POLITECNICO DI TORINO

Master Degree Course in Computer Engineering

Master Degree Thesis

A multi-technology hardware-aware layout synthesis library for quantum circuits compilation

Supervisors
Prof. Mariagrazia Graziano
Prof. Giovanna Turvani
Prof. Maurizio Zamboni

Candidate
Andrea Russo

July 2022

Summary

Quantum computing paradigm interest has grown very fast in recent years. The contemporary Noisy Intermediate-Scale Quantum (NISQ) devices have permitted to demonstrate that quantum computation is very promising not only from a theoretical point of view, but also from a practical one, showing the potential of this fascinating model of computation and providing the proof of concept of its feasibility.
Quantum algorithms are designed using an ideal high-level quantum circuit description without considering the main quantum hardware restrictions such as the limited set of applicable gates and limited connectivity. Quantum compilation toolchains aim to refine the original quantum circuit description, making it executable on the target hardware while optimising some desired figure of merits. This process is composed of two steps: the logic synthesis, decomposing the original circuit using the target technology native gates, and the layout synthesis, solving the coupling-constraint of the target NISQ device, due to the hardware limitations.

The target of this thesis is twofold: developing a flexible multi-technology library to perform the layout synthesis phase, and integrating it inside the template-based compilation toolchain available at the VLSI Lab of Politecnico di Torino. The library is written entirely in Python, targeting quantum circuits described using the OpenQASM 2.0 language. The supported technologies are: superconducting qubits, quantum dots (partially connected), Nuclear Magnetic Resonance (NMR) and trapped ions (fully connected).

The first chapter provides an overview of the current state of quantum computing, as well as its key concepts. The focus of this introduction is on the NISQ devices limitations, underlining the need for quantum circuits compilation. All the constraints that must be considered for the correct quantum circuit execution are highlighted for each state-of-the-art quantum technology.

In the second chapter, the focus is on quantum circuits compilation, with a detailed explanation on the tasks accomplished during the logic and layout synthesis steps. At the end of the chapter, the original VLSI compilation toolchain developed by M. Avitabile for his Master thesis dissertation is presented, highlighting its main missing component, that is, a complete layout synthesis procedure.

The third chapter provides a high-level overview of the layout synthesis library's

implementation. All the classes and their relationships are explained, demonstrating how they can be used to perform the layout synthesis task. Following that, the layout synthesis tool is introduced, which was designed to allow the general user to use all of the incorporated heuristics without any programming experience. At the end of the chapter, the final and complete VLSI compilation toolchain is presented, capable of performing the logic and layout synthesis for all the quantum technologies supported by the library.

In the fourth chapter, an in-depth explanation of the placement sub-step of the layout synthesis task is presented. It aims to map the logical qubits used for describing the quantum algorithm to the physical qubits of the NISQ device. For the placement, the three implemented strategies are introduced: a trivial technology-agnostic one, and two strategies using simulated annealing to find a potentially optimal solution. The first aims to find the sub-graph of most connected physical qubits in the target hardware, the latter exploits the quantum gates features to find an optimal hardware-aware placement.

In the fifth chapter, an in-depth explanation of the routing sub-step of the layout synthesis task is presented. It aims to ensure that each two-qubit interaction is allowed in hardware, adding swap gates to make the final circuit compliant with the target coupling-graph. After presenting how the routing step is internally implemented, the simplest hardware-unaware heuristic, the Basic Routing, is explained.

The sixth chapter focuses on the hardware-aware routing algorithms which have been implemented. These algorithms exploit the execution time and error rate of the native gates of a quantum device during the swap insertion phase. The implementation of the original hardware-aware algorithm (adapted to target all the supported technologies) is presented, and an extended version of the latter, to target fully-connected topologies, is also underlined. This adaption was required for NMR and trapped ions devices. Specifically, a modified version of the original method was devised, allowing it to shift the two-qubit interactions towards the stronger interacting nodes, to optimise the output circuit.

In the seventh chapter, the focus is on the verification of the implemented heuristics and an evaluation of the obtained results. The divergence between the discrete probability distributions of the measured eigenstates, before and after applying each implemented algorithm is computed to prove the functional equivalence between the input and output circuit of the layout synthesis procedure, and thus validate the methodology. The integrated procedures are compared with IBM's Qiskit and Cambridge Quantum Computing's t|ket⟩ compilation toolchains, exploiting the figure of merits routinely used for the comparison in the associated literature: the number of swap gates added and the execution time and fidelity of the final quantum circuit.

In the eighth and last chapter, a final overview of the implemented library, with a focus on the obtained results and future perspectives, is presented.

Contents

Part I

Introductory concepts

Chapter 1

The current state of Quantum Computing

Quantum computing is a fascinating emerging computational model that could solve problems considered **hard** for other **classical** (non-quantum) models, exploiting the peculiarities of the **quantum mechanical framework**.

A problem is considered hard if there exists no algorithm capable of solving it in an **efficient** way. Computer Science has developed the idea of **computational complexity** in order to measure how much an algorithm is efficient or inefficient (both in terms of execution time and memory occupation). In a nutshell and focusing on time complexity, if an algorithm requires an amount of time that is **polynomial** on the size of its input it is considered efficient, instead if the time required is **exponential** (or worst) it is considered inefficient.

Quantum computers **can provide a significant speed-up** which could even be exponential for solving specific computational hard problems [1]. In quantum literature, the term **"Quantum Advantage"** is used for expressing the potential of the quantum computational model over a classical one in this context. The time complexity for solving the problem using a quantum computer could even be exponentially lower, **if a quantum efficient algorithm can be found**, which is a non-trivial burden.

David Deutsch was one of the pioneers in this field, demonstrating in 1985 the feasibility of a **universal quantum computer**. Peter Shor and Lov Grover are other remarkable names whose publications demonstrated the potential of such a computational model for solving computationally hard problems with no efficient solution in the classical computation. The first for finding the prime factors of an integer [2], the latter for the development of an algorithm for searching an item in an unstructured data set [3].

This new model revealed to be perfectly suited not only for solving classes of problems with no efficient solutions in the classical domain, but also for improving

the current state-of-the-art in **cryptographic** algorithms [4, A.5] and in **quantum systems simulation** [4, A.2], exploiting the exponential speed-up to perform simulations beyond the current classical limits.

The main motivations behind the study of this powerful research field were only briefly synthesised here, following [5, Ch. 1], which is a suggested reading for the history and development of quantum computation and quantum information. Moreover, a general understanding of the fundamental ideas of **quantum computation** and of the **quantum circuit model of computation** are required before proceeding to the following sections and chapters. Indeed, [4] is a good introduction to this previously cited concepts.

1.1 Logical vs physical qubit

Before proceeding, it is essential to formalise the qubit definition in order to avoid any future ambiguity.

Definition 1.1.1. A **quantum bit**, or **qubit**, is the unit of quantum information. It is the quantum counterpart of the classical bit. [4, Sec. 5.1]

A qubit is the **mathematical representation** of a two-dimensional quantum system. The state of a qubit is described following the quantum mechanics superposition principle as the linear combination of the eigenstates that it can assume after measurement (**basis states**), which corresponds to the possible states of classical bits. Following the **Dirac notation** [5, Sec. 2.1], the qubit basis state can be written as a column vector of two elements:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \tag{1.1}$$

Consequently, the qubit state can be written as:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,, \tag{1.2}$$

where:

- $|\psi\rangle$ is the state of the qubit, that is, the state of the quantum system.
- α and β are complex numbers such that: $|\alpha|^2 + |\beta|^2 = 1$.

Measuring a qubit makes it **collapse** in the basis state $|0\rangle$ with probability $|\alpha|^2$ or in the basis state $|1\rangle$ with probability $|\beta|^2$.

Quantum algorithms are mainly described using the circuit model of quantum computation, composed of **unitary transformations called quantum gates**. These gates can act on a single qubit (**single-qubit gates**), on two qubits (**two-qubit gates**) or more (**multi-qubit gates**). All the gates described in a quantum

circuit cause the evolution of qubits from an initial state to a final one (most likely to be measured). **These qubits and gates used in a quantum circuit are abstract entities**. They are required only for specifying the logic, that is, the functionality of a quantum algorithm.
Being relieved from the burden of the physical details when writing an algorithm is essential in the quantum computing world in the same way as it is in the classical realm. Indeed, a programmer that is writing software for a classical computer does not care how, physically, a bit is implemented. These entities are **logical (abstract)**.
Algorithms are abstract, but computers, both classical and quantum, are real entities and exactly as classical bits have their physical implementation, qubits have also their physical counterpart (the **physical qubits**).

Formalizing these concepts:

Definition 1.1.2. Logical qubits are the abstract qubits that evolve in time following a quantum algorithm [6, Sec. 2.3], [7].

Definition 1.1.3. Physical qubits or **nodes** are physical two-state quantum systems that can encode a logical qubit [6, Sec. 2.4], [7].

In the following of this thesis, if the term "qubit" alone appears in the presented work, then it must be intended as "logical qubit". For the physical qubits it is always used the explicit terminology or the "node" term.

1.2 The NISQ era of quantum computation

Different research activities are focused on the physical implementation of quantum computation using different technologies for building a quantum computing device. The feasibility of a quantum information processing hardware, regardless of the selected technology, depends on the notorious **DiVincenzo's criteria** [8]: which is, a set of requirements that all the quantum computers must respect to work correctly.

The current era of quantum computing technology is known as **NISQ: Noisy Intermediate-Scale Quantum** [9]. This term perfectly describes the achievements accomplished and the current challenges for contemporary quantum computers. The *intermediate scale* term refers to the **number of qubits** available on current devices (a few hundred at most). This number is too low for employing any error correction code, but still, NISQ devices have the great responsibility of **demonstrating the computational potential** of quantum information and computation [10, Sec. 1].
The *noisy* term is linked to the non-ideal phenomena, which affects qubit itself, the effect of quantum gates and measurement operations and **limits the computational capabilities** of these devices. Indeed, due to these noise effects, for today's

quantum computers it is **not possible to execute long quantum circuits**, and thus implement complex quantum algorithms effectively exploiting the exponential speed-up. Specifically, due to the impossibility of isolating a quantum system from the external environment, the limitation in qubit control and manipulation, and intrinsic device limits, there is an inevitable loss of information. One of the most important effects is known as **decoherence** [4, Ch. 11] and it reduces the **fidelity** [5, Ch. 9] of the quantum circuit, which is a fiducial scalar quantity for the estimation of the distance between the qubit noisy state and the corresponding ideal one. The best approach to limit this problem is to employ quantum gates with a **low execution time**, to make the decoherence effects negligible.

Several technologies have been proposed to implement a **NISQ device** satisfying all DiVincenzo's criteria, each with its advantages and disadvantages. The most used quantum technologies, at the time of writing, are **superconducting**, **quantum dots**, **nuclear magnetic resonance (NMR)** and **trapped ions**. The following Sections 1.4 to 1.7 intend to give to the reader a general overview of the main features of these technologies. These sections focus on the limitations of the NISQ devices more than on the physical theory behind logical qubits representation, implementation of unitary transformations and measurement operations.

1.3 NISQ devices main limitations

In the previous section, the characteristics of the current quantum computers (the **NISQ devices**) were presented, underlining their limitations in terms of computational capabilities. Besides this, there are **further main constraints** that must be considered when a quantum circuit is executed on a real device. Understanding these requirements is necessary to justify the need for quantum circuits compilation, on which this thesis is focused.

First, not all quantum gates can be executed on any NISQ device. Each technology has its own set of **native gates** that forms a **universal quantum gates set**. This is labelled as *universal*, to specify that every unitary transformation can be implemented with a combination of the gates forming the set [4, Sec. 5.4]. Moreover, this native gates set usually consists of only single-qubit and two-qubit gates, thus **the multi-qubit gates cannot be directly executed in hardware**.

Another important limitation to understand, fundamental for the presented work, is the coupling-constraint (also called connectivity-constraint). Indeed, the whole purpose of this thesis is the development of a library for obtaining the layout of a quantum circuit for the execution on a real device by solving these connectivity requirements (presented in details in Chapters 3 to 5).

Definition 1.3.1. For the contemporary NISQ devices, not all the interactions are allowed in hardware. The set of allowed two-qubit interactions form the **coupling-constraint** (also called **connectivity-constraint**) [11, Sec. 2] [12, Ch. 15].

These coupling-constraint are usually mathematically represented by using the so called coupling-graph.

Definition 1.3.2. The **coupling-graph** is a graph ($G = (V, E)$) used for representing the coupling-constraint [12, Sec. 2.2]. The vertices V of this graph are the physical qubits of the quantum computing device. The edges E represent the allowed two-qubit interactions. Indeed, a two-qubit gate can be applied to two generic nodes n_i and n_j if and only if $(n_i, n_j) \in E$.

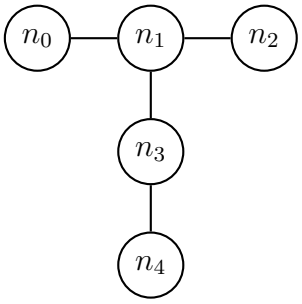

Figure 1.1. Graph representing the coupling-constraint of a NISQ device. The black circles and lines show the coupling-graph of the backend (modelling the *ibmq_lima* [13] superconducting device topology).

1.4 Superconducting devices

Superconducting is one of the most used quantum technology for the implementation of a quantum computing device. Indeed, it is adopted by the notorious **IBM Quantum systems** [14], allowing researchers from all around the globe to run quantum circuits on their machines through a cloud-based connection. Because of this, all the superconducting devices described in this thesis are related to the IBM's devices, which are the most used in the quantum literature. In this technology, a logical qubit is encoded as the state of an anharmonic oscillator, built using an LC circuit, where the non-linear inductor L is realised by using a **Josephon Junction**. All the information reported in this section is based on [15, Sec. 1.3.4] and [1], which are a suggested reading for further details on the superconducting implementation of quantum computation.

1.4.1 Native gates set

Since the target devices analysed in this work are the IBM's ones, the native single-qubit gates set is composed by IBM's U_1, U_2 and U_3 gates.
Specifically:

- the $U_1(\lambda)$ gate implements a R_z gate;
- the $U_2(\phi, \lambda)$ gate implements a R_x or R_y gate;
- the $U_3(\theta, \phi, \lambda)$, also called the U gate, can implement any single-qubit rotations.

In this technology, the native two-qubit gate, in the compilation procedure, is the **CX** gate.

1.4.2 Connectivity

Superconducting devices are typically **non-fully-connected**. Moreover, this connectivity is **nearest-neighbour**, meaning that only adjacent physical qubits are allowed to interact. Therefore, the complete layout synthesis procedure, composed of placement and routing, **is mandatory** for this technology.

1.5 NMR devices

Nuclear magnetic resonance (NMR) quantum computers are a type of spin $\frac{1}{2}$ qubit quantum device. The logical qubits are encoded by using the nuclei's energy levels of a liquid-state molecule, which depend on the nuclei spin (due to the Zeeman effect, after the application of a static magnetic field). It is also possible to perform quantum computation by employing a solid-state molecule, but this implementation was not considered in this thesis. Therefore, for the proposed work, when referring to the NMR technology the first family of quantum hardware is implicitly intended. All the information reported in this section is based on [15, Sec. 1.3.2] and [16], which are a suggested reading for further details on the NMR technology.

1.5.1 Native gates set

The single-qubit unitary transformations can be implemented by using the notorious: R_x, R_y and R_z gates.

The native two-qubit gate is the R_{zz} gate, constructed exploiting the **J-coupling** interaction of the nuclei composing the molecule. In particular, this spin-spin coupling is providing information on the interaction strength: the higher absolute value the J-coupling, the lower are the gate error rate and execution time.
Having the R_{zz} gate as the native two-qubit interaction, the CZ gate can be easily constructed following the decomposition shown in Figure 6.6.

1.5.2 Connectivity

NMR devices are typically **fully-connected**. This is because every couple of molecule's nuclei have a J-coupling constant which can be exploited for implementing a two-qubit interaction. However, for some pairs of physical qubits, the coupling strength might be so low in absolute value that it becomes impractical to consider such interaction as allowed. Therefore, **only the placement operation is mandatory** for this technology. However, the routing task might still be useful for **optimising** the final quantum circuit's main figure of merits.

1.6 Quantum dots devices

In the quantum literature, a lot of attention is given to quantum dots, which is a good candidate technology for the implementation of quantum computers. Quantum dots are a semiconductor technology based on confinement of electrons. Logical qubits can then be encoded in different ways: into the spin of the single trapped electron or using two communicating quantum dots, and, in this case, the state of the qubit depends on where the electron is confined. All the information reported in this section is based on [17], which is a suggested reading for further details on the quantum dots technology.

1.6.1 Native gates set

The native single-qubit gates used for quantum dot technology are the same used in NMR devices: R_x, R_y and R_z gates.

The native two-qubit gate is the R_{zz} gate, constructed exploiting the **Exchange-Interaction J**. In particular, this spin-spin coupling provides information on the interaction strength: the higher absolute value the J, the lower are the gate error rate and execution time.
Having the R_{zz} gate as the native two-qubit interaction, the CZ gate can be easily constructed following the decomposition shown in Figure 6.6.

1.6.2 Connectivity

Quantum dots devices are typically **non-fully-connected**. Moreover, currently they present a linear-topology, which means that the coupling-graph is a linear chain. Therefore, the complete layout synthesis procedure, composed of placement and routing, **is mandatory** for this technology.

1.7 Trapped ions devices

Trapped ions or **ion trap**, like superconducting, is one of the most used quantum technology for the implementation of a quantum computing device. The key idea is to encode a logical qubit by using the energy state of an ion (usually Ca^+ or Yb^+) confined in an electromagnetic trap forming a linear chain: the **Paul trap**. Scalability is the main limitation of these devices. Increasing the number of ions inside the trap makes it more difficult to physically apply quantum gates, and thus increases the error rate. To scale this technology, the most promising implementation, at the time of writing, is using a **Quantum Charge Coupled Device (QCCD)** [18]. This is a distributed and interconnected multi-trap system. In this way, the number of ions in each trap is kept limited, and whenever two separated ions must be coupled, an operation called **ion shuttling** is performed, moving the ion from one trap to another one. In this thesis work, only single linear trap architecture are considered. All the information reported in this section is based on [15, Sec. 1.3.3] and [19], which are a suggested reading for further details on the trapped ions implementation of quantum computation.

1.7.1 Native gates set

All the single-qubit unitary transformations are performed using the **generic rotation gate** $R(\theta, \phi)$. However, due to compatibility reasons, for the proposed work, also the more typical R_x, R_y and R_z are considered native gates for this technology.

The native two-qubit gate is the **Mølmer–Sørensen gate (MS gate)** which implements the $R_{xx}(\chi)$ [20], where the parameter χ depends on the interacting pair of ions. Using the MS gate, it is possible to easily implement the **CX** gate, according to the decomposition presented in Figure 6.14. To correctly implement the CX gate, the parameter χ must be fixed to $\pm\frac{\pi}{4}$, where the sign of the rotation depends on the pair of ions that interact.

