LONDON'S GLOBAL UNIVERSITY
Department of Computer Science

Dynamic Circuit Mapping Algorithms for Networked Quantum Computers

Natasha Valentina Santoso

23223788

Supervisors:

Abhishek Agarwal (NPL), Lachlan Lindoy (NPL)

A project report presented in partial fulfillment of the degree

MSc Quantum Technologies

Department of Physics and Astronomy

University College London

August 2024

Abstract

This project focuses on simulating qubit placement on physical devices and incorporating swap gates to address connectivity constraints in networked quantum computers. A key challenge in transpilation is the NP-hard problem of mapping logical qubits to physical qubits, particularly given the limited connectivity of physical qubits. This problem requires strategies for initial qubit placement and the insertion of swap gates when connectivity constraints are violated. The proposed approach utilizes physical qubit connectivity and gate priorities to generate an efficient initial mapping, followed by a dynamic lookahead window to insert swap gates more effectively. Although the Lookahead Swap method introduces slightly more swap gates than the default optimized Swap method, it significantly reduces circuit depth, making it very effective for noisy quantum hardware. In distributed quantum computing settings, layouts with higher connectivity perform better because of shorter distance between nodes, and organizing qubits into fewer groups further improves efficiency by simplifying information exchange and reducing circuit depth. This approach is particularly beneficial for networked quantum computers.

Keywords— quantum computing - quantum circuit mapping - networked quantum computers

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Abhishek Agarwal and Lachlan Lindoy, for their invaluable guidance, support, and encouragement throughout the development of this thesis. Their insight and expertise have been crucial to the successful completion of this project. I am also immensely grateful to Indonesia Endowment Fund for Education Agency (Lembaga Pengelola Dana Pendidikan), Ministry of Finance of Republic Indonesia for their full financial support throughout my master's study, which made this work possible.

I want to extend my heartfelt thanks to my family and friends for their unwavering support and understanding during this journey. Their patience and encouragement have been a constant source of strength.

Additionally, I would like to acknowledge IBM Qiskit team for providing the foundational framework that facilitated my research. As part of my contribution to this project, I developed and integrated transpilation algorithms, enhancing the overall functionality of the Qiskit software package, and I hope it will be beneficial to the quantum computing community.

Declaration

I, Natasha Valentina Santoso, declare that the thesis has been composed by myself and that the work has not be submitted for any other degree or professional qualification. I confirm that the work submitted is my own. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Natasha Valentina Santoso	23 August 2024
Signature	*Date*

Table of Contents

List of Tables

List of Algorithms

Acronyms

DAG Directed Acyclic Graph. 12, 16, 17, 21, 34, 35

DQC Distributed quantum computing. 1

ECR Echoed Cross-Resonance. 3

ISA Instruction Set Architecture. 6, 7

QBN Qubit Interaction Neighborhood. 12, 14

QPI Qubit Pair Interaction. v, 11, 12, 13, 14, 34

QPU Quantum Processing Unit. 1, 3, 9, 39

1 | Introduction and Background

Quantum computers are expected to surpass classical computers in processing complex data, potentially revolutionizing various fields such as quantum chemistry [1], machine learning [2], and quantum simulation [3]. Currently, quantum computing is in the Noisy Intermediate-Scale Quantum (NISQ) era, characterized by quantum processors with tens to hundreds of qubits [4]. To expand the number of qubits, both academic and industrial efforts are shifting towards utilizing quantum networks to link multiple smaller quantum chips. However, significant challenges exist in connecting multiple quantum computers and executing circuits across them simultaneously, along with algorithmic difficulties due to device limitations. This project will focus on simulating qubit placement on physical devices and incorporating swap gates to address networked quantum computers connectivity constraints.

1.1 Distributed Quantum Computing

Distributed quantum computing (DQC) involves breaking down a large quantum computation into smaller parts that are executed on multiple interconnected processors, rather than relying on a single large quantum computer with many qubits [5]. As shown in Figure 1.1, this approach uses smaller quantum devices, each with a limited number of qubits, working together to perform complex computations. In this system, qubits can be shared and entangled between different quantum processors across a network, allowing the execution of quantum algorithms that require more resources than any single quantum processor could handle on its own. A primary goal in DQC architecture is to minimize the number of operations, including connecting gates and interactions between different Quantum Processing Unit (QPU) [6].

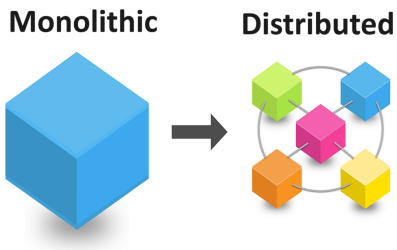


Figure 1.1: Monolithic versus distributed illustration. Adapted from [7]

1.2 Quantum Computing Architecture

Gate-based quantum computing is the foundation of most quantum computing architectures today. These systems rely on a limited set of native quantum gates, including single-qubit gates (such as Pauli-X, Y, Z, Hadamard, and phase gates) and two-qubit gates like the Controlled-NOT (CNOT) gate. While single-qubit gates can be applied directly to qubits, a CNOT gate requires the qubits to be adjacent on the coupling map. Despite the limited number of these gates, any quantum circuit can be constructed by carefully combining these operations [8]. This universality is a key aspect of quantum computing, enabling the design and execution of complex quantum algorithms within the constraints of available gate sets.

However, implementing these gates physically presents challenges, especially regarding qubit connectivity. In some quantum platforms, such as those using superconducting qubits [9], the connectivity between qubits is constrained by the architecture's coupling graph. This graph defines the possible interactions between qubits, meaning not all qubits can directly interact with each other through two-qubit gates. As a result, additional operations, such as swap gates, may be needed to bring qubits into positions where the desired two-qubit gate can be applied. This added complexity can affect the efficiency and fidelity of quantum circuits, especially as the scale of the computation grows [10].

In this context, it is important to distinguish between logical qubits and physical qubits. According to Itoko, Raymond, Imamichi, *et al.*, logical qubits refer to the abstract entities on which quantum algorithms and operations are defined (different from the concept of logical qubits in quantum error correction). These qubits exist independently of the physical hardware, representing the idealized qubits that are free from noise and hardware constraints [11]. On the other hand, physical qubits are the actual qubits within a quantum processor where quantum operations are executed. The performance of physical qubits is influenced by hardware limitations such as decoherence, gate fidelity, and connectivity constraints [12]. Understanding the relationship between logical and physical qubits is crucial for effectively mapping quantum algorithms onto real quantum hardware, ensuring that the physical resources are used optimally while minimizing the effect of hardware imperfections.

In the 127-qubit IBM quantum system, *ibm_kyoto* supports four single-qubit gates (ID,

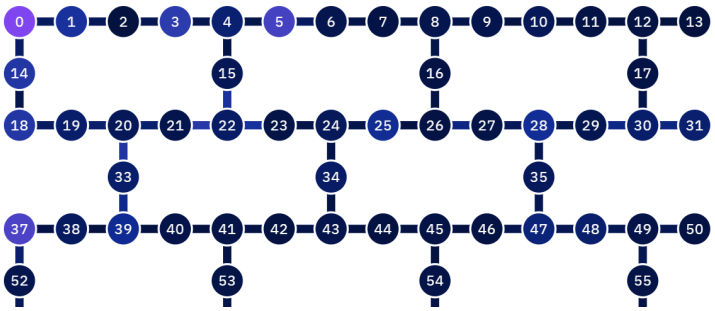


Figure 1.2: Partial coupling map of 127-qubit *ibm_kyoto*. Adapted from IBMQuantum [14]. The nodes represent physical qubits, and the edges between nodes represent possible two-qubit gate operations.

RZ, SX, X) and ECR gates for two-qubit operations. From Figure 1.2, a CNOT operation between q_1 and q_2 can be executed directly, but between q_0 and q_2, additional steps are needed. This highlights the requirement of a preliminary circuit preprocessing step, known as *quantum transpiling* [13], before running a quantum circuit.

1.3 Quantum Circuit Mapping

A quantum circuit compiler translates a logical quantum circuit into instructions that can be directly executed on a real QPU. The compiler converts high-level programming languages, like Qiskit in Python [15], into low-level instructions, such as OpenQASM [16]. Quantum algorithms, as defined in quantum circuits, are generally hardware-agnostic and may not account for specific hardware limitations [17]. To run these algorithms on a quantum processor, they must be adapted through a process called "quantum circuit mapping" (also known as quantum transpiling), which can increase gate count (g) or circuit depth (d), depending on the circuit and hardware constraints.
Figure 1.3 presents a 9-qubit device with two CNOT gates: (q_1, q_2) (blue) and (q_3, q_4) (green). The left side shows the original mapping and two different optimization approaches are explained:

1. **Depth First**: SWAP (q_2, q_9), (q_1, q_5) can be directly performed to connect CNOT (q_1, q_2), and also SWAP (q_4, q_8), (q_3, q_7) to connect CNOT (q_3, q_4). This method adds 4 SWAPs but only increases circuit depth by 1.
2. **Gates First**: SWAP (q_2, q_9) is done first, followed by swaps on (q_2, q_3) and (q_4, q_8). This method only adds 3 SWAPs but increases circuit depth by 2.

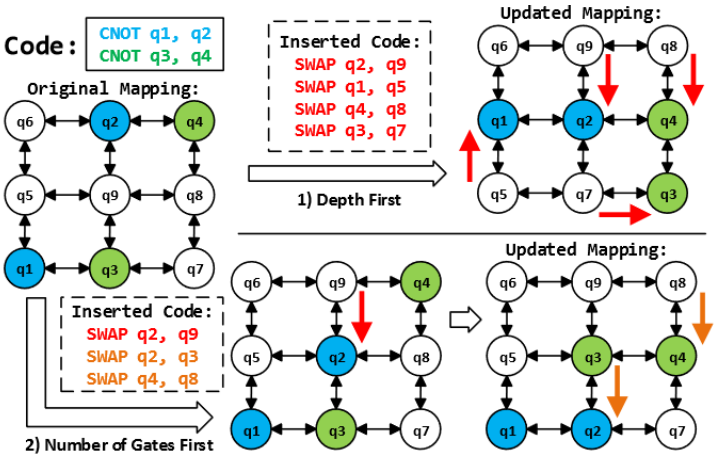


Figure 1.3: Example of a trade-off between depth-first and gate-first. Adapted from [18]

1.4 Qiskit Framework

Qiskit is a quantum software program that converts quantum circuits into a format that can be executed on the IBM Quantum Platform [14]. It utilizes a "transpilation" process to transform a given logical circuit to be compatible with a specific target device while also optimizing the circuit to enhance performance and achieve better results.

1.4.1 Pass Manager

The compilation process consists of six stages known as Pass Managers [19] (Figure 1.4), which include:

1. `init`: Runs initial passes to prepare the circuit for the backend, including unrolling custom instructions and converting multi-qubit gates into 1- and 2-qubit gates.
2. `layout`: Maps the logical qubits in the circuit to the physical qubits on the target quantum device.
3. `routing`: Inserts SWAP gates into the circuit to align with the physical connectivity of the backend, ensuring qubits that need to interact are adjacent.
4. `translation`: Converts the circuit's gates into the basis gates supported by the target backend, making the circuit executable on the specific hardware.
5. `optimization`: Performs multiple iterations of optimization to reduce circuit depth and gate count until a specified condition, such as a fixed depth, is met.
6. `scheduling`: Manages the timing and order of gate executions on the hardware to ensure optimal performance during execution.

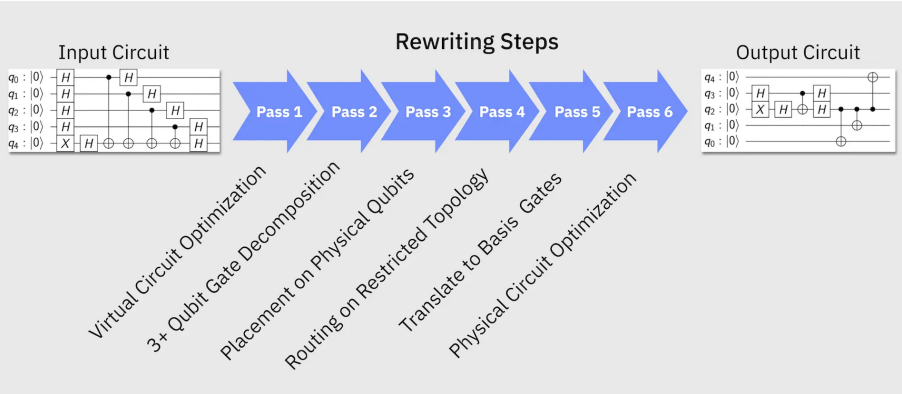


Figure 1.4: Pass manager stages. Adapted from [19]

This project will only focus on the first four stages until the translation stage.

1.4.2 Initialization stage

Most layout and routing algorithms only handle single- or two-qubit gates. Therefore, this stage unrolls any multi-qubit gates into sequences of single- or two-qubit gates, potentially increasing circuit size and depth [20]. Two specific gates require decomposition by the backend:

1. A SWAP gate is decomposed into three CNOT gates, shown in Figure 1.5. While swap gates are essential for mapping a circuit onto the limited physical connectivity of a quantum device, they are expensive operations to execute on noisy quantum systems.

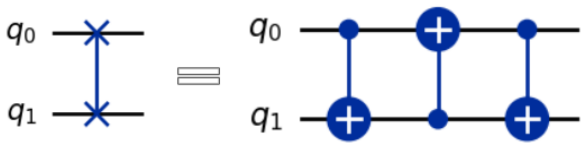


Figure 1.5: Decomposed swap gate

2. A Toffoli, also known as a controlled-controlled-not gate (CCX) gate, is a three-qubit operation. Since a Toffoli gate is not a native gate on most quantum hardware, it needs to be decomposed. The standard decomposition of a Toffoli gate involves up to 6 CNOT gates and several single-qubit gates, such as Hadamard (H) and T-gates, making it a resource-intensive process [21], as illustrated in Figure 1.6.

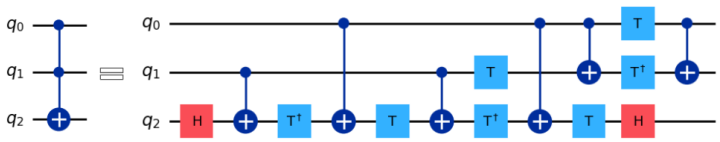


Figure 1.6: Decomposed Toffoli gate

1.4.3 Layout Stage

Quantum circuits consist of quantum gates that used qubits in the computations, which must be directly mapped onto the physical qubits of a quantum device. Choosing the right initial layout is key to minimizing swap operations and reducing noise-related losses on physical qubits. This mapping, stored as a `Layout` object, is a part of the constraints within a backend's Instruction Set Architecture (ISA). For example, Figure 1.7 shows how a quantum circuit is mapped onto a device's coupling graph.

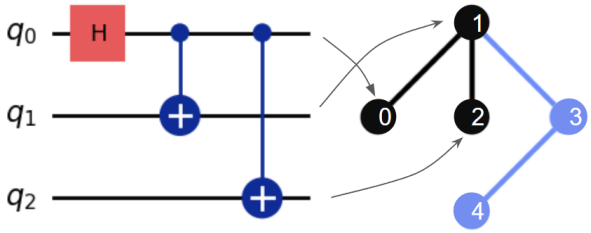


Figure 1.7: Mapping logical qubits to physical qubits

1.4.4 Routing Stage

The routing stage is a key part of the transpilation process that ensures two-qubit gates can be executed on a quantum device, considering its limited qubit connectivity. Since not all qubits on a device can interact directly, the routing stage adds SWAP gates to reposition qubits so that the required two-qubit operations are adjacent on the device's gate map [22], as shown in Figure 1.8. Since swap gates are costly and introduce noise, minimizing their number is crucial. Qiskit uses a stochastic algorithm called `SabreSwap` [18] to find an effective, though not always optimal, swap mapping. Because this method is stochastic, the resulting circuits can vary across runs, leading to different circuit depths and gate counts.

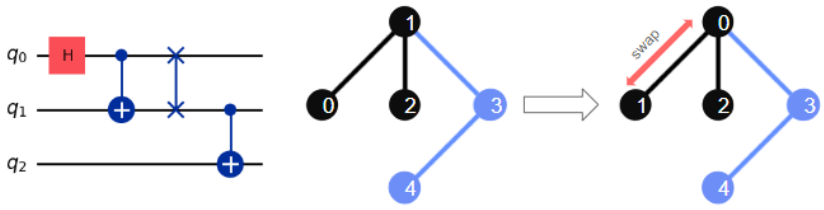


Figure 1.8: Swap qubit states between physical Q_0 and and Q_1

1.4.5 Translation Stage

When writing a quantum circuit, any arbitrary quantum gates can be used, including non-gate operations like qubit measurements and reset operations. Since most quantum devices only support a limited set of basic gates and non-gate operations, the quantum gates are translated into the native basis gates of a specific backend, which are part of the definition of a target's ISA [23]. For example, `GenericBackendV2` class supports only a limited set of basic gates ['cx', 'id', 'rz', 'sx', 'x', 'reset', 'delay', 'measure'], while the quantum circuit (Figure 1.9) has H, X, and controlled-P gates. After the transpilation process, a quantum circuit that initially had 5 logic gates is transformed into one with 12 gates.

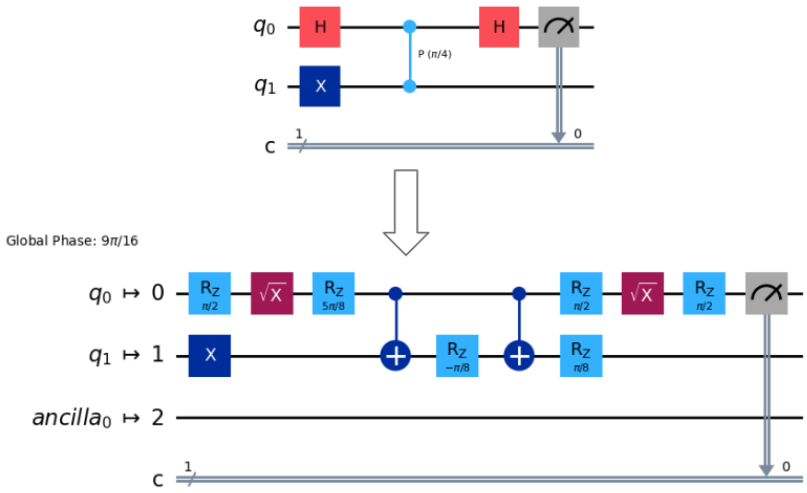


Figure 1.9: Translation basis gates

2 | Methodology

One of the main challenges in transpilation is mapping logical qubits to physical qubits, especially since physical qubits often have limited connectivity. The qubit-mapping problem is classified as NP-hard [24], meaning that while optimal solutions may be achievable for small quantum circuits, the problem becomes increasingly difficult for larger circuits. Because most quantum hardware only allows two-qubit gates to operate between physically adjacent qubits, two important strategies are required: determining the initial placement of qubits and inserting swap gates when a gate execution violates these connectivity constraints [25].

For the overview of the algorithms, the initial step in addressing the qubit-mapping problem involves utilizing the physical qubit connectivity and gate priorities to generate an initial qubit mapping that is more efficient than a simple one-to-one layout. The subsequent step introduces a dynamic lookahead window to insert swap gates more efficiently taking into account gates that give positive effect to the swap candidate. This proposed algorithms successfully address the coupling limitations of quantum devices and reduce the overhead of inserting swap gates, leading to better performance on networked quantum hardware.

Table 2.1 provides the definitions for the notations used in the analysis:

Table 2.1: Definition of Notations

Notation	**Definition**
n	number of logical qubits
$q_{1,2,...,n}$	logical qubits q_L in quantum circuit
N	number of physical qubits
$Q_{1,2,...,N}$	physical qubits Q_P on quantum device
$C_{1,2,...,N}$	classical register for quantum device
g_i	number of gates in the circuit
d	depth of the circuit
$G(V, E)$	the coupling graph of the backend
$D[\,][\,]$	the distance matrix of the physical qubits between Q_i, Q_j
$\pi()$	a mapping from $q_{1,2,...,n}$ to $Q_{1,2,...,N}$
$\pi^{-}1()$	a mapping from $Q_{1,2,...,N}$ to $q_{1,2,...,n}$
F	Front layer of quantum circuit

2.1 Layout

2.1.1 Distributed Coupling Graph

Simulating a distributed quantum computer involves creating a coupling map that represents the connectivity of qubits across different "groups", with each group acting as a separate node or quantum processor within the quantum network. These groups consist of multiple Quantum Processing Unit (QPU), with connections both within each group and between different groups. The coupling graph defines the topology of the QPU within each group, and this is managed using `CouplingMap` [26] class in Qiskit. In this coupling graph, the last qubit of one group is connected to the first qubit of the next group, forming a larger distributed coupling map. By modifying the internal topology of a node (quantum computer) and the distribution of qubits across different groups, various configurations of the quantum network can be explored, as illustrated in Appendix B.
The following functions are used to define the topology within each group:

- `from_line`: creates a coupling map of n qubits connected in a line.
- `from_grid`: creates a coupling map of qubits arranged in a grid with a specified number of rows and columns.
- `from_ring`: creates a coupling map of n qubits connected in a ring.
- `from_full`: creates a fully connected coupling map on n qubits.
- `from_t_horizontal`: creates a coupling map of five qubits arranged in a T-shape, connected at the longer end.
- `from_t_vertical`: creates a coupling map of five qubits arranged in a T-shape, connected at the shorter end.

2.1.2 Interaction Mapping

In this section, an algorithm called `InteractionLayoutMapping` will be discussed to determine the most appropriate mapping from logical qubits to physical qubits. During the initialization step, the connectivity of the physical qubits on the backend is determined, and the priority of two-qubit gates is assigned based on their order of appearance in the quantum circuit, with gates appearing earlier (on the left side) being given higher priority [27]. Additionally, a metric is calculated to measure qubit interactions that quantify how qubits interact with each other within a circuit, particularly on how qubits are connected

through multi-qubit gates, such as CNOT gates [12]. This calculation will help to define the order in which quantum gates are prioritized.

Definition 1. Physical connectivity is the number of neighbours of physical qubits (Q_P).

A physical quantum device is represented by a coupling graph, which is a directed graph (V, E). In this graph, $Q = \{Q_0, Q_1, ..., Q_{N-1}\}$ represents the set of physical qubits as vertices (V), and the edges (E) represent the direct connections between two physical qubits, Q_i and Q_j, where a two-qubit gate can be applied. When a physical qubit has higher connectivity, the logical qubit mapped to it has a greater chance of connecting to other logical qubits without needing additional swap gates. Conversely, if a physical qubit has low connectivity, more qubit movements will be needed to satisfy the coupling constraints before a logical gate can be operated. These additional movements increase the need for swap gate insertions, leading to a larger circuit size and depth, which may negatively impact gate fidelity and runtime. For example, Qiskit `FakeLondonV2` [28] (Figure 2.1) is a simulated 5-qubit backend with a coupling graph that illustrates the physical connectivity on each qubit.

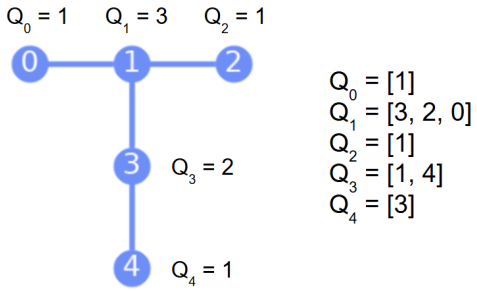


Figure 2.1: `FakeLondonV2` coupling map and its physical connectivity list containing the number of neighbours of each physical qubit Q_P.

Definition 2. Logical priority calculates the number of two-qubit gates acting on each logical qubit in a quantum circuit.

If two logical qubits have the same priority value, the one with the earlier index is given precedence. The logical neighbours list identifies other logical qubits that interact with a specific logical qubit q_i in the circuit. Figure 2.2 lists the number two-qubit gates operating on the wire of the quantum circuit.

Definition 3. Gate weight reflects the importance of each gate g_i in a quantum circuit,

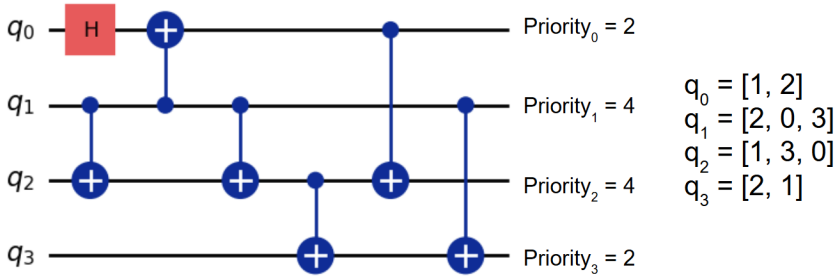


Figure 2.2: Logical priority of a quantum circuit and its logical neighbours list containing the indices of qubits that interact with the logical qubit q_i.

based on when it occurs in the sequence of operations.

In a quantum circuit, consider a set of logical qubits $q_L = \{q_0, q_1, ..., q_{n-1}\}$ and an ordered sequence of gates $g = \{g_0, g_1, ..., g_{m-1}\}$. It is important to note that the ordering of gates introduces some arbitrariness, as certain gates can be implemented in parallel. For example, the last two gates in Figure 2.3, CNOT (q_0, q_2) and CNOT (q_1, q_3), could be executed simulatenously, making the weights of g_6 and g_7 interchangeable.
Each quantum gate g_i is assigned a weight w_i, which is calculated as:

$$w_i = m - i \tag{2.1}$$

where m is the total number of gates in the quantum circuit, and i is the gate's position in the sequence. A higher weight suggests that the gate occurs earlier in the sequence, potentially having a greater influence on the subsequent operations and the final state of the quantum system. Figure 2.3 shows the gate weights in descending order, with earlier gates assigned higher values.

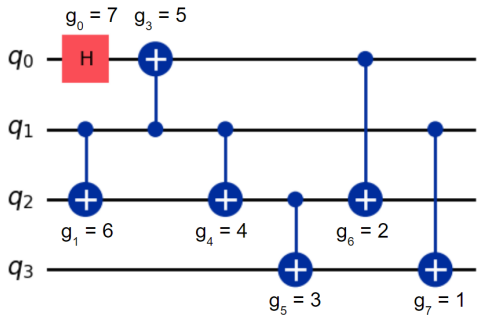


Figure 2.3: Gate weight of a quantum circuit, with front gates assigned higher weights

Definition 4. Qubit Pair Interaction (QPI) refers to the measure of interaction logi-

cal qubit pair (q_i, q_j) when they are involved in a multi-qubit quantum gate, such as controlled-NOT (CNOT) gate or any other two-qubit gate.

QPI is determined by summing all gate weights that involve the qubit pair and is represented as a 2D matrix, as in Table 2.2:

$$\text{QPI}(q_i, q_j) = \sum_{k \in \mathcal{G}(q_i, q_j)} w_k \tag{2.2}$$

where $\mathcal{G}(q_i, q_j)$ represents the set of indices k corresponding to gates that involve both qubits q_i and q_j, and w_k is the weight of gate g_k.

(q_i, q_j)	(0, 1)	(0, 2)	(1, 2)	(1, 3)	(2, 3)
QPI	5	2	10	1	3

Table 2.2: QPI value of gates on logical qubits (q_i, q_j) from quantum circuit on Figure 2.3.

Definition 5. Qubit Interaction Neighborhood (QBN) refers to the sum of weights of the qubit pairs interactions (QPI values) that involve the logical qubit being considered and its neighbouring physical qubits.

A higher QBN value indicates that the qubit is more connected or plays a more significant role within the quantum circuit. The QBN value is defined as:

$$\text{QBN}(q_i) = \sum_j \text{QPI}(q_i, Q_{P,j}) \tag{2.3}$$

where q_i is the logical qubit under consideration, $Q_{P,j}$ denotes each physical qubit that is adjacent to or interacts with q_i, and $\text{QPI}(q_i, Q_{P,j})$ represents the interaction strength (weight) between q_i and the j-th adjacent physical qubit.

2.1.3 Interaction Mapping Algorithm

Algorithm 1 `InteractionLayoutMapping` outlines the steps for creating an initial qubit mapping, where the inputs are the device's coupling map and the circuit's Directed Acyclic Graph (DAG), and the output is a layout that maps logical qubits (q_L) to physical qubits (Q_P). The algorithm starts by calculating physical connectivity from the coupling map and determining the logical qubits' priorities, which are sorted in descending order.

The logical qubit with the highest priority is then mapped to the physical qubit with the most connections. For subsequent placements, the algorithm checks if the assigned physical qubits have adjacent neighbours. If they do, it evaluates Qubit Pair Interaction (QPI) between the logical neighbours and their corresponding qubits, selecting the highest QPI candidate for mapping. If no adjacent neighbours meet this criterion, the algorithm assigns the next highest physically connected qubits to the logical qubits. This process generates all possible mappings for the quantum circuit, which are then ranked based on their total QPI index, and the final output is one initial mapping with the highest QPI rank. If multiple mappings have the same top QPI rank, the final output may be different with each run.

Algorithm 1 Interaction Layout Mapping

Input Coupling Graph G(V, E), Circuit DAG

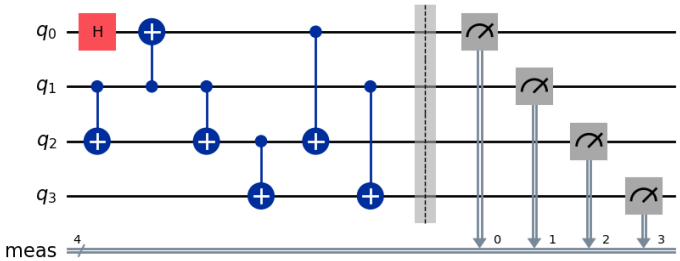
physical connectivity $\leftarrow E$ on Q_P
logical priority $\leftarrow g$ on q_L
if coupling graph is mostly fully connected **then**
    return one-to-one index mapping
end if
maps = [ ]
while logical priority is not empty **do**
    for maps **do**
        $q_i \leftarrow$ max (logical priority)
        temp physical connectivity $\leftarrow$ neighbours of assigned Q_P
        if all q_i neighbours already assigned **then**
            $Q_X \leftarrow$ max(temp physical connectivity)
            maps $\leftarrow (q_i, Q_X)$
        else
            Q_X neighbours[ ] $\leftarrow$ adjacent assigned Q_P
            Get all assigned q_L from maps
            for available Q_X neighbours **do**
                $q_j \leftarrow \pi^{-1}$(available Q_X)
                QBN candidate $\leftarrow$ QPI$[q_i][q_j]$
            end for
            $Q_X \leftarrow$ max (QBN value)
            maps $\leftarrow (q_i, Q_X)$
        end if
    end for
end while

Example 1. The initial quantum circuit shown in Figure 2.4a is processed through Algorithm 1 `InteractionLayoutMapping` to obtain a more efficient solution for qubit

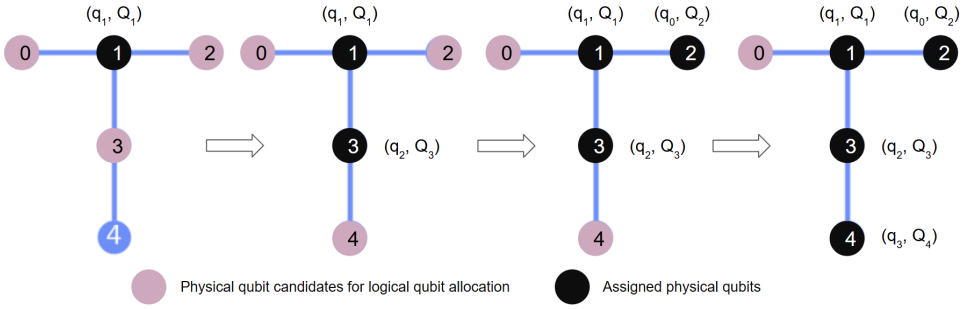
placement. The backend device used is a simulated five-qubit system, `FakeLondonV2`. First, logical qubit q_1 is directly assigned to Q_1, which has the most connections with three neighbours. The current mapping is [[(1, 1)]], where the first value in the tuple represents the logical qubit (q_L) and the second value represents the physical qubit (Q_P). Next, logical qubit q_2 is to be assigned to the available physical candidates Q_3, Q_2, and Q_0. The QPI(q_1, q_2) value is 10 for all these candidates, so they are all included in the mapping. The current mapping is $[[(1,1),\ (2,\ 3)\ ],[(1,1),(2,2)],[(1,1),(2,0)]]$, and the sum of QBN is recorded. Figure 2.4b illustrates the path when physical qubit Q_3 is chosen for logical qubit q_2. The next logical qubit, q_0, has possible physical candidates Q_0, Q_2, and Q_4. Using the mapped logical qubits and their assigned physical qubits, the QPI values are calculated, and the QBN values are summed to determine the maximum value. For the neighbour of Q_1, the QPI between q_0 and q_1 is 5, while for the neighbor of Q_3, the QPI between q_0, q_2 is 2. The maximum value occurs for both Q_0 and Q_2. Therefore, the updated mapping for path $[(1,1),(2,3)]$ is $[[(1,1),(2,3),\ (0,\ 2)\ ],[(1,1),(2,3),(0,0)]]$. With the last mapping of (q_0, Q_2), the last logical qubit is for q_3 has available candidates Q_0 and Q_4. If q_3 is assigned to Q_0, the QPI value is 1, while for Q_4, the QPI value is 3. The final assignment is (q_3, Q_4) and QBN rank is updated accordingly. The algorithm not only generates all possible paths for mapping but also identifies the path with the highest QBN rank. The chosen layout is $[(1,1),(2,3),(0,0),\ (3,\ 4)\ ]$, as shown in Figure 2.4c, with a total rank of 18.0. The initial quantum circuit is redrawn as in Figure 2.4d.

2.1.4 Analysis Pass

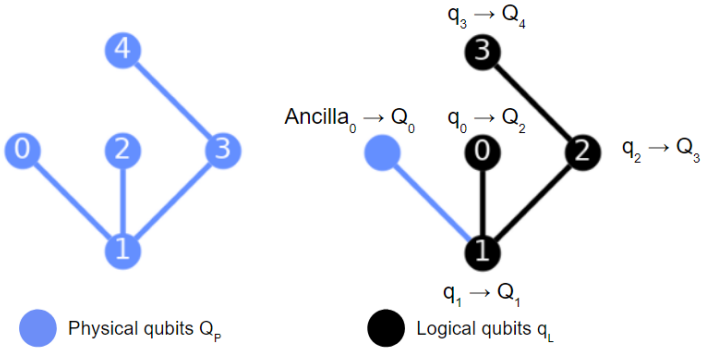
Qiskit framework provides specific `AnalysisPass` [29] to change the property set of a quantum circuit. The custom analysis pass requires layout mapping as input, builds the mapping to Qiskit `Layout` [30] class, and implements the abstract run function. The class layout constructs a bijective dictionary, mapping virtual qubits q_L to physical qubits Q_P. An AnalysisPass might be used to evaluate the effectiveness of a given qubit layout, guiding the optimization process and ensuring that the final circuit is well-suited for the physical quantum hardware it will run on.



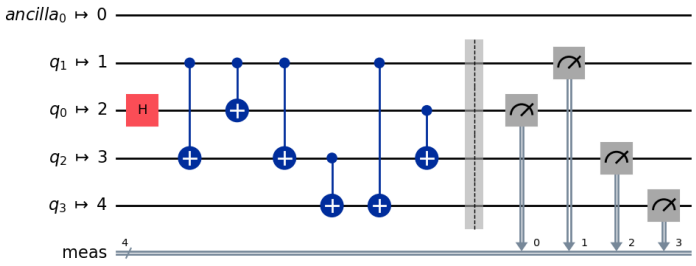
(a) Initial quantum circuit



(b) Steps to assign logical qubit to backend coupling map



(c) $\pi(q_L, Q_P)$ mapping in the coupling map, where the blue nodes on the left are physical qubits Q_P, and the black nodes on the right are logical qubits q_L assigned to the physical qubits.



(d) Redraw quantum circuit in embed coupling map

Figure 2.4: Interaction layout mapping (q_L, Q_P)

2.2 Routing

2.2.1 Lookahead Swap

When addressing the qubit mapping problem, all two-qubit gates must meet the hardware connectivity constraints. Single-qubit gates, which act on individual physical qubits, are not relevant and can be ignored for the mapping problem. In most cases, it is not feasible to find a mapping that satisfies all connectivity constraints for the entire quantum circuit. Therefore, whenever a gate does not meet these constraints, the mapping between logical and physical qubits must be updated using swap gates. However, adding swap gates can greatly affect the order of subsequent logical gates. The main metrics for evaluating these adjustments are the number of additional swap gates needed and the resulting circuit depth.

Definition 6. DAGCircuit [31] is a Directed Acyclic Graph (DAG) representation of a quantum circuit. This intermediate representation allows the manipulation and optimization of quantum circuits.

In the DAG, each node corresponds to a quantum gate or a measurement applied to qubits, while the edges indicate the flow of quantum information between operations, connecting the output of one operation to the input of another. The DAG structure can be analysed by counting the number of qubits and assessing the circuit depth, which may help simplify the circuit by merging gates, eliminating redundant operations, and applying other techniques to reduce the circuit's complexity [18]. Figure 2.5 illustrates the flow of a quantum circuit, where green nodes represent logical qubit inputs, blue nodes represent logical gates g, and red nodes represent logical qubit outputs. The edges in the graph depict the quantum information flow from input to output. Additionally, each level in the graph corresponds to the quantum circuit depth, so when multiple nodes appear at the same level, it indicates that the gate operations are applied to disjoint physical qubits.

Definition 7. The Gate Front Layer refers to a series of quantum gates that have no pending predecessors on their associated qubits.

A layer in a quantum circuit consists of gates that operate on non-overlapping qubits, and

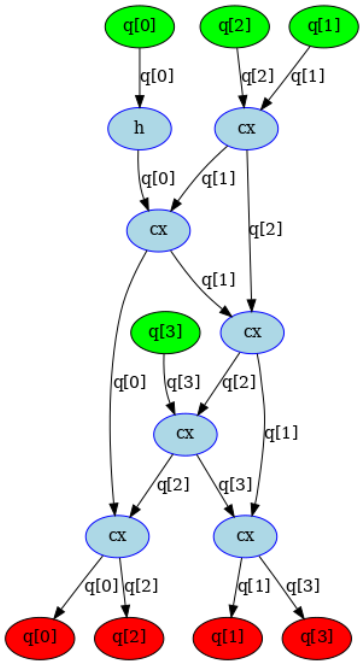


Figure 2.5: Directed Acyclic Graph (DAG) of the previous quantum circuit (Figure 2.4a). DAG consists of three nodes: DAG input nodes (green), DAG operation nodes (blue), and DAG output nodes (red). The arrows indicate the quantum information flow.

the total number of layers corresponds to the circuit depth, denoted as d. From Qiskit documentation, these layers are generated using a greedy algorithm. The resulting layer includes new DAG operation nodes, DAG input nodes, and DAG output nodes. In the DAG illustrated in Figure 2.5, the front layer consists of the CX and H gates.

Definition 8. Nearest neighbour distance refers to the shortest distance between a given qubit and its closest neighbouring qubit in a quantum device.

In most quantum hardware, two-qubit gates (like CNOT or CZ gates) can only be directly implemented between qubits that are physically adjacent. If the two qubits involved in the gate are not neighbours, they need to be moved closer together through a series of SWAP operations, which exchange the positions of qubits on the device.
In the setup Figure 2.6, the nearest neighbour distance between Q_0 and Q_1 is 1, the nearest neighbour distance between Q_1 and Q_2 is 1, and the nearest neighbour distance between Q_0 and Q_2 is 2, as they are not direct neighbours.

Figure 2.6: Example of nearest neighbour distance setup, where distance $Q_0 - Q_1$ is 1, and distance $Q_0 - Q_2$ is 2.

Definition 9. The dependency list for q_i contains the quantum gates g_i that are located at logical qubits q_L. This list only considers two-qubit gates.

The dependency list starts with the front layer of quantum gates that have no preceding gates, which is the leftmost gate in each dependency list_i. A two-qubit gate is included in two related dependency lists and becomes an active gate when it is the front gate for both associated logical qubits at the same time. As shown in Figure 2.7, gate g_1, as the front gate for both dlist_1 and dlist_3, makes it an active gate. Single-qubit gates are not included in the dependency list because they can be directly executed on a physical qubit without requiring special consideration.

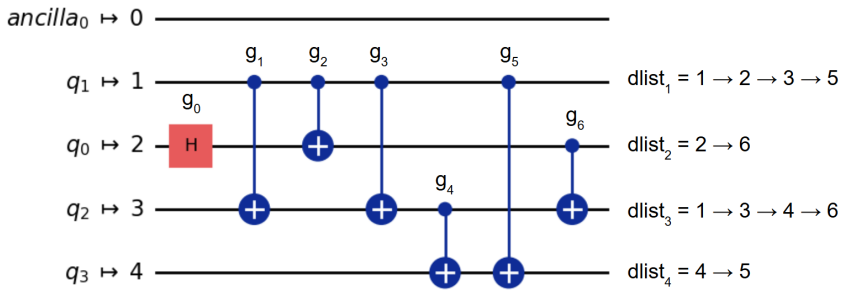


Figure 2.7: The dependency list of a quantum circuit, where dlist_1 contains $g_1, g_2, g_3,$ and g_5, which are two-qubit gates that lie on q_1.

Definition 10. The effect of a swap gate on a specific two-qubit gate is defined as the change in the nearest neighbour distance of that gate before and after the swap is applied.

When a swap gate is applied, it alters the nearest neighbour distance for subsequent two-qubit gates that share a logical qubit with the swap. Each swap impacts only one distance, leading to one of three outcomes: the distance can decrease by 1 (positive effect), increase by 1 (negative effect), or remain unchanged (no effect) [32]. In figure 2.8, each rectangle represents a two-qubit gate, with " $+$ " indicates a non-negative effect (0 or $+1$), while " $-$ " signifies a negative effect (-1).

The effect of a swap gate applied to (Q_i, Q_j) changes the nearest neighbour distance of g_i, as determined by the logical to physical mapping before the swap (π_1) and after the swap (π_2).

$$\text{effect}(\text{SWAP}, g_i) = \text{dist}(g_i, \pi_1) - \text{dist}(g_i, \pi_2) \tag{2.4}$$

Definition 11. The look-ahead technique is a heuristic cost function used to evaluate

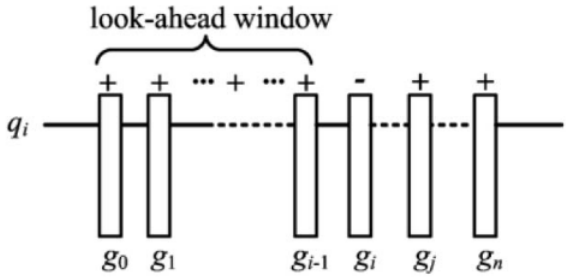


Figure 2.8: Diagram for the dynamic look-ahead technique. Adapted from [32]

how effectively a swap gate ensures that the circuit meets connectivity constraints.

The method used in Qiskit relies on a fixed window size that is determined before the mapping process begins [33]. This fixed window approach has limitations because it considers a predetermined number of gates across all qubits, regardless of their relevance. Additionally, with a fixed-size window, the heuristic cost function might select a swap gate that helps some gates scheduled later but negatively impacts those that are executed earlier. In extreme cases, the chosen swap could be harmful to current active gates but may be beneficial for future ones, potentially slowing down or even blocking the heuristic algorithm to progress.

To overcome these issues, a dynamic look-ahead technique is introduced as the heuristic cost function. This approach evaluates the effect of a swap gate by focusing only on the two-qubit gates directly affected by the swap. It selectively looks ahead at the two logical qubits (q_i and q_j) that experience a positive effect, stopping when a negative effect is encountered. As illustrated in Figure 2.8, lookahead window only considers from g_0 to g_{i-1}, where the values are non-negative. The look-ahead window sums all the non-negative (1 or 0) effects on the logical qubit q_L:

$$\text{Lookahead}(\text{SWAP}, q_i) = \sum_{g \in \text{window}} \text{effect}(g) \tag{2.5}$$

The value of Lookahead(SWAP) includes both logical qubits q_i and q_j where the swap gate is inserted,

$$\text{Lookahead}(\text{SWAP}) = \text{Lookahead}(\text{SWAP}, q_i) + \text{Lookahead}(\text{SWAP}, q_j) \tag{2.6}$$

All these definitions are incorporated to Algorithm 2 `Lookahead Swap Routing`.

Algorithm 2 Lookahead Swap Routing

Input Coupling Graph G(V, E), Circuit DAG
Output Circuit DAG

procedure CHECK GATE CONNECTIVITY(g_i)
    $Q_i \leftarrow \pi(q_i)$
    $Q_j \leftarrow \pi(q_j)$
    $D[Q_i][Q_j]$
    if $D[Q_i][Q_j] == 1$ **then**
        dlist[q_i] pop front
        dlist[q_j] pop front
        new_dag apply operation back(g_i)
    else
        act_list.append(g_i)
    end if
end procedure

dlist[ ] $\leftarrow$ g_i on q_n iff two-qubit gates
new_dag = dag.copy_empty()
act_list = [ ]
for dag layers **do** ▷ traverse circuit depth per level
    for g_i in dag operation nodes **do**
        if g_i is a two-qubit gate of (q_i, q_j) **then**
            check gate connectivity(g_i)
        else
            if g_i is 'measure' operation **then**
                $C_i \leftarrow q_i$ the corresponding updated Q_P for register
            end if
            new_dag apply operation back(g_i)
        end if
    end for
    while act_list is not empty **do**
        check gate connectivity(g_i)
        candidate list = generate possible physical swaps
        for swap(Q_x, Q_y) in candidate list **do**
            Lookahead (swap(Q_x, Q_y)) = $\sum_g$ effect($Q_x(g)$) + $\sum_g$ effect($Q_y(g)$)
        end for
        highest swap (Q_x, Q_y) $\leftarrow$ max(lookahead)
        new_dag apply operation back(SwapGate(Q_x, Q_y))
    end while
end for

2.2.2 Lookahead Swap Routing Algorithm

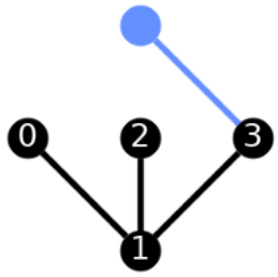
Figure 2.9b demonstrates the transpilation process using Qiskit `TrivialLayout` [34] and `BasicSwap` [35] routing methods. The `TrivialLayout` assigns virtual qubits to physical qubits by mapping n logical qubits to device qubits $0, 1, ..., , N-1$ in ascending order. The `BasicSwap` then performs minimal adjustments by inserting one or more swap gates before a two-qubit gate until the quantum information is located at adjacent physical qubits.

In contrast, Figure 2.9d illustrates the process using the interaction layout from Algorithm 1. Initially, the algorithm converts the circuit to DAG and creates a dependency list for all two-qubit gates, as shown in Figure 2.7. It then iterates over the DAG's level layers, checking the gate operation's coupling map distance $D[q_i][q_j]$. If the distance is 1, indicating that the logical qubits are adjacent, the gate is directly applied, and the gate nodes are added to the DAG circuit. However, if the distance is greater than 1, the node is inserted into the active list.

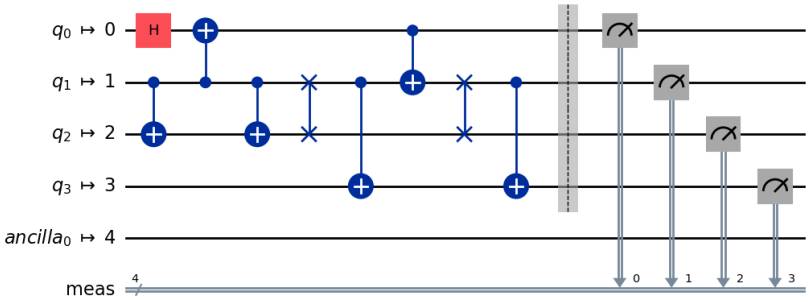
The first four layers have a distance of 1, allowing gates g_0 through g_4 to be executed directly. The next layer includes $g_5(q_1, q_3)$ and $g_6(q_0, q_2)$, for which the algorithm generates a list of physical swap candidates for each gate. For gate g_5, the swap candidates are $[(Q_2, Q_1), (Q_3, Q_1), (Q_3, Q_4)]$, while gate g_6 adds the swap candidate (Q_1, Q_0). The algorithm then calculates the lookahead heuristic value for subsequent gates on the dependency list. If a negative effect is encountered during the iteration, the process terminates, and the lookahead window only considers non-negative values. After the calculation, the swap between Q_3 and Q_1 has the highest lookahead value. The next step is to recalculate the distance between the new swap positions for the gates. If the gates can now be executed directly, the gate operation is added to the DAG. The final step is to determine the order of the measurement operations to the classical registers, taking into account the updated logical-to-physical qubit mapping after the swaps.

2.2.3 Transformation Pass

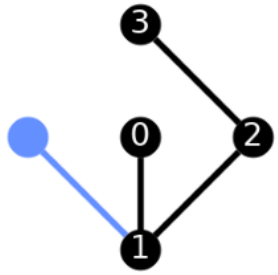
The Pass Manager responsible for routing requires the use of a `TransformationPass` [36] to modify the DAG structure representing the quantum circuit. This pass manager takes the device's coupling map and the circuit's DAG as inputs and modifies the DAG by inserting the necessary swap gates to ensure that the circuit adheres to the hardware



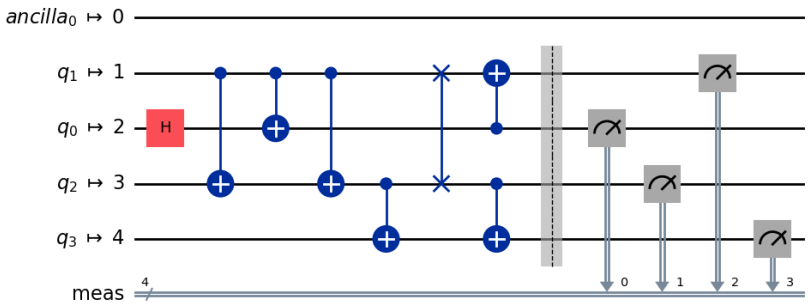
(a) One-to-one mapping



(b) Transpiled quantum circuit with basic swap routing



(c) Interaction layout mapping



(d) Transpiled quantum circuit with lookahead swap routing

Figure 2.9: Comparison between layout and routing implementation

coupling constraints. The output of this process is a transpiled quantum circuit that is compatible with the physical device and ready for execution.

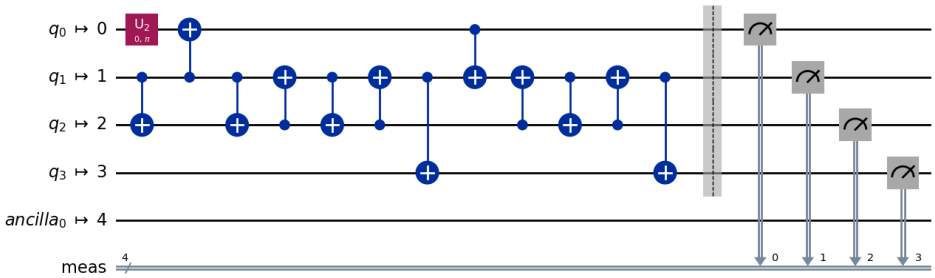
2.3 Integration

2.3.1 Staged Pass Manager

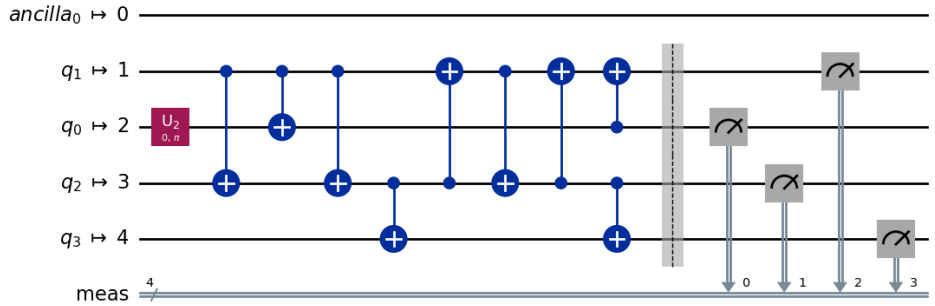
The final step involves integrating both Algorithm 1 `Interaction Layout Mapping` and Algorithm 2 `Dynamic Lookahead Swap Routing` into the Staged Pass Manager. In this pass manager, the `init` stage unrolls gates involving more than three qubits, and `layout` applies the most effective interaction layout mapping. The `routing` generates the minimal number of SWAP gates required to satisfy the device's coupling constraints. The final stage, `translation`, converts the logical gates into the available basis gate set for the selected device backend. By employing this staged pass manager, the initial quantum circuit is fully translated and optimized for execution on the target backend.

For instance, the initial circuit shown in Figure 2.4a has a circuit size of 11 and a circuit depth of 6. When using the `TrivialLayout` and `BasicSwap` methods, as depicted in Figure

2.9b, the circuit size increases to 17 and depth to 13 (Figure 2.10a). In contrast, the `InteractionLayout` and `LookaheadSwap` methods, as shown in Figure 2.9d, produce a more compact circuit with a size of 14 and a depth of 9 (Figure 2.10b).



(a) Translated final layout with `BasicSwap` routing, with additional 2 SWAPs



(b) Translated final layout with `Lookahead` routing, with additional 1 SWAP

Figure 2.10: Comparison for swap gate routing methods

2.3.2 Verification

The verification process involves retrieving the results of a quantum computation after a job has been executed on a quantum device or simulator. For this work, `GenericBackend2` [37] is used, and the backend is run without noise. The result is returned as a dictionary, where the keys represent the bitstrings of the measured qubits, and the values represent the number of times each bitstring was measured. This output allows for the analysis of the probability distribution of the qubit states after the quantum circuit has been executed.

The `Result` class [38] of the transpiled quantum circuit, run on Qiskit's preset pass manager, is then compared with the result obtained from the algorithm's swap implementation. It is observed that several of the highest occurrences are identical in both cases (Figure C.1a and Figure C.1b), indicating consistency between the two methods, as shown in Appendix C.

3 | Results and Analysis

3.1 Configurations

The evaluation is configured as follows:

1. Metrics: The evaluation focuses on two main metrics, the total number of additional swap gates (g) and the resulting circuit depth (d) after transpilation.
2. Experiment Software: The algorithm is implemented in Python version 3.10.12 with IBM Qiskit framework. The code is available on Github https://github.com/natashaval/qubit-mapping-distributed-qc.
3. Experiment Hardware: The tests were conducted on a standard personal computer equipped with an Intel i7-8700 CPU and 16 GB of RAM.
4. Layout: The algorithms were tested using 20 qubits across the layouts detailed in Table 3.1. The layout names follow the format: *<layout name>_<number of qubits>_<number of groups>*. Each layout is a symmetric coupling graph, enabling bidirectional operation of two-qubit gates between any pair of connected physical qubits. Coupling graphs are illustrated in Appendix **??**.

Table 3.1: Number of qubits and groups for layouts

Layout	Number of qubits	Number of groups
Full	20	1
Full	10	2
Grid	9	2
Ring	10	2
Full	7	3
Grid	8	3
Ring	7	3
Full	5	4
Grid	6	4
Ring	5	4
T Horizontal	5	4
T Vertical	5	4
Line	1	20

5. Benchmarks: The benchmarks listed in Table 3.2 are taken from MQTBench [39]. The quantum circuits are composed of native gates from the Qiskit library and are represented in the QASM 2.0 language.

Table 3.2: Circuit size and circuit depth of benchmark algorithms for 5, 10, and 15 qubits

Algorithm	**n = 5**		**n = 10**		**n = 15**	
	gate	depth	gate	depth	gate	depth
ghz	7	7	12	12	17	17
dj	36	11	79	17	118	22
graphstate	50	22	100	26	150	29
qft	71	38	270	78	591	118
wstate	73	45	163	90	253	135
qftentangled	78	42	282	82	608	122
vqe	83	21	168	26	253	31
qaoa	95	31	190	34	285	34
realamprandom	130	37	335	57	615	77
twolocalrandom	130	37	335	57	615	77
su2random	150	41	375	61	675	81
qnn	154	58	459	108	914	158
portfolioqaoa	195	72	615	132	1260	192
random	223	97	646	155	1992	412
portfoliovqe	310	107	1145	217	2505	327

3.2 Evaluation

The data is processed from the table in the appendix F.

3.2.1 Distribution of Additional Swap Gates and Circuit Depth

The box plot in Figure 3.1 compares the average number of additional swap gates required by three different strategies, namely `BasicSwap`, `SabreSwap`, and `LookaheadSwap`, across various layout configurations. `BasicSwap` [35] strategy employs a brute force approach, adding one or more swap gates whenever a connectivity constraint is encountered, iterating through each layer until the circuit is compatible. The `SabreSwap` strategy uses a heuristic search to minimize the number of swap gates and reduce circuit depth [18], [40]. Meanwhile, the `LookaheadSwap` strategy combines elements from both algorithms (Algorithm 1 and 2) that has been described in the previous section. A lower number of swap gates or fewer circuit depth indicates better strategy performance.

The `BasicSwap` consistently exhibits the highest number of swap gates across all layouts, with particularly high values exceeding 5000 additional swap gates in layouts such as *ring_10_2*, *t_horizontal_5_4*, *t_vertical_5_4*, and performing worst in the *line_1_20* layout. The large spread and height of the boxes indicate both poor and inconsistent performance. In contrast, `SabreSwap` performs significantly better with the range of around

1000 additional swap gates for all layouts. `LookaheadSwap` performs almost as well as `SabreSwap`, with the best results seen in the *full_7_3* layout. However, in some cases like *full_5_4* and *ring_5_4*, the `LookaheadSwap` strategy has exceptionally low values that are likely outliers due to timeouts, which may skew the perception of its effectiveness in these instances.

Overall, the layouts that show better performance, as indicated by lower swap gate counts, are *grid_8_3*, *grid_6_4*, *full_10_2* and *full_7_3*. These results suggest that *grid* and *full* layouts are generally more efficient than other configurations. It is also important to note that *full_20_1* and *line_1_20* serve as baselines; *full_20_1* represents a maximally connected coupling map and is expected to deliver the best performance, while *line_1_20*, being the least connected, is likely to show the poorest performance.

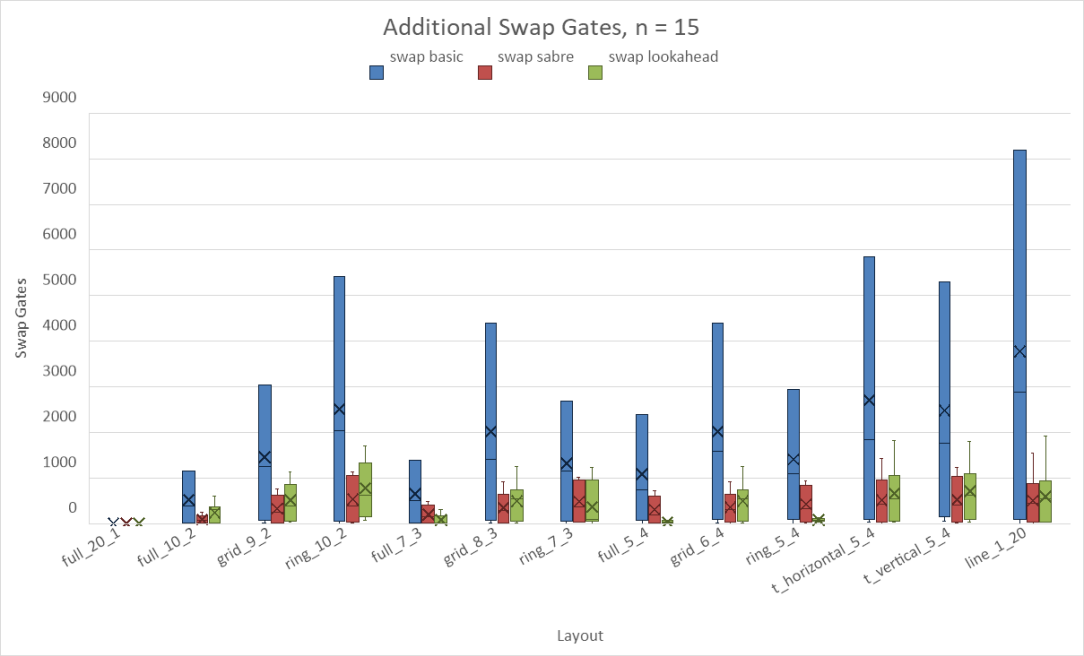


Figure 3.1: Box plot additional swap gates for a circuit size of 15

Similarly, the analysis of quantum circuit depths across various layouts, as shown in Figure 3.2, reveals significant differences in the effectiveness of the three optimization methods. The depths associated with `BasicSwap` consistently result in the highest circuit depths often exceeding 1000, and in some cases approach 2000 or more in layouts like *line_1_20*, *t_horizontal_5_4*, and *ring_10_2*. In contrast, `SabreSwap` and `LookaheadSwap` both achieve significantly shallower circuit depths, generally clustering below 500. How-

ever, some layouts, such as *full_5_4* and *ring_5_4* show occasional outliers.

Overall, the `SabreSwap` and `LookaheadSwap` strategies perform comparably well, with the latter slightly outperforming `SabreSwap` in certain configurations. The most efficient layouts, characterized by lower circuit depths, include *full_7_3* and *grid_6_4*, where both `SabreSwap` and `LookaheadSwap` consistently maintain minimal circuit depth. This suggests that for minimizing circuit depth, `LookaheadSwap` is generally the most effective strategy, closely followed by `SabreSwap`.

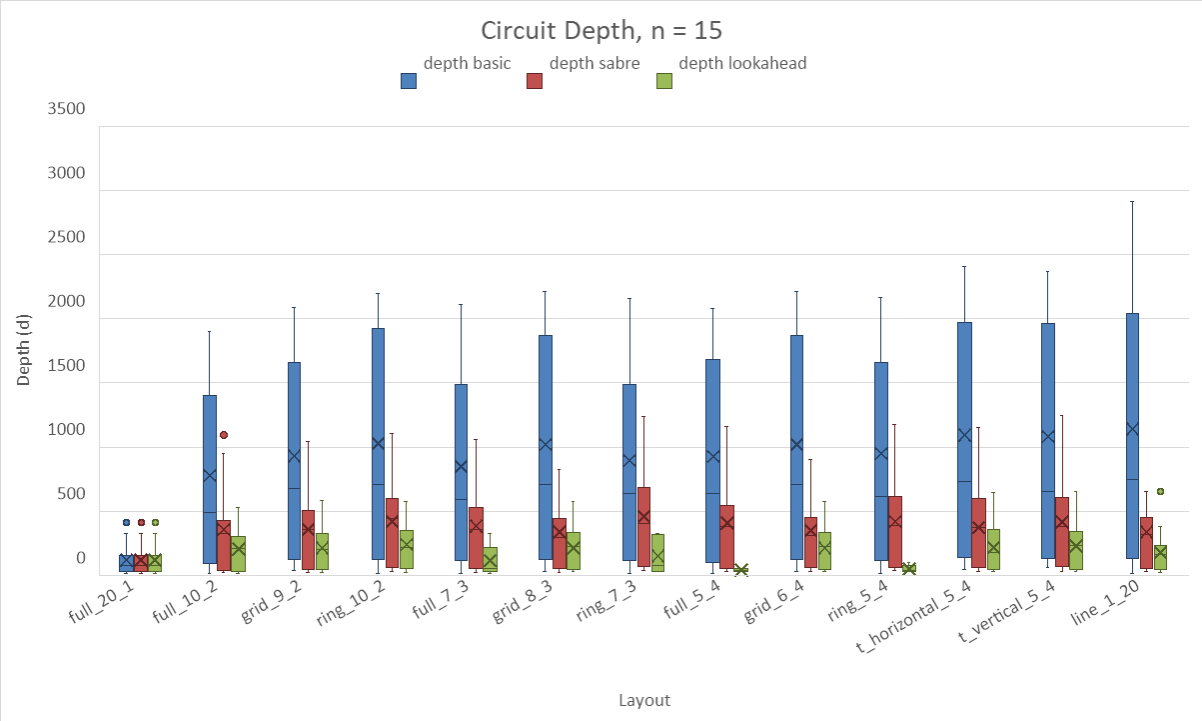
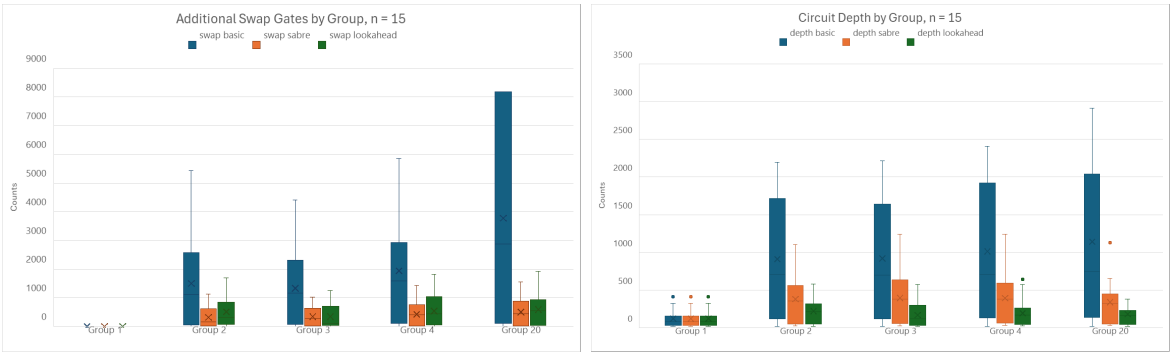


Figure 3.2: Box plot circuit depth for a circuit size of 15

The box plots present the additional swap gates 3.3a and circuit depth 3.3b categorized by group for a circuit size of 15. Group 1 and Group 20 serve as baselines, with Group 1 representing a fully connected graph and Group 20 a line coupling graph. Among other groups, Group 2 stands out as the best performer. In Group 2, the `LookaheadSwap` strategy performs particularly well, showing very low swap gate counts and moderate circuit depths, reflecting consistent and efficient performance. Group 3, while similar to Group 2, displays slightly more variability in circuit depths and a minor increase in swap gate counts, particularly with the `BasicSwap` strategy, though the `LookaheadSwap` approach still outperforms the others. Group 4, on the other hand, shows the most variability, especially with the `BasicSwap` strategy, which has a much higher median and

a widespread, suggesting inconsistent performance. Although the `LookaheadSwap` strategy in Group 4 still achieves the lowest swap gate counts and circuit depths, its performance is less consistent compared to Groups 2 and 3. Therefore, Group 2 emerges as the top performer in minimizing swap gates and lowering circuit depths with the `LookaheadSwap` strategy, followed by Group 3, while Group 4 shows the least favourable results.



(a) Box Plot Additional Swap Gates by Group (b) Box Plot Circuit Depth by Group

Figure 3.3: Swap strategies categorized in group for a circuit size of 15

3.2.2 The Total Number of Algorithms Successfully Run for Each Layout

The previous section mentioned that the algorithms for the *full_5_4* and *ring_5_4* layouts show outliers, indicating that they do not execute all the algorithms. Therefore, the chart in Figure 3.4 specifies the total number of successful algorithms run across different layouts, focusing on circuit sizes ranging from 5, 10, and 15 qubits, with a total of 15 algorithms per round. Most layouts perform consistently well, achieving the maximum count of 15 successful runs per benchmark.

One thing to notice is that *full_7_3* and *full_5_4* layouts fail to complete the benchmarks for the largest circuit size of 15 qubits, with *full_7_3* achieving only 9 successful runs and *full_5_4* barely completing 3 runs. Additionally, *ring_7_3* and *ring_5_4* layouts also encounter difficulties, particularly as the circuit size increases beyond 10 qubits. *Ring_7_3* can only achieve 10 and 8 successful runs for circuit sizes 10 and 15, while *ring_5_4* struggles even at the 5-qubit benchmark, with its performance worsening as the circuit size increases. The failed algorithms and layouts are listed in Appendix E.

These variations suggest that certain layouts are more robust across different run sizes,

while others, particularly *ring_7_3* and *ring_5_4* face challenges with medium run sizes, indicating possible inefficiencies or challenges in these configurations. In general, the layouts in groups 1 and 2 are more consistent across different run sizes, while those in groups 3 and 4 tend to show more variability.

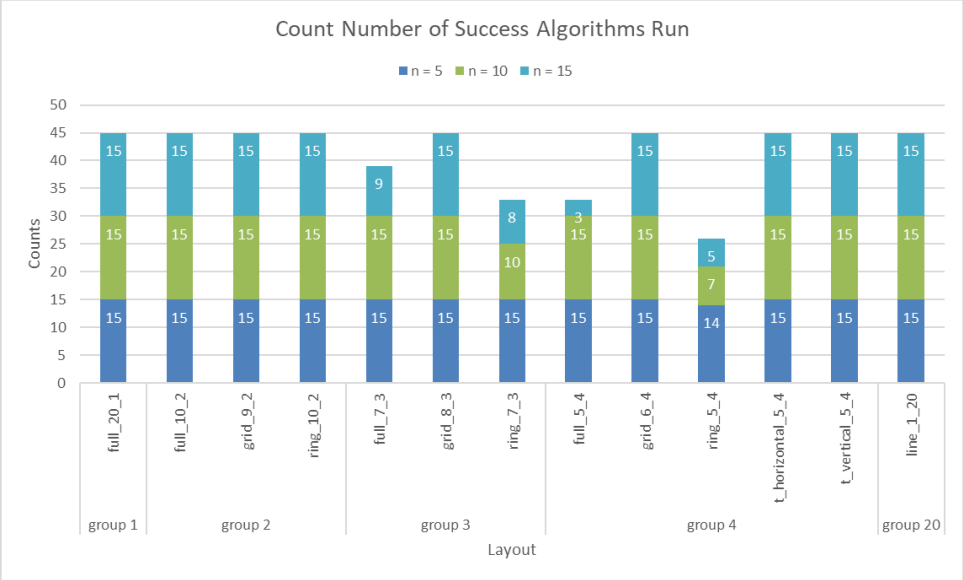


Figure 3.4: Total count of successful algorithm runs across various layouts, $n = 15$

In addition, the chart in Figure 3.5 highlights the total count of successful layout executions across different algorithms. Most algorithms like "dj", "graphstate" and "qaoa" maintain consistent performance across all benchmark sizes with 100% success rate. However, some algorithms, like "qft", "qftentangled", "realamprandom", and "twolocalrandom", show slightly reduced success rates, particularly for $n = 10$, where the count drops to 12. The algorithms "qnn" and "random" display even more pronounced variations, with success rates for $n = 10$ and $n = 15$ decreasing to 11 and 9, respectively.
While most algorithms perform reliably across all benchmarks, a few show inconsistencies, especially with $n = 10$ and $n = 15$, suggesting that certain algorithms may struggle with larger problem sizes.

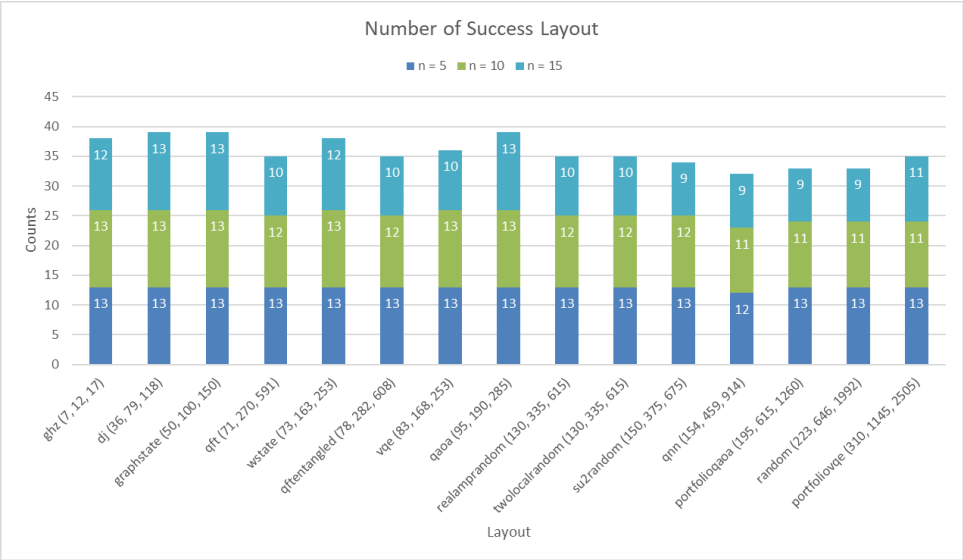


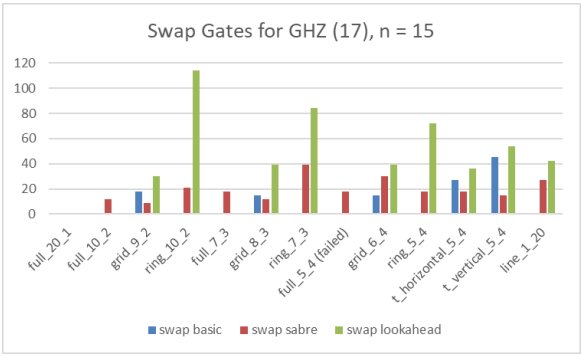
Figure 3.5: Total count of successful layouts across different algorithms

3.2.3 Layouts for Algorithms

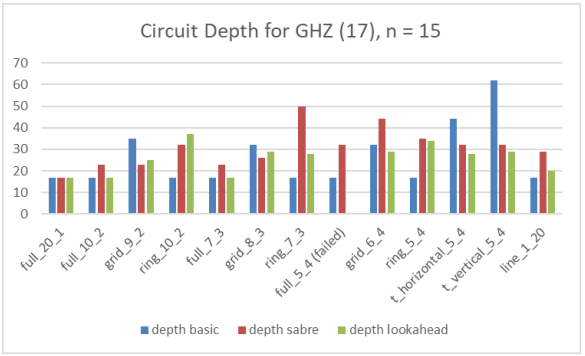
In this section, a selection of representative algorithms - GHZ, Deutsch-Jozsa, Graph State, VQE, and Portfolio QAOA - are analyzed and compared across various layouts. The baseline values for the total circuit gates and circuit depth are provided in parentheses, and the results discussed are based on using 15 qubits. This comparison highlights how different layouts perform when subjected to different swap routing methods.

3.2.3.1 GHZ

The GHZ charts compare the number of additional swap gates needed for different layouts (Figure 3.6a) and the resulting circuit depth (Figure 3.6b) when running with 17 qubits. It shows that the `BasicSwap` strategy generally performs best, requiring the fewest swaps, particularly in all full, ring, and line layouts, where it results in 0 additional swap gates. In this context, `SabreSwap` tend to perform worse than `BasicSwap`, while the `LookaheadSwap` strategy performs significantly worse than the others, with a notable spike in the *ring_10_2* layout. Regarding circuit depth, the results for `LookaheadSwap` are nearly comparable to those of `SabreSwap`, with both strategies yielding a circuit depth of approximately 30.



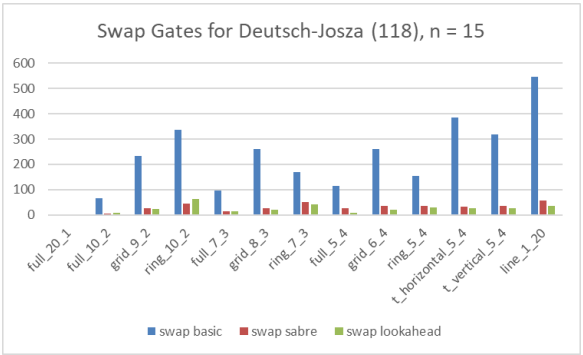
(a) Additional swap gates for GHZ



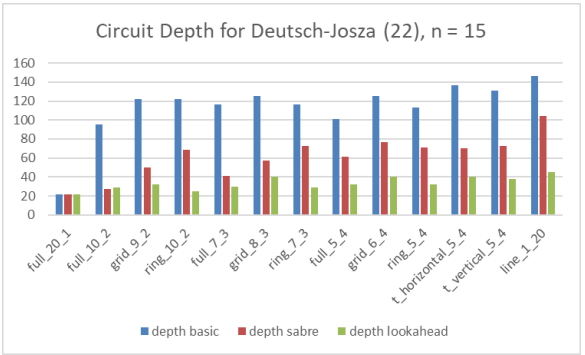
(b) Circuit depth for GHZ

3.2.3.2 Deutsch-Jozsa

The charts for running the Deutsch-Jozsa algorithm with 118 gates (Figure 3.7a) and a circuit depth of 22 (Figure 3.7b) show that the `SabreSwap` and `LookaheadSwap` strategies consistently outperform the `BasicSwap` strategy in minimizing swap gates. The *line_20_1* layout performs poorly, with the `BasicSwap` method leading to an extremely high number of additional swap gates, exceeding 500. Although the additional swap gates for `SabreSwap` and `LookaheadSwap` are similar across all layouts, remaining below 50, the circuit depth with `LookaheadSwap` is significantly lower, staying under 40 for all layouts, compared to `SabreSwap`, which is around 60.



(a) Additional swap gates for Deutsch-Josza

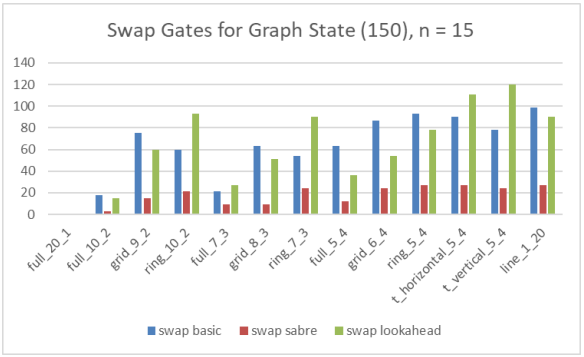


(b) Circuit depth for Deutsch-Josza

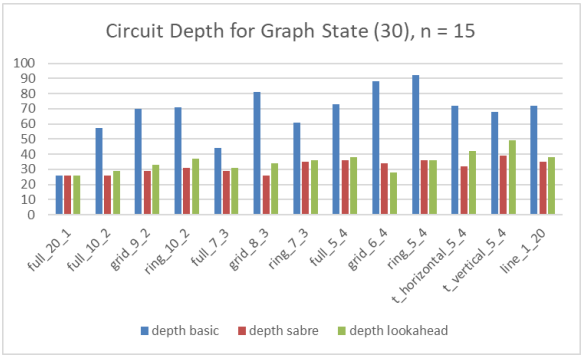
3.2.3.3 Graph State

The Graph State charts compare additional swap gates with an initial count of 150 gates (Figure 3.8a) and a circuit depth of 30 (Figure 3.8b) across different layouts. In this algorithm, the performance of `LookaheadSwap` is similar to `BasicSwap`, with the poorest performance observed in the *t_vertical_5_4* layout, which requires 120 additional swap gates. The `SabreSwap` strategy, on the other hand, results in significantly fewer swap

gates, with counts around 20. Despite the difference in additional swap gates between `SabreSwap` and `LookaheadSwap` being as much as 200%, the resulting circuit depth is almost comparable, ranging from 30 to 40 across all layouts.



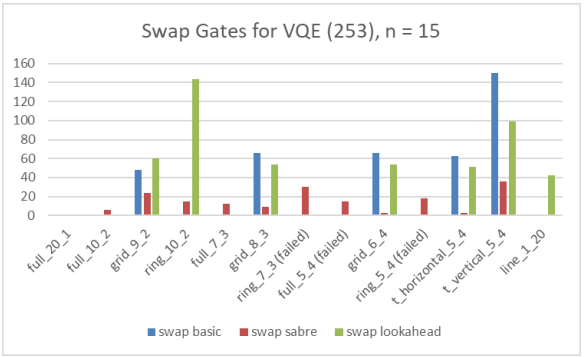
(a) Additional swap gates for Graph State



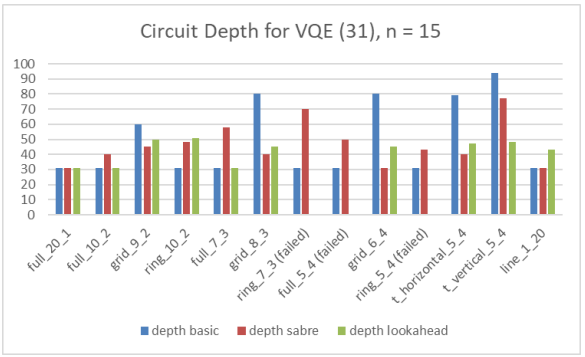
(b) Circuit depth for Graph State

3.2.3.4 VQE

The chart illustrates that for the VQE algorithm with 253 gates (Figure 3.9a), the `SabreSwap` strategy generally performs best across most layouts, particularly in *grid_8_3*, *grid_6_4* and *t_horizontal_5_4*, where it minimizes the number of additional swap gates. In contrast, the `LookaheadSwap` strategy tends to require significantly more swaps, especially in layouts like *ring_10_2* and *t_vertical_5_4*. The `BasicSwap` strategy shows similar behaviour to `LookaheadSwap`, with the exception of the *t_vertical_5_4* layout, where it exceeds 140 additional swap gates. When examining circuit depth (Figure 3.9b), `BasicSwap` does not show much difference from the baseline, remaining at 31. Meanwhile, the circuit depth performance of `LookaheadSwap` is comparable to that of `SabreSwap`, with layouts like *full_7_3* and *t_vertical_5_4* delivering significantly better results, with a reduction of nearly 40%.



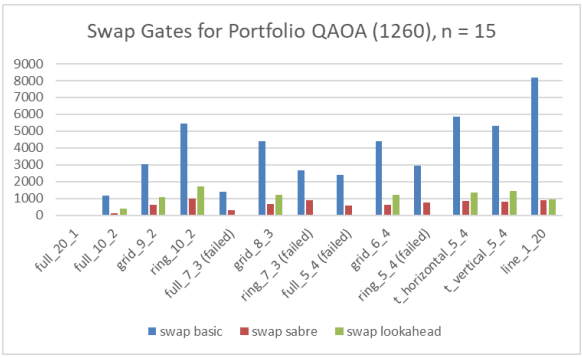
(a) Additional swap gates for VQE



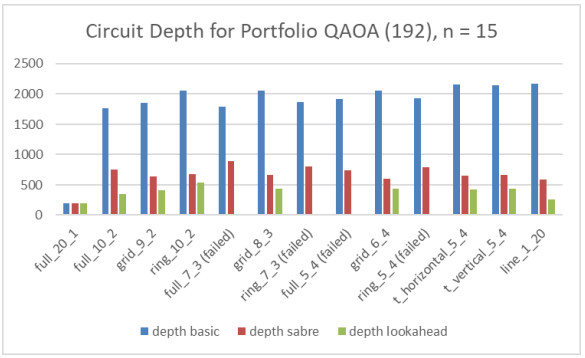
(b) Circuit depth for VQE

3.2.3.5 Portfolio QAOA

These final charts serve as representatives for various other algorithms listed in Table 3.2, ranging from "qft" to "portfoliovqe". The charts compare the performance of swap strategies for the Portfolio QAOA algorithm with 1260 gates (Figure 3.10a) and a circuit depth of 192 (Figure 3.10b). The `BasicSwap` strategy performs poorly, especially in layouts like *ring_10_2*, *t_horizontal_5_4*, and *line_1_20* with swap counts exceeding 5000 in some cases. On the other hand, both `SabreSwap` and `LookaheadSwap` significantly reduce the number of swap gates across most layouts to just below 2000, with `SabreSwap` generally requiring fewer swaps than `LookaheadSwap`. The second chart showing circuit depth follows a similar pattern: `BasicSwap` results in much deeper circuits, often exceeding 2000 in depth, particularly in the same problematic layouts. `SabreSwap` and `LookaheadSwap` again perform better, with `SabreSwap` consistently achieving lower circuit depths around 700, closely followed by `LookaheadSwap` just below 500.



(a) Additional swap gates for Portfolio QAOA



(b) Circuit depth for Portfolio QAOA

4 | Discussion

4.1 Time Complexity

4.1.1 Interaction Mapping Layout

The Algorithm 1 `InteractionLayoutMapping`, particularly the `calculate_final_maps` method, can be computationally intensive, especially in the worst-case scenario. During its execution, several key operations contribute to its complexity. Initially, the algorithm performs pre-processing steps, such as generating the Qubit Pair Interaction (QPI) matrix and calculating logical priorities and physical connectivity. These steps run in polynomial time relative to the number of physical qubits and two-qubit operations in the DAG. However, the main challenge arises in the core loop of the `calculate_final_maps` function, where the algorithm iteratively assigns logical qubits to physical qubits.

During this process, the algorithm checks and updates possible mappings in a `while logical_priority` loop, which continue until all logical qubits are assigned. Each iteration may create several new mappings, causing the number of possibilities to increase exponentially, as the algorithm must consider at a rate of $O(2^k \times |P|)$, where k is the recursion depth and $|P|$ is the number of physical qubits. As a result, the overall time complexity in the worst case is **exponential**, potentially reaching $O(2^k \times |P| \times n)$, where n is the number of logical qubits. This exponential growth is due to the combinatorial nature of the problem, where multiple candidate mappings are explored and expanded. In practical implementations, this could result in significant computational overhead as the number of qubits increases.

4.1.2 Lookahead Swap Routing

The Algorithm 2 `DynamicLookaheadSwap` is a heuristic method designed to map logical qubits to physical qubits in quantum circuits, especially on hardware with limited connectivity. The goal of the algorithm is to reduce the number of additional swap gates required when implementing the circuit on the target hardware, consequently, reducing the circuit depth as well.

The algorithm begins with an initialization phase, where it preprocesses the quantum circuit and sets up necessary data structures. This phase runs in linear time proportional to the number of gates. The algorithm then categorizes and analyzes dependencies be-

tween gates, focusing on two-qubit operations. This step is also a linear time complexity of $O(G)$, where G is the total number of two-qubit gates in the DAG.

In the main loop, the algorithm iterates over each layer of the circuit and checks whether the current qubit layout allows direct execution of two-qubit gates or if a swap operation is necessary. This `check_gate_connectivity` has a time complexity of $O(G \times n^2)$, where n is the number of qubits. The algorithm then `generate_possible_swaps` operations by considering the neighbours of the involved qubits in the coupling map, with a complexity of $O(G \times n^2 \times D)$, where D is the degree of connectivity in the coupling map.

For each potential swap, the algorithm evaluates its effect on the circuit using the `sum_effect` method. This method assesses how the swap improves qubit placement for future gates by exchanging physical qubits and reducing the distance between qubits. This swap evaluation process can be computationally intensive, with a worst-case time complexity of $O(S \times G)$, where S is the number of candidate swaps. If a beneficial swap is identified, it is applied to the layout at $O(1)$, which further modifies the circuit structure.

Overall, the worst-case time complexity of the `DynamicLookaheadSwap` algorithm is approximately $O(G \times n^2 \times D + S \times G)$, reflecting the number of two-qubit gates, qubits, and the connectivity of the coupling map. Despite the potential for high computational costs, the algorithm's use of heuristics and the pruning of less beneficial swaps helps manage its complexity, making it a practical approach for optimizing quantum circuits on devices with limited qubit connectivity.

4.2 Coupling Graph Options

4.2.1 Choosing Layout

The algorithm tested involves around 20 qubits, which aligns with previous studies [18], [32] that used 5 to 20 qubits in quantum programs sourced from IBM Qiskit [41], [42] and RevLib [43]. The chosen layout configurations - *full*, *grid*, *ring*, *t_horizontal*, and *t_vertical* - were selected to determine which layout offers the best performance based on node connectivity. The *t_horizontal* and *t_vertical* layouts, derived from a 5-qubit T-shaped IBM backend, were specifically tested to assess how the chain's length at either end impacts qubit mapping performance. An illustration of how logical qubits are placed on physical qubits is provided in Appendix D.

When analyzing the distribution of additional swap gates, the *full* layout consistently

outperforms the *grid* and *ring* layouts. Full connectivity allows any qubit to interact directly with any other qubit, eliminating the need for intermediate swaps and ensuring the shortest path between qubits is always direct [44]. The *grid* layout ranks second, as it has several nodes with three neighbours, compared to the *ring* layout, where each node has only two neighbours. The *grid* layout requires additional operations to manage qubit connectivity, and the *ring* linear layout structure further limits efficient algorithm implementation, leading to higher resource use and less efficient computation [45]. The *t_horizontal_5_4* and *t_vertical_5_4* layouts show no significant difference in additional swap gates and are similar to the *line_20_1* layout, indicating they share a linear arrangement. In summary, the number of neighbouring qubits significantly impacts the reduction of additional swap gates needed.

Surprisingly, the *grid* layout proved to be the most stable in terms of circuit depth, successfully running all algorithms compared to the *full* and *ring* layouts. The *t_horizontal* and *t_vertical* layouts also performed well, running all 15 algorithms. This stability likely stems from the presence of qubits with three neighbours in these layouts, which enhances their ability to handle circuit depth effectively. Although *t_horizontal_5_4* and *t_vertical_5_4* performed similarly to *line_20_1* in terms of additional swap gates, they exhibited slightly higher variance in circuit depth. The `DynamicLookaheadSwap` algorithm works well with these three layouts because it prioritizes executing as many active gates as possible from the dependency list while performing SWAP operations simultaneously [46]. In contrast, the *ring* layout is limited by its linear qubit configuration, and the *full* layout struggles with large traversal nodes when evaluating swap candidates. This suggests that an optimal number of neighbouring qubits - neither too many nor too few - is necessary to maintain a balanced circuit depth when using `InteractionLayoutMapping` algorithm.

4.2.2 Choosing Number of Groups

According to the results shown in Figure 3.3, Group 2 outperformed the other groups with low variability. This success is attributed to dividing tasks into fewer groups, which reduces the number of communication channels required between groups. With only two groups, communication overhead is minimized, as there is only one direct communication link, simplifying data exchange and reducing potential bottlenecks from waiting on other groups [47]. In contrast, increasing the number of groups complicates coordination,

requiring more complex synchronization and qubit allocation across groups [48]. This added complexity can introduce overhead in the initial qubit mapping, as seen in the failed interaction mapping for the *full_5_4* layout. Fewer groups result in simpler coordination, fewer dependencies to manage, and less overall system complexity.

In the context of networked quantum computers, it is still crucial to maintain high connectivity within the individual quantum computers inside each group. Even as tasks are divided to reduce inter-group communication overhead, the internal connectivity of each quantum computer must remain robust to ensure efficient qubit interactions. High connectivity within a quantum computer allows for more flexible and efficient execution of quantum operations, minimizing the need for additional swap gates and reducing circuit depth. This balance between minimizing communication overhead across groups and maintaining strong internal connectivity is key to optimizing the overall performance of quantum networks.

4.3 Limitation and Errors

Appendix E details the layouts where timeouts occurred during quantum circuit transpilation. The `InteractionLayoutMapping` algorithm generally succeeds in mapping logical to physical qubits, except for the *full_5_4* layout, where it failed due to a timeout during qubit placement. This issue arises because the algorithm, when traversing possible neighbours, encounters an excessively large tree. To address this, the algorithm includes an exit function that checks if the coupling map is fully connected by calculating the number of neighbours for each physical qubit. If more than 80% of physical qubits have the maximum number of neighbours, the algorithm exits early and returns a one-to-one mapping. For example, in the best case, the *full_10_2* layout has 9 neighbours for 18 qubits and 10 neighbours for 2 qubits that connecting groups, allowing the algorithm to detect the full connectivity and return the one-to-one mapping quickly. However, the *full_5_4* layout, with 4 neighbours for 14 qubits and 5 neighbours for 6 qubits, fails to meet the 80% threshold, causing the algorithm to continue exploring all possibilities until it runs out of memory. A suggested solution to overcome this problem is to analyze the local graph characteristics and check if the graph is k-connected [49].

Similarly, the `DynamicLookaheadSwap` algorithm generally works well but begins to fail for certain algorithms at a circuit size of 10 in the *ring_7_3* and *ring_5_4* layouts,

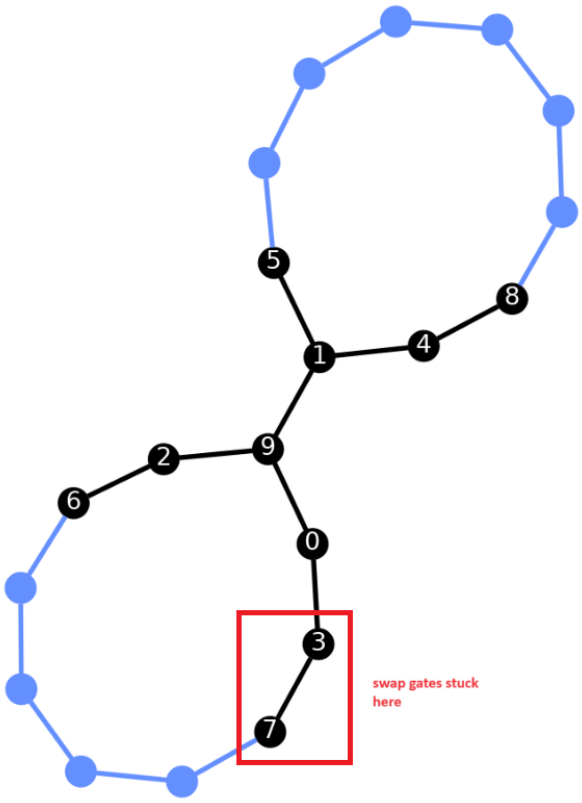


Figure 4.1: The swap operation is stuck because the highest Lookahead value is at Q_7 and Q_3.

and at a circuit size of 15 in the *full_7_3* layout, as well as the previous two layouts. The swap timeout error likely occurs due to the greedy nature of the qubit placement, which assigns logical qubits to the physical qubits with the highest connectivity. In ring layouts, the algorithm starts mapping from the center and branches out. If there are multiple active gates, the algorithm may direct one gate to a branch end but still fail to find adjacent physical qubits to operate the gate, leading to an infinite loop between two nodes at the end until timeout, as illustrated in Figure 4.1. One strategy to address this issue is to keep an empty list to track assigned physical qubits, but this can cause problems when multiple gates are active. If one gate completes, the next might get stuck because the necessary qubits to go back are already assigned. A potential solution could involve allowing the algorithm to backtrack to certain points in the layout and explore alternative paths [50].

4.4 Direction of Future Work

This section outlines potential future enhancements to optimize the current algorithm:

1. **Incorporating Noise**: The current work assumes quantum devices without noise. However, in practice, the noise levels can vary between physical qubits on the same device, which should be factored into the algorithm [51]. Future work could integrate noise considerations by configuring constraints, instruction sets, qubit properties, operation timing, and other parameters using the IBM `Target` [52] class.
2. **Addressing Communication Costs in Distributed QPU**: The current approach does not account for the communication cost between groups in a distributed QPU. Communication between distant quantum computers can introduce higher noise and errors, affecting gate fidelity and disrupting entanglement [53]. Future work could incorporate communication costs into the initial qubit mapping process [54], either by evenly distributing logical qubits when communication costs are low or by grouping them locally when communication costs are high.
3. **Unidirectional Connectivity Constraints**: The coupling graph used in this work assumes bidirectional connections, where two-qubit gates can operate in both directions. A future enhancement could involve considering unidirectional connectivity constraints, which are more restrictive, and refining the search strategy to improve performance under these conditions [55].
4. **Utilizing Optimization Techniques**: Currently, the algorithm only calculates the difference in additional swap gates before and after transpilation. Future work could introduce additional optimization stages, such as applying passes that check gate commutation rules [56] and optimizing gate decomposition [57], to further enhance performance.

5 | Conclusion

To address the connectivity constraints between logical and physical qubits, two algorithms were introduced: "Interaction Mapping Layout" and "Dynamic Lookahead Swap Routing". Although the `LookaheadSwap` method generates slightly more additional swap gates compared to the `SabreSwap` method, it excels in minimizing circuit depth, achieving the lowest circuit depth among the three strategies. In testing various layouts and groups, *full* and *grid* layouts demonstrated superior performance in distributed settings, offering better connectivity due to shorter paths between nodes and more even load distribution. Additionally, organizing qubits in fewer groups is generally preferable, as it simplifies information exchange leading to fewer additional swap gates and more direct operation of quantum gates within the groups, leading to a lower circuit depth. Overall, choosing layouts with higher connectivity and fewer groups can enhance circuit execution efficiency and performance on quantum hardware, making these approaches particularly well-suited for networked quantum architectures.

References

[1] Y. Cao, J. Romero, J. P. Olson, *et al.*, "Quantum chemistry in the age of quantum computing," *Chemical Reviews*, vol. 119, no. 19, pp. 10 856–10 915, Oct. 2019, ISSN: 0009-2665. DOI: 10.1021/acs.chemrev.8b00803.

[2] A. Paler, L. Sasu, A.-C. Florea, and R. Andonie, "Machine learning optimization of quantum circuit layouts," *ACM Transactions on Quantum Computing*, vol. 4, no. 2, pp. 1–25, Feb. 2023. DOI: 10.1145/3565271.

[3] A. Peruzzo, J. McClean, P. Shadbolt, *et al.*, "A variational eigenvalue solver on a photonic quantum processor," *Nature Communications*, vol. 5, no. 1, p. 4213, Jul. 2014, ISSN: 2041-1723. DOI: 10.1038/ncomms5213.

[4] J. Preskill, "Quantum Computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, Aug. 2018, ISSN: 2521-327X. DOI: 10.22331/q-2018-08-06-79.

[5] D. Cuomo, M. Caleffi, and A. S. Cacciapuoti, "Towards a distributed quantum computing ecosystem," *IET Quantum Communication*, vol. 1, no. 1, pp. 3–8, 2020. DOI: https://doi.org/10.1049/iet-qtc.2020.0002.

[6] M. Caleffi, M. Amoretti, D. Ferrari, J. Illiano, A. Manzalini, and A. S. Cacciapuoti, "Distributed quantum computing: A survey," *Computer Networks*, p. 110 672, 2024, ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2024.110672.

[7] T. Boroglu, *Microservices*, en, May 2024. [Online]. Available: https://tolgaboroglu.medium.com/microservices-28b727e7f153 (visited on 08/22/2024).

[8] A. Barenco, C. H. Bennett, R. Cleve, *et al.*, "Elementary gates for quantum computation," *Phys. Rev. A*, vol. 52, pp. 3457–3467, 5 Nov. 1995. DOI: 10.1103/PhysRevA.52.3457.

[9] P. Krantz, M. Kjaergaard, F. Yan, T. P. Orlando, S. Gustavsson, and W. D. Oliver, "A quantum engineer's guide to superconducting qubits," *Applied Physics Reviews*, vol. 6, no. 2, p. 021 318, Jun. 2019, ISSN: 1931-9401. DOI: 10.1063/1.5089550.

[10] S. Khandavilli, I. Palanisamy, M. V. Nguyen, T. V. Le, T. N. Nguyen, and T. N. Dinh, "Towards Fidelity-Optimal Qubit Mapping on NISQ Computers," en, in *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, Bellevue, WA, USA: IEEE, Sep. 2023, pp. 89–98, ISBN: 9798350343236. DOI: 10.1109/QCE57702.2023.00019.

[11] T. Itoko, R. Raymond, T. Imamichi, and A. Matsuo, "Optimization of quantum circuit mapping using gate transformation and commutation," *Integration*, vol. 70, pp. 43–50, Jan. 2020, ISSN: 01679260. DOI: 10.1016/j.vlsi.2019.10.004.

[12] M. Bandic, C. G. Almudever, and S. Feld, "Interaction graph-based characterization of quantum benchmarks for improving quantum circuit mapping techniques," *Quantum Machine Intelligence*, vol. 5, no. 2, p. 40, Oct. 2023, ISSN: 2524-4914. DOI: 10.1007/s42484-023-00124-1.

[13] D. Ferrari, A. S. Cacciapuoti, M. Amoretti, and M. Caleffi, "Compiler Design for Distributed Quantum Computing," en, *IEEE Transactions on Quantum Engineering*, vol. 2, pp. 1–20, 2021, ISSN: 2689-1808. DOI: 10.1109/TQE.2021.3053921.

[14] IBMQuantum, *Compute resources*. [Online]. Available: https://quantum.ibm.com/services/resources (visited on 03/13/2024).

[15] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, *et al.*, *Qiskit: An open-source framework for quantum computing*, version 0.7.2, Feb. 2019. DOI: 10.5281/zenodo.2562111. [Online]. Available: https://doi.org/10.5281/zenodo.2562111.

[16] A. W. Cross, L. S. Bishop, J. A. Smolin, and J. M. Gambetta, *Open quantum assembly language*, 2017. arXiv: 1707.03429 [quant-ph]. [Online]. Available: https://arxiv.org/abs/1707.03429.

[17] A. Ash-Saki, M. Alam, and S. Ghosh, "QURE: Qubit Re-allocation in Noisy Intermediate-Scale Quantum Computers," en, in *Proceedings of the 56th Annual Design Automation Conference 2019*, Las Vegas NV USA: ACM, Jun. 2019, pp. 1–6, ISBN: 978-1-4503-6725-7. DOI: 10.1145/3316781.3317888. (visited on 07/31/2024).

[18] G. Li, Y. Ding, and Y. Xie, "Tackling the qubit mapping problem for NISQ-era quantum devices," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence RI USA: ACM, Apr. 4, 2019, pp. 1001–1014, ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304023.

[19] IBMQuantum, *Transpiler*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/transpiler (visited on 06/13/2024).

[20] Y. Ding and F. T. Chong, "Circuit synthesis and compilation," in *Quantum Computer Systems: Research for Noisy Intermediate-Scale Quantum Computers*, Springer, 2020, pp. 91–125.

[21] V. V. Shende and I. L. Markov, *On the cnot-cost of toffoli gates*, 2008. arXiv: 0803.2316 [quant-ph]. [Online]. Available: https://arxiv.org/abs/0803.2316.

[22] R. Wille and L. Burgholzer, "MQT QMAP: Efficient Quantum Circuit Mapping," in *Proceedings of the 2023 International Symposium on Physical Design*, arXiv:2301.11935 [quant-ph], Mar. 2023, pp. 198–204. DOI: 10.1145/3569052.3578928. (visited on 02/19/2024).

[23] P. Gokhale, T. Tomesh, M. Suchara, and F. T. Chong, *Faster and More Reliable Quantum SWAPs via Native Gates*, 2021. DOI: 10.48550/ARXIV.2109.13199. (visited on 08/23/2024).

[24] A. Botea, A. Kishimoto, and R. Marinescu, "On the Complexity of Quantum Circuit Compilation," en, *Proceedings of the International Symposium on Combinatorial Search*, vol. 9, no. 1, pp. 138–142, Sep. 2021, ISSN: 2832-9163, 2832-9171. DOI: 10.1609/socs.v9i1.18463.

[25] A. Cowtan, S. Dilkes, R. Duncan, A. Krajenbrink, W. Simmons, and S. Sivarajah, "On the Qubit Routing Problem," en, *arXiv preprint arXiv:1902.08091*, 32 pages, 1190396 bytes, 2019, ISSN: 1868-8969. DOI: 10.4230/LIPICS.TQC.2019.5.

[26] IBMQuantum, *CouplingMap*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.CouplingMap (visited on 07/18/2024).

[27] H. Liu, B. Zhang, Y. Zhu, H. Yang, and B. Zhao, "QM-DLA: An efficient qubit mapping method based on dynamic look-ahead strategy," en, *Scientific Reports*, vol. 14, no. 1, p. 13 118, Jun. 2024, ISSN: 2045-2322. DOI: 10.1038/s41598-024-64061-0. (visited on 07/01/2024).

[28] IBMQuantum, *FakeLondonV2*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit-ibm-runtime/qiskit_ibm_runtime.fake_provider.FakeLondonV2 (visited on 07/28/2024).

[29] IBMQuantum, *AnalysisPass*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.AnalysisPass (visited on 07/28/2024).

[30] IBMQuantum, *Layout*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.Layout (visited on 07/29/2024).

[31] IBMQuantum, *DAGCircuit*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.dagcircuit.DAGCircuit (visited on 08/11/2024).

[32] P. Zhu, Z. Guan, and X. Cheng, "A Dynamic Look-Ahead Heuristic for the Qubit Mapping Problem of NISQ Computers," en, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4721–4735, Dec. 2020, ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2020.2970594.

[33] IBMQuantum, *LookaheadSwap*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.passes.LookaheadSwap (visited on 07/31/2024).

[34] IBMQuantum, *TrivialLayout*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.passes.TrivialLayout (visited on 07/17/2024).

[35] IBMQuantum, *BasicSwap*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.passes.BasicSwap (visited on 07/31/2024).

[36] IBMQuantum, *TransformationPass*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.TransformationPass (visited on 07/11/2024).

[37] IBMQuantum, *GenericBackendV2*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.providers.fake_provider.GenericBackendV2 (visited on 07/29/2024).

[38] IBMQuantum, *Result*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.result.Result (visited on 08/14/2024).

[39] N. Quetschlich, L. Burgholzer, and R. Wille, "MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing," *Quantum*, vol. 7, p. 1062, Jul. 2023, arXiv:2204.13719 [quant-ph], ISSN: 2521-327X. DOI: 10.22331/q-2023-07-20-1062.

[40] IBMQuantum, *SabreSwap*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.passes.SabreSwap (visited on 07/31/2024).

[41] M. Y. Siraichi, V. F. D. Santos, C. Collange, and F. M. Q. Pereira, "Qubit allocation," en, in *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, Vienna Austria: ACM, Feb. 2018, pp. 113–125, ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168822. (visited on 04/23/2024).

[42] A. Zulehner, A. Paler, and R. Wille, "An efficient methodology for mapping quantum circuits to the ibm qx architectures," *IEEE Transactions on Computer-Aided*

Design of Integrated Circuits and Systems, vol. 38, no. 7, pp. 1226–1236, 2019. DOI: 10.1109/TCAD.2018.2846658.

[43] R. Wille, D. Gro, L. Teuber, G. W. Dueck, and R. Drechsler, "RevLib: An Online Resource for Reversible Functions and Reversible Circuits," in *38th International Symposium on Multiple Valued Logic (ismvl 2008)*, ISSN: 0195-623X, Dallas, TX, USA: IEEE, May 2008, pp. 220–225, ISBN: 978-0-7695-3155-7. DOI: 10.1109/ISMVL.2008.43.

[44] B. Hayes, "Computing science: Graph theory in practice: Part i," *American Scientist*, vol. 88, no. 1, pp. 9–13, 2000, ISSN: 00030996. (visited on 08/22/2024).

[45] T. Peham, L. Burgholzer, and R. Wille, "On Optimal Subarchitectures for Quantum Circuit Mapping," en, *ACM Transactions on Quantum Computing*, vol. 4, no. 4, pp. 1–20, Dec. 2023, ISSN: 2643-6809, 2643-6817. DOI: 10.1145/3593594.

[46] C. Zhang, Y. Chen, Y. Jin, W. Ahn, Y. Zhang, and E. Z. Zhang, *A Depth-Aware Swap Insertion Scheme for the Qubit Mapping Problem*, 2020. DOI: 10.48550/ARXIV.2002.07289. (visited on 08/21/2024).

[47] D. Cuomo, M. Caleffi, K. Krsulich, *et al.*, "Optimized Compiler for Distributed Quantum Computing," en, *ACM Transactions on Quantum Computing*, vol. 4, no. 2, pp. 1–29, Jun. 2023, ISSN: 2643-6809, 2643-6817. DOI: 10.1145/3579367.

[48] C. Cicconetti, M. Conti, and A. Passarella, "Resource allocation in quantum networks for distributed quantum computing," in *2022 IEEE International Conference on Smart Computing (SMARTCOMP)*, 2022, pp. 124–132. DOI: 10.1109/SMARTCOMP55677.2022.00032.

[49] A. Cornejo and N. Lynch, "Reliably detecting connectivity using local graph traits," in *Principles of Distributed Systems*, C. Lu, T. Masuzawa, and M. Mosbah, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 87–102, ISBN: 978-3-642-17653-1.

[50] P. Parízek and O. Lhoták, "Fast detection of concurrency errors by state space traversal with randomization and early backtracking," *International Journal on Software Tools for Technology Transfer*, vol. 21, no. 4, pp. 365–400, Aug. 2019, ISSN: 1433-2787. DOI: 10.1007/s10009-018-0484-7.

[51] S. Niu, A. Suau, G. Staffelbach, and A. Todri-Sanial, "A hardware-aware heuristic for the qubit mapping problem in the nisq era," *IEEE Transactions on Quantum Engineering*, vol. 1, pp. 1–14, 2020. DOI: 10.1109/TQE.2020.3026544.

[52] IBMQuantum, *Target*, en. [Online]. Available: https://docs.quantum.ibm.com/api/qiskit/qiskit.transpiler.Target (visited on 07/30/2024).

[53] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. Martonosi, "Noise-Adaptive Compiler Mappings for Noisy Intermediate-Scale Quantum Computers," en, in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence RI USA: ACM, Apr. 2019, pp. 1015–1029, ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304075.

[54] M. Houshmand, Z. Mohammadi, M. Zomorodi-Moghadam, and M. Houshmand, "An evolutionary approach to optimizing teleportation cost in distributed quantum computation," *International Journal of Theoretical Physics*, vol. 59, no. 4, pp. 1315–1329, Apr. 2020, ISSN: 1572-9575. DOI: 10.1007/s10773-020-04409-0.

[55] S. Sanaei and N. Mohammadzadeh, "Qubit mapping of one-way quantum computation patterns onto 2d nearest-neighbor architectures," *Quantum Information Processing*, vol. 18, no. 2, p. 56, Jan. 2019, ISSN: 1573-1332. DOI: 10.1007/s11128-019-2177-x.

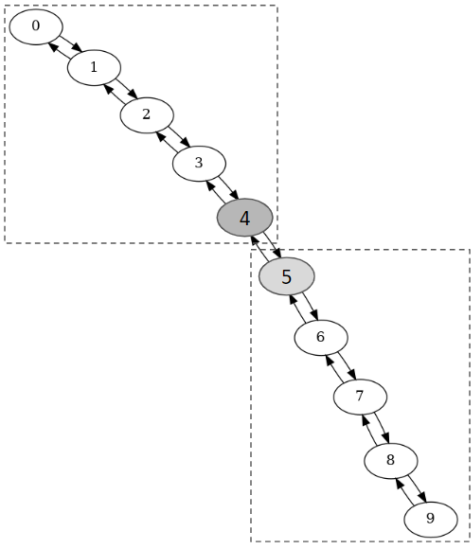
[56] T. Itoko, R. Raymond, T. Imamichi, A. Matsuo, and A. W. Cross, "Quantum circuit compilers using gate commutation rules," en, in *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, Tokyo Japan: ACM, Jan. 2019, pp. 191–196, ISBN: 978-1-4503-6007-4. DOI: 10.1145/3287624.3287701.

[57] M. S. Rudolph, J. Chen, J. Miller, A. Acharya, and A. Perdomo-Ortiz, "Decomposition of matrix product states into shallow quantum circuits," *Quantum Science and Technology*, vol. 9, no. 1, p. 015 012, Jan. 2024, ISSN: 2058-9565. DOI: 10.1088/2058-9565/ad04e6.

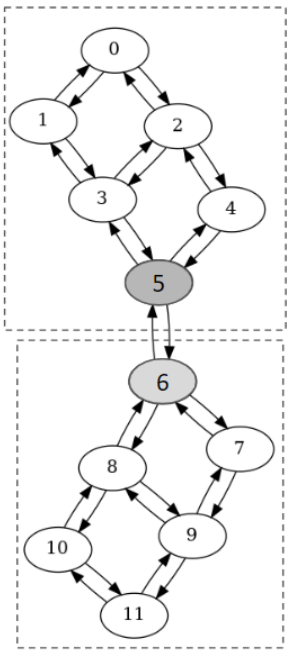
A | Source Code

Source code for all of the methods implemented in Chap. 2 for the project can be found in the GitHub repository:
https://github.com/natashaval/qubit-mapping-distributed-qc.

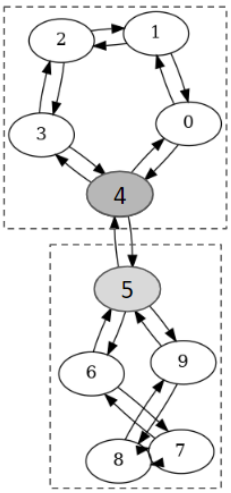
B | Coupling Graph by Group



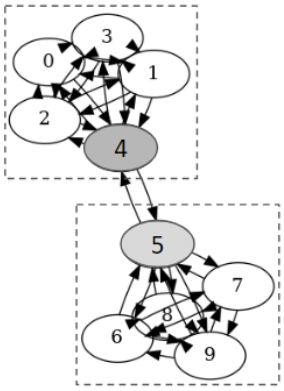
(a) Coupling graph line_10_2



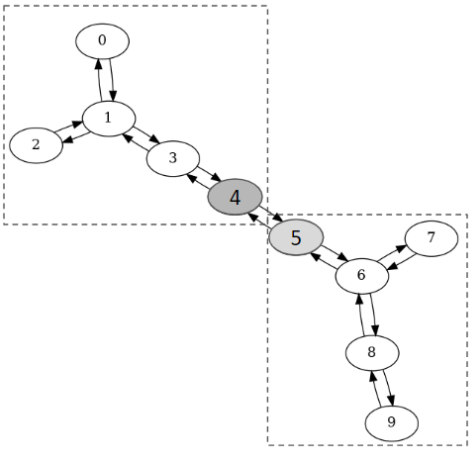
(b) Coupling graph grid_6_2



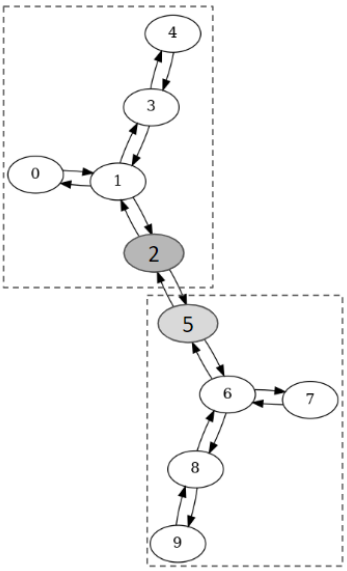
(c) Coupling graph ring_5_2



(d) Coupling graph full_5_2



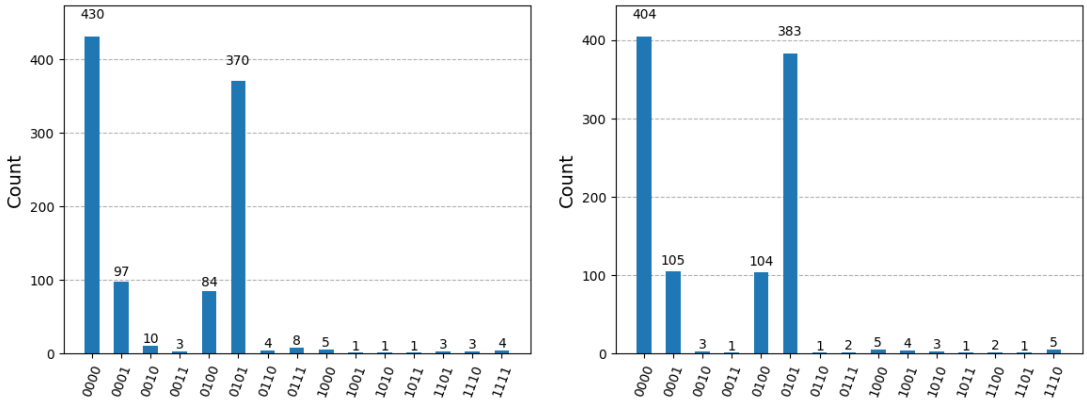
(e) Coupling graph t_horizontal_5_2



(f) Coupling graph t_vertical_5_2

Figure B.1: Generate group coupling graph

C | Verification Probability Distributions

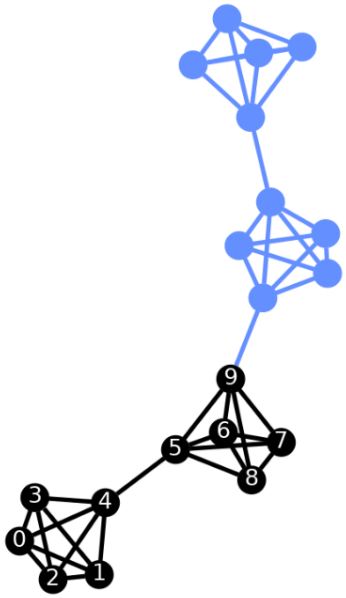


(a) Result counts from `BasicSwap` method

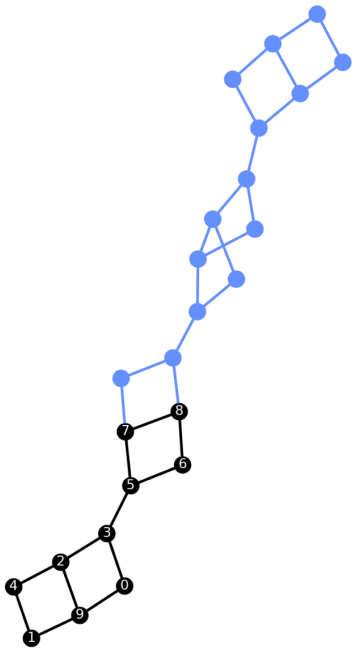
(b) Result counts of `Lookahead`

Figure C.1: Result counts from running `FakeLondonV2` backend. The counts vary due to being drawn from probability distributions, with the highest occurrences being compared.

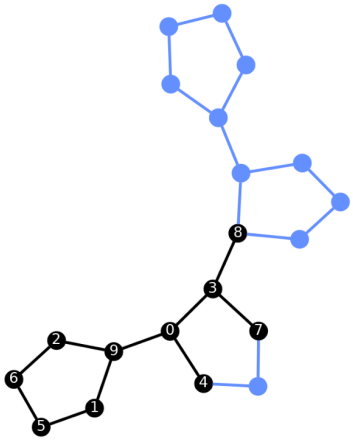
D | Qubit Mapping Illustration



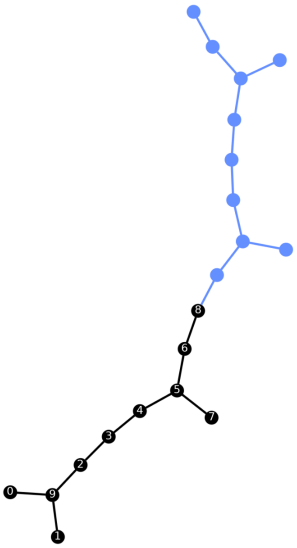
(a) full_5_4 layout



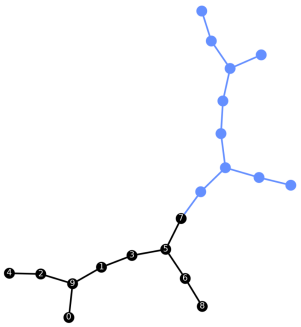
(b) grid_6_4 layout



(c) ring_5_4 layout



(d) t_horizontal_5_4 layout



(e) t_horizontal_5_4 layout

Figure D.1: Deutsch-Jozsa n = 10 in coupling map. Black nodes with numbers illustrate the location of logical qubits, and blue nodes illustrate the available physical qubits.

E | Failed Algorithms by Layout and Group

Table E.1 lists failed algorithms of 10 and 15 qubits, showing the layouts where timeout occured.

Table E.1: Failed algorithms grouped by circuit size

Algorithms	**n = 10**		**n = 15**	
	mapping timeout	swap timeout	mapping timeout	swap timeout
ghz			full_5_4	
qft		ring_7_3 ring_5_4	full_5_4	
wstate			full_5_4	
qftentangled		ring_7_3	full_5_4	ring_7_3 ring_5_4
vqe		ring_7_3 ring_5_4	full_5_4	
twolocalrandom		ring_5_4	full_5_4	full_7_3 ring_5_4
su2random		ring_5_4	full_5_4	full_7_3 ring_5_4
qnn		ring_7_3 ring_5_4	full_5_4	full_7_3 ring_7_3 ring_5_4
portfolioqaoa		ring_7_3 ring_5_4	full_5_4	full_7_3 ring_7_3 ring_5_4
random		ring_7_3 ring_5_4	full_5_4	full_7_3 ring_7_3 ring_5_4
portfoliovqe		ring_7_3 ring_5_4	full_5_4	full_7_3 ring_7_3 ring_5_4

F | Benchmark Result Table by Group

These tables present the benchmark results for various layouts running different algorithms on circuit sizes of 5, 10, and 15 qubits.

layout: layout configuration, where the first number indicates the number of qubits and the second number represents the number of groups.

benchmark: benchmark algorithms used for testing

g: total number of gates

d: circuit depth

s_B: total additional swap gates for "swap basic"

s_S: total additional swap gates for "swap sabre"

s_L: total additional swap gates for "swap lookahead"

Δs_B: gate difference between "swap basic-lookahead" (%)

Δs_S: gate difference between "swap sabre-lookahead" (%)

d_B: circuit depth for "swap basic"

d_S: circuit depth for "swap sabre"

d_L: circuit depth for "swap lookahead"

Δd_B: depth difference between "swap basic-lookahead" (%)

Δd_S: depth difference between "swap sabre-lookahead" (%)