30 s**	**61.07 s**	**60.42 s**	**61.00 s**	**60.91 s**	**60.74 s**

Number of pins 16	Parallel execution (4 processes)					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	35.49 s	36.00 s	35.50 s	35.48 s	35.61 s	35.62 s
Our Method	**10.56 s**	**10.62 s**	**10.62 s**	**10.85 s**	**10.77 s**	**10.68 s**

(a) 16 Qubits for "Xmon + Direct Coupler" (Number of grids 248×248)

Number of pins 16	Serial execution					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	191.20 s	190.80 s	192.80 s	191.20 s	191.20 s	191.44 s
Our Method	**68.28 s**	**68.16 s**	**68.19 s**	**68.28 s**	**68.13 s**	**68.21 s**

Number of pins 16	Parallel execution (4 processes)					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	42.00 s	42.00 s	42.00 s	30.00 s	45.20 s	40.24 s
Our Method	**12.77 s**	**12.74 s**	**12.74 s**	**12.72 s**	**12.74 s**	**12.74 s**

(b) 16 Qubits for "Transmon + Bus Resonator Coupler" (Number of grids 248×248)

Number of pins 40	Serial execution					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	958.39 s	962.41 s	962.28 s	959.01 s	960.07 s	960.43 s
Our Method	**476.10 s**	**470.37 s**	**474.63 s**	**468.89 s**	**475.20 s**	**473.04 s**

Number of pins 40	Parallel execution (4 processes)					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	427.56 s	421.80 s	423.09 s	425.00 s	421.95 s	423.88 s
Our Method	**183.62 s**	**189.77 s**	**186.36 s**	**183.40 s**	**185.91 s**	**185.81 s**

(c) 24 Qubits for "Xmon + Direct Coupler" (Number of grids 380×380)

Number of pins 40	Serial execution					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	1002.60 s	1005.50 s	1005.50 s	1009.10 s	1003.50 s	1005.24 s
Our Method	**495.25 s**	**496.00 s**	**498.40 s**	**508.70 s**	**495.90 s**	**498.85 s**

Number of pins 40	Parallel execution (4 processes)					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	452.95 s	451.31 s	464.47 s	454.00 s	451.58 s	454.86 s
Our Method	**205.03 s**	**202.25 s**	**204.50 s**	**203.73 s**	**203.31 s**	**203.76 s**

(d) 24 Qubits for "Transmon + Bus Resonator Coupler" (Number of grids 380×380)

Number of pins 56	Serial execution					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	1271.00 s	1267.00 s	1256.40 s	1266.10 s	1263.40 s	1264.78 s
Our Method	**662.10 s**	**664.80 s**	**662.60 s**	**661.20 s**	**664.30 s**	**663.00 s**

Number of pins 56	Parallel execution (4 processes)					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	658.22 s	660.04 s	659.01 s	656.35 s	660.44 s	658.81 s
Our Method	**316.36 s**	**316.15 s**	**316.25 s**	**315.33 s**	**314.88 s**	**315.79 s**

(e) 32 Qubits for "Xmon + Direct Coupler" (Number of grids 408×408)

Table 2 (*Continued*)

Number of pins 104	Serial execution					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	6712.10 s	6709.10 s	6726.50 s	6683.50 s	6717.00 s	6709.64 s
Our Method	**3595.00 s**	**3588.50 s**	**3595.80 s**	**3595.50 s**	**3597.10 s**	**3594.38 s**
Number of pins 104	Parallel execution (4 processes)					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	3131.87 s	3047.78 s	3081.13 s	3030.00 s	3198.48 s	3097.85 s
Our Method	**1552.25 s**	**1552.41 s**	**1552.33 s**	**1554.56 s**	**1549.61 s**	**1552.23 s**

(f) 64 Qubits for "Xmon + Direct Coupler" (Number of grids 318×318)

Number of pins 288	Serial execution					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	10,210.80 s	10,184.70 s	10,207.80 s	10,165.00 s	10,207.10 s	10,195.08 s
Our Method	**5764.00 s**	**5757.00 s**	**5731.80 s**	**5739.30 s**	**5750.40 s**	**5748.50 s**
Number of pins 288	Parallel execution (16 processes)					**Avg.**
	1	2	3	4	5	
Improved A* algorithm	3480.24 s	3476.60 s	3480.49 s	3485.56 s	3483.29 s	3481.24 s
Our Method	**2035.74 s**	**2031.35 s**	**2028.98 s**	**2027.36 s**	**2028.50 s**	**2030.39 s**

(g) 128 Qubits for "Xmon + Direct Coupler" (Number of grids 748×748)

time cost of pin assignment. At the same time, this also indicates that there is still room for improvement and optimization in our method. In each subscale, the execution time of our method outperforms the improved A star algorithm in both serial and parallel modes. In five qubit scales, compared with the improved A star algorithm, the average execution time of our method is at least 43.61% shorter in serial mode and 41.68% shorter in parallel mode. The main reason is that the bidirectional A star algorithm can perform bidirectional search from both the starting and ending nodes simultaneously, reducing unnecessary search space and accelerating the path search process. Secondly, we use the binary tree-based Min-heap to implement the open list data structure of the bidirectional A star algorithm, which reduced the time complexity of operations such as nodes insertion, deletion, and sorting, ensuring efficient path searching. In addition, building multiple processes based on the divided areas can parallelly execute routing methods. Compared to the serial mode, the average execution time of our method in the parallel mode has been reduced by at least 52.37%, which verifies the rationality of the parallel pins assignment and routing design by dividing regions.

In addition, when comparing the execution time between the "Xmon + Direct Coupler" design and the "Transmon + Bus Resonator Coupler" design, the "Xmon + Direct Coupler" design has a shorter execution time. In our method, the serial average execution time based on the "Xmon + Direct Coupler" design is reduced by at least 5.17% compared to the "Transmon + Bus Resonator Coupler" design, while the parallel average execution time is reduced by at least 8.81%. This is mainly because the "Transmon + Bus Resonator Coupler" design introduces more obstacles after component projection, requiring the algorithm to search through more nodes to find a routing path. Figure 11(a) shows a comparison of the obstacle placement after component projection for both designs on the same 24-qubit topology. One can see that the "Transmon + Bus Resonator Coupler" design (on the left) has more obstacles, which limits the number of feasible routing paths. As a result, the algorithm needs to explore more nodes to find a viable path, thereby increasing the time cost.

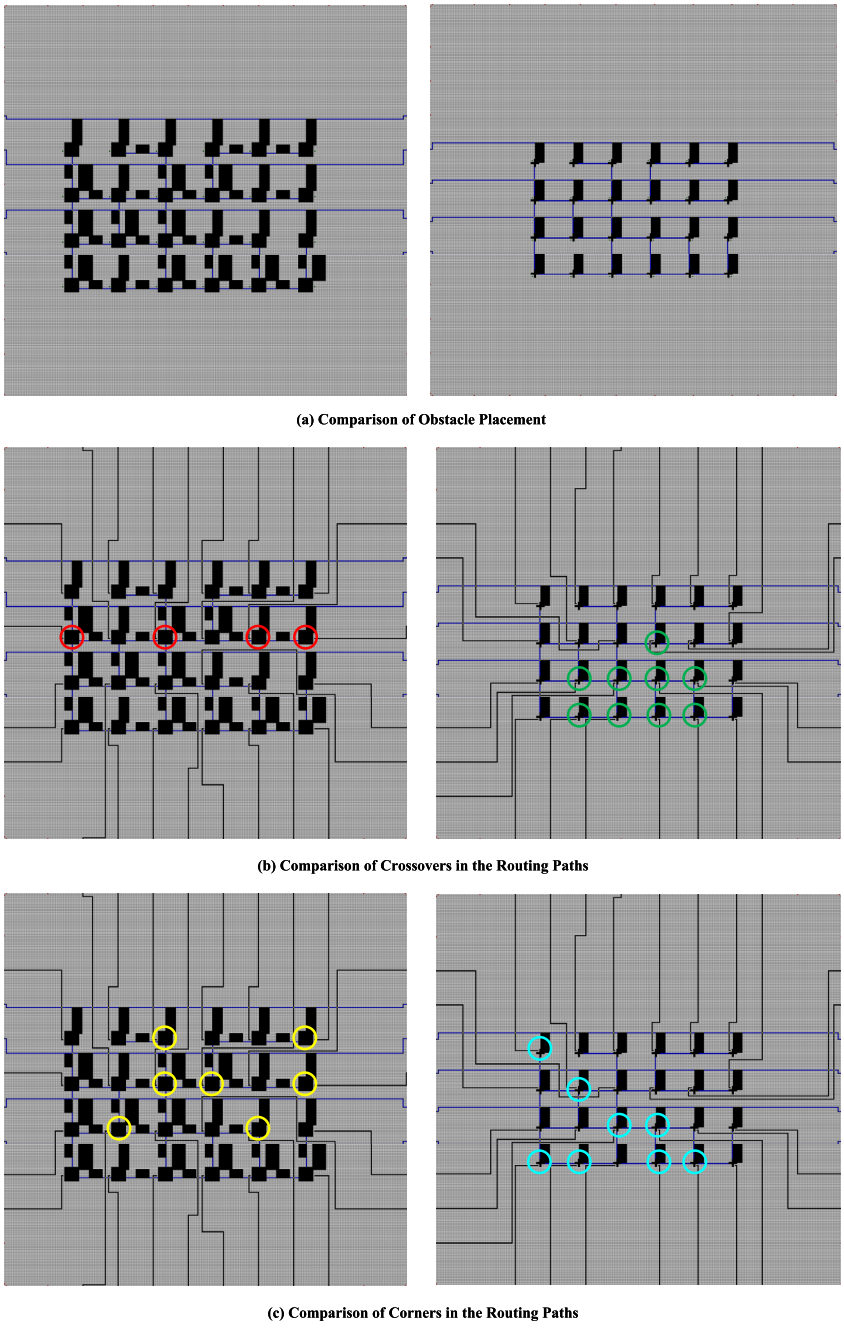


Figure 11 Example of routing comparison between the "Transmon + Bus Resonator Coupler" design (left) and the "Xmon + Direct Coupler" design (right) on a 24-qubit topology. The qubits marked with red (yellow) circles indicate that the number of crossovers (corners) in their routing paths is lower in the "Transmon + Bus Resonator Coupler" design than that in the "Xmon + Direct Coupler" design. Conversely, the qubits marked with green (cyan) circles indicate that the number of crossovers (corners) in their routing paths is lower in the "Xmon + Direct Coupler" design than that in the "Transmon + Bus Resonator Coupler" design

3.2 Comparison of crossovers and corners optimization

In order to verify the effectiveness of our method for the optimization of the number of crossovers and corners, we compare it with the standard bidirectional A star algorithm,

the bidirectional A star algorithm without corner removal step (No corner removal), the path search algorithm of Ref. [35], and random pins assignment algorithm. Among them, the standard bidirectional A star algorithm takes the shortest distance as the optimization objective in the path search, and does not consider the number of crossovers and corners. No corner removal is the same as our method, except for not considering the corner removal step. Ref. [35] calculates the desired node based on the parent node index to reduce the number of corners, without considering crossovers optimization. The random pins assignment algorithm does not consider our pins assignment strategy, and it generates 5 feasible pins assignment schemes (ensuring that each qubit can complete routing), while the rest is consistent with our method. We randomly generated three topologies for each qubit scale to compare the optimization effects of different methods on the number of crossovers and corners. The experimental results are shown in Fig. 12, Table 3, and Table 4.

Figure 12 and Table 3 show the comparison of the number of crossovers and corners in routing designs using five different methods based on the planar architecture. In Fig. 12, the red star represents our method, the green circle represents No corner removal method, the blue triangle represents the path search algorithm proposed in Ref. [35], and the black

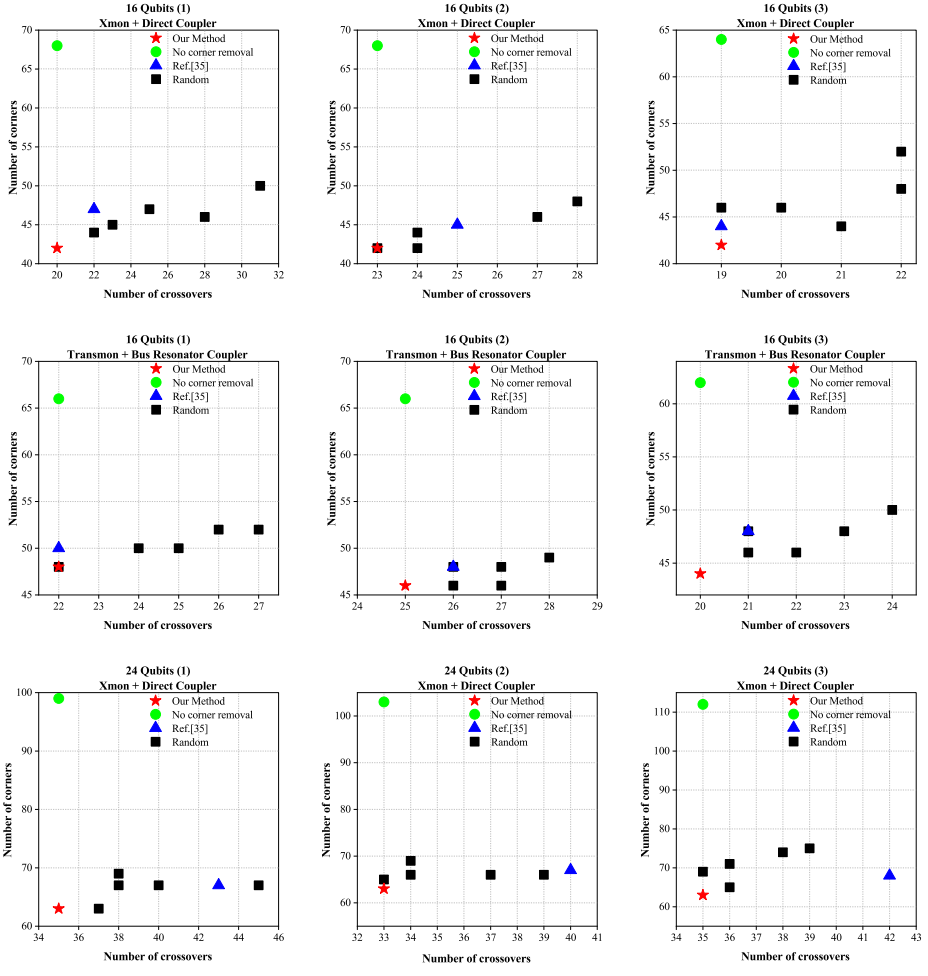


Figure 12 Comparison of the crossover and corner numbers

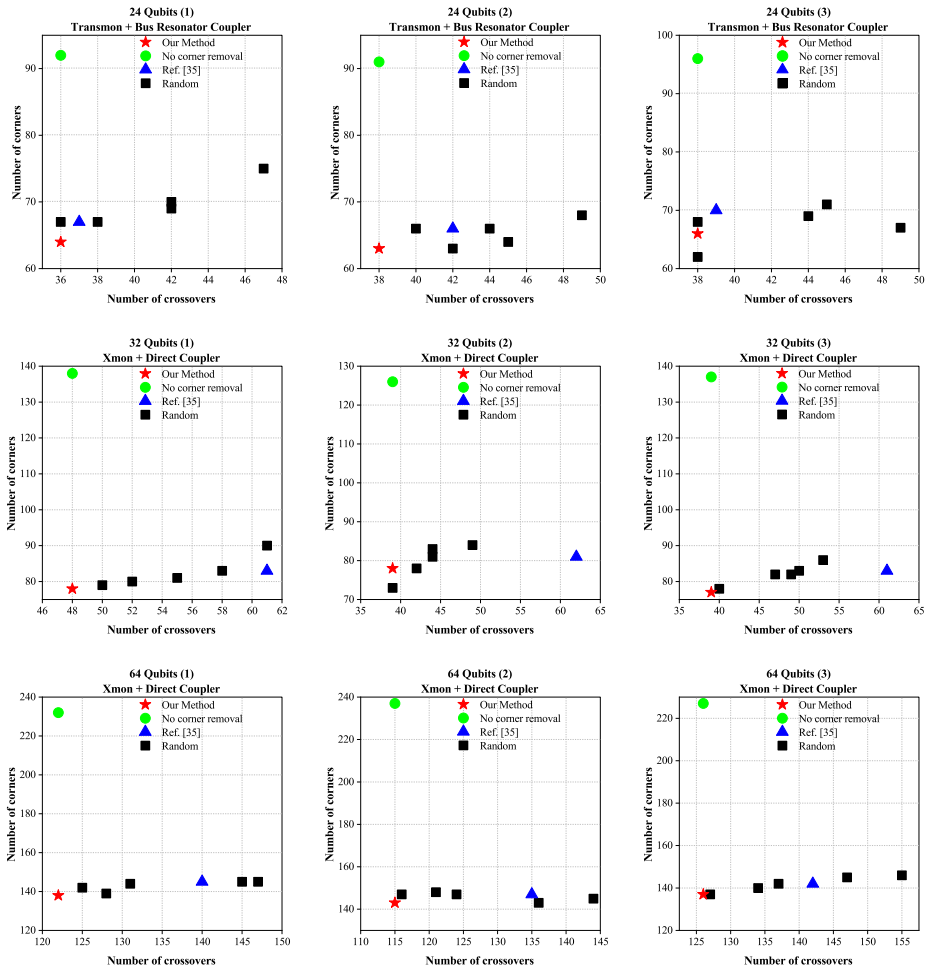


Figure 12 Continued

Table 3 The number of crossovers and corners of the standard bidirectional A star algorithm

	Standard bidirectional A star algorithm	
	Number of crossovers	Number of corners
16 Qubits (1) Xmon + Direct Coupler	22	690
16 Qubits (2) Xmon + Direct Coupler	25	698
16 Qubits (3) Xmon + Direct Coupler	19	710
16 Qubits (1) Transmon + Bus Resonator Coupler	22	585
16 Qubits (2) Transmon + Bus Resonator Coupler	26	577
16 Qubits (3) Transmon + Bus Resonator Coupler	21	632
24 Qubits (1) Xmon + Direct Coupler	43	793
24 Qubits (2) Xmon + Direct Coupler	40	806
24 Qubits (3) Xmon + Direct Coupler	42	812
24 Qubits (1) Transmon + Bus Resonator Coupler	37	744
24 Qubits (2) Transmon + Bus Resonator Coupler	42	715
24 Qubits (3) Transmon + Bus Resonator Coupler	39	709
32 Qubits (1) Xmon + Direct Coupler	61	958
32 Qubits (2) Xmon + Direct Coupler	62	991
32 Qubits (3) Xmon + Direct Coupler	61	974
64 Qubits (1) Xmon + Direct Coupler	140	924
64 Qubits (2) Xmon + Direct Coupler	135	908
64 Qubits (3) Xmon + Direct Coupler	142	897

square represents the random pins assignment algorithm. Due to the large number of corners in the standard bidirectional A star algorithm, the number of crossovers and corners for this algorithm are presented separately in Table 3. Combining Fig. 12 and Table 3, it can be seen that our method and No corner removal method have lower crossover numbers for each processor. Compared to the standard bidirectional A star algorithm and the path search algorithm in Ref. [35], the average number of crossovers is reduced by 12.59%. Compared to the random pins assignment algorithm, the average number of crossovers is reduced by 10.63%. The main reason is that our method and No corner removal method increase the crossover cost in the heuristic evaluation function, while the standard bidirectional A star algorithm and Ref. [35] do not consider optimizing the number of crossovers in the path search. Although the random pins assignment algorithm uses the same heuristic evaluation function as our method and achieves comparable crossover numbers to our method in some results, its outcomes are probabilistic due to the lack of consideration for global crossovers optimization during pins assignment.

In terms of corner numbers comparison, the standard bidirectional A star algorithm only considers distance cost, resulting in the largest number of corners. No corner removal method adds corner costs in the heuristic evaluation function, reduces the number of corners by at least 73.9% compared to the standard bidirectional A star algorithm. Ref. [35] optimized the number of corners by calculating the desired nodes, and compared to No corner removal method, the minimum number of corners was reduced by 14. Our method can further reduces the number of corners by performing the corner removal step after adding corner cost. The average reduction in the number of corners is 5.96% compared to Ref. [35], and the minimum reduction in the number of corners compared to No corner removal method is 18. This verifies the effectiveness of our method in adding corresponding penalty terms to the heuristic evaluation function and conducting the corner removal step. In the same way, due to the probabilistic nature of the random pins assignment algorithm, it cannot guarantee to obtain the optimal number of corners every time. Compared to the random pins assignment algorithm, our method reduces the number of corners by an average of 5.42%. However, in the cases of 24 Qubits (3) and 32 Qubits (2), the random pins assignment algorithm obtained better pins assignment schemes with fewer corners. The specific analysis is shown in Fig. 13. It shows that the routing results of our method and the random pins assignment algorithm on the 24 Qubits (3) and 32 Qubits (2) grid maps. Figures 13(a) and 13(c) are the routing results of our method, while Figs. 13(b) and 13(d) are the routing results of the random pins assignment algorithm. We have marked the different pins assignment schemes for the same qubits under the two methods with circles and numbers, specifically divided into two cases: red circles and green circles. The reason for marking with red circles is that our method in the simulation experiment set a larger weight for the path length γ in the objective function $f(path_{ij})$. As a result, in some path searches, the variation in the number of corners is overlooked, leading to our method not selecting the pins assignment scheme with fewer corners. We changed β and γ from 0.3 and 0.2 to 0.4 and 0.1 respectively for corresponding qubits. That is, by increasing the weight of corners, we can obtain the same corner results as the random pins assignment algorithm. The reason for marking with green circles is that our method divides the global grid map into regions, performing local pin assignments for qubits in different regions. In contrast, the random pins assignment algorithm randomly assigns pins across the entire grid map. As a result, the pins available for selection differ between the two methods,

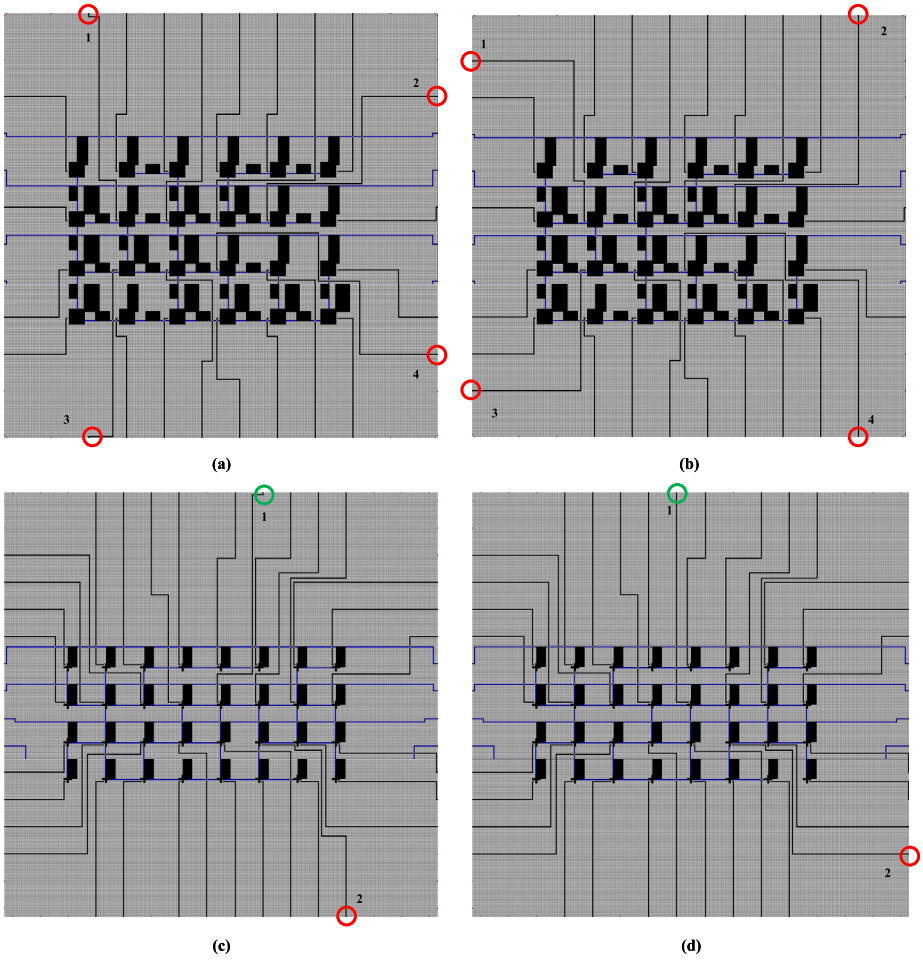


Figure 13 Analysis of special cases

leading to different final routing results. In the next step of the research, we will comprehensively consider the pins assignment results from both the local and global perspectives to improve our method, further optimizing the number of crossovers and corners.

Moreover, in our method, the routing paths based on the "Xmon + Direct Coupler" design have fewer crossovers and corners compared to the "Transmon + Bus Resonator Coupler" design. Compared to the "Transmon + Bus Resonator Coupler" design, the routing path based on the "Xmon + Direct Coupler" design shows a maximum reduction of 13.16% in the number of crossovers, with an average reduction of 7.65%. The number of corners is reduced by a maximum of 12.5%, with an average reduction of 5.31%. This is mainly due to the more complex layout of components in the "Transmon + Bus Resonator Coupler" design. We take the topology of the 24 Qubits (3) as an example for analysis (Fig. 11). Figure 11(a) shows the projection results of two designs under the same topology. One can see that after component projection, the placement of obstacles in the grid map based on the "Transmon + Bus Resonator Coupler" design is more complex. Compared to the "Xmon + Direct Coupler" design, it introduces more routing path search constraints, which may lead to an increased number of crossovers and corners. Figure 11(b) and Fig. 11(c) show a comparison of the crossovers and corners in the routing paths based

Table 4 Comparison of the number of corners in the flip-chip architecture (128 Qubits)

	Our method	No corner removal	Ref. [35]	Standard bidirectional A* algorithm	Random				
					1	2	3	4	5
(1)	**408**	610	413	5662	418	416	408	418	411
(2)	**408**	610	413	5662	432	413	424	422	427
(3)	**408**	610	413	5662	424	425	416	421	430

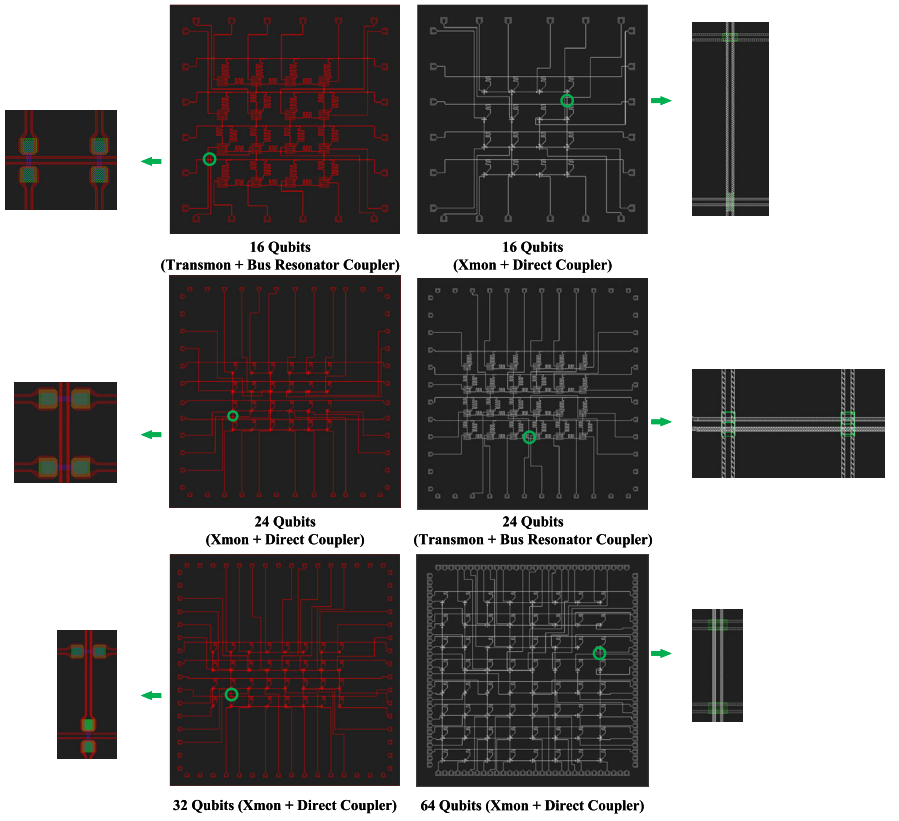
on the two designs, respectively. The qubits highlighted with red (yellow) circles indicate that their routing paths have fewer crossovers (corners) in the "Transmon + Bus Resonator Coupler" design than in the corresponding qubits of the "Xmon + Direct Coupler" design. In contrast, the qubits highlighted with green (cyan) circles indicate that their routing paths have fewer crossovers (corners) in the "Xmon + Direct Coupler" design than in the corresponding qubits of the "Transmon + Bus Resonator Coupler" design. One can see that, due to the simpler obstacle placement in the "Xmon + Direct Coupler" design, more qubits have routing paths with fewer crossovers and corners. Overall, compared to the "Transmon + Bus Resonator Coupler" design, the number of crossovers and corners is reduced by 7.89% and 4.55%, respectively. When there are more qubits and connections in the processor, the "Xmon + Direct Coupler" design has a greater advantage for routing design. Therefore, we chose to conduct larger-scale routing design experiments based on the "Xmon + Direct Coupler" design.

In addition, we conduct a comparison of the number of corners (128 qubits) for five routing methods under the flip-chip architecture, as shown in Table 4. Since we have set the control lines not to be on the same layer as the coupling components in the placement design based on the flip-chip architecture, and the transmission lines and the existing control lines are not considered as crossover areas (see Sect. 2.1.2), there are no crossover points in the routing design based on the flip-chip architecture in this paper. Meanwhile, since the processor topologies are not in the same layer as the routing schemes and do not consider crossover constraints in the mathematical model and algorithm design, each routing scheme has the same number of corners (except for random assignment) under different topology structures. From the Table 4, it can be seen that the number of corners in our method is more optimal. Compared to the standard bidirectional A star algorithm, No corner removal method, and the path search algorithm in Ref. [35], our method has a minimum average reduction of 1.21% in the number of corners. This further demonstrates the design validity of our method. Compared to the random pins assignment algorithm, our method reduces the number of corners by an average of 12, and the result is not random, which further demonstrates the rationality of the pins assignment strategy in our method.

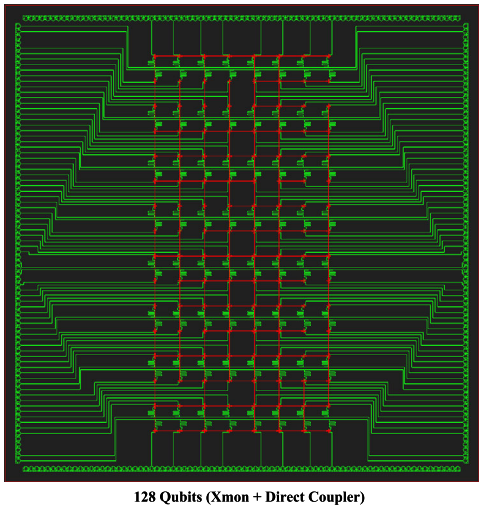
Finally, we compared the routing results of the standard bidirectional A star algorithm and our method under the planar architecture with 32 Qubits and the flip-chip architecture with 128 Qubits, to demonstrate the optimization effects of our method on the number of crossovers and corners. Details can be found in Appendix J.

3.3 The display of GDS routing

After optimizing for crossovers and corners on the grid map routing, the GDS routing will be conducted based on the grid paths. We also chose five qubit scales (16, 24, 32, 64 and 128) for GDS routing experiments. The components placement of the processors are the same as before, and the routing results are shown in Fig. 14. Figure 14(a) shows the GDS



(a) GDS Routing Results for Planar Architecture



(b) GDS Routing Results for Flip-chip Architecture

Figure 14 GDS routing results for planar and flip-chip architectures

routing results for 16, 24, 32, and 64 qubit scales based on the planar architecture. For the 16 and 32 qubit scales, airbridges are used for crossover processing (shown in red layout), while for the 24 and 64 qubit scales, insulations are used for crossover handling (shown in gray layout). Meanwhile, we zoom in to show the two crossover types (marked by green arrows). Figure 14(b) shows the routing results of a 128 qubits QPU layout based on flip-

chip architecture. Among them, red represents Q-Chip, and green represents C-Chip. As can be seen from Fig. 14, our method supports the routing design of both planar and flip-chip architectures. It also supports the automatic processing of airbridge and insulation crossovers, to assist in the completion of superconducting quantum processor designs.

4 Conclusion and outlook

In this paper, we proposed an automated routing method to address the routing challenges faced in the design of large-scale superconducting quantum processors. Our method can efficiently complete the processor routing design while optimizing the number of crossovers and corners. Our method is mainly implemented through the bidirectional star algorithm, the backtracking algorithm, and the greedy strategy, specifically including three steps: layout projection, grid map routing, and GDS routing. The experimental results indicate that our method demonstrates significant improvement in both serial and parallel execution efficiency, as well as optimization of crossovers and corners. These validate the effectiveness and reliability of our method. Additionally, our method supports the processor routing design of planar and flip-chip architectures. It is also capable of automatically handling two types of crossovers, airbridge and insulation. These can efficiently assist in the design of large-scale superconducting quantum processors.

Of course, our method also has its shortcomings. Due to the prioritized use of routing resources by earlier routing designs, our method of routing according to the assignment order may result in backtracking and rerouting, increasing the cost of the routing design. However, simultaneously routing can avoid the above problem. By optimizing the design of the heuristic function, the computational cost of simultaneously routing can be reduced. Even if there are no routing conflicts, the simultaneous routing scheme can still reduce the number of routing path searches. Therefore, as the routing scale becomes increasingly complex, one should consider simultaneous routing solution. In future, we will consider automated routing designs that include simultaneous routing design, tunable couplers, a combination of local and global region routing designs, and through-silicon via (TSV) architecture.

Appendix A: Grid map initialization

The calculation for the horizontal and vertical ranges of the grid map is as follows:

$$S_{\text{grid\_map\_X}} = \left\lfloor (S_{\text{processor\_GDS\_Y}} - d_{\text{pins\_t}} - d_{\text{pins\_b}})/\Delta_{\text{set}} + 0.5 \right\rfloor \tag{A.1}$$

$$S_{\text{grid\_map\_Y}} = \left\lfloor (S_{\text{processor\_GDS\_X}} - d_{\text{pins\_l}} - d_{\text{pins\_r}})/\Delta_{\text{set}} + 0.5 \right\rfloor \tag{A.2}$$

where $d_{\text{pins\_t}}$ represents the distance from the port of the top-side pin to the top-side boundary of the processor. Similarly for $d_{\text{pins\_b}}$, $d_{\text{pins\_l}}$, and $d_{\text{pins\_r}}$, as shown in Fig. 3(a). The connection ports of the top and bottom pins are oriented downwards and upwards, respectively, while the connection ports of the left and right pins are oriented to the right and left. Assuming the pin port node is Node(x, y), when the pin port is oriented downwards or upwards, Node($x + 1$, y) (or Node(x-1, y)) is the mandatory node that the routing path must pass through to ensure the correct connection between the qubit control line and the pin. Similarly, when the pin port is oriented to the right (or left), Node(x, $y + 1$) (or Node(x, y-1)) is the mandatory node that the routing path must pass through. When the

starting point of the control line is on the left (right) side of the qubit, its connection port faces to the left (right). Assuming the starting point of the control line is Node(x, y), when Node(x, y) is on the left (right) side of the qubit, Node(x, y-1) (or Node(x, y + 1)) is the mandatory node that the routing path must pass through. This ensures the correct connection of the starting port of the control line. Since the boundaries of the routing paths involve only the pin ports, Eqs. (A.1) and (A.2) subtract the distances from the pin ports to the processor boundaries during the calculation to compress the data stored in the grid map. To simplify calculations, in this paper, $d_{\text{pins\_t}} = d_{\text{pins\_b}} = d_{\text{pins\_l}} = d_{\text{pins\_r}}$. Meanwhile, in Eqs. (A.1) and (A.2), rounding is applied to $S_{\text{grid\_map\_X}}$ and $S_{\text{grid\_map\_Y}}$ to ensure that the grid coordinates are non-negative integers. The reason is that we use a two-dimensional array to store the grid map, where each item in the array represents a grid. Since the indexes of the items are typically non-negative integers, we ensure that the grid coordinates are also non-negative to correspond to them. Furthermore, the coordinate axis direction of the grid map is different from the QPU layout (referring to Fig. 3(a) and Fig. 3(b)). Therefore, we calculate $S_{\text{processor\_GDS\_Y}}$ and $S_{\text{processor\_GDS\_X}}$ based on $S_{\text{grid\_map\_X}}$ and $S_{\text{grid\_map\_Y}}$, respectively. Each grid is denoted by Grid[x,y],$x \in \left[0, S_{\text{grid\_map\_X}} - 1\right]$,$y \in \left[0, S_{\text{grid\_map\_Y}} - 1\right]$, with an initial value of Grid[x,y] = 0.

Appendix B: Obstacle placement

We define pin, qubit, direct coupler, bus resonator coupler, readout resonator, and transmission line as components. For each component, the boundary range of the component projection is determined based on defined mark points (recording the corresponding coordinates of the component in the QPU layout), and combined with Δ_{set} to convert the boundary range into grid coordinates for obstacle placement. The golden dots in Fig. 4 represent the marked points. In the setting of this paper, the processor is designed as a square structure, with the number of pins (assignable) in the processor being greater than or equal to the number of qubits, and the number of pins (overall) on each boundary of the processor is the same. Here, all readout resonators are projected into rectangles with the maximum boundary range. Similar to the existing works on routing designs [45–47], we define the crossover areas based on the planar architecture as the Direct Coupler path, the straight paths of the Bus Resonator Coupler, and the non-corner paths on transmission lines (represented in blue, with the corresponding grid value of 4). The crossover points in the routing paths (represented in yellow, with the corresponding grid value of 5) are selected only from the crossover areas. When a crossover point exists in the routing path, it indicates that the circuit can physically cross the corresponding component in the crossover area using an airbridge or insulation to avoid a short circuit [40]. In terms of code processing, when selecting a node in the crossover area as the crossover point (*i.e.*, Grid[x,y] = 4 → 5), the following conditions must be met:

I. Grid[x,y] = 4;
II. Grid[$x - 1$,y] ≠ 5 and Grid[$x + 1$,y] ≠ 5 and Grid[x,$y - 1$] ≠ 5 and Grid[x,$y + 1$] ≠ 5;
III. (Grid[$x - 1$,y] = 4 and Grid[$x + 1$,y] = 4 and Grid[x,$y - 1$] = 0 and Grid[x,$y + 1$] = 0) or (Grid[$x - 1$,y] = 0 and Grid[$x + 1$,y] = 0 and Grid[x,$y - 1$] = 4 and Grid[x,$y + 1$] = 4).

II indicates that two consecutive crossover points cannot exist. III indicates that there should be no obstacles on both sides of the crossover point, allowing the crossover. When a node in the crossover area is marked as a crossover point, it can no longer be considered a searchable node, while other nodes in the crossover area can still be considered searchable.

In the design based on the flip-chip architecture, it includes two chips [42]. One is the quantum chip (Q-Chip), which contains qubits and coupling components (in the QPU layout of Fig. 4(b), shown in red). The other is the control chip (C-Chip), which includes pins, readout resonators, transmission lines, and control lines (in the QPU layout of Fig. 4(b), shown in green). The grid value for readout resonators, transmission lines, and control lines is set to 6. Here, we set that when Grid[x,y] = 6, Node(x,y) cannot be used as a routing path node, crossover point, or crossover area. Therefore, in the flip-chip architecture, the transmission line paths are no longer considered as crossover areas. In addition, no crossover is allowed between control lines in both processes. Furthermore, to avoid crosstalk caused by excessive overlap in space between the control lines on the C-Chip and the components on the Q-Chip, we project the components from both chips onto the same grid map. This means that during the routing process, obstacles from both the C-Chip and Q-Chip are considered, aiming to find reasonable routing paths. Simultaneously, since the conversion of component grid coordinates requires rounding of GDS coordinates to integers, different components may have the same grid coordinates, *i.e.* there may be overlapped obstacles.

Below, we take the projection and obstacle placement of pin port and transmon qubit as examples to introduce the conversion process of grid coordinates. The center point of the pin port is defined as the mark point for the pin. The conversion process from GDS coordinates to grid coordinates is shown in Pseudocode 1. Due to the different origin coordinates of the grid map and the QPU layout (see Fig. 3(a) and Fig. 3(b)), it is necessary to perform a transformation on $k_i^{\text{Pin\_y}}$ when calculating $G_i^{\text{Pin\_x}}$ (The first line of Pseudocode 1). Meanwhile, due to $G_i^{\text{Pin\_x}} \in \left[0, S_{\text{grid\_map\_X}} - 1\right], G_i^{\text{Pin\_y}} \in \left[0, S_{\text{grid\_map\_Y}} - 1\right]$, the grid coordinates of the pin ports on the right and bottom sides of the processor need to undergo boundary-check processing (Lines 3 to 8 of Pseudocode 1). The mark points of the transmon qubit are the top left and bottom right vertices. The projection range is determined based on the grid coordinates of these two points, and obstacle placement is completed accordingly. The detailed process is shown in Pseudocode 2. Similarly, based on the mark points defined for the remaining components, we also calculate the grid coordinates and complete obstacle placement according to the projection range.

Algorithm 1 The projection and obstacle placement of pin ports

Input: mark point of pin port i ($k_i^{\text{Pin\_x}}, k_i^{\text{Pin\_y}}$), grid map **G_M**

Output: grid coordinate of pin port i ($G_i^{\text{Pin\_x}}, G_i^{\text{Pin\_y}}$), grid map has completed obstacle placement **G_M**

1: $G_i^{\text{Pin\_x}} = \text{int}(S_{\text{processor\_GDS\_Y}} - d_{\text{pins\_t}} - d_{\text{pins\_b}} - k_i^{\text{Pin\_y}})//\Delta_{\text{set}}$
2: $G_i^{\text{Pin\_y}} = \text{int}(k_i^{\text{Pin\_x}})//\Delta_{\text{set}}$
3: **if** $\text{int}(S_{\text{processor\_GDS\_Y}} - d_{\text{pins\_t}} - d_{\text{pins\_b}} - k_i^{\text{Pin\_y}})\%\Delta_{\text{set}} == 0$ & $G_i^{\text{Pin\_x}} != 0$ **then**
4: $\quad G_i^{\text{Pin\_x}} -= 1$
5: **end if**
6: **if** $\text{int}(k_i^{\text{Pin\_x}})\%\Delta_{\text{set}} == 0$ & $G_i^{\text{Pin\_y}} != 0$ **then**
7: $\quad G_i^{\text{Pin\_y}} -= 1$
8: **end if**
9: **G_M**$[G_i^{\text{Pin\_x}}, G_i^{\text{Pin\_y}}] = 2$

Algorithm 2 The projection and obstacle placement of transmon qubits

Input: mark points of transmon qubit i $(k_i^{\text{Qtl\_x}}, k_i^{\text{Qtl\_y}})$, $(k_i^{\text{Qbr\_x}}, k_i^{\text{Qbr\_y}})$, grid map **G_M**

Output: grid map has completed obstacle placement **G_M**

$G_i^{\text{Qtl\_x}} = \text{int}(S_{\text{processor\_GDS\_Y}} - d_{\text{pins\_t}} - d_{\text{pins\_b}} - k_i^{\text{Qtl\_y}})//\Delta_{\text{set}}$
$G_i^{\text{Qtl\_y}} = \text{int}(k_i^{\text{Qtl\_x}})//\Delta_{\text{set}}$
$G_i^{\text{Qbr\_x}} = \text{int}(S_{\text{processor\_GDS\_Y}} - d_{\text{pins\_t}} - d_{\text{pins\_b}} - k_i^{\text{Qbr\_y}})//\Delta_{\text{set}}$
$G_i^{\text{Qbr\_y}} = \text{int}(k_i^{\text{Qbr\_x}})//\Delta_{\text{set}}$
for $p \leftarrow G_i^{\text{Qtl\_x}}$ to $G_i^{\text{Qbr\_x}}$ **do**
    for $q \leftarrow G_i^{\text{Qtl\_y}}$ to $G_i^{\text{Qbr\_y}}$ **do**
        G_M$[p, q] = 1$
    end for
end for

Appendix C: Corner removal step

Algorithm 3 Corner_Removal_Step

Input: grid path Grid_Path, the number of corners that need to be merged corner_number

Output: grid path after corner removal step new_Grid_Path, endpoint of the path *e_p*

s_p, *e_p*, *t_p* = Grid_Path[0]
direct, *count* = 0
for *i* ← Grid_Path.index(*s_p*)+1 to len(Grid_Path) **do**
    if *i* == 1 **then**
        if Grid_Path[*i*][0] == *t_p*[0] **then**
            direct = 0, *count*+ = 1
        else if Grid_Path[*i*][1] == *t_p*[1] **then**
            direct = 1, *count*+ = 1
        end if
    else
        if Grid_Path[*i*][0] == *t_p*[0] **then**
            temp = 0
            if *temp* != *direct* **then**
                direct = *temp*, *count*+ = 1
            end if
        else if Grid_Path[*i*][1] == *t_p*[1] **then**
            temp = 1
            if *temp* != *direct* **then**
                direct = *temp*, *count*+ = 1
            end if
        end if
    end if
    t_p = Grid_Path[*i*]
    if *count* == *corner_number* **then**
        e_p = Grid_Path[*i* – 1], **break**
    end if
    if *t_p* == Grid_Path[len(Grid_Path)-1] **then**
        e_p = *t_p*, **break**

end if
end for
if *s_p* == *e_p* **then return** Grid_Path, *e_p*
end if
t = Grid_Path.index(*s_p*) + 1
new_Grid_Path = list()
new_Grid_Path.append(*s_p*)
if *s_p*[0] > *e_p*[0] **then**
    x_d = −1
else if *s_p*[0] < *e_p*[0] **then**
    x_d = 1
else
    x_d = 0
end if
if *s_p*[1] > *e_p*[1] **then**
    y_d = −1
else if *s_p*[1] < *e_p*[1] **then**
    y_d = 1
else
    y_d = 0
end if
flag = **True**
if Grid_Path[*t*][0] == *s_p*[0] then **then**
    i = 1
    temp_x = *s_p*[0] + *i***x_d*, new_Grid_Path.append([*temp_x*, *s_p*[1]])
    while *temp_x* != *e_p*[0] **do**
        i+ = 1
        temp_x = *s_p*[0] + *i***x_d*, new_Grid_Path.append([*temp_x*, *s_p*[1]])
    end while
    i = 1
    temp_y = *s_p*[1] + *i***y_d*, new_Grid_Path.append([*e_p*[0], *temp_y*])
    while *temp_y* != *e_p*[1] **do**
        i+ = 1
        temp_y = *s_p*[1] + *i***y_d*, new_Grid_Path.append([*e_p*[0], *temp_y*])
    end while
    if new_Grid_Path not meeting routing constraints **or** increase the number of crossovers **then**
        flag = **False**
    end if
    if *flag* **then**
        for *k* ← Grid_Path.index(*e_p*)+1 to len(Grid_Path) **do**
            new_Grid_Path.append(Grid_Path[*k*])
        end for
    else
        flag = **True**, new_Grid_Path.clear()
        i = 1

```
        temp_y = s_p[1] + i*y_d, new_Grid_Path.append([s_p[0], temp_y])
        while temp_y != e_p[1] do
            i+ = 1
            temp_y = s_p[1] + i*y_d, new_Grid_Path.append([s_p[0], temp_y])
        end while
        i = 1
        temp_x = s_p[0] + i*x_d, new_Grid_Path.append([temp_x, e_p[1]])
        while temp_x != e_p[0] do
            i+ = 1
            temp_x = s_p[0] + i*x_d, new_Grid_Path.append([temp_x, e_p[1]])
        end while
        if new_Grid_Path not meeting routing constraints or increase the number of crossovers then
            flag = False
        end if
        if flag then
            for k ← Grid_Path.index(e_p)+1 to len(Grid_Path) do
                new_Grid_Path.append(Grid_Path[k])
            end for
        end if
    end if
else if Grid_Path[t][1] == s_p[1] then
    i = 1
    temp_y = s_p[1] + i*y_d, new_Grid_Path.append([s_p[0], temp_y])
    while temp_y != e_p[1] do
        i+ = 1
        temp_y = s_p[1] + i*y_d, new_Grid_Path.append([s_p[0], temp_y])
    end while
    i = 1
    temp_x = s_p[0] + i*x_d, new_Grid_Path.append([temp_x, e_p[1]])
    while temp_x != e_p[0] do
        i+ = 1
        temp_x = s_p[0] + i*x_d, new_Grid_Path.append([temp_x, e_p[1]])
    end while
    if new_Grid_Path not meeting routing constraints or increase the number of crossovers then
        flag = False
    end if
    if flag then
        for k ← Grid_Path.index(e_p)+1 to len(Grid_Path) do
            new_Grid_Path.append(Grid_Path[k])
        end for
    else
        flag = True, new_Grid_Path.clear()
        i = 1
        temp_x = s_p[0] + i*x_d, new_Grid_Path.append([temp_x, s_p[1]])
```

```
        while temp_x != e_p[0] do
            i+ = 1
            temp_x = s_p[0] + i*x_d, new_Grid_Path.append([temp_x, s_p[1]])
        end while
        temp_y = s_p[1] + i*y_d, new_Grid_Path.append([e_p[0], temp_y])
        while temp_y != e_p[1] do
            i+ = 1
            temp_y = s_p[1] + i*y_d, new_Grid_Path.append([e_p[0], temp_y])
        end while
        if new_Grid_Path not meeting routing constraints or increase the number of
    crossovers then
            flag = False
        end if
        if flag then
            for k ← Grid_Path.index(e_p)+1 to len(Grid_Path) do
                new_Grid_Path.append(Grid_Path[k])
            end for
        end if
    end if
end if
if flag then
    Return new_Grid_Path, e_p
else
    Return Grid_Path, e_p
end if
```

Appendix D: Starting node conversion algorithm

Algorithm 4 Starting_Node_Conversion

Input: grid map **G_M**, starting nodes list start_control_grid_points, pin ports list in the region end_grid_points

Output: new starting nodes list start_control_grid_points

```
flag = Ture
boundary_intermediate_node = list()
c_1, c_2 = 0
T_1 = [end_grid_points[0][0], end_grid_points[0][1]] // Upper boundary pin coordinates in the region
T_2 = [end_grid_points[-1][0], end_grid_points[-1][1]] // Lower boundary pin coordinates in the region
while flag do
    if G_M[T_1[0]][T_1[1] + d] == 0 then // d is the exploration length
        T_1[1] += 1, c_1+ = 1
    else
        boundary_intermediate_node.append(T_1)
        flag = False
    end if
end while
```

flag = **Ture**
while *flag* **do**
    if G_M[T_2[0]][T_2[1] + d] == 0 **then**
        T_2[1] += 1, c_2+ = 1
    else
        boundary_intermediate_node.append(T_2)
        flag = **False**
    end if
end while
if c_1 < c_2 == 0 **then**
    boundary_intermediate_node.pop()
else
    boundary_intermediate_node.pop(0)
end if
intermediate_node_list =list()
for $i \leftarrow$ 0 to int(len(start_control_grid_points)/2)-1 **do**
    intermediate_node_list.append([start_control_gri_points[0][0]+(n_s*i), boundary_intermediate_node[0][1]-(d*i)]) //n_s is node spacing
    bidirectional_A_star_algorithm_with_corner_removal_step(start_control_grid_points[i],intermediate_node_list[-1])
end for
temp_index = len(start_control_grid_points) - 1
for $i \leftarrow$ 0 to int(len(start_control_grid_points)/2)-1 **do**
    intermediate_node_list.append([start_control_grid_points[int (len(start_control_grid_points)/2)][0]-(n_s*i),boundary_intermediate_node[0][1]-(d*temp_index)])
    bidirectional_A_star_algorithm_with_corner_removal_step(start_control_grid_points[i+int (len(start_control_grid_points)/2)], intermediate_node_list[-1])
    temp_index -= 1
end for
new_start_control_grid_points = boundary_intermediate_node.copy()
return new_start_control_grid_points

Appendix E: Pins assignment strategy algorithm

Algorithm 5 Pins_Assignment_Strategy

Input: starting nodes list start_control_grid_points, pin ports list in the region end_grid_points, assignment scheme *assignment*
Output: assignment scheme *assignment*

if len(*assignment*) == len(start_control_grid_points) **then**
    return *assignment*
end if
current_start = start_control_grid_points[len(*assignment*)]
min_cost = **INF**
best_end = **None**

```
for end ← end_grid_points[0] to end_grid_points[-1] do
    if end not in assignment.values() and end not in tabu_dict[(current_start[0], current_start[1])] then
        cost = cal_f(current_start, end)
        if cost < min_cost then
            min_cost = cost, best_end = end
        end if
    end if
end for
if best_end is None then
    return None
end if
assignment[(current_start[0],current_start[1])] = best_end
path, temp_grid_map_n = bidirectional_A_star_algorithm_with_corner_removal_step(current_start, best_end)
all_grid_path_temp.append(path)
temp_grid_map.append(temp_grid_map_n)
result = Pins_Assignment_Strategy(start_control_grid_points, end_grid_points, assignment)
if result is None then
    if temp_tabu is None then
        temp_tabu = current_start
    else if temp_tabu != current_start then
        tabu_dict[(temp_tabu[0], temp_tabu[1])].clear()
        temp_tabu = current_start
    end if
    tabu_dict[(temp_tabu[0], temp_tabu[1])].append(best_end)
    del assignment[(current_start[0], current_start[1])]
    all_grid_path_temp.pop()
    if len(temp_grid_map) > 1 then then
        temp_grid_map.pop()
    end if
    return Pins_Assignment_Strategy(start_control_grid_points, end_grid_points, assignment)
else
    return best_end
end if
```

Appendix F: Grid path to GDS path algorithm

Algorithm 6 Grid_Path_to_GDS_Path

Input: grid path Grid_Path

Output: gds path GDS_Path

```
i = len(Grid_Path)-2 // Starting from the next grid coordinate of the pin port.
temp_point = Grid_Path[len(Grid_Path)-1]
length = 0
GDS_Path = list()
```

direction = ‘’// The direction of segmented paths in GDS_Path.
while *i* >=0 **do**
  if *i* == len(Grid_Path)-2 **then** // Determine the initial direction of GDS_Path.
    if (Grid_Path[*i*][0] == temp_point [0]) **and** (Grid_Path[*i*][1] < temp_point[1]) **then**
      length+ = 1, *direction* = ’-x’
    else if (Grid_Path[*i*][0] == temp_point [0]) **and** (Grid_Path[*i*][1] > temp_point[1]) **then**
      length+ = 1, *direction* = ’+x’
    else if (Grid_Path[*i*][1] == temp_point [1]) **and** (Grid_Path[*i*][0] < temp_point[0]) **then**
      length+ = 1, *direction* = ’+y’
    else if (Grid_Path[*i*][1] == temp_point [1]) **and** (Grid_Path[*i*][0] > temp_point[0]) **then**
      length+ = 1, *direction* = ’-y’
    end if
  else
    if (Grid_Path[*i*][0] == temp_point [0]) **and** (Grid_Path[*i*][1] < temp_point[1]) **then**
      td = ’-x’
      if td != *direction* **then**
        length -= 1, GDS_Path.append([*length* $*$ Δ_{set}, *direction*])
        if *direction* == ’+y’ **then**
          GDS_Path.append([Δ_{set}, ’l’])
        else if *direction* == ’-y’ **then**
          GDS_Path.append([Δ_{set}, ’r’])
        end if
        length = 1, *direction* = td
      else
        length+ = 1
      end if
    else if (Grid_Path[*i*][0] == temp_point [0]) **and** (Grid_Path[*i*][1] > temp_point[1]) **then**
      td = ’+x’
      if td != *direction* **then**
        length -= 1, GDS_Path.append([*length* $*$ Δ_{set}, *direction*])
        if *direction* == ’+y’ **then**
          GDS_Path.append([Δ_{set}, ’r’])
        else if *direction* == ’-y’ **then**
          GDS_Path.append([Δ_{set}, ’l’])
        end if
        length = 1, *direction* = td
      else
        length+ = 1
      end if

else if (Grid_Path[i][1] == temp_point [1]) **and** (Grid_Path[i][0] < temp_point[0]) **then**
    td = '+y'
    if td != *direction* **then**
        length -= 1, GDS_Path.append([*length* $*$ Δ_{set}, *direction*])
        if *direction* == '+x' **then**
            GDS_Path.append([Δ_{set}, 'l'])
        else if *direction* == '-x' **then**
            GDS_Path.append([Δ_{set}, 'r'])
        end if
        length = 1, *direction* = td
    else
        length+ = 1
    end if
else if (Grid_Path[i][1] == temp_point [1]) **and** (Grid_Path[i][0] > temp_point[0]) **then**
    td = '-y'
    if td != *direction* **then**
        length -= 1, GDS_Path.append([*length* $*$ Δ_{set}, *direction*])
        if *direction* == '+x' **then**
            GDS_Path.append([Δ_{set}, 'r'])
        else if *direction* == '-x' **then**
            GDS_Path.append([Δ_{set}, 'l'])
        end if
        length = 1, *direction* = td
    else
        length+ = 1
    end if
end if
end if
if i == 0 **then**
    GDS_Path.append([*length* $*$ Δ_{set}, *direction*])
end if
end while
return GDS_Path

Here, we reversely convert the GDS path from the pin port. Among them, *direction* $\in \{'+x', '-x', '+y', '-y'\}$, and [*length* $*$ Δ_{set}, *direction*] indicates moving a distance of *length* $*$ Δ_{set} along the positive or negative directions of the X or Y axes on the QPU layout. [Δ_{set}, 'r'] and [Δ_{set}, 'l'] are corner processing, representing a 1/4 arc (with radius Δ_{set}) rotated to the right or left along the current path direction.

Appendix G: GDS path correction algorithm

Algorithm 7 GDS_Path_Correction

Input: gds path GDS_Path_{ij}, the starting coordinate of the control line for qubit i in the QPU layout $(x_i^{\text{Qubit\_ctrl}}, y_i^{\text{Qubit\_ctrl}})$, the coordinate of the pin port j in the QPU layout, $(x_i^{\text{Pin}}, y_i^{\text{Pin}})$

Output: corrected gds path GDS_Path_{ij}

temp_c = Find_Coordinate(GDS_Path_{ij}, GDS_Path_{ij}) // Calculate the end point coordinate of the path based on the GDS_Path_{ij} with the pin port j
distance_x = $x_i^{\text{Qubit\_ctrl}}$ - temp_c[0] // Deviation value on the horizontal coordinate.
for $m \leftarrow$ len(GDS_Path_{ij})-1 to 0 **do**
    if distance_x > 0 **then**
        if $\text{GDS\_Path}_{ij}[m][1]$ == '+x' **then**
            $\text{GDS\_Path}_{ij}[m][0]$ += abs(distance_x), **break**
        else if $\text{GDS\_Path}_{ij}[m][1]$ == '-x' **then**
            $\text{GDS\_Path}_{ij}[m][0]$ -= abs(distance_x)
            if $\text{GDS\_Path}_{ij}[m][0]$ <= 0 **then**
                $\text{GDS\_Path}_{ij}[m][0]$ += abs(distance_x), **continue**
            end if
            break
        end if
    else if distance_x < 0 **then**
        if $\text{GDS\_Path}_{ij}[m][1]$ == '-x' **then**
            $\text{GDS\_Path}_{ij}[m][0]$ += abs(distance_x), **break**
        else if $\text{GDS\_Path}_{ij}[m][1]$ == '+x' **then**
            $\text{GDS\_Path}_{ij}[m][0]$ -= abs(distance_x)
            if $\text{GDS\_Path}_{ij}[m][0]$ <= 0 **then**
                $\text{GDS\_Path}_{ij}[m][0]$ += abs(distance_x), **continue**
            end if
            break
        end if
    end if
end for
distance_y = $y_i^{\text{Qubit\_ctrl}}$ - temp_c[1] // Deviation value on the vertical coordinate.
for $m \leftarrow$ len(GDS_Path_{ij})-1 to 0 **do**
    if distance_y > 0 **then**
        if $\text{GDS\_Path}_{ij}[m][1]$ == '+y' **then**
            $\text{GDS\_Path}_{ij}[m][0]$ += abs(distance_y), **break**
        else if $\text{GDS\_Path}_{ij}[m][1]$ == '-y' **then**
            $\text{GDS\_Path}_{ij}[m][0]$ -= abs(distance_y)
            if $\text{GDS\_Path}_{ij}[m][0]$ <= 0 **then**
                $\text{GDS\_Path}_{ij}[m][0]$ += abs(distance_y), **continue**
            end if
            break
        end if
    else if distance_y < 0 **then**
        if $\text{GDS\_Path}_{ij}[m][1]$ == '-y' **then**
            $\text{GDS\_Path}_{ij}[m][0]$ += abs(distance_y), **break**
        else if $\text{GDS\_Path}_{ij}[m][1]$ == '+y' **then**
            $\text{GDS\_Path}_{ij}[m][0]$ -= abs(distance_y)
            if $\text{GDS\_Path}_{ij}[m][0]$ <= 0 **then**
                $\text{GDS\_Path}_{ij}[m][0]$ += abs(distance_y), **continue**

```
            end if
            break
        end if
    end if
end for
return GDS_Path_ij
```

Appendix H: Crossover coordinate correction algorithm

Algorithm 8 Crossover_Coordinate_Correction

Input: corrected gds path GDS_Path, the coordinate of the pin port in the QPU layout, $(x^{\mathrm{Pin}}, y^{\mathrm{Pin}})$, crossover coordinate converted based on the grid coordinate $(x^{\mathrm{Cs}}, y^{\mathrm{Cs}})$

Output: corrected crossover coordinate $(x^{\mathrm{Cs}}, y^{\mathrm{Cs}})$

```
seg_c = (x^Pin, y^Pin)
seg_c_list = list()
direct = ''
for i ← 0 to len(GDS_Path)-1 do
    if i == 0 then
        seg_c_list.append(seg_c)
        if GDS_Path[i][1] == '+x' then
            seg_c[0] += GDS_Path[i][0]
        else if GDS_Path[i][1] == '-x' then
            seg_c[0] -= GDS_Path[i][0]
        else if GDS_Path[i][1] == '+y' then
            seg_c[1] += GDS_Path[i][0]
        else if GDS_Path[i][1] == '-y' then
            seg_c[1] -= GDS_Path[i][0]
        end if
        direct = GDS_Path[i][1]
    else
        if GDS_Path[i][1] == '+x' then
            seg_c[0] += GDS_Path[i][0]
            direct = GDS_Path[i][1]
        else if GDS_Path[i][1] == '-x' then
            seg_c[0] -= GDS_Path[i][0]
            direct = GDS_Path[i][1]
        else if GDS_Path[i][1] == '+y' then
            seg_c[1] += GDS_Path[i][0]
            direct = GDS_Path[i][1]
        else if GDS_Path[i][1] == '-y' then
            seg_c[1] -= GDS_Path[i][0]
            direct = GDS_Path[i][1]
        else if GDS_Path[i][1] == 'l' then
            seg_c_list.append(seg_c)
            if direct == '+x' and GDS_Path[i + 1][1] == '+y' then
                seg_c[0] += GDS_Path[i][0]
```

```
            seg_c[1] += GDS_Path[i][0]
        else if direct == '-x' and GDS_Path[i + 1][1] == '-y' then
            seg_c[0] -= GDS_Path[i][0]
            seg_c[1] -= GDS_Path[i][0]
        else if direct == '+y' and GDS_Path[i + 1][1] == '-x' then
            seg_c[0] -= GDS_Path[i][0]
            seg_c[1] += GDS_Path[i][0]
        else if direct == '-y' and GDS_Path[i + 1][1] == '+x' then
            seg_c[0] += GDS_Path[i][0]
            seg_c[1] -= GDS_Path[i][0]
        end if
        seg_c_list.append(seg_c)
    else if GDS_Path[i][1] == 'r' then
        seg_c_list.append(seg_c)
        if direct == '+x' and GDS_Path[i + 1][1] == '-y' then
            seg_c[0] += GDS_Path[i][0]
            seg_c[1] -= GDS_Path[i][0]
        else if direct == '-x' and GDS_Path[i + 1][1] == '+y' then
            seg_c[0] -= GDS_Path[i][0]
            seg_c[1] += GDS_Path[i][0]
        else if direct == '+y' and GDS_Path[i + 1][1] == '+x' then
            seg_c[0] += GDS_Path[i][0]
            seg_c[1] += GDS_Path[i][0]
        else if direct == '-y' and GDS_Path[i + 1][1] == '-x' then
            seg_c[0] -= GDS_Path[i][0]
            seg_c[1] -= GDS_Path[i][0]
        end if
    end if
    seg_c_list.append(seg_c)
end for
new_seg_c_list = list()
for i ← 0 to len(seg_c_list)-1 do
    if i % 2== 0 then
        new_seg_c_list.append([seg_c_list[i], seg_c_list[i + 1]])
    end if
end for
seg_c_list.clear()
for i ← 0 to len(new_seg_c_list)-1 do
    if i % 2== 0 then
        seg_c_list.append(new_seg_c_list[i])
    end if
end for
if the direction of (x^Cs, y^Cs) is longitudinal then
    for i ← 0 to len(seg_c_list)-1 do
```

if seg_c_list[i][0][0] == seg_c_list[i][1][0] **then**
if seg_c_list[i][0][1] <= y^{Cs} <= seg_c_list[i][1][1] **or** seg_c_list[i][1][1] <= y^{Cs} <= seg_c_list[i][0][1] **then**
x^{Cs} = seg_c_list[i][0][0]
end if
end if
end for
else
for $i \leftarrow 0$ to len(seg_c_list)-1 **do**
if seg_c_list[i][0][1] == seg_c_list[i][1][1] **then**
if seg_c_list[i][0][0] <= x^{Cs} <= seg_c_list[i][1][0] **or** seg_c_list[i][1][0] <= x^{Cs} <= seg_c_list[i][0][0] **then**
y^{Cs} = seg_c_list[i][0][1]
end if
end if
end for
end if
return (x^{Cs}, y^{Cs})

Appendix I: Parameters of QPU layout for different numbers of qubits

Table 5 Parameters of QPU layout for different numbers of qubits

	Number of pins (including transmission line pins)	Processor size (μm^2)	The distance between qubits (μm)
16 Qubits Transmon + Bus Resonator Coupler	6×4	15,000×15,000	2200
24 Qubits Transmon + Bus Resonator Coupler	12×4	22,000×22,000	2200
16 Qubits Xmon + Direct Coupler	6×4	15,000×15,000	1800
24 Qubits Xmon + Direct Coupler	12×4	22,000×22,000	1800
32 Qubits Xmon + Direct Coupler	16×4	23,000×23,000	1800
64 Qubits Xmon + Direct Coupler	30×4	18,500×18,500	1800
128 Qubits Xmon + Direct Coupler	80×4	40,000×40,000 (C-Chip)	2100

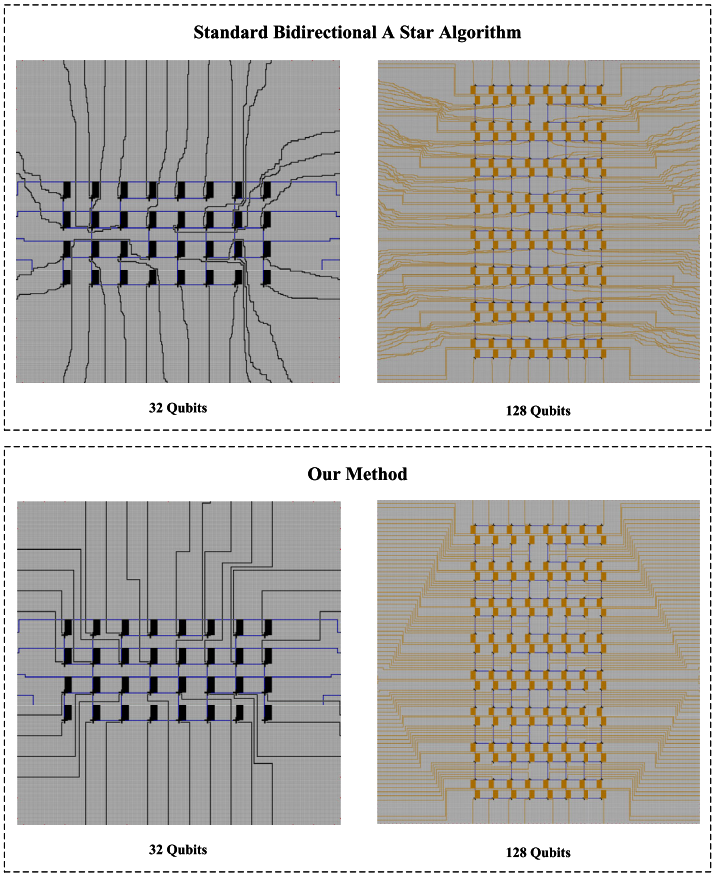


Figure 15 Comparison of routing results between standard bidirectional A star algorithm and our method

Appendix J: Comparison of routing results with the standard bidirectional a star algorithm

Figure 15 shows the routing results of the standard bidirectional A star algorithm and our method under the planar architecture with 32 Qubits and the flip-chip architecture with 128 Qubits. It can be seen that since the standard bidirectional A star algorithm only takes the shortest distance as the goal for the path search, its routing paths have more corners and crossover points. Our method through adds penalty terms to the heuristic evaluation function and performs corner removal step, has a more optimal number of corners and crossovers. On 32 Qubits, the number of crossovers in our method is reduced by 37.10% compared to the standard bidirectional A star algorithm. In the comparison of corners on 32 and 128 Qubits, the number of corners in our method is less than 8% of that in the standard bidirectional A star algorithm, further validating the effectiveness of our method.

Abbreviations

QPU, quantum processing unit; EDA, electronic design automation; QEDA, quantum electronic design automation; PCB, printed circuit board; GUI, graphical user interface; GDS, graphic data system; CPW, co-planar waveguides; TSV, through-silicon via.

Supplementary information

Supplementary information accompanies this paper at https://doi.org/10.1140/epjqt/s40507-025-00320-x.

Additional file 1. (PDF 325 kB)

Acknowledgements
The authors thank Laboratory for Advanced Computing and Intelligence Engineering for equipment supply.

Author contributions
Tian Yang : Conceptualization, Investigation, Methodology, Software, Visualization, Formal Analysis, Writing - Original Draft; Chen Liang: Software, Visualization, Formal Analysis, Data Curation; Weilong Wang : Data Curation, Validation, Supervision, Writing - Review & Editing; Bo Zhao: Methodology, Supervision; Lixin Wang: Investigation, Visualization; Qibing Xiong: Software, Visualization; Xuefei Feng: Software, Data Curation; Zheng Shan : Conceptualization, Resources, Validation, Supervision, Writing - Review & Editing.

Funding
This work was supported by the Major Science and Technology Projects in Henan Province(CN), Grant No.: 221100-210400, 221100-210600.

Data Availability
The data that support the findings of this study can be obtained from the corresponding author upon a reasonable request.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
All authors have approved the publication.

Competing interests
The authors declare no competing interests.

Author details
[1]Laboratory for Advanced Computing and Intelligence Engineering, Information Engineering University, Zhengzhou, 450001 Henan, China. [2]Songshan Laboratory, Zhengzhou, 450008 Henan, China. [3]School of Cyber Science and Engineering, Zhengzhou University, Zhengzhou, 450002 Henan, China.

Received: 26 July 2024 Accepted: 27 January 2025 Published online: 10 February 2025

References

1. Wendin G. Quantum information processing with superconducting circuits: a review. Rep Prog Phys. 2017;80(10):106001. https://doi.org/10.1088/1361-6633/aa7e1a.
2. Huang H-L, Wu D, Fan D, Zhu X. Superconducting quantum computing: a review. Sci China Inf Sci. 2020;63:1–32.
3. Gambetta J. The hardware and software for the era of quantum utility is here. IBM Research Blog. 2023.
4. Wu Y, Bao W-S, Cao S, Chen F, Chen M-C, Chen X, Chung T-H, Deng H, Du Y, Fan D, et al. Strong quantum computational advantage using a superconducting quantum processor. Phys Rev Lett. 2021;127:180501. https://doi.org/10.1103/PhysRevLett.127.180501.
5. Google Quantum AI. Suppressing quantum errors by scaling a surface code logical qubit. Nature. 2023;614(7949):676–81.
6. Kim Y, Eddins A, Anand S, Wei KX, Van Den Berg E, Rosenblatt S, Nayfeh H, Wu Y, Zaletel M, Temme K, et al. Evidence for the utility of quantum computing before fault tolerance. Nature. 2023;618(7965):500–5.
7. Ezratty O. Perspective on superconducting qubit quantum computing. Eur Phys J A. 2023;59(5):94.
8. Li G, Ding Y, Xie Y. Towards efficient superconducting quantum processor architecture design. In: Proceedings of the twenty-fifth international conference on architectural support for programming languages and operating systems. 2020. p. 1031–45.
9. Deb A, Dueck GW, Wille R. Exploring the potential benefits of alternative quantum computing architectures. IEEE Trans Comput-Aided Des Integr Circuits Syst. 2021;40(9):1825–35. https://doi.org/10.1109/TCAD.2020.3032072.
10. Lin W-H, Tan B, Niu MY, Kimko J, Cong J. Domain-specific quantum architecture optimization. IEEE J Emerg Sel Top Circuits Syst. 2022;12(3):624–37. https://doi.org/10.1109/JETCAS.2022.3202870.
11. Park SH, Bang J, An S, Hahn S. Design and performance analysis of hexagonal transmon qubit in a superconducting circuit. IEEE Trans Appl Supercond. 2021;31(5):1–5.
12. Wang H, Zhao Y-J, Wang R, Xu X-W, Liu Q, Wang J, Jin C. Frequency adjustable resonator as a tunable coupler for xmon qubits. J Phys Soc Jpn. 2022;91(10):104005. https://doi.org/10.7566/JPSJ.91.104005.
13. Wang C, Li X, Xu H, Li Z, Wang J, Yang Z, Mi Z, Liang X, Su T, Yang C, et al. Towards practical quantum computers: transmon qubit with a lifetime approaching 0.5 milliseconds. npj Quantum Inf. 2022;8(1):3.
14. Li F-Y, Jin L-J. Quantum chip design optimization and automation in superconducting coupler architecture. Quantum Sci Technol. 2023;8(4):045015.
15. Gely MF, Steele GA. Qucat: quantum circuit analyzer tool in python. New J Phys. 2020;22(1):013025.
16. Groszkowski P, Koch J. Scqubits: a python package for superconducting qubits. Quantum. 2021;5:583.
17. Aumann P, Menke T, Oliver WD, Lechner W. Circuitq: an open-source toolbox for superconducting circuits. New J Phys. 2022;24(9):093012.

18. Rahamim J, Behrle T, Peterer M, Patterson A, Spring P, Tsunoda T, Manenti R, Tancredi G, Leek P. Double-sided coaxial circuit qed with out-of-plane wiring. Appl Phys Lett. 2017;110(22):222602.
19. Yost D-RW, Schwartz ME, Mallek J, Rosenberg D, Stull C, Yoder JL, Calusine G, Cook M, Das R, Day AL, et al. Solid-state qubits integrated with superconducting through-silicon vias. npj Quantum Inf. 2020;6(1):59.
20. Tamate S, Tabuchi Y, Nakamura Y. Toward realization of scalable packaging and wiring for large-scale superconducting quantum computers. IEICE Trans Electron. 2022;105(6):290–5.
21. Chitta SP, Zhao T, Huang Z, Mondragon-Shem I, Koch J. Computer-aided quantization and numerical analysis of superconducting circuits. New J Phys. 2022;24(10):103020.
22. Shammah N, Saha Roy A, Almudever CG, Bourdeauducq S, Butko A, Cancelo G, Clark SM, Heinsoo J, Henriet L, Huang G, et al. Open hardware solutions in quantum technology. APL Quantum. 2024;1(1):011501.
23. Yan T, Wong MDF. Recent research development in pcb layout. In: 2010 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). 2010. p. 398–403. https://doi.org/10.1109/ICCAD.2010.5654190.
24. Chen T, Xiong S, He H, Yu B. Trouter: thermal-driven pcb routing via nonlocal crisscross attention networks. IEEE Trans Comput-Aided Des Integr Circuits Syst. 2023;42(10):3388–401. https://doi.org/10.1109/TCAD.2023.3243544.
25. Riley F, et al. The electronics assembly handbook. Berlin: Springer; 1988. https://doi.org/10.1007/978-3-662-13161-9_77.
26. Krantz P, Kjaergaard M, Yan F, Orlando TP, Gustavsson S, Oliver WD. A quantum engineer's guide to superconducting qubits. Appl Phys Rev. 2019;6(2):021318.
27. Gao YY, Rol MA, Touzard S, Wang C. Practical guide for building superconducting quantum devices. PRX Quantum. 2021;2(4):040202.
28. Cucurachi D, Das SR, Giri R, Gusenkova D, Guthrie A, Heinsoo J, Inel S, Janzsó D, Jenei M, Juliusson K, Kotilahti J, Landra A, Moretti R, Mylläri T, Ockeloen-Korppi C, Räbinä J, Savola N, Smirnov P, Takala E, Wubben L. KQCircuits. https://doi.org/10.5281/zenodo.4944796.
29. Minev ZK, McConkey TG, Drysdale J, Shah P, Wang D, Facchini M, Harper G, Blair J, Zhang H, Lanzillo N, Mukesh S, Shanks W, Warren C, Gambetta JM. Qiskit Metal: an Open-Source Framework for Quantum Device Design & Analysis. 2021. https://doi.org/10.5281/zenodo.4618153.
30. Wang S. Design and Fabrication of Two-dimensional Resonator-coupled Quantum Chips and Research on Two-dimensional Quantum Walks. (PHD Thesis). China: University of Science and Technology of China; 2022.
31. Pozar DM. Microwave engineering: theory and techniques. 2021.
32. Wei KX, Lauer I, Pritchett E, Shanks W, McKay DC, Javadi-Abhari A. Native two-qubit gates in fixed-coupling, fixed-frequency transmons beyond cross-resonance interaction. PRX Quantum. 2024;5(2):020338.
33. Chow J, Dial O, Gambetta J. Ibm quantum breaks the 100-qubit processor barrier. IBM Research Blog. 2021.
34. Hugh C. Ibm unveils 400 qubit-plus quantum processor and next-generation ibm quantum system two. IBM Research Blog. 2022.
35. Li B, Tan C, Lian Y, Shao Y, Xu H. Mobile robot global planning based on improved a* algorithm path planning research. In: Proceedings of the 2023 international conference on advances in artificial intelligence and applications. 2023. p. 305–11. https://doi.org/10.1145/3603273.3635241.
36. Saian PON, et al. Optimized a-star algorithm in hexagon-based environment using parallel bidirectional search. In: 2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE). 2016. p. 1–5. https://ieeexplore.ieee.org/document/7863246.
37. Carlsson B, Johansson S, Boman M. Generous and greedy strategies. In: Proceedings of complex systems. vol. 98; 1998.
38. Civicioglu P. Backtracking search optimization algorithm for numerical optimization problems. Appl Math Comput. 2013;219(15):8121–44.
39. Gabrielli LH. Gdspy: a Python module for creation and manipulation of GDSII stream files. 2009. https://github.com/heitzmann/gdspy.
40. Liu W. Synthesizing Three-body Interactions with Superconducting Qubits. (PHD Thesis). China: Zhe Jiang University; 2021.
41. Sun Y, Ding J, Xia X, Wang X, Xu J, Song S, Lan D, Zhao J, Yu Y. Fabrication of airbridges with gradient exposure. Appl Phys Lett. 2022;121(7):074001.
42. Kosen S, Li H-X, Rommel M, Shiri D, Warren C, Grönberg L, Salonen J, Abad T, Biznárová J, Caputo M, et al. Building blocks of a flip-chip integrated superconducting quantum processor. Quantum Sci Technol. 2022;7(3):035018.
43. Yuan B, Wang W, Liu F, He H, Shan Z. Comparison of lumped oscillator model and energy participation ratio methods in designing two-dimensional superconducting quantum chips. Entropy. 2022;24(6):792.
44. He H, Wang W, Liu F, Yuan B, Shan Z. Suppressing the dielectric loss in superconducting qubits through useful geometry design. Entropy. 2022;24(7):952.
45. Mukai H, Sakata K, Devitt SJ, Wang R, Zhou Y, Nakajima Y, Tsai J-S. Pseudo-2d superconducting quantum computing circuit for the surface code: proposal and preliminary tests. New J Phys. 2020;22(4):043013.
46. Krinner S, Lacroix N, Remm A, Di Paolo A, Genois E, Leroux C, Hellings C, Lazar S, Swiadek F, Herrmann J, et al. Realizing repeated quantum error correction in a distance-three surface code. Nature. 2022;605(7911):669–74.
47. Marques JF, Varbanov B, Moreira M, Ali H, Muthusubramanian N, Zachariadis C, Battistel F, Beekman M, Haider N, Vlothuizen W, et al. Logical-qubit operations in an error-detecting surface code. Nat Phys. 2022;18(1):80–6.

Publisher's Note