1.7.2 Connectivity

Trapped ions devices are typically **fully-connected**. Indeed, all the ions in the linear trap can interact. However, there are some implementations in which the two-qubit gate error rate and execution time are proportional to the distance of the involved ions [18]. Therefore, **only the placement operation is mandatory** for this technology, however the routing task might still be useful for **optimising** the final quantum circuit's main figure of merits.

Chapter 2

Quantum circuits compilation

In the **quantum computing circuit model**, quantum algorithms are designed as a combination, in series and in parallel, of **unitary transformations** called **quantum gates** [4, Ch. 5]. Typically, such circuits are designed having **only the quantum algorithm functionality** as the main concern [21, Sec. 1]. This model is extremely useful, allowing the design of a quantum program without having to consider the actual **implementation of each desired transformation** in hardware, with their limitations. Therefore, not only does it **simplify** and **speed up** the development process, but it is also **independent of the quantum technology adopted**.

However, this description assumes an **ideal quantum computing device** that does not exist in reality. As already stated, the contemporary NISQ devices have many limitations (see Section 1.3), all of which must be taken in consideration for the correct quantum algorithm execution. A **quantum circuits compilation toolchain** (or simply **quantum compiler**) aims to **translate** this quantum circuit description in an equivalent circuit compatible with the chosen quantum device by adding all the necessary low-level details for its correct execution.
It is essential to underline that not only a quantum compiler must make executable an ideal quantum circuit, but it must also **maximize its fidelity** and **minimize its execution time**, remembering that NISQ devices are indeed "noisy" [21, Sec. 1].

2.1 Quantum circuits compilation process

As introduced, quantum circuits are usually **designed in an abstract way**, describing only the logic (that is, the functionality) of the quantum algorithm. Just

as high-level software must be refined before it can be executed on a specific classical computer, the same must be done also for quantum programs (modelled as quantum circuits).
In the quantum literature there are **different names** used for referring to this refinement task (transpilation [22], compilation [21], qubit mapping problem [23], etc.). Indeed, it is essential to clarify what is the terminology used for the presented work, to avoid ambiguity.

The **complete refinement task**, taking an abstract quantum circuit description (all gates and interactions allowed) as input and producing a transformed circuit (executable on a specific quantum computing device) as output, is called **quantum circuits compilation**. This process is composed of two essential steps: the **logic synthesis** and the **layout synthesis**.

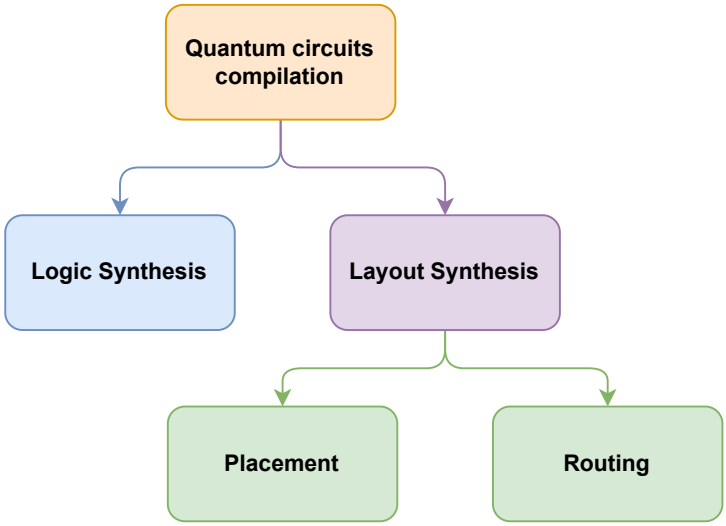

Figure 2.1. Quantum circuits compilation structure.

2.1.1 Logic synthesis

The first step of quantum circuits compilation is the **logic synthesis**. It is called **logic** because it works with and produces a logic (abstract) quantum circuit description. The NISQ devices limitations solved in this step are related to the l**imited native gates set** [7, Sec. 1].
First, **all the multi-qubit quantum gates are decomposed** by using only single-qubit and two-qubit gates. Subsequently, **all the quantum gates are translated** to obtain a quantum circuit composed of only the limited native gates of a specific quantum technology. Moreover, not only must this be accomplished,

but the logic synthesis usually employs **heavy transformations** in order to optimise the final quantum circuits in terms of some desired figure of merits (circuit depth, execution time, fidelity).

Usually, the two-qubit interactions are **not decomposed** in the native coupling gates. It is common to leave the CX or/and CZ gates untouched after the logic synthesis process to ease the following task performed during the layout synthesis [15, Sec. 1.4].

2.1.2 Layout synthesis

The second fundamental step performed during the compilation of a quantum circuit is the **layout synthesis**, and it is the main focus of the presented thesis work. During this process, the logic quantum circuit description is converted into a **physical one**, executable on a real device. This task is divided into two sub-steps: the **placement** and the **routing**.

The placement task has the important job of **mapping** every logical qubit to a specific physical qubit. An in-depth explanation of this procedure is given in Chapter 4. The input of the placement is the output of the logic synthesis, which is, a logical (or abstract) quantum circuit description. Indeed, all the qubits used in this description are logical (see Section 1.1). These qubits are not the physical qubits of a NISQ device, but are only used for describing the functionality of the quantum circuit.
Even if the logical synthesis is completed and the quantum circuit is placed, this cannot yet be performed on the real hardware, as the connectivity limits of the device have not yet be considered. Indeed, as already stated in Section 1.3, NISQ devices have a set of coupling-constraint, specifying the allowed two-qubit interactions. The routing task has the important role of solving these connectivity-constraint, modifying the input quantum circuit. For a more in-depth explanation of the routing procedure, see Chapter 5.

Summarising, the target of the layout synthesis procedure is the production of the **spacetime coordinates** (t_j, x_j) for each gate j composing the quantum circuit [7, Sec. 2]. They indicate **when** and **where** to apply each specific transformations, meaning in which order and to which nodes apply each gate, to correctly implement the desired quantum algorithm (thus, having all the transformations allowed in hardware).

2.2 The VLSI compilation toolchain

At the *VLSI Lab* of *Politecnico di Torino*, a quantum circuits compilation toolchain was developed by *Manfredi Avitabile* in 2021, as a work for his Master thesis dissertation [15]. The Avitabile's toolchain is completely written using the **Python** language, and can target quantum circuits designed using **IBM's OpenQASM**

2.0 language [24], which is the standard description language used for modelling quantum circuits. This compiler can perform the logic and layout synthesis targetting the most used NISQ devices quantum technologies: **superconducting**, **NMR** and **trapped ions**.

Instead of using **exact algorithms** or **heuristics/metaheuristics** to perform the compilation task (remembering that optimising the final circuit is essential in quantum circuits compilation), the VLSI toolchain uses a novel approach: **the template-based approach**. The idea is to find **circuit patterns** inside the input quantum circuit description and substitute this patterns using some predefined **templates** (that is, **circuital identities**), aiming to optimise the output quantum circuit.

2.2.1 VLSI toolchain - structure

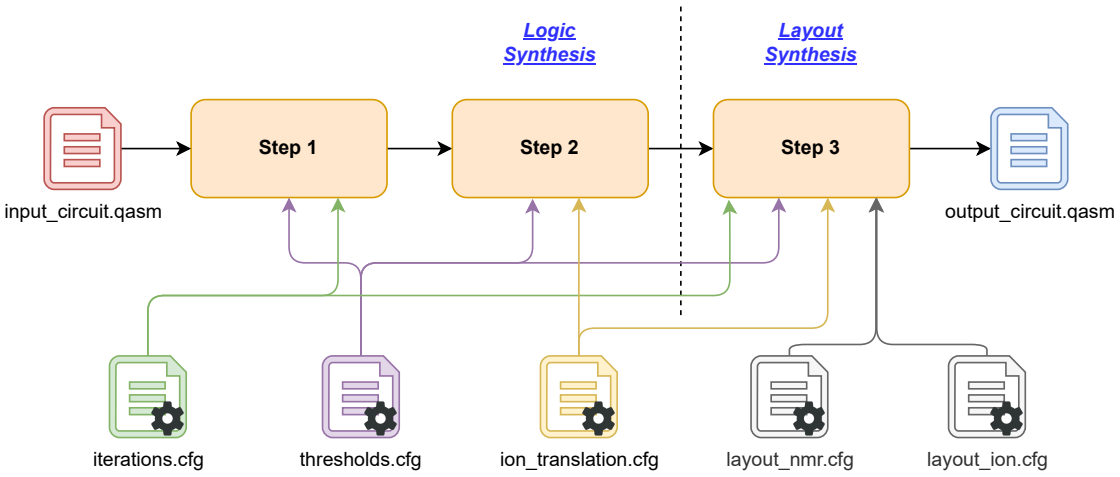

Figure 2.2. Original VLSI toolchain structure.

The VLSI toolchain is structured in **three different steps**, as shown in Figure 2.2. Step 1 and Step 2 compose the logic synthesis block of quantum circuits compilation, while the Step 3 implements the layout synthesis. Step 1 takes the original abstract quantum circuit description as input, while the Step 2 and Step 3 takes the circuit produced by the preceding step as input. Moreover, to correctly perform its task, the toolchain exploits some **configuration files (.cfg)**. They contain some mandatory but configurable settings used by the compiler during the compilation process.

2.2.2 Step 1

Step 1 is the first step which composes the logic synthesis block of the VLSI toolchain. The main aim of this step is to apply some **technology-agnostic templates** in order to increase as much as possible the number of circuital null operations, which can then be discarded. Moreover, it also tries to increase the number of R_z gates, in order to optimise the final quantum circuit when these gates are **implemented virtually** (with zero execution time and error rate) [15, Sec. 1.2.5]. This step supports only the quantum gates composing the **extended Clifford + T gate set** [25] and all the IBM's native gates: U_1, U_2 and U_3 (no optimisation is performed on these latter gates at this step). The output of Step 1 is a quantum circuit still described using the OpenQASM 2.0 language, but compacted and composed of only $R_x, R_y, R_z, U_1, U_2, U_3, CX, CZ$ gates.

An in-depth explanation of the Step 1 of the VLSI toolchain can be found in [15, Ch. 2]

2.2.3 Step 2

The **Step 2** is the final step of to the logic synthesis of the VLSI quantum circuits compilation toolchain. At this step, all the goals of the logic synthesis (presented in Section 2.1.1) must be fulfilled. Indeed, this step applies some **technology-specific** translation and optimisation templates, producing a technology-compliant final quantum circuit description. Specifically, the CZ gates are translated to CX gates for the superconducting and trapped ions technology. Moreover, the input quantum circuit is optimized using a powerful compaction section called the **Eulercombo**, to compact as much as possible the number of single-qubit gates.

In order to work correctly, this step requires that the input quantum circuit is composed only of $R_x, R_y, R_z, U_1, U_2, U_3, CX, CZ$ gates. Moreover, each single-qubit gate must not be adjacent to a single-qubit gate of the same type. This requirement is always fulfilled if the input of Step 2 is the output of Step 1.
The output basis gates for each supported technology are:

- **NMR:** $\{R_x, R_y, R_z, CX, CZ\}$. Even if the CZ gates can be easily constructed using the native coupling gate, the CX gates are not decomposed in CZ yet, since for the first ones stronger templates are available in the following Step 3.

- **Trapped ions:** $\{R_x, R_y, R_z, CX\}$ or $\{R(\theta, \phi), R_z, CX\}$, depending on the ***Iontran*** boolean parameter inside the *ion_translation.cfg* configuration file. $R(\theta, \phi)$ is the native rotation gate for this technology, but the classic R_x, R_y, R_z gates are selectable for compatibility reasons.

- **Superconducting:** $\{U_1, U_2, U_3, CX\}$.

It is essential to remark that the CX gates for trapped ions and the CZ gates for NMR **are not natively supported by these technologies**. However, their decomposition into native gates is postponed in order to apply powerful templates during the Step 3 and also simplify the layout synthesis procedure (as anticipated in Section 2.1.2).

An in-depth explanation of the Step 2 of the VLSI toolchain can be found in [15, Ch. 3]

2.2.4 Step 3

Step 3 is the last step of the VLSI quantum circuits compilation toolchain and is related to the layout synthesis task. It must be underlined, that actually the main problems related to the layout synthesis, namely placement and routing (see Section 2.1.2), **are not solved during this step**, as a strong assumption is made: **all the target technologies are fully-connected**. Indeed, all the tasks performed in this step are related to the final quantum circuit optimisation, and to the translation of all the two-qubit gates by using the supported native gates set. The operations performed during this final step are:

1. Apply strong CX optimisations templates, to decrease as much as possible these gates inside the final quantum circuit.

2. Translate all the two-qubit gates by using the native coupling gate of each target technology. For the decomposition, some parameters and a simplified layout (fully-connected) is required for each technology. Indeed, this information is contained inside the *layout_nmr.cfg* and the *layout_ion.cfg* configuration files.

3. Apply for the last time the Eulercombo section, aiming at performing further single-qubit gates compaction.

In order to work correctly, this step requires that the input quantum circuit is composed only of:

- **NMR:** $\{R_x, R_y, R_z, CX, CZ\}$.
- **Trapped ions:** $\{R_x, R_y, R_z, CX\}$ or $\{R(\theta, \phi), R_z, CX\}$, depending on the **Iontran** boolean paramter inside the *ion_translation.cfg* configuration file.
- **Superconducting:** $\{U_1, U_2, U_3, CX\}$.

Moreover, each single-qubit gate must not be adjacent to a single-qubit gate of the same kind. This requirement is always fulfilled if the input of Step 3 is the output of Step 2.

The output basis gates for each supported technology are the same of Step 2, with the possibility of decomposing the CX and CZ gates using the technology

native two-qubit interaction gate. Specifically, this translation is configurable with the ***ioncxtrans*** and ***cztranslat*** parameters inside respectively the *layout_ion* and *layout_nmr* configuration files.

An in-depth explanation of the Step 3 of the VLSI toolchain can be found in [15, Ch. 4]

Part II

Implementation of the layout synthesis library and tool

Chapter 3

The layout synthesis library and tool

The target of this thesis is the development of a **flexible layout synthesis library** that can be easily integrated into any quantum circuits compilation toolchain. The entire library was written by using the **Python** language, which was selected for its simplicity and suitability for projects with a concern for an easy expansion. Indeed, the majority of the quantum compilation frameworks employ this language and by adopting it, the interfacing with these famous tools is simplified. For example, Python is also the selected language for the IBM's consolidated Qiskit [26] compiler. The proposed library has a quantum circuits described by using **IBM's OpenQASM 2.0** language [24], which is one of the most used circuit description languages in the quantum literature, as input. The proposed library can perform the refinement of the circuit for running it on a target **NISQ devices**, described by a configuration file, belonging the most used quantum technology in the state-of-the-art: **superconducting**, **quantum dots**, **NMR** and **trapped ions**. The variety of the supported technologies demonstrates the versatility and completeness of the presented library, which can manage different kinds of topologies (non-fully-connected, fully-connected).
A general overview of the Python classes which compose the library is provided in Section 3.1. More details on the implementation of the placement and routing procedures are given in respectively Chapter 4 and Chapter 5. The proposed work was **integrated into the current VLSI toolchain** (presented in Section 2.2), substituting the incomplete layout synthesis phase of it. The modularity of the developed library makes comfortable this task, and would make easy also the integration in other compilers. The result of this integration was a complete quantum circuits compiler, presented in Section 3.2.

Additionally, a **tool was developed**, allowing the common user (without any programming knowledge) to exploit the implemented layout synthesis heuristics

for solving the connectivity limitations of today's NISQ devices. Specifically, two command-line applications were developed in Python, giving a textual user interface for interacting with the library Application Programming Interface (API). The tools are presented in Section 4.4.1 and Section 5.3.1.

3.1 The layout synthesis library

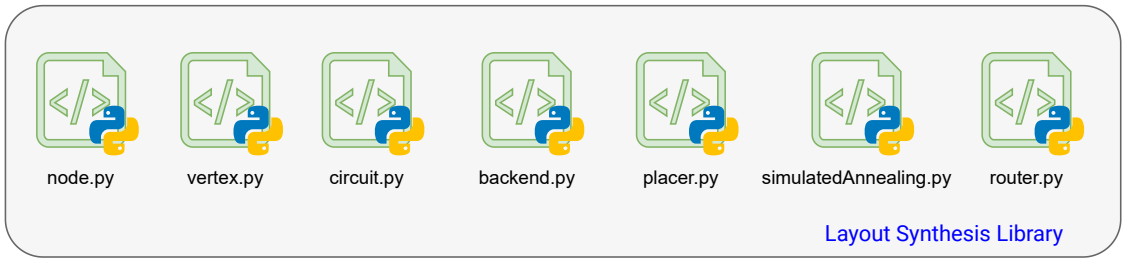

Figure 3.1. Representation of the python scripts composing the implemented layout synthesis library.

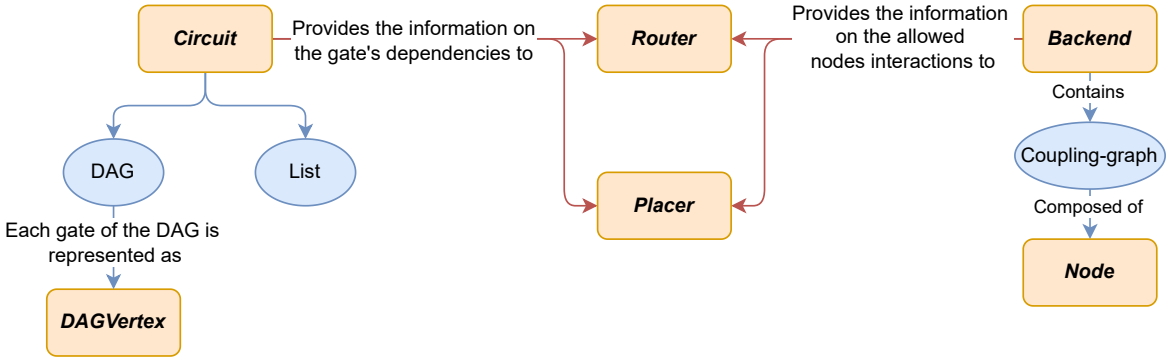

Figure 3.2. Layout synthesis library, classes relations.

The developed library is composed of a set of Python classes that form a **complete framework** for solving the placement and routing problems of quantum circuits compilation. This structure is inspired by the Qiskit quantum compiler [26], offering different classes that can be used to model a NISQ device, to describe a quantum circuit and to solve the connectivity-constraint. The relationships among these classes, which underline how everything joins to perform the layout synthesis, are shown in Figure 3.2.

Specifically, the following classes are implemented:

***Circuit*:** represents a quantum circuit. The main attributes of this class are a **directed acyclic graph (DAG)**, used for storing the quantum gates dependencies, and a **list**, for storing the measures operations. More details are presented in Section 3.1.1.

DAGVertex*:** represents a generic vertex in the directed acyclic graph (DAG) representation of the quantum circuit. There are two kinds of *DAGVertex*: ***GateDAGVertex and ***MeasureDAGVertex***. The first is used to represent a quantum gate, the latter to represent a measure operation. In particular, the *MeasureDAGVertex* is added for completeness, but it is not currently employed since the measure operations are stored in a separate data structure (a list). Currently, it is possible to model only single-qubit and two-qubit gates, as explained in Section 3.1.1.

***Backend*:** represents a generic NISQ device, target of the layout synthesis process. More details are presented in Section 6.1

***Node*:** represents a physical qubit of a NISQ device. At the time of writing, only the node's identifier (ID) is necessary to completely describe a physical qubit, but, having this class available, future improvements can be easily incorporated.

***Placer*:** takes care of performing the placement step for a quantum circuit, having a specific NISQ device as the target of the placement process. More details are presented in Section 4.2.

***Router*:** takes care of performing the routing step for a quantum circuit, having a specific NISQ device as the target of the routing process. More details are presented in Section 5.2.1.

3.1.1 The Circuit class

The ***Circuit*** class, written inside the *circuit.py* Python script of the library, models the quantum circuit under compilation.
In order to represent a quantum circuit, this class employs two essential data structures:

- A **directed acyclic graph (DAG)**, where each **vertex** of it represents a quantum gate (single-qubit or two-qubit), while each **edge** represents a dependency among two gates. In particular, each vertex is dependent on all the other vertices that are pointing to it.

- A **list**, which is used to store all the measure operations in a separate data structure. In this way, the layout synthesis library can focus only on the quantum gates during the placement and routing phases, and this ensures that during the reconstruction of the final quantum circuit OpenQASM 2.0 description, the measure operations are placed at the end.

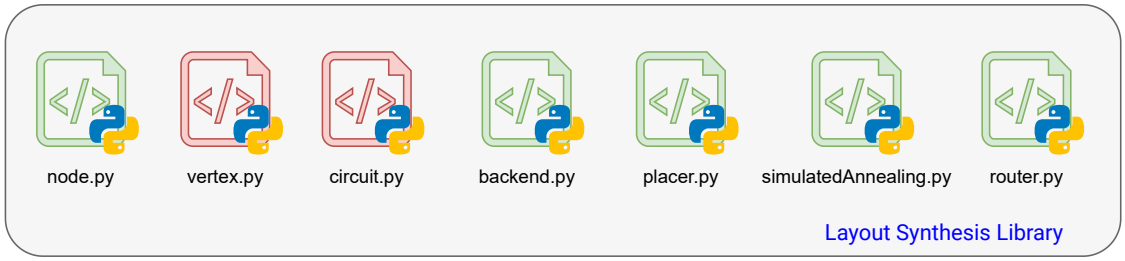

Figure 3.3. Representation of the python scripts composing the implemented layout synthesis library. The ones containing the *Circuit* and *DAGVertex* classes are highlighted in red.

It is possible to construct a quantum circuit manually, exploiting the methods offered by the *Circuit* class to append quantum gates, like in Qiskit and t|ket⟩. On the other hand, it is also possible to instantiate a circuit **automatically parsing** an IBM's OpenQASM 2.0 description file. However, there are some **constraints** that this quantum circuit description must respect to be compatible with the library. Specifically, every description must begin with the following lines:

```
1 OPENQASM 2.0;
2 include "qelib1.inc";
3 qreg q[n];
4 creg c[m];
```

This implies that the circuit can have **one and only one** quantum register of size n and **one and only one** classical register of size m. It is mandatory that these registers are named respectively "q" and "c".
Only single-qubit and two-qubit gates are currently supported. This is common for layout synthesis tools since the multi-qubit gates are decomposed during the logic synthesis step (see Section 2.1.1).
Theoretically, every single-qubit and two-qubit quantum gate can be used in this circuit description. In practice however, the presented library was tested only with the $R_x, R_y, R_z, U_1, U_2, U_3, CZ, CX$ gates, thus the stability of the library in other scenarios is unpredictable. Even if the tested gates set might seem limited, it actually incorporates all the expected scenarios, remembering that usually the

layout synthesis procedure is performed after the logic synthesis and before the two-qubit interactions are decomposed in the native two-qubit gates. The measures operations must be placed at the end of the quantum circuit. This implies that no transformations can be performed on a qubit after it is measured.

Example 3.1.1. Example showing the DAG representation of a quantum circuit.

Figures 3.4 and 3.5 depict respectively the OpenQASM 2.0 and the graphical representation of the same quantum circuit. The proposed library internally stores the quantum gates dependencies using a DAG. Figure 3.6 shows the DAG representation for modelling the same circuit.
In this figure, it is noticeable that each vertex composing the DAG is grouped in numbered sets, called **layers**. Each layer is a set of quantum gates that can be executed in parallel since they have no dependencies on one another. The first layer (*Layer 1*) contains all the quantum gates having no dependencies at all. The second layer (*Layer 2*) contains all the gates having dependencies to at least one gate of layer 1, and so on for the further layers. Slicing a quantum circuit into layers is extremely useful to simplify the placement and routing tasks, in order to reconstruct a quantum circuit **respecting the original gates dependencies**.

```
1  OPENQASM 2.0;
2  include "qelib1.inc";
3
4  qreg q[4];
5  creg c[4];
6
7  h q[0];
8  x q[1];
9  cz q[0], q[1];
10 cx q[1], q[2];
11 y q[2];
12 x q[3];
13
14 measure q->c;
```

Figure 3.4. Example of a quantum circuit represented in the OpenQASM 2.0 language.

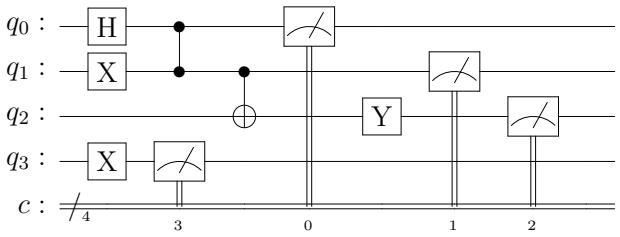

Figure 3.5. Graphical representation for the quantum circuit depicted in Figure 3.4.

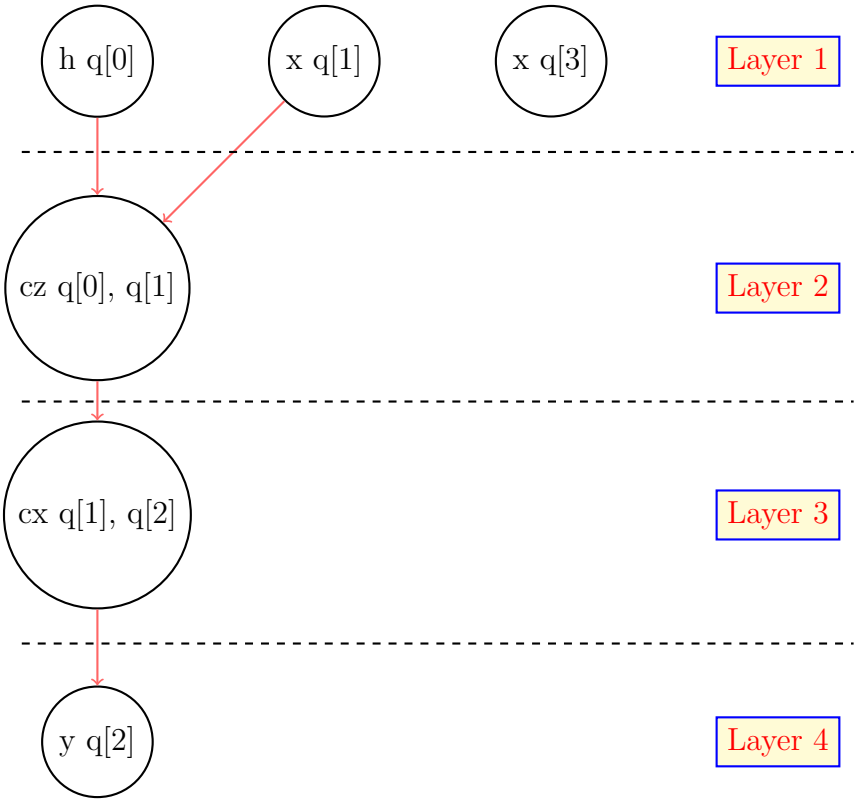

Figure 3.6. DAG representing the dependencies among the gates of the quantum circuit depicted in Figures 3.4 and 3.5.

3.2 The complete VLSI compilation toolchain

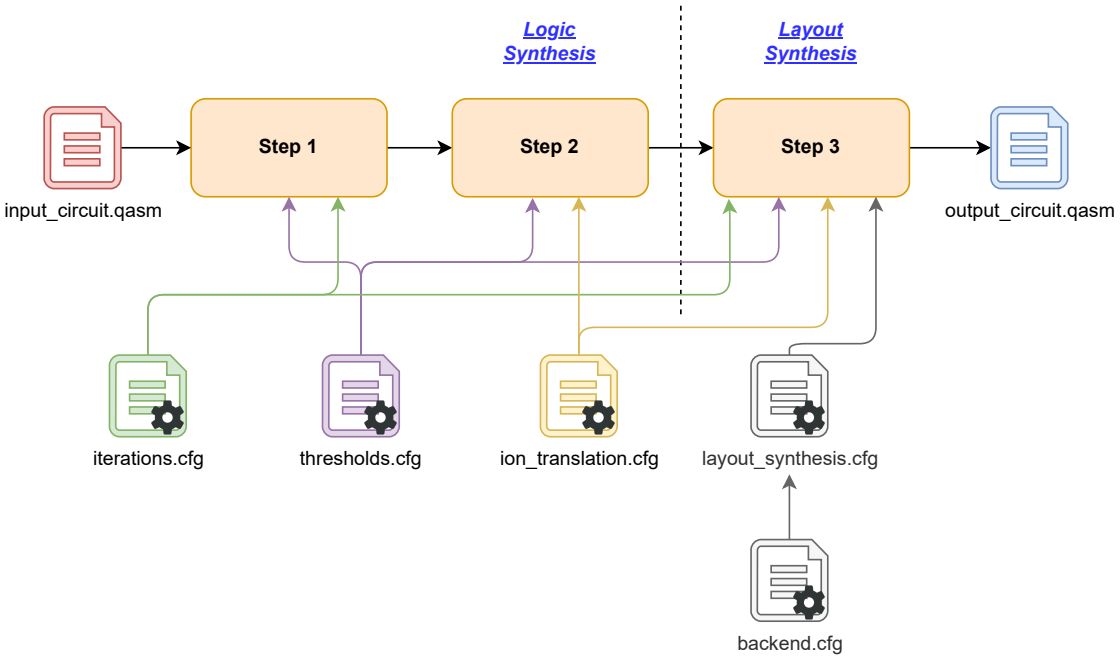

Figure 3.7. Complete VLSI toolchain structure.

The original VLSI quantum circuits compilation toolchain (presented in Section 2.2) was incomplete. Specifically, only the logic synthesis was implemented, and thus the placement and routing steps were not performed during the compilation. Step 3 of the toolchain performs strong two-qubit gates compactions, but a full layout synthesis is still mandatory for building a complete quantum compiler. After the development of the proposed layout synthesis library was completed, it was integrated into the VLSI toolchain. In particular, **two main improvements** are performed:

1. The VLSI toolchain is expanded to support also the quantum dots NISQ technology. This expansion was performed thanks to the similarities between the NMR and quantum dots native gates set [17, 16], as explained in Section 3.2.1.

2. The VLSI toolchain Step 3 is completed, to perform the layout synthesis phase. It now implements both the placement and the routing having a specific backend as target NISQ device. This integration is presented in Section 3.2.2.

3.2.1 Quantum dots technology integration

The original toolchain is intended to support only **superconducting**, **NMR** and **trapped ions** NISQ devices. Focusing on the quantum technologies limitations (see Section 1.3), NMR is very similar to another technology used for encoding a qubit: **quantum dots** (Section 1.6). Specifically, these technologies have the same native gates set and thus the same native coupling gate.

Thanks to these similarities, and because the proposed library can also target quantum dots devices, this family of quantum computing hardware was incorporated inside the VLSI compilation toolchain. In particular, the following improvements to the original compiler are performed:

- For **Step 1**, no actions are required. Indeed, this step is technology-agnostic.
- For **Step 2**, the same procedures employed for the NMR technology are now also used for quantum dots. This step is technology-dependent, thus the original toolchain asked for a user-input specifying the target technology. After the update, the current possible parameters are: **"S"** for superconducting technology, **"I"** for trapped ions technology, **"M"** for NMR technology and **"Q"** for quantum dots technology.
- For **Step 3** the same procedures employed for the NMR technology are now also employed for the quantum dots technology.

3.2.2 Layout synthesis library integration

As already explained, the original VLSI toolchain Step 3, performing the layout synthesis, was actually **missing the whole placement and routing procedures**. To finalise the work started by M. Avitabile, the Step 3 was completed, integrating the developed layout synthesis library for producing a quantum circuit description that is actually executable in hardware. To achieve this, the workflow for each supported technology is expanded.

The new Step 3 structure, after the integration, is shown in Figure 3.8. For all the available technology's workflows, the first task performed is the employment of the powerful CX reduction templates. After that, the developed library heuristics can be used to place the quantum circuit and solve the coupling-constraint. Indeed, the CX and CZ gates are not decomposed with the native two-qubit interactions, in order to simplify the routing task. The decomposition of these gates with a last optimisation effort is postponed to the last blocks of Step 3.

The added initial mapping and routing phase is completely configurable by the end-user through the ***layout_synthesis.cfg*** configuration file. In this file, it is possible to indicate which placement and which routing algorithm to employ, as well as their optional parameters. Moreover, inside the same file, the path to the ***backend.cfg*** configuration file must be specified. This description file contains the

information on the allowed interactions and on the quantum gate features (more details are presented in Section 6.1). Therefore, these configuration files completely replace the old ***layout_nmr.cfg*** and ***layout_ion.cfg***, required by the original toolchain to obtain respectively the sign of the J-coupling constant (Section 1.5) and the interaction sign (Section 1.7), used for the correct CX/CZ gates decomposition.
For all the NISQ technologies, when the placement and routing is complete, the two-qubit interactions are decomposed by using the native coupling-gate (if requested). Afterwards, a last compaction attempt is made, trying to achieve a last optimisation, after all the constraints have been satisfied. In particular:

- For NMR, quantum dots and trapped ions technology, the original ***Special Templates*** are applied, to optimise the newly added R_{zz} and R_{xx} gates. Successively, the strong *Eulercombo* section is employed to compact the single-qubit gates included after the layout synthesis procedure. Indeed, remembering that this template requires that each single-qubit gate must not be adjacent to a single-qubit gate of the same type, for these technologies' workflow, the ***Finalcombo*** procedure [15, Sec. 2.2.6] is priorly executed, assessing that this requirement is satisfied.

- For superconducting technology instead, since there is no new two-qubit gate inserted (the CX gates are the native two-qubit interaction), the last applied block simply tries to compact the IBM's U_1, U_2, U_3 gates.

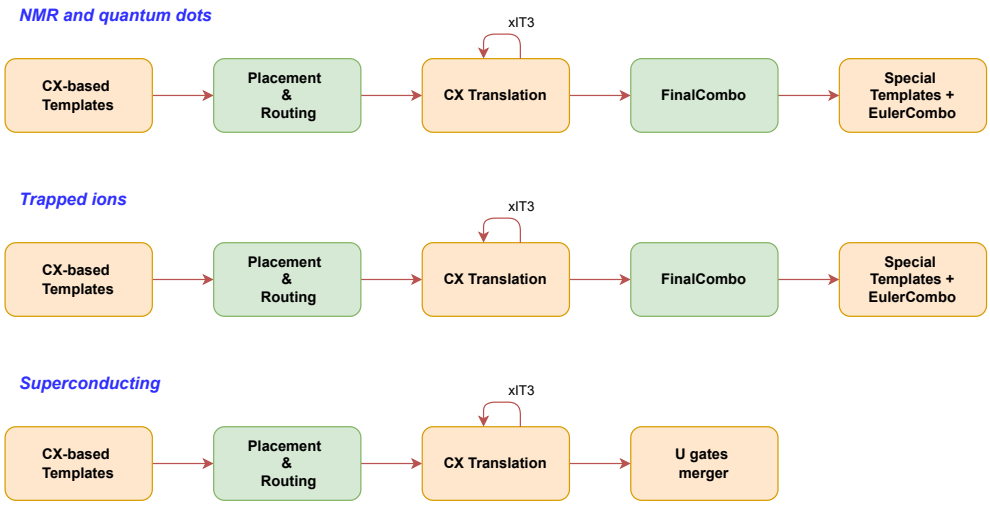

Figure 3.8. Final Step 3 structure of the complete VLSI quantum circuits compilation toolchain. The original blocks are depicted in orange, while the added blocks are highlighted in green.

Chapter 4

Placement

Placement is the first crucial operation performed by the layout synthesis block of a quantum circuit compilation toolchain.

Definition 4.0.1. The **placement** or **initial mapping** is a static one-to-one mapping between the logical qubits of a quantum circuit with the physical qubits of a quantum computing device [11, Sec. 3.2], [23, Sec. 3], [21, Sec. 7].

Following the definition 4.0.1 and remembering the task performed by a generic layout synthesis tool, the initial mapping produces the **initial spacetime coordinates** (t_j, x_j) for each gate j of the quantum circuit [6, Sec. 2].
These coordinates (and thus logical to physical qubits mapping) may or may not be the final ones. If the mapping is such that all the coupling-constraint of the target quantum computing device are respected, the layout synthesis would be complete. If instead some constraints are violated by the current mapping, the routing phase will perform some **circuit transformations** (adding additional gates) to allow a **dynamic** change of the current logical to physical qubits mapping, conforming to the target architecture. The details on how this is done are explained in Chapter 5.

Example 4.0.1. The following example will clarify the placement procedure and prepare the reader for the conventions used to show the input and output of the initial mapping step during the whole chapter.

The simple quantum circuit shown in Figure 4.1 is the selected case of study to explain the job performed during the placement step. Figure 4.2 shows both the coupling-graph of the quantum computing device target of the placement and the selected mapping of logical to physical qubits.
The result of the step can be analysed by looking at Figure 4.3. The space dimension is explicit in the circuit diagram, the information displayed is to which nodes the gates composing the quantum circuit are applied. The time dimension is instead implicit, the topological order of the circuit (from left to right) shows the order of execution of the gates.
The selected initial mapping for the example was: $\pi = \{q_0 \to n_0, q_1 \to n_1, q_2 \to n_2\}$.

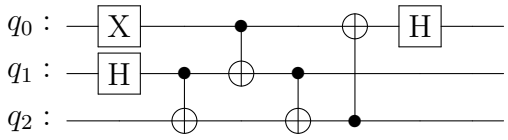

Figure 4.1. Original quantum circuit input of the placement step. All the quantum gates are applied to logical qubits, used only for describing the algorithm implemented by the circuit.

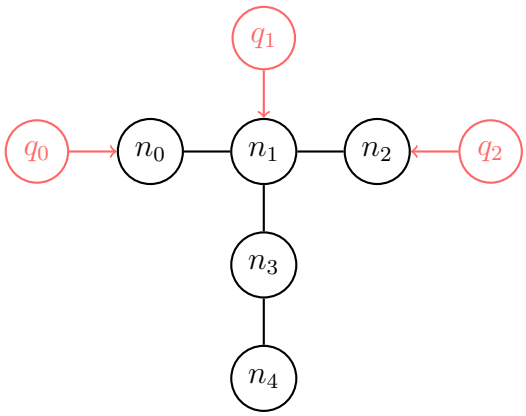

Figure 4.2. Graph representation of the NISQ device target of the placement plus the initial mapping to apply. The black circles and lines show the coupling-graph of the backend (modelling the *ibmq_lima* [13] superconducting device topology). The red circles and arrows represent the initial mapping, each logical qubit (red circle) is connected to a physical qubit (black circle). The nodes n_3 and n_4 are not mapped because the quantum circuit of Figure 4.1 is composed of only 3 logical qubits.

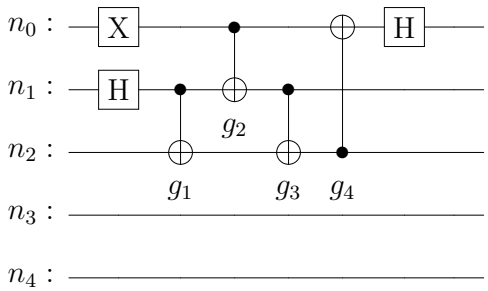

Figure 4.3. Output quantum circuit obtained after the placement is complete. All the gates are applied to physical qubits of the target backend. The two-qubit gates are labeled as g_1, g_2, g_3 and g_4.

Focusing the attention on the two-qubit gates of the circuit, g_1, g_2 and g_3 satisfy the constraints imposed by the architecture, while g_4 does not. This implies that the quantum circuit is still not executable on the target NISQ device. Hence, the routing step needs to perform further modifications to ensure that all the two-qubit gates are applied to nodes connected in the coupling-graph.

The placement is thus the first attempt to go from the abstract quantum algorithm described as a quantum circuit, to a physical set of transformations applied to nodes of a quantum computing device.
In the implemented **layout synthesis library** a class named *Placer* provides all the required data structures and methods to correctly perform this task.

4.1 State-of-the-art

In general, the existence of an initial mapping for a specific quantum circuit capable of solving all the connectivity constraints of a quantum computing device is not guaranteed [6, Sec. 3.1], [23, Example 2.3], [12, Sec. 15.2]. To be possible, there must exist a **subgraph monomorphism** between the interaction-graph G_I and the coupling-graph G_D [21, Sec. 7]. The first graph is one such that its nodes are the logical qubits of a quantum circuit, and there is an edge between two of its nodes if there is at least one two-qubit gate in the circuit acting on that pair of qubits. An example of such a graph is illustrated in Figure 4.4. The latter is the classical way of representing a quantum computer, an example is the one composed of the black circles and lines of Figure 4.2.

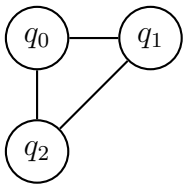

Figure 4.4. Interaction-graph G_I for the quantum circuit of Figure 4.1. The nodes of the graph are logical qubits and the edges show which qubits are interacting in the circuit.

For this reason, the need for a subsequent routing phase is considered compulsory. Hence, the placement searches for an initial mapping capable of reducing the **cost** of the following circuit transformations [23, Sec. 3]. This cost depends on some metrics that change based on the requirements (number of additional gates, circuit depth, fidelity, ...).

The placement problem is a real and difficult **combinatorial optimization problem** [27, Sec. 1]. The computer science literature is full of models and methods for solving this family of optimisation problems. Generally, the solutions are

divided into two main branches [28, Sec. 1]: the **exact** algorithms aiming at finding the global optimal solution and the **heuristic (or approximate)** algorithms that produce a locally optimal solution that may or may not be the global optimal one. The former produce the best solution, but they are most likely to be computationally heavy, while the latter may not produce the best solution, but the computational complexity is reduced.
Examples of exact algorithms are Integer Linear Programming [29], Branch and Bound [30] or Dynamic Programming [31].

Enumerating all the possible placements for a quantum circuit, comparing the metrics to optimise and select the best one, which would lead to the optimal solution in the search space, is computationally unfeasible.
For this reason, heuristic solutions were preferred for the development of this layout synthesis library. This decision is shared also by the majority of the associated literature [32], [23], [33], [34], [35] and by two of the main used quantum circuit compilation toolchains, namely **Qiskit** [26] and **t|ket⟩** [21].
Since these two famous quantum frameworks will be used for comparison during the benchmarking phase, Section 4.1.1 and Section 4.1.2 intend to introduce the main placement algorithms that they incorporate.

4.1.1 Qiskit placement algorithms

At the time of writing, Qiskit allows selecting **different initial mapping strategies** to perform the placement of a quantum circuit. The simplest initial mapping strategy which is available is the **TrivialLayout** [36]. Each logical qubit q_i is mapped to the corresponding physical qubit with the same id: n_i. Clearly, this policy does not optimise any metric, but it has the advantage of being computationally simple, and it can work on any quantum technology without modifications (technology-agnostic). For these reasons, this strategy was included as an available initial mapping strategy for the presented layout synthesis library.

A second placement policy is the **DenseLayout** [37]. The main idea here is to find the sub-graph of the coupling-graph of the size equal to the number of logical qubits of the quantum circuit with the maximum connectivity (where connectivity is intended to be the number of allowed interactions). If calibration data are provided to this algorithm, it can select a sub-graph also comparing the average CX error rates and the average readout error rates.

Another placement strategy available in Qiskit, exploiting the gates features of the target NISQ device during the mapping process, is the **NoiseAdaptiveLayout** [38]. It is the implementation of the initial mapping strategy presented in [18].

An initial mapping algorithm implemented in Qiskit that is relevant for the presented work is the **SABRE Layout** [39]. This is related to the SABRE [32] routing algorithm that performs a heuristic search for finding the best possible swap gate to insert whenever an interaction is not allowed in the target device.

Besides the routing strategy, in the article, a placement strategy is suggested, the **Reverse Traversal SABRE Initial Mapping** [32, Sec. 4.3.2]. The main idea here is to exploit the reversibility of a quantum circuit for generating a better initial mapping. The starting point is a random initial mapping, which is fed to the SABRE routing heuristic. The obtained final mapping is then used as an initial mapping for a new run of the heuristic, but for the reversed quantum circuit. The last mapping obtained in output is used as the selected initial mapping.

4.1.2 t|ket⟩ placement algorithms

t|ket⟩, at the time of writing, has available **three main placement strategies** to perform the initial mapping of a quantum circuit. All the algorithms implemented in the Cambridge quantum compilation toolchain focus on a simple, yet effective strategy aimed at maximising the number of two-qubit gates in the first layers of the quantum circuit that can be executed without any swap gate addition.

The simplest algorithm of the toolchain is the **LinePlacement** [11, Sec. 3.2]. Starting from the interaction graph, generates lines of interacting logical qubits that are mapped to lines of interacting nodes in the coupling-graph.
GraphPlacement [21, Sec. 7.1] is another strategy available. The algorithm finds a subgraph monomorphism between the interaction graph and the coupling-graph. If a monomorphism cannot be found, the algorithm removes one edge in the latest (higher) circuit layer and attempts the graph matching again. If there are multiple placements, the algorithm just selects the first one.
An improvement to this algorithm that takes into consideration also the fidelity of the nodes is the **NoiseAwarePlacement** [21, Sec. 9.2]. Placements are found using the GraphPlacement strategy, but then the output placement is selected by assigning an overall fidelity score to all the placements and picking the highest-scoring one.

All the strategies depicted before are not guaranteed to generate a complete initial mapping, but they might also produce a partial one. The routing algorithm implemented in t|ket⟩ then fills these gaps with an on-the-fly mapping.

4.2 Layout Synthesis Library - Placement

This section and the following Section 4.3 are devoted to explaining how the placement step was implemented in the proposed layout synthesis library.
The task of generating and applying an initial mapping to a quantum circuit is held by the ***Placer* class** inside the *placer.py* python script composing the library. It provides the necessary functionalities required to relabel the input quantum circuit to start from a set of gates that are applied to the abstract logical qubits and end with one that contains gates applied to the nodes of a specific target NISQ device.

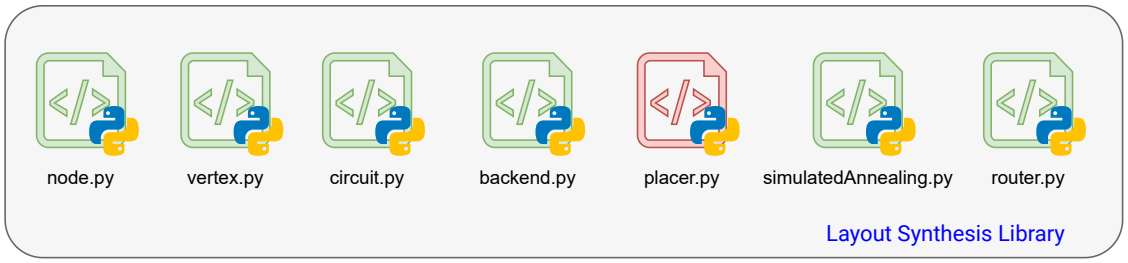

Figure 4.5. Representation of the python scripts composing the implemented layout synthesis library. The one containing the *Placer* class is highlighted in red.

4.2.1 Applying the initial mapping to a quantum circuit

The *Placer* class provides a simple method, *place*, which allows applying a user-defined initial mapping to a specific quantum circuit.
There are two tasks performed by the aforementioned method, required to correctly perform the placement:

- **Expand the quantum register size** of the circuit to match the number of physical qubits of the target quantum computer.
- **Relabel** all the gates and measures in the circuit following the provided initial mapping.

The input of this first step is a quantum circuit described using the **OpenQASM 2.0** language, in which all the gates are considered to be applied to logical qubits. This input description is thus the representation of the abstract quantum algorithm. The output is still a quantum circuit described using the **OpenQASM 2.0**. In this description, all the gates and measures must be considered to be applied to the nodes in the target NISQ device with the corresponding id. For example, if the output circuit OpenQASM 2.0 file contains the line: *cx q[0], q[1]*, that CX gate has the node 0 as the control qubit and the node 1 as the target qubit. Before the placement, all the qubits in a quantum circuit description are always considered to be logical ones, but after, all the spatial coordinates are well-defined, and this information must be taken into consideration for later execution or simulation.

Example 4.2.1. Example of the produced output of the placement step applying the user-provided initial mapping: $\pi_{init} = \{q_0 \rightarrow n_2, q_1 \rightarrow n_0, q_2 \rightarrow n_1\}$ using the 5-nodes backend of Figure 4.2 as target quantum computing device.

Figure 4.7 shows the result of applying the provided initial mapping to the input quantum circuit description shown in Figure 4.6. The quantum register size before applying the mapping was 3, the number of logical qubits of the circuit (even if the

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[3];
creg c[3];

x q[0];
y q[1];
cx q[0], q[1];
measure q -> c;
```

Figure 4.6. Input OpenQASM 2.0 description of the circuit. All the gates are considered to be applied to logical qubits.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
creg c[3];

x q[2];
y q[0];
cx q[2], q[0];

measure q[2] -> c[0];
measure q[0] -> c[1];
measure q[1] -> c[2];
```

Figure 4.7. Output OpenQASM 2.0 generated by the placement step applying the initial mapping π_{init}. All the quantum gates in this description are applied to physical qubits. The measure operations are expanded.

third logical qubit was not used in the algorithm). After the placement, the size became 5 matching the number of nodes of the target NISQ device.

Each quantum gate is also relabelled:

- *X q[0];* $\rightarrow$ *X q[2];* because the target logical qubit is q_0 and $\pi_{init}(q_0) = n_2$.
- *Y q[1];* $\rightarrow$ *Y q[0];* because the target logical qubit is q_1 and $\pi_{init}(q_1) = n_0$.
- *CX q[0], q[1];* $\rightarrow$ *CX q[2], q[0];* because the control logical qubit is q_0, the target logical qubit is q_1 and $\pi_{init}(q_0) = 2, \pi_{init}(q_1) = 0$.

All the measures operations that were in the compact form in the input quantum circuit are expanded. This is required to correctly describe to which classical bit each node must be measured.
One important thing to understand is that the measure location of each logical qubit does not change after the placement step. If the logical qubit i was measured in the classical bit j, then after the placement the i-th logical qubit of the circuit will still be measured in the same j-th classical bit:

- *measure q[2]* $\rightarrow$ *c[0];* because the logical qubit q_0 is measured into the classical bit c_0 in the original circuit, and $\pi_{init}(q_0) = n_2$.
- *measure q[0]* $\rightarrow$ *c[1];* because the logical qubit q_1 is measured into the classical bit c_1 in the original circuit, and $\pi_{init}(q_1) = n_0$.

- *measure q[1]* $\rightarrow$ *c[2];* because the logical qubit q_2 is measured into the classical bit c_2 in the original circuit, and $\pi_{init}(q_2) = n_1$.

4.2.2 Internal implementation

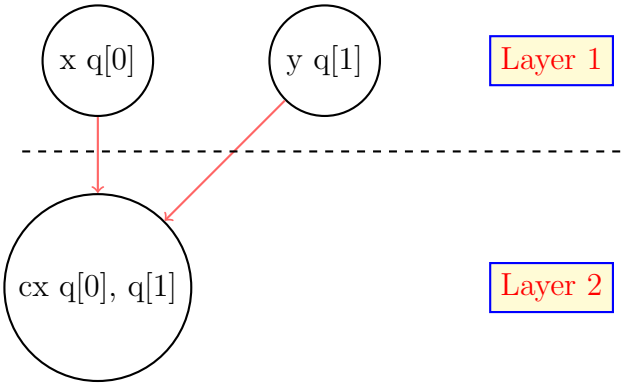

Figure 4.8. DAG representing the dependencies among the gates of the quantum circuit target of the placement step. All the gates are applied to logical qubits.

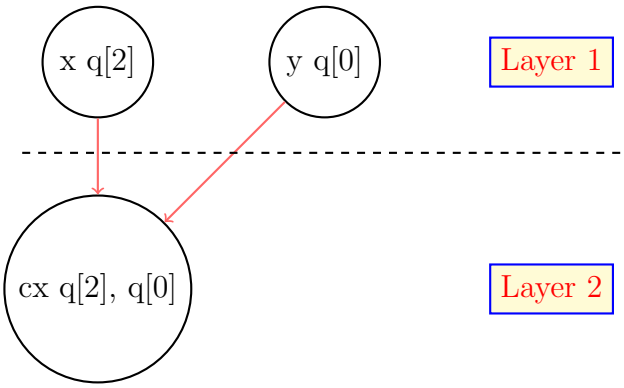

Figure 4.9. DAG representing the dependencies among the gates of the quantum circuit after the placement step was completed. All the gates are applied to physical qubits.

This section is devoted to showing the details of how the transformations, described in Section 4.2.1, are applied to the input quantum circuit during the placement step.
The *Placer.place* method **modifies the input quantum circuit**, accomplishing two main tasks:

- Relabel all the vertices of the circuit DAG following the provided mapping.

- Relabel all the measure operations following the provided mapping.

These two operations are easily understandable, remembering that the quantum circuit is internally represented using two data structures: a **DAG** for the quantum gates dependencies and a **list** for the measures.

Example 4.2.2. Internal modifications for applying the placement that is described in Example 4.2.1.

Figure 4.8 shows the DAG representation of the input quantum circuit. During the initial mapping, these gates are relabelled without modifying the dependencies among them, only the qubits ids change. Figure 4.9 shows the internal representation of the circuit after the step is complete.

The measure operations are kept in a separated data structure (a list) to ease the task performed by the router. The last modification required is to update the measure lines matching the provided initial mapping:

- *measure q[0]→c[0]* ⟶ *measure q[2]→c[0].*
- *measure q[1]→c[1]* ⟶ *measure q[0]→c[1].*
- *measure q[2]→c[2]* ⟶ *measure q[1]→c[2].*

4.2.3 Generating an initial mapping

Besides allowing the user to specify a desired initial mapping and applying it to the quantum circuit, the *Placer* class offers heuristics for automatically generating a possible placement. The available placement strategies for each targetable quantum technology, implemented inside the layout synthesis library, are depicted in Figure 4.10.

The **three implemented heuristics** are:

TrivialMapping: it is the simplest implemented strategy, inspired by the Qiskit **TrivialLayout** explained in Section 4.1.1. It is **technology-agnostic**, since it can be applied to all the supported quantum technologies without modifications, and **hardware-unaware**, since it is not requiring the calibration data of the target device. In this mapping methodology, the logical qubits are mapped to the nodes in the NISQ device with the same id: $\pi_{trivial} = \{q_0 \rightarrow n_0, q_1 \rightarrow n_1, q_2 \rightarrow n_2, q_3 \rightarrow n_3, \dots\}$, like shown in Figure 4.11.

SimulatedAnnealingDenseMapping: this is the first of two implemented mapping methodologies using the Simulated Annealing metaheuristic for performing the task. It is **technology-agnostic**, since it can be applied to all the supported quantum technologies without modifications, and **hardware-unaware**, since it is not requiring the calibration data of the target device.

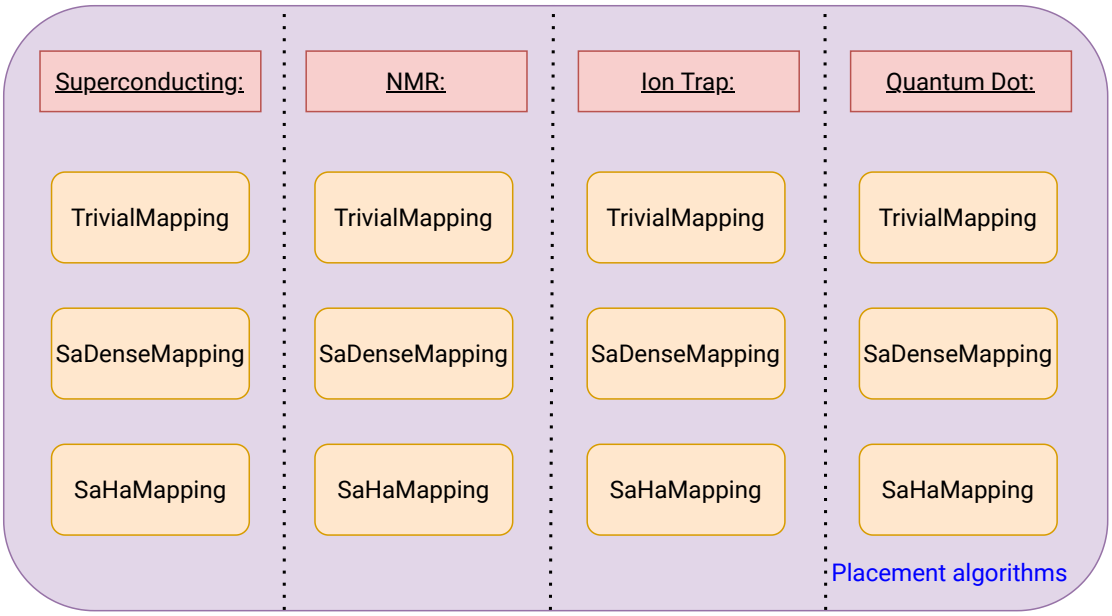

Figure 4.10. Available placement algorithms, developed and incorporated into the layout synthesis library, for each targetable quantum technology.

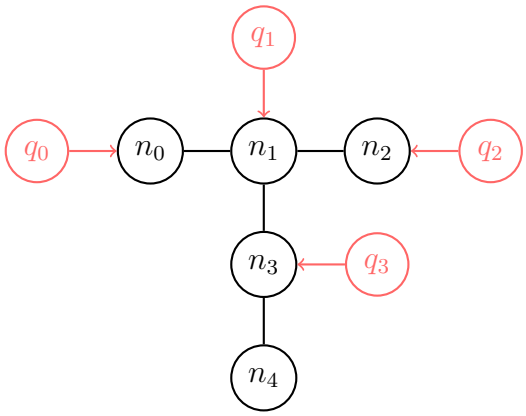

Figure 4.11. Graph representation of a trivial initial mapping applied to a quantum computing device. The black circles and lines show the coupling-graph of the backend (modelling the *ibmq_lima* [13] superconducting device topology). The red circles and arrows represent the initial mapping, each logical qubit (red circle) is connected to a physical qubit (black circle) with the same id. The node n_4 is not mapped because the target quantum circuit is supposed to have only 4 logical qubits.

The aim of this initial mapping strategy is to find a sub-graph of the coupling-graph with the maximum connectivity. This algorithm is explained in detail in Section 4.3.3.

SimulatedAnnealingHardwareAwareMapping: this is the second and last implemented mapping methodology using the Simulated Annealing metaheuristic for performing the task. It is **technology-agnostic**, since it can be applied to all the supported quantum technologies without modifications, and **hardware-aware**, since it is requiring the calibration data of the target device. The aim of this initial mapping strategy is to find a possible placement minimising the overall distance between each interacting qubits (the distance takes in consideration: physical distance between the nodes, error rate of a nodes' interaction, execution time of a nodes' interaction). This algorithm is explained in detail in Section 4.3.4

All of the aforementioned placement algorithms generate a logical to physical qubits mapping that can be applied to the target quantum circuit, using the *place* method described in Section 4.2.1.

4.3 Initial mapping generation using the Simulated Annealing metaheuristic

Besides the simple method illustrated in Section 4.2.3, two clever initial mapping strategies were implemented using the **Simulated Annealing metaheuristic** [40]. A metaheuristic [41] is a high-level algorithm used to guide the search for a solution to a generic optimisation problem (non-problem-specific). It tries to explore in a smarter way the search space in the prospect of finding a near-optimal solution [27, Sec. 1]. Metaheuristics usually adopt some mechanisms to perform an exploration of the search space avoiding being trapped in local optimal solutions, allowing a temporary non-optimal move.

The main reason why Simulated Annealing was considered a valid option is because it is a suggested placement approach used in [42, Sec. 3.B], the article that introduced the core routing algorithm implemented in the presented work. All the details and adaptations of this routing strategy are underlined in Chapter 6.
Their idea was to combine a hardware-aware search space exploration to a classical Simulated Annealing algorithm to improve the obtained initial mapping.

The computational complexity required to explore the solution space of the placement problem, combined with the interest in the proposed Hardware-Aware Simulated Annealing exploration strategy, led to the development and integration of two initial mapping algorithms:

1. **Simulated Annealing Dense Mapping**.

2. **Simulated Annealing Hardware-Aware Mapping**.

The former was implemented to have another placement that is hardware-unaware, the latter to have a cleverer way of generating the initial mapping, analysing also the calibration data of the target NISQ device.

4.3.1 Simulated Annealing

Simulated Annealing (SA) is among the oldest metaheuristic algorithms, inspired by the metallurgy annealing: the material is repeatedly heated and cooled down, in order to obtain the desired chemical and physical properties [27, Sec. 3.2].
The strategy that SA adopts to escape local minimum solutions is explicit. It uses a **temperature parameter (T)** which is linked to the probability of accepting non-optimal local solutions (uphill moves). This can be analysed looking at Figure 4.12: if the algorithm only considered local optimal moves, it would be trapped in a suboptimal solution. The SA procedure can accept temporary bad moves to "climb" the local optimal hill, reaching a better placement.

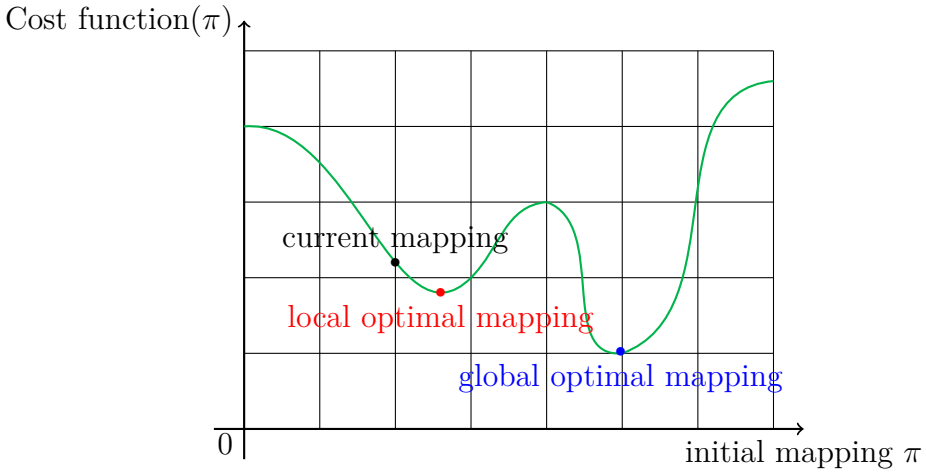

Figure 4.12. In green, it is plotted a generic cost function (that has an initial mapping as independent variable) that should be minimised. The black dot represents the current solution (an initial mapping) in the iterative search exploration process of the metaheuristic algorithm. The red dot shows the local optimal solution, reachable from the current one by iteratively adopting the locally optimal move. The blue dot is the global optimal solution that minimises the cost function.

SA, like any other metaheuristic algorithm, is problem independent. It is a generic high-level iterative algorithm to find the solution of an optimisation problem. In Figure 4.13 it is presented the flow chart of the implemented algorithm, already adapted for the placement problem. This implies that:

- The cost function has a possible placement π as independent variable.
- The neighbours' solution exploration method (*getNeighbour*) has a placement $\pi_{current}$ as the independent variable, and returns a new placement $\pi_{neighbour}$.
- The algorithm returns the optimal placement $\pi_{optimal}$ found.

The Simulated Annealing algorithm, presented in Figure 4.13, has the goal of minimising a generic cost function. The changes required in order to maximise a cost function are minimal.

The steps performed by the SA procedure are the following:

1. The starting point of the algorithm is the initialisation of the first possible placement $\pi_{initial}$ as the beginning of the solution space exploration. Another parameter initialised is the current temperature $T_{current}$. This is linked to the probability of accepting non locally optimal solution (the higher the temperature, the higher the probability).

2. The SA procedure then iterates for a fixed amount of times. At each iteration, a new mapping neighbour of the current one is sampled $\pi_{neighbour} = getNeighbour(\pi_{current})$. If the cost function computed for the new solution is lower than the previous one, this new placement is the new current mapping for keeping exploring the solution space.

3. If the new cost is higher instead, the neighbour solution is accepted with a probability that is proportional to $T_{current}$ and $costFunction(\pi_{neighbour}) - costFunction(\pi_{current})$. This probability usually follows the Boltzmann distribution reported in Equation (4.1). When the current temperature is higher, more locally worse neighbour solutions are accepted to perform the hill climbing.
$$\exp\left\{-\frac{costFunction(\pi_{neighbour}) - costFunction(\pi_{current})}{T_{current}}\right\} \tag{4.1}$$

4. At each iteration, $T_{current}$ is reduced. The task of deciding the temperature for the new iteration is handled by the **cooling strategy**. Complex non-monotonic strategies that incorporate period of heating (increasing $T_{current}$) and period of cooling (decreasing $T_{current}$) can be followed [27, Sec. 3.2]. The strategy adopted here, which is the most used one, is to cool down the temperature of a fixed amount Δ at every step, following Equation (4.2).
   The algorithm proceeds for a fixed amount of steps. When a final temperature T_{final} is reached, the optimal initial mapping found is returned.
   Other strategies might be adopted for implementing an SA procedure, like having a threshold on the maximum number of cycles or the maximum number of cycles without finding a better solution.
$$T_{current} = \Delta \cdot T_{current}, \Delta \in [0, 1] \tag{4.2}$$

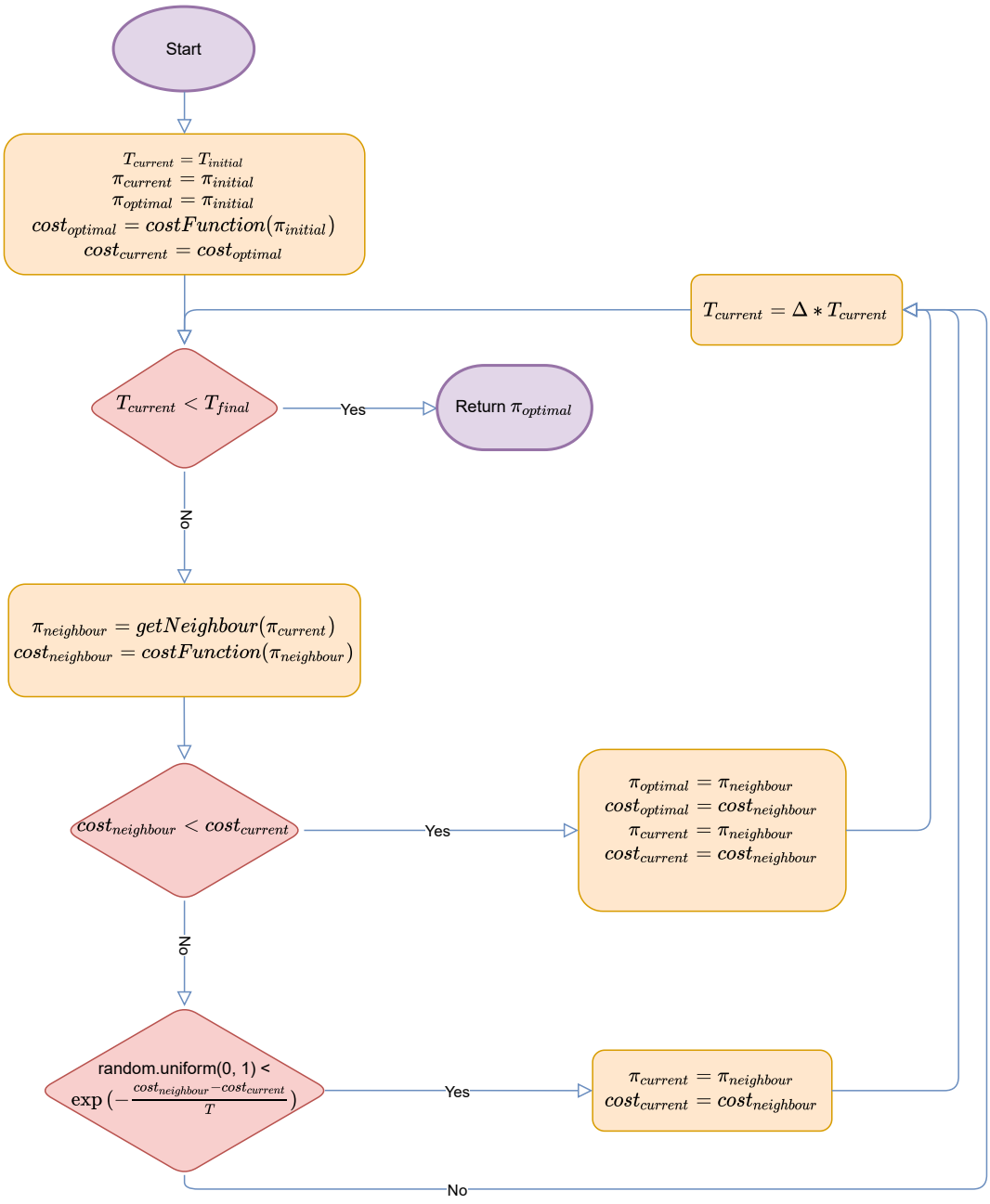

Figure 4.13. Flow chart of the Simulated Annealing algorithm adapted to solve the initial mapping problem. The presented algorithm aims at minimising the given cost function.

4.3.2 Simulated Annealing Implementation

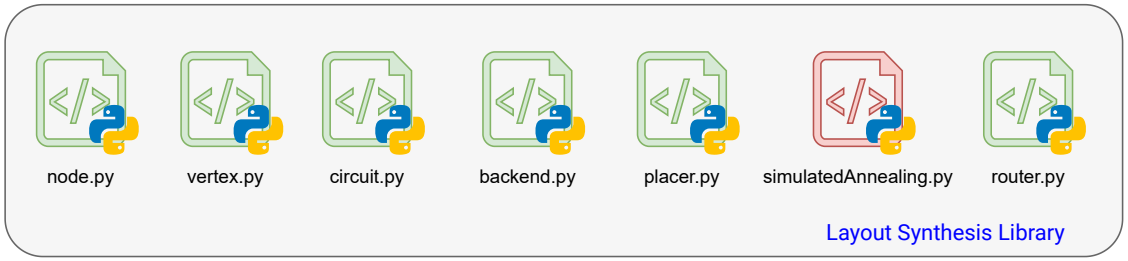

Figure 4.14. Representation of the python scripts composing the implemented layout synthesis library. The one containing the methods related to the Simulated Annealing is highlighted in red.

All the methods required for the development of initial mapping strategies utilising the SA methodology, were implemented in the *simulatedAnnealing.py* python script.

The core procedure of the SA iterative search was implemented as presented in Section 4.3.1, following the information available in [42, Algorithm 2] and [27, Sec. 3.2]. The implemented algorithm is mainly generic:

- It requires a generic cost function to optimise.
- It can both minimise and maximise the cost function, returning the optimal solution found.
- It requires a generic method to get a new neighbour mapping, in order to continue the solution space exploration.

The key idea presented in [42, Sec. 3.B] is to adopt a hardware-aware solutions' space exploration, to improve the quality of the selected initial mapping. Following this idea, the implemented cost functions and neighbour exploration methods were developed in two kinds: the **random (hardware-unaware)** ones and the **hardware-aware** ones. For the latter, in the following discussion, the concept of **distance** is used: it is a metric that takes into consideration the physical distance (edges), the error rate and the execution time of a nodes' interaction. For a complete explanation on how this distance is computed, Section 6.2.1 explains this in detail.

Two cost functions were written to be optimised by the SA main procedure: **costFunctionConnectivity** and **costFunctionHardwareAwareTotalDistance**. The first returns the total number of allowed interactions for the nodes used in the provided initial mapping, the second, the total distance between the interacting nodes (given a quantum circuit).

For the solutions' space exploration, two branches of policies were implemented: the **high-level** ones and the **low-level** ones. The first randomly choose which low-level policy to call, while the latter actually search for a new possible placement. The low-level *getNeighbour* policies have two counterparts: the random (hardware-unaware) ones and the hardware-aware ones.

The following is a presentation and description of the implemented low-level policies:

- **getNeighbourRandomReset**
  - Random (hardware-unaware) policy.
  - It completely resets the provided initial mapping, substituting it with a new random one.
- **getNeighbourHaReset**
  - Hardware-aware policy.
  - Returns a new initial mapping starting from a random logical qubit mapped to a random node and expanding the mapping until a complete one is generated. For the expansion, it is added the optimal node with the minimum distance from the last inserted one.
- **getNeighbourRandomShuffle**
  - Random (hardware-unaware) policy.
  - Returns a new initial mapping obtained by randomly shuffling the nodes of the input one. No new node is added to the mapping.
- **getNeighbourRandomExpand**
  - Random (hardware-unaware) policy.
  - Returns a new initial mapping obtained by replacing one random node of the mapping with a new random node of the backend that is not part of the mapping.
- **getNeighbourHaExpand**
  - Hardware-aware policy.
  - Returns a new initial mapping obtained by replacing one node of the mapping with a new node of the backend that is not part of the mapping. The node removed from the mapping is the one with the worst connectivity (number of edges) that maximises the total distance between the interacting nodes in the quantum circuit target of the placement. The replacement node is the one that minimises this total distance.

The two high-level policies implemented are:

- **getNeighbourRandom**
  - Random (hardware-unaware) policy.
  - Randomly selects which low-level policy to execute among *getNeighbourRandomShuffle*, *getNeighbourRandomReset* and *getNeighbourRandomExpand*. Each can be picked with a probability of respectively 90%, 2% and 8% (the same probability values used in the original paper [42, Sec. 4.A]).
- **getNeighbourHardwareAware**
  - Hardware-aware policy.
  - Randomly selects which low-level policy to execute among *getNeighbourRandomShuffle*, *getNeighbourHaReset* and *getNeighbourHaExpand*. Each can be picked with a probability of respectively 90%, 2% and 8% (the same probability values used in the original paper [42, Sec. 4.A]).

4.3.3 Simulated Annealing Dense Mapping

The first utilisation of the Simulated Annealing metaheuristic was to implement a placement strategy inspired by the Qiskit **DenseLayout** [37], presented in Section 4.1.1, that is, the **Simulated Annealing Dense Mapping**. This new initial mapping generation strategy uses a hardware-unaware solution space exploration for the SA procedure, with the intent of finding the combination of nodes that maximises the **connectivity**, that is, the total number of allowed interactions considering the selected nodes.

The followings are the details used for implementing this methodology:

- The core SA method underlined in Figure 4.13 and presented in Section 4.3.1 is called.
- As cost function to optimise, it is passed the **costFunctionConnectivity** explained in Section 4.3.2. For this cost function, the optimal solution is the **maximal** one.
- As neighbour solution exploration method, the high-level policy **getNeighbourRandom** is used, in order to obtain a random neighbour exploration.

This new placement algorithm was implemented to provide the layout synthesis library with a new methodology, besides the **TrivialMapping** presented in Section 4.2.3, capable of finding a possible placement, without requiring the calibration data of the target NISQ device.

4.3.4 Simulated Annealing Hardware-Aware Mapping

Another initial mapping strategy, that uses Simulated Annealing in order to retrieve the solution, is the **Simulated Annealing Hardware-Aware Mapping**. The idea here is to explore the possible placements in a smarter hardware-aware way, excluding some possible mappings in advance, using the calibration data of the target quantum computing device.
The mapping that this algorithm tries to obtain is one such that the total distance, among the interacting nodes in the quantum circuit target of the placement, is minimised, while the connectivity (number of edges) of the selected nodes is maximised. This distance between nodes is computed using the D matrix explained in Section 6.2.1.

The followings are the details used for implementing this methodology:

- The core SA method underlined in Figure 4.13 and presented in Section 4.3.1 is called.
- As cost function to optimise, it is passed the **costFunctionHardwareAware-TotalDistance** explained in Section 4.3.2. For this cost function, the optimal solution is the **minimal** one.
- As neighbour solution exploration method, the high-level policy **getNeighbourHardwareAware** is used, in order to obtain a neighbour mapping exploiting the calibration data information.

This new placement algorithm was implemented to have an initial mapping strategy linked to the core routing method implemented in the presented work, that is, the **Hardware-aware routing** explained in Chapter 6. The aim is to have an initial mapping capable of simplifying the work done by this routing strategy, in the following layout-synthesis phase.

4.4 Layout Synthesis Tool - Placement

Besides offering a **flexible python library** for integrating the layout synthesis phase in any quantum compilation toolchain, an additional set of scripts was developed, composing the **layout synthesis tool**: it is a **command-line application** (console application), designated to the end-user for performing the placement and routing operations without having a programming knowledge.

The layout synthesis tool is composed of two scripts:

layout_synthesis_tool_placement.py: it is a command-line application allowing the user to **apply all the placement algorithms described in this chapter** to any quantum circuit, targeting a superconducting, ion trap, NMR or quantum dots NISQ device.

layout_synthesis_tool_routing.py: it is a command-line application allowing the user to **apply all the routing algorithms described in Chapter 5** to any quantum circuit, targeting a superconducting, ion trap, NMR or quantum dots NISQ device.

4.4.1 Placement tool overview

Inputs: all the inputs must be passed as **console arguments** to the python script *layout_synthesis_tool_placement.py*. The inputs are divided into two categories:

Required arguments: mandatory for performing the placement.

- **inputQasmFile:** the relative (to the working directory) or absolute path to the OpenQASM 2.0 description of the quantum circuit target of the placement. Any quantum gates available in the OpenQASM 2.0 language can be used. All the **barriers and comments** inside the description are ignored and not included in the output placed circuit.
- **outputQasmFolder:** the relative (to the working directory) or absolute path to the folder where to store the output OpenQASM 2.0 description of the quantum circuit after the placement operation is performed.
- **backendConfigurationFile:** the relative (to the working directory) or absolute path to the .cfg configuration file describing the NISQ device target of the placement step. This can be a superconducting, NMR, ion trap or quantum dots device. A detailed explanation on how to model a generic device using a configuration file is reported in Section 6.1.

Optional arguments: for setting all the configurable parameters. Any missing optional argument is set to a default value.

- **-a:** the initial mapping generation algorithm to use in order to perform the placement step. Available strategies are: *Trivial Mapping*, *Simulated Annealing Dense Mapping*, *Simulated Annealing Hardware-Aware Mapping*. It is set to *Trivial Mapping* by default.
- **–swapNumberWeight:** the coefficient to be multiplied by the S matrix in the D matrix computation (used for the *Simulated Annealing Hardware-Aware Mapping* algorithm). For a detailed explanation on this placement strategy, see Section 4.3.4. It is set to 0.5 by default.
- **–swapErrorWeight:** the coefficient to be multiplied by the E matrix in the D matrix computation (used for the *Simulated Annealing Hardware-Aware Mapping* algorithm). For a detailed explanation on this placement strategy, see Section 4.3.4. It is set to 0.5 by default.

- **–swapTimeWeight:** the coefficient to be multiplied by the T matrix in the D matrix computation (used for the *Simulated Annealing Hardware-Aware Mapping* algorithm). For a detailed explanation on this placement strategy, see Section 4.3.4. It is set to 0 by default.
- **–isRZvirtual:** used only for NMR, quantum dots and ion trap technologies. *True* if the RZ gates are implemented virtually, *False* otherwise (used for the *Simulated Annealing Hardware-Aware Mapping* algorithm). For a detailed explanation on this placement strategy, see Section 4.3.4. It is set to *False* by default.
- **–Ti:** the initial temperature parameter, used by the simulated annealing metaheuristic (used for the *Simulated Annealing Dense Mapping* and *Simulated Annealing Hardware-Aware Mapping* algorithms). It is set to 10 by default.
- **–Tf:** the final temperature parameter, used by the simulated annealing metaheuristic (used for the *Simulated Annealing Dense Mapping* and *Simulated Annealing Hardware-Aware Mapping* algorithms). It is set to 10^{-6} by default.
- **–delta:** the temperature cooling parameter, used by the simulated annealing metaheuristic (used for the *Simulated Annealing Dense Mapping* and *Simulated Annealing Hardware-Aware Mapping* algorithms). It is set to 0.9 by default.

Outputs: the placement step produces two outputs:

- **outputQasmFile:** the OpenQASM 2.0 description of the placed quantum circuit. The file is written inside the *outputQasmFolder* directory.
- **initialMapping:** the initial logical to physical qubits mapping applied by the placement procedure.

In Figure 4.16 is shown a schematic illustrating the inputs and outputs of the layout synthesis placement tool.

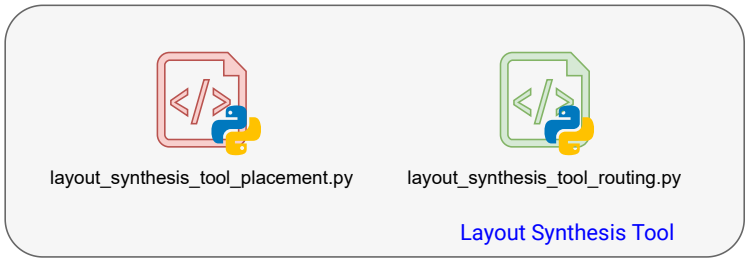

Figure 4.15. Representation of the python scripts composing the implemented layout synthesis tool. The script performing the placement phase is highlighted in red.

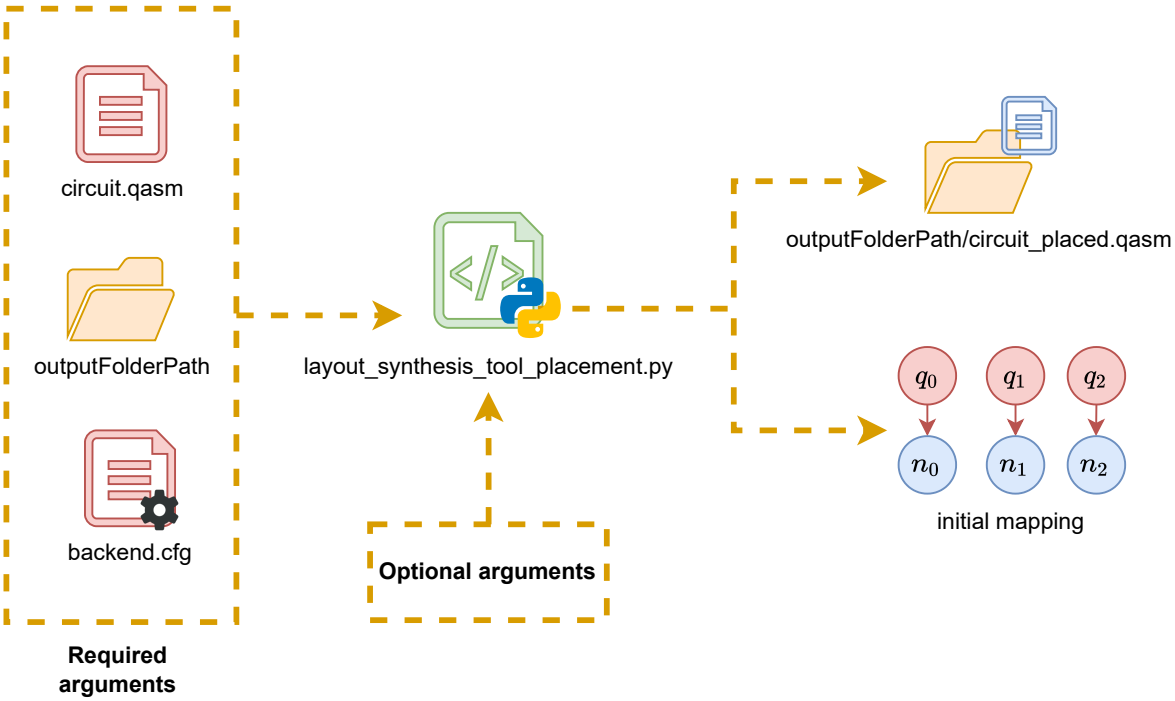

Figure 4.16. Schematic representation of the inputs and outputs of the layout synthesis placement tool. The inputs of the tool are divided into two categories: required and optional, as explained in Section 4.4.1. The outputs of the tool are: the placed quantum circuit and the initial mapping applied.

Chapter 5

Routing

As explained in Chapter 4, the **static** mapping between logical and physical qubits, established during the placement step, is usually not enough to satisfy all the coupling-constraint of the target NISQ device. **This mapping must dynamically change during the circuit execution**, to solve all the connectivity requirements. To accomplish this, a second fundamental step in order to complete the layout synthesis operation is required, the routing.

Definition 5.0.1. The **routing** consists in a transformation of the quantum circuit, adding additional gates, allowing a dynamic change of the mapping between the logical and physical qubits, to fulfil the coupling-constraint of a target quantum computing device [11, Sec. 3.3], [6, Sec. 3.2], [21, Sec. 7], [43, Sec. 3.A].

The most common way to implement this dynamic change is through the addition of **swap** gates (Figure 5.1) to the quantum circuit. A swap gate applied to two physical qubits **exchanges the logical qubits** to which they are mapped. After a swap gate is added to the circuit, **the following gates must be relabelled**, to ensure that the unitary transformations applied to each logical qubit in the input and output circuits are the same (thus a swap gate changes the interacting nodes of the circuit). Swap gates are added until all the nodes' interactions are allowed in the target coupling-graph.
Another option available to solve a coupling-constraint violation is relabelling a CX gate with a **bridge** gate when possible. A bridge gate allows implementing a CX between two disconnected nodes having one common neighbour without altering the current mapping, like shown in Figure 5.2.

After the routing is complete, the spatial and temporal coordinates (t_j, x_j) for each gate j composing the quantum circuit are the final ones [6, Sec. 2]. They indicate **when** and **where** to apply each specific transformations, meaning in which order and to which nodes apply each gate, to correctly implement the desired quantum algorithm.

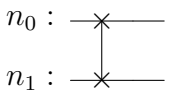

Figure 5.1. Swap gate example.

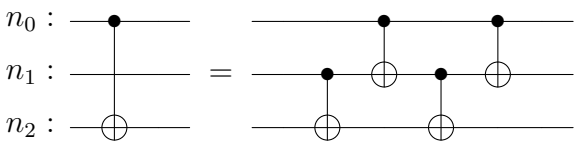

Figure 5.2. Bridge gate example.

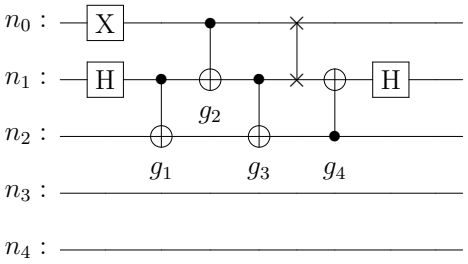

Figure 5.3. Output quantum circuit obtained after the routing is complete. All the gates are applied to physical qubits of the target backend. The two-qubits gates are labelled as g_1, g_2, g_3 and g_4. Each two-qubit gate respects the coupling-constraint of the target coupling-graph illustrated in Figure 5.4.

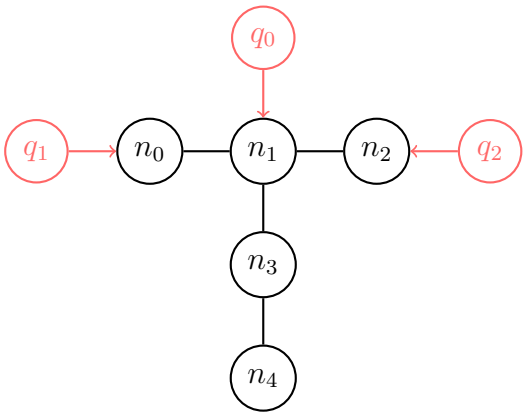

Figure 5.4. Graph representation of the NISQ device target of the routing plus the final logical to physical qubit mapping (after it changed dynamically from π_{init} at the end of the quantum circuit). The black circles and lines show the coupling-graph of the backend (modelling the *ibmq_lima* [13] superconducting device topology). The red circles and arrows represent the final mapping, each logical qubit (red circle) is connected to a physical qubit (black circle). The nodes n_3 and n_4 are not mapped because the quantum circuit of Figure 4.1 is composed of only 3 logical qubits.

Example 5.0.1. To have a better grasp of the routing procedure, the following example will continue the layout synthesis for the scenario exposed in Example 4.0.1.

After the quantum circuit of Figure 4.1 was placed for the NISQ device depicted in Figure 4.2, where also the applied initial mapping $\pi_{init} = \{q_0 \to n_0, q_1 \to n_1, q_2 \to n_2\}$ is indicated, some problematic was highlighted: the two-qubit gate g_4 did not respect the coupling-constraint of the target device, thus the quantum circuit was still not executable, as it can be noted by looking at Figure 4.3.
The routing procedure needs to **transform the placed quantum circuit**, adding additional gates, to solve this coupling-constraint violation.

A possible solution might be the final circuit presented in Figure 5.3. There, it is noticeable that the quantum circuit was altered during the routing phase, in particular, a swap gate was added before the problematic gate g_4. The swap gate **dynamically changes the current mapping** of logical to physical qubits, bringing it from π_{init} to $\pi_{final} = \{q_0 \to n_1, q_1 \to n_0, q_2 \to n_2\}$. The gate g_4, that in the original circuit of Figure 4.1 was an interaction between the nodes n_0 and n_2, became in the output circuit an interaction between n_1 and n_2 (the interacting logical qubits are still q_0 and q_2), completely executable in the target NISQ device.
A further transformation observable in Figure 5.3 is that the Hadamard gate (H gate), applied after the g_4 gate, was relocated. In the placed circuit (Figure 4.3) it was applied to n_0, in the final one it is applied to n_1. This is to ensure that the **same transformations are applied to the same logical qubits**, for the output circuit of the routing procedure.

5.1 State-of-the-art

The layout synthesis problem is NP-Complete [23, Theorem 3.1]. Splitting it into two sub-problems, namely placement and routing, makes it manageable but does not reduce the computational complexity of each individual phase.
Like the placement, also the routing has two main branches of approaches to find a valid solution [6, Sec. 4]: **reformulate the problem** with an equivalent mathematical model to find the exact solution employing an appropriate solver [44], [45], [46], [47], or **use heuristics** to cut off the search exploration time, hoping to find a near optimal solution [32], [42], [23], [33], [34], [35].

Since **Qiskit** [26] and **t|ket⟩** [21] are the selected quantum frameworks used for comparison during the benchmarking phase, Section 5.1.1 and Section 5.1.2 intend to prepare the reader on the main routing strategies that they incorporate. Section 5.1.3 explains an essential algorithm required for understanding the presented work, the **SABRE heuristic**.

5.1.1 Qiskit routing algorithms

The Qiskit framework, at the time of writing, offers **four main routing strategies** that can be utilised during the quantum circuit transpilation process [22].

The first and the simplest strategy is the **BasicSwap** [48]. When a two-qubits gate interaction is not allowed in the backend coupling-graph, it inserts one or more swap gates in front of it to make it compatible (following the shortest path between the involved physical qubits). This routing strategy was included in the presented layout synthesis library, as explained in Section 5.2.3.

The default heuristic used by the transpiler is the **StochasticSwap** [49]. This strategy adopts a stochastic (randomized) algorithm, and this implies that the output result will not be the same for repeated runs. Indeed, a distribution of quantum circuits with different characteristics will be obtained.

An optional routing strategy is the **LookaheadSwap** [50]. This is the implementation of the Sven Jandura's algorithm presented for the 2018 Qiskit Developer Challenge [51].
The algorithm explores the quantum gates composing the input circuit and marks each gate that can be executed (meaning that respects the target coupling-graph constraints) as executed. If a gate cannot be executed, at least one swap gate must be added to the quantum circuit. The algorithm searches for the **best swap gate** to add using the following procedure:

1. The **distance** among each pairs of logical qubits, q_i and q_j, is defined as the length of the shortest path connecting the nodes $\pi(q_i)$ and $\pi(q_j)$ (where π is the current mapping) in the coupling-graph. Among all the possible swap gates, the **four most promising ones** are found: these are the swap gates minimising the total distance between the interacting logical qubits for the following two-qubit gates.

2. For each of the four most promising swap gates, repeat the swap gate search procedure explained in Step 1 with a depth of four, supposing that the swap gate is applied to the circuit. The maximum number of final mappings explored is $4^4 = 256$.

3. The circuit transformation that allowed for the most two-qubit gates to be executed is selected, and **the first swap gate in this path is added to the input quantum circuit**. The algorithm then continues up to completion.

The last routing strategy worth to mention, integrated in Qiskit, is the **SabreLayout** one [39], that is explained in Section 5.1.3 since it is relevant for the presented work and needs its own space.

5.1.2 t|ket⟩ routing algorithms

The main algorithm available in the t|ket⟩ quantum framework, at the time of writing, is the **LexiRouteRoutingMethod** that uses the lexicographical comparison approach explained in [11, Sec. 3].
This strategy is completely **technology-agnostic**, since it can be run without modifications for any quantum technology, and **hardware-unaware**, since it does not require the calibration data of the target NISQ device. The only requirement is the target coupling-graph in order to specify the allowed two-qubits interactions. Another peculiarity of this routing method is that it can work on quantum circuits that were placed using a **partial initial mapping**: that is, a placement such that not all the involved logical qubits are mapped to a physical one, but that will be completed by the routing algorithm **on-the-fly**.

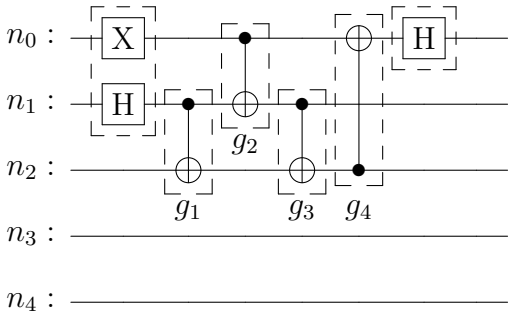

Figure 5.5. Example showing the t|ket⟩ quantum framework slicing the quantum circuit of Figure 4.3 into timesteps. Each timestep is numbered, starting from one, from left to right.

In order to work, the algorithm requires the input quantum circuit to be divided into **timesteps (or slices)**. A timestep is a set of gates that can be executed in parallel since there are no dependencies among them. **Each timestep is numbered** based on the dependencies among the gates composing the circuit: the timestep having gates with no dependencies at all (the ones at the topological start of the quantum circuit) constitute the timestep s_1. The timestep having gates dependent only to the ones of s_1 constitute the timestep s_2, and so on. Figure 5.5 shows an example of quantum circuit slicing performed by the Cambridge quantum compilation tool.

The LexiRouteRoutingMethod takes as input:

- The **sliced input quantum circuit**, already placed with a partial or complete initial mapping.
- The **initial mapping**, applied during the placement step.

- The **lookahead parameter**, employed when picking a swap gate: it defines the number of two-qubits gates to consider when finding the best swap gate to add.

The routing procedure iteratively builds a new output quantum circuit, appending gates respecting the dependencies imposed by the timesteps slicing, and inserting swap gates when necessary, to fulfil the coupling-constraint requirements. Since the routing can work on a partial initial mapping, in such scenario the partial mapping is automatically completed: for a two-qubits interaction where one logical qubit is not mapped to a physical one, this logical qubit is mapped to the nearest free node, with the aim of minimising the topological distance between the interacting nodes.

The algorithm proceeds timestep by timestep starting from the first one. All single-qubit gates and the two-qubits gates that are connected in hardware can be directly appended to the output quantum circuit. If after this procedure the current timestep is empty, the strategy proceeds to the next timestep, otherwise one or more swap gates must be added to the routed circuit.

In order to proceed with the swap gate insertion, the **best candidate swap** must be computed. To accomplish this, a list of candidate swap gates is generated to select the best one, with the aim of minimising the total number of swap gates added by the procedure. This list is generated as follows:

1. $\Sigma_0 = swaps(s_0)$ is generated, containing all the swap gates that can change the mapping of an interacting logical qubit in the current timestep.

2. $\Sigma_{t+1} = \arg\min_{\sigma \in \Sigma_t} d(s_t, \sigma \bullet m)$ is generated up to $\Sigma_{lookaheadParameter}$, where:

   - s_t denotes the timestep t.
   - Σ_t denotes the set of candidate swap gates for the timestep s_t
   - $\sigma \bullet m$ denotes the action of applying the swap gate σ to the mapping m, that is, the new mapping after that swap gate is applied.
   - $d(s, m)$ is the **distance vector**, estimating the number of swap gates to add to the timestep s with the mapping m, to make all of its two-qubits gates executable in the target hardware.
   - $d(s_t, \sigma \bullet m)$ is estimating the number of swap gates to add to the timestep s_t to make all of its two-qubits gates executable in the target hardware, after the swap gate σ is applied to m.
   - $\arg\min_{\sigma \in \Sigma_t} d(s_t, \sigma \bullet m)$ is computing the set of swap gates in Σ_t minimising the distance vector $d(s_t, \sigma \bullet m)$: that is, the set of swap gates minimising the approximated number of swap gates required to make all of the two-qubits gates in s_t executable in the target hardware, after the swap gate σ is applied.

- $\Sigma_{lookaheadParameter}$ is the last set of candidate swap gates that can be generated. The higher the lookahead parameter, the more timesteps are considered for the best swap gate computation.

3. The algorithm proceeds until $|\Sigma_t| = 1$ or a predefined maximum number of iterations is reached. The best swap gate found is appended to the routed circuit. If a swap gate could not be found (in some cases the strategy gets stuck), swap gates are added connecting the furthest interacting nodes, following the shortest path between them in the coupling-graph.

5.1.3 SABRE routing algorithm

Because the core routing algorithm implemented in the presented work is the **hardware-aware** one, explained in details in Chapter 6, it is essential to mention the **father** of this routing strategy, expanded by the authors of [42]: the **swap-based bidirectional heuristic search algorithm (SABRE)** [32]. It was presented as a novel swap insertion heuristic labelled, when published, as capable of outperforming the best known routing algorithms. For these reasons, it was also incorporated as an available routing strategy inside the Qiskit quantum framework. The objectives at the heart of this routing strategy were to build an algorithm **flexible** for targeting any NISQ device topology, aiming at **maximising the final circuit fidelity** and **minimizing the circuit depth** (trying to avoid the insertion of non parallelisable swap gates) and being **scalable** to work with quantum computing devices composed of hundreds of physical qubits with a reasonable execution time.

The heuristic approach used by SABRE is composed of two main operations:

Preprocessing: it is the first operation required, that initialises the needed data structures [32, Sec. 4.1].
The tasks performed are:

- **Distance matrix** $D[\,][\,]$ **computation**, applying the **Floyd-Warshall algorithm** [52] to the coupling-graph of the target quantum computing device. It computes the **all-pairs shortest path** between all the nodes of the input graph: that is, the lengths of the shortest paths between each pair of nodes. $D[i][j]$ contains the minimum number of swap gates required to apply a swap between node i and node j (satisfying the coupling-constraint).
- **Circuit DAG generation**. The input quantum circuit is represented using a directed acyclic graph (DAG), storing the dependencies among the quantum gates. The first nodes of the DAG from a topological perspective (the nodes having zero in-degree) are **immediately executable**, since they have no dependencies. A generic quantum gate represented as a

node of the DAG can be executed **only if all of its predecessors have been executed**.

- **Front layer initialisation**. The front layer F is the set of quantum gates, composing the circuit DAG, that have no dependencies among them and on any other non executed gate. This means that all the gates in F can always be executed from a dependency perspective, even if this might not be the case from a hardware perspective (for example, if it is a two-qubits gate and the interacting nodes are not connected in the target coupling-graph).
  This front layer will constantly be updated during the routing process, removing the gates that can be executed and inserting their successors. This is done in order to explore the quantum gates composing the circuit in topological order, respecting their dependencies.

SWAP-Based Heuristic Search: The second operation is the main routing heuristic, that scans the input quantum circuit in topological order, thus respecting the DAG dependencies, and inserts swap gates when a two-qubits interaction is not allowed in the target NISQ device [32, Sec. 4.2]. Its inputs are the distance matrix D, the circuit DAG, the front layer F, the coupling-graph $G_{coupling}$ and the initial mapping π_{init}, which is the starting point of the routing.

The main steps performed by the algorithm are:

1. The algorithm begins the topological scan starting from the initial F, and **labels as executed** the quantum gates that can be executed (both considering the DAG dependencies and the hardware constraints). If the front layer F is empty, the algorithm ends since all the quantum gates have been executed. Otherwise, a list called *Execute_gate_list* is constructed: it contains all the single-qubit gates and all the two-qubits gates in F that can be executed. A two-qubits gate inside F can be executed if its interacting nodes are connected by an edge in $G_{coupling}$.
2. If *Execute_gate_list* is not empty, all of its gates are removed from F (that is equivalent to label them as executed). Every time that a gate is removed from F its successors are evaluated, to check if they must be inserted inside the front layer: if a successor of the removed gate have no dependency with a gate in F, it is added to the front layer. After this the algorithm repeats from Step 1.
3. Otherwise, if *Execute_gate_list* is empty, it means that all the gates composing F are two-qubits gates whose interacting nodes are not connected in hardware. A swap gate must be inserted to satisfy the connectivity constraints. The list *SWAP_candidate_list* is constructed: it contains all the possible swap gates for each node involved in an interaction contained

in F (that is, the swap gates that can bring the interacting logical qubits in F closer in hardware).

4. Each swap inside *SWAP_candidate_list* is rated using a heuristic cost function H. The swap with the lowest score is added to the output quantum circuit, and the algorithm repeats from Step 1.

   For the rating, it is possible to use two heuristics: Equation (5.1) and Equation (5.2), both reported from [32, Sec. 4.4].

$$H = \sum_{gate \in F} D[\pi(gate.q1)][\pi(gate.q2)] \tag{5.1}$$

$$\begin{aligned} H = \max((&decay(\text{SWAP.q1}), decay(\text{SWAP.q2})) \\ &\cdot \Big\{ \frac{1}{|F|} \sum_{gate \in F} D[\pi(gate.q1)][\pi(gate.q2)] \\ &+ W \cdot \frac{1}{|E|} \sum_{gate \in E} D[\pi(gate.q1)][\pi(gate.q2)] \Big\} \end{aligned} \tag{5.2}$$

   The former is the simplest version of the function, summing the **shortest path distances** between the interacting nodes for each gate inside the front layer, where:

   - $D[\,][\,]$ is the distance matrix computed during the preprocessing phase.
   - π is the current mapping supposing the candidate swap gate, for which H must be computed, was added to the quantum circuit.
   - $\pi(gate.q1)$ and $\pi(gate.q2)$ are the interacting nodes of *gate* given the current mapping π.

   The latter is the most complete version of the heuristic, taking in consideration also the impact of a swap gate insertion for the future gates (thus, adding the look-ahead ability), and also allows flexibility for preferring swap gates that are parallelisable.

   The final H presented in Equation (5.2) is composed of:

   - $decay(q_i)$ is the **decay effect**. It is used in order to prefer the addition of non overlapping swap gates, that is, swap gates acting on different nodes. Parallelisable swap gates might be preferred in order to avoid increasing the final circuit depth. This decay effect depends on a tunable parameter δ, in this way **the user can decide the importance of reducing the final circuit depth**.
   - $|F|$ is the number of quantum gates inside the front layer F.
     $\frac{1}{|F|}$ is used to **normalise** the summation of the distances between the gates in F.

- $|E|$ is the number of quantum gates inside the **extended layer** E. This set contains some successors of the gates in F, where the size of E is a tunable parameter. In this way, **the user can decide the depth of the look-ahead ability**.
  $\frac{1}{|E|}$ is used to **normalise** the summation of the distances between the gates in E.
- W is the **look-ahead parameter**. It is a real value such that $0 \leq W < 1$ that is **used to reduce the importance of the look-ahead ability**.

This routing heuristic is the same procedure used by the hardware-aware routing algorithm. The only modifications required are related to the preprocessing phase, and heuristic cost function definitions. For a more in-depth explanation and a view of the flow-chart illustrating the implemented algorithm, see Section 6.2.

5.2 Layout Synthesis Library - Routing

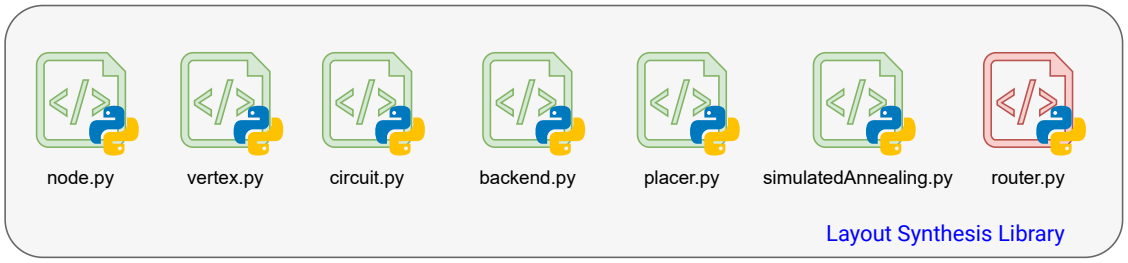

Figure 5.6. Representation of the python scripts composing the implemented layout synthesis library. The one containing the *Router* class is highlighted in red.

This section and the following Chapter 6 are devoted to explaining how the routing phase was implemented in the proposed layout synthesis library.
The task of transforming the input quantum circuit, allowing a dynamic change of the logical to physical qubits mapping and making all the gates compliant to the target topology is held by the ***Router* class**, inside the *router.py* python script composing the library.
It encapsulates all the functionalities and routing strategies available to solve the routing problem.

5.2.1 Performing the routing of a quantum circuit

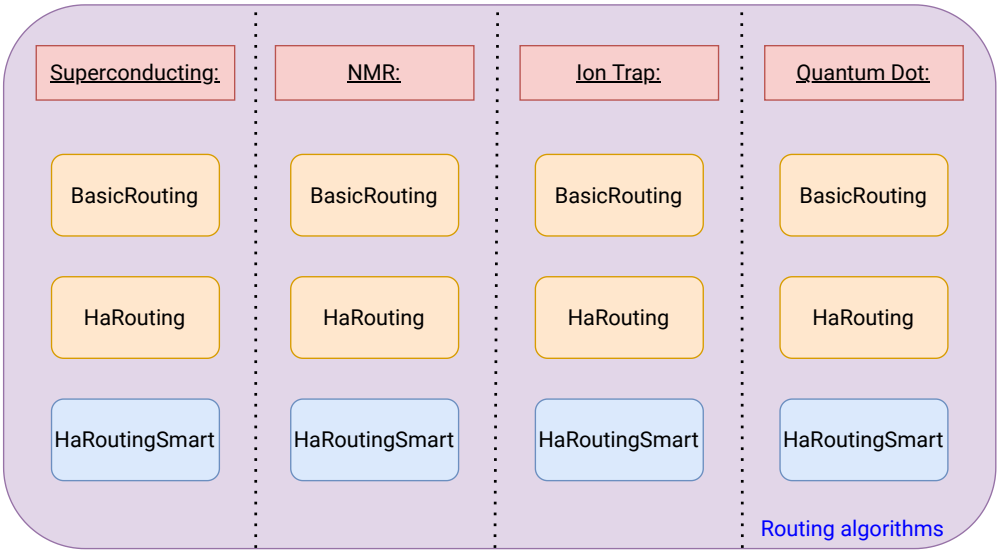

Figure 5.7. Available routing algorithms, developed and incorporated into the layout synthesis library, for each targetable quantum technology. The ones depicted in yellow are suited for non-fully-connected topologies, while the ones depicted in light blue are suited for fully-connected topologies.

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
creg c[4];

x q[0];
y q[1];
cx q[0], q[2];
h q[2];
x q[3];

measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];
measure q[3] -> c[3];
```

Figure 5.8. Input OpenQASM 2.0 description of the circuit. All the gates are considered to be applied to nodes of the architecture represented in Figure 4.11, that is also indicating the initial mapping applied: a trivial mapping (see Section 4.2.3).

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
creg c[4];

x q[0];
y q[1];
swap q[2], q[1];
cx q[0], q[1];
h q[1];
x q[3];

measure q[0] -> c[0];
measure q[2] -> c[1];
measure q[1] -> c[2];
measure q[3] -> c[3];
```

Figure 5.9. Output OpenQASM 2.0 generated by the routing step. All the quantum gates in this description are applied to physical qubits and all the two-qubits interactions are allowed in the target coupling-graph.

The available routing strategies for each targetable quantum technology, implemented inside the layout synthesis library, are depicted in Figure 5.7.
The *Router* class provides **three route methods** in order to perform the routing phase:

routeBasicRouting: it is the simplest strategy available, suited for targeting **non-fully-connected topologies** and **hardware-unaware** (does not require the calibration data of the target NISQ device). This algorithm is explained in details in Section 5.2.3.

routeHardwareAwareRouting: it is the most performant heuristic implemented in the proposed layout synthesis library. Suited for targeting **non-fully-connected topologies**, it is **hardware-aware** (it requires the calibration data of the target NISQ device), and it has the intent of optimising the **fidelity**, **number of additional gates** and **total execution time** in a way that is completely configurable by the user. This algorithm is explained in details in Section 6.2.

routeHardwareAwareRoutingSmart: it is a **smart adaptation** of the *HardwareAwareRouting* algorithm suited for targeting **fully-connected topologies**. It is **hardware-aware** (it requires the calibration data of the target NISQ device), and it has the intent of optimising the total execution time of the output quantum circuit. This algorithm is explained in details in Section 6.3.2.

Each of the aforementioned methods performs the following required tasks:

- **Constructs** an output quantum circuit in which the same quantum gates of the input circuit are applied in the same order (respecting the dependencies among them), and to the same logical qubits, ensuring that the output circuit implements the same initial unitary transformation (thus not altering the **functional behaviour of the circuit**).

- Whenever an interaction is not allowed in the target NISQ device (according to the coupling-graph), **swap gates are inserted** to dynamically change the logical to physical qubits mapping, or **a CX gate is relabelled with a bridge one**, when possible. The following gates must be updated in accordance to the transformation.

- In the output quantum circuit, **all the two-qubits gates must be applied to a couple of nodes that are allowed to interact** in the target NISQ device.

The input of this step is a quantum circuit described using the **OpenQASM 2.0** language that **was already placed**, thus all the gates are considered to be

applied to physical qubits. Besides the target circuit and architecture, also the applied **initial mapping** must be passed, in order to compute and show the **final mapping** between logical and physical qubits (reached dynamically at the end of the quantum circuit execution).
The output of this step is still a quantum circuit described using the **OpenQASM 2.0** language, that is completely executable on the specified target quantum computing device because it respects all its connectivity constraints (the gates composing the circuit must be native gates available in the target quantum technology, this requirement is fulfilled during the logic synthesis block of a quantum circuit compilation toolchain).

Example 5.2.1. The following example will clarify the tasks performed during the routing phase for a quantum circuit violating some coupling-constraint.

The presented case of study is the routing of the circuit described by the OpenQASM 2.0 code of Figure 5.8, using the NISQ device shown in Figure 4.11 as the target coupling-graph, where also the initial mapping applied during the placement is indicated: $\pi_{init} = \{q_0 \rightarrow n_0, q_1 \rightarrow n_1, q_2 \rightarrow n_2, q_3 \rightarrow n_3\}$.
Figure 5.9 shows the result of the routing phase transformations:

- *X q[0];* and *Y q[0];* remained unaltered since they are single-qubit gates, thus have no connectivity constraint.
- The quantum gate *CX q[0], q[2];* of the input circuit was not executable, because there is not an edge in the target coupling-graph between the nodes n_0 and n_2. For this reason, the routing phase added the line *swap q[2], q[1]*, bringing after its execution the interacting logical qubits q_0 and q_2, connected in the quantum hardware. In particular, once the added swap gate has been executed, the logical to physical qubits mapping dynamically changes from π_{init} to $\pi_{final} = \{q_0 \rightarrow n_0, q_1 \rightarrow n_2, q_2 \rightarrow n_1, q_3 \rightarrow n_3\}$.
  It is essential to notice that after the swap gate is added, all the subsequent gates are relabelled accordingly:
  - *CX q[0], q[2];* → *CX q[0], q[1];* because the interacting logical qubits are q_0 and q_2 and, after the swap gate gets executed, $\pi_{final}(q_2) = n_1$.
  - *H q[2];* → *H q[1];* because the target logical qubit is q_2 and, after the swap gate gets executed, $\pi_{final}(q_2) = n_1$.

  The modifications explained before underline that the logical qubits on which each gate is applied to do not change after the routing.
- The quantum gate *X q[3]* remains unchanged because the target logical qubit is q_3 and $\pi_{init}(q_3) = \pi_{final}(q_3) = n_3$.
- One important thing to understand is that the measure location of each logical qubit does not change after the routing step. If the logical qubit i was measured

in the classical bit j, then after the routing the i-th logical qubit of the circuit will still be measured in the same j-th classical bit:

- *measure q[0] → c[0];* ⟶ *measure q[0] → c[0];* because $\pi_{init}(q_0) = \pi_{final}(q_0) = n_0$.
- *measure q[1] → c[1];* ⟶ *measure q[2] → c[1];* because $\pi_{final}(q_1) = n_2$.
- *measure q[2] → c[2];* ⟶ *measure q[1] → c[2];* because $\pi_{final}(q_2) = n_1$.
- *measure q[3] → c[3];* ⟶ *measure q[3] → c[3];* because $\pi_{init}(q_3) = \pi_{final}(q_3) = n_3$.

5.2.2 Internal implementation

This section is devoted to showing the details of how each routing method presented in Section 5.2.1 performs its duty.

Before focusing on this, it is mandatory to recall that the presented layout synthesis library internally represents the input quantum circuit using the ***Circuit*** class (see Section 3.1.1), containing two data-structures: a **DAG** for the quantum gates dependencies and a **list** for the measures. The target quantum computing device of the routing step is abstracted using a class, ***Backend***, that contains a **graph** (the coupling-graph) representing the connectivity constraints among its nodes (see Section 6.1 for further details).

The implemented library offers some **basic methods**, giving to the routing algorithms the necessary required functionalities. These basic methods are:

checkGateSatisfiesConnectivity: checks whether a quantum gate satisfies the connectivity constraints of the target quantum device. This is true in two conditions: if the gate is a single-qubit gate or if it is a two-qubits gate and there exists an edge in the coupling-graph connecting the interacting nodes.

Swap: appends a swap gate to the routed quantum circuit while relabelling all the gates in the original (placed) circuit DAG accordingly. The method also updates the measures list of the routed circuit to ensure that the classical bit where each logical qubit is measured is the same as in the original input one. Every time that a swap gate is added, the internal information on the mapping between logical and physical qubits $\pi_{current}$ is updated. This is required to output the final mapping π_{final}.

Bridge: appends a bridge gate to the routed quantum circuit, substituting a CX gate.

These aforementioned methods are the basic building blocks to any routing strategy. Each algorithm implementing the routing procedure follows the following structure for **iteratively generating an output routed quantum circuit**:

1. **Extract the first layer** of the original quantum circuit (a layer is a set of quantum gates having no dependencies among them, see Section 3.1.1 for details on how the circuit is sliced in layers underling the gates dependencies). The first layer is **removed** from the original circuit during the extraction, employing the *extractLayer1* method of the *Circuit* class composing the library.

2. **Insert in the output (routed) quantum circuit** all the single-qubit gates and the two-qubits gates of the extracted first layer, respecting the coupling-constraint of the target device. Whenever a gate is added to the output circuit, it is also removed from the first layer.

3. If the **first layer is not empty** at this point, it means that it contains some two-qubits interactions that are not allowed in the target coupling-graph. Swap or bridge (if possible) gates must be added to make the interacting logical qubits connected in hardware. The strategy used for selecting such swap or bridge gates depends on the implemented routing algorithm.

4. **Until the original circuit is not empty** (meaning all of its gates have been extracted and inserted in the output circuit), repeat from Step 1.

The structure depicted before allows building an output quantum circuit respecting the gates dependencies of the original one, having all the nodes' interactions allowed in the target NISQ device. It is used by the basic routing strategy explained in Section 5.2.3 and the hardware-aware ones explained in Chapter 6.

It is worth to mention that all the basic methods underlined before and all the data structures and methods used for representing quantum circuits (see Section 3.1.1), offered by the proposed layout synthesis library, allow implementing **any routing strategy**. The structure explained before is just the suggested structure to follow when implementing a routing algorithm, but it is not mandatory.

5.2.3 BasicRouting strategy

The simplest strategy implemented as solution to the routing problem is the **BasicRouting**, inspired by the Qiskit **BasicSwap** [48] explained in Section 5.1.1. The algorithm is suited to solve the connectivity violations for **non-fully-connected topologies**, otherwise no modifications are performed to the input quantum circuit. It is also **hardware-unaware**, because it does not require the calibration data of the target NISQ device. Indeed, the only hardware information needed is the coupling-graph in order to check for the allowed two-qubits interactions.

Its core idea is the following: when a two-qubit gate interaction is not allowed in the backend coupling-graph, it inserts one or more swap gates in front of it to make it compatible (following the shortest path between the involved physical qubits) with the hardware device.

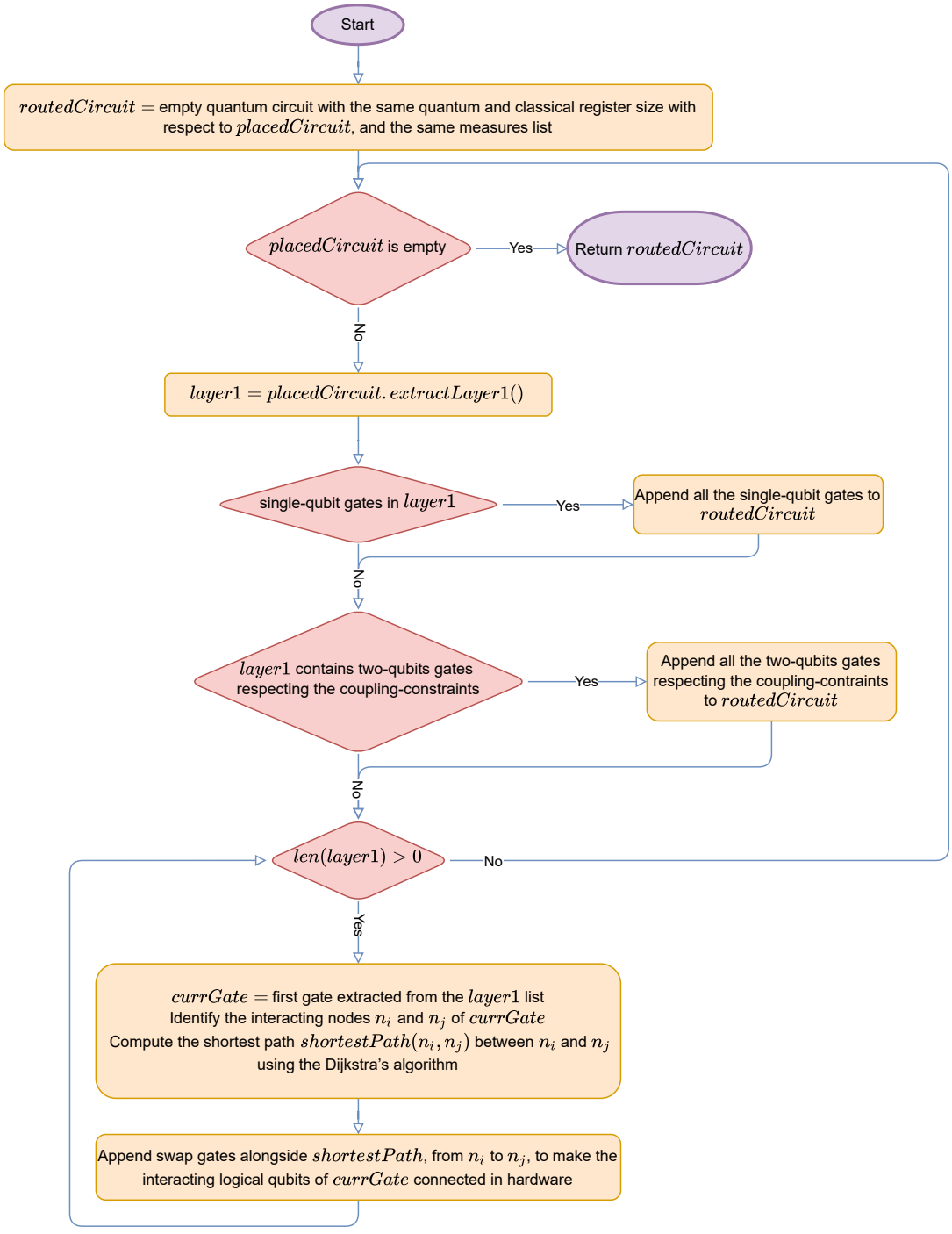

Figure 5.10. Flow chart of the BasicRouting algorithm.

Figure 5.10 shows the flow-chart of the presented routing strategy.
Its required inputs are:

- *placedCircuit:* the input quantum circuit of the routing procedure for which the placement was already performed.

- π_{init}**:** the initial mapping applied during the placement. It is required since during the routing phase, the current (considering all the swap gates added to the circuit) mapping between logical and physical qubits is tracked to output the final mapping π_{final} (the mapping at the end of the quantum circuit execution, before the measure operations).

The main steps followed by the presented strategy are:

1. The first step is an **initialisation phase**. The algorithm constructs step-by-step a new quantum circuit (***routedCircuit***), extracting gates from the placed circuit and inserting them in the new one, respecting their dependencies.
   During this initialisation the *routedCircuit* is instantiated: at the beginning it contains no quantum gates; the quantum and classical registers size match the ones of the original circuit; the measures operations are the same as the placed circuit ones.

2. The **first layer of the input quantum circuit is extracted** (thus it is removed from the original circuit). All of its composing single-qubit gates and the two-qubits gates applied to nodes connected in the target coupling-graph are simply added to the output circuit.

3. If at this step the **first layer is not empty**, it means that it contains one or more two-qubits interactions that are not allowed on the target hardware. For each of the remaining gates inside the first layer, the following operations are preformed: its interacting nodes n_i and n_j are identified; the shortest path between them in the target coupling-graph is computed, resorting to the Dijkstra's algorithm [53]; swap gates are added following the shortest path from n_i to n_j in order to connect the interacting logical qubits.

4. The algorithm repeats from Step 1 until all the gates of the input circuit are appended to the output one.

It is necessary to mention that **the input *placedCircuit* is not modified during the routing procedure**. The heuristic works on a copy of the original quantum circuit DAG to avoid modifying it.

5.3 Layout Synthesis Tool - Routing

Besides offering a **flexible python library** for integrating the layout synthesis phase in any quantum compilation toolchain, an additional set of scripts was developed, composing the **layout synthesis tool**: it is a **command-line application** (console application), designated to the end-user for performing the placement and routing operations without having a programming knowledge.

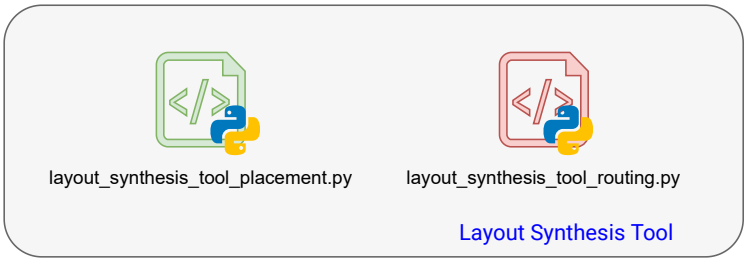

Figure 5.11. Representation of the python scripts composing the implemented layout synthesis tool. The script performing the routing phase is highlighted in red.

The layout synthesis tool is composed of two scripts:

layout_synthesis_tool_placement.py: it is a command-line application allowing the user to **apply all the placement algorithms described in Chapter 4** to any quantum circuit, targeting a superconducting, ion trap, NMR or quantum dot NISQ device.

layout_synthesis_tool_routing.py: it is a command-line application allowing the user to **apply all the routing algorithms described in this chapter** to any quantum circuit, targeting a superconducting, ion trap, NMR or quantum dot NISQ device.

5.3.1 Routing tool overview

Inputs: all the inputs must be passed as **console arguments** to the python script *layout_synthesis_tool_routing.py*. The inputs are divided into two categories:

Required arguments: mandatory for performing the routing.

- **inputQasmFile:** the relative (to the working directory) or absolute path to the OpenQASM 2.0 description of the placed quantum circuit, target of the routing procedure. Any quantum gates available in the OpenQASM 2.0 language can be used. All the **barriers and comments** inside the description are ignored and not included in the output routed circuit.

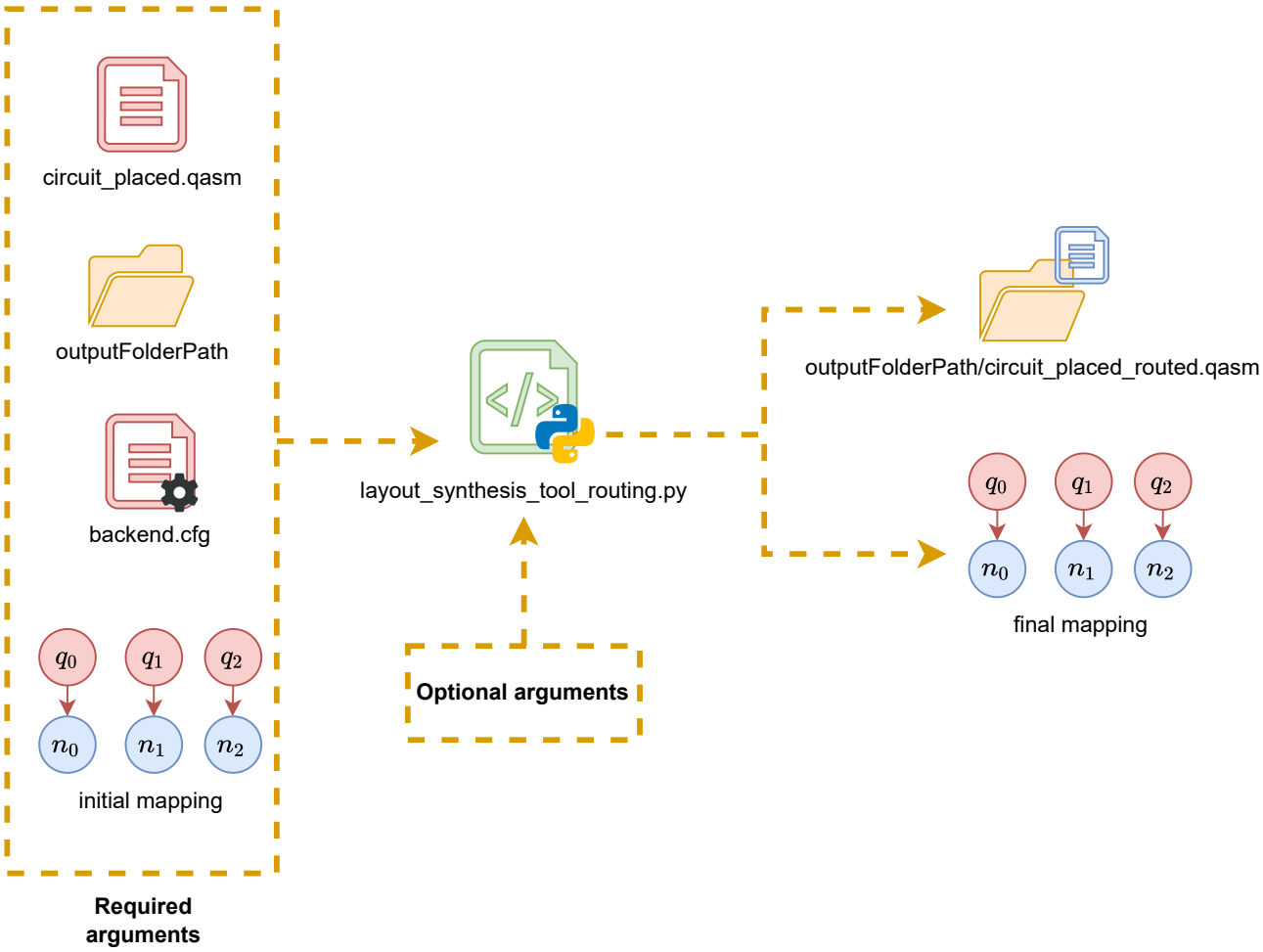

Figure 5.12. Schematic representation of the inputs and outputs of the layout synthesis routing tool. The inputs of the tool are divided into two categories: required and optional, as explained in Section 5.3.1. The outputs of the tool are: the routed quantum circuit and the final mapping.

- **outputQasmFolder:** the relative (to the working directory) or absolute path to the folder where to store the output OpenQASM 2.0 description of the quantum circuit, after the routing operation is performed.
- **backendConfigurationFile:** the relative (to the working directory) or absolute path to the .cfg configuration file describing the NISQ device target of the routing step. This can be a superconducting, NMR, ion trap or quantum dot device. A detailed explanation on how to model a generic device using a configuration file is reported in Section 6.1.
- **initialMapping:** the initial mapping applied during the placement step.

Optional arguments: for setting all the configurable parameters. Any missing optional argument is set to a default value.

- **-a:** the routing algorithm to use. Available strategies are: *Basic Routing*, *Hardware-Aware Routing*, *Hardware-Aware Routing Smart*. It is set to *Basic Routing* by default.
- **–swapNumberWeight:** the coefficient to be multiplied by the S matrix in the D matrix computation (used for the *Hardware-Aware Routing* and *Hardware-Aware Routing Smart* algorithms). For a detailed explanation of these routing strategies, see Chapter 6. It is set to 0.5 by default.
- **–swapErrorWeight:** the coefficient to be multiplied by the E matrix in the D matrix computation (used for the *Hardware-Aware Routing* and *Hardware-Aware Routing Smart* algorithms). For a detailed explanation of these routing strategies, see Chapter 6. It is set to 0.5 by default.
- **–swapTimeWeight:** the coefficient to be multiplied by the T matrix in the D matrix computation (used for the *Hardware-Aware Routing* and *Hardware-Aware Routing Smart* algorithms). For a detailed explanation of these routing strategies, see Chapter 6. It is set to 0 by default.
- **–isRZvirtual:** used only for NMR, quantum dot and ion trap technologies. *True* if the RZ gates are implemented virtually, *False* otherwise (used for the *Hardware-Aware Routing* and *Hardware-Aware Routing Smart* algorithms). For a detailed explanation of these routing strategies, see Chapter 6. It is set to *False* by default.
- **–heuristic:** the heuristic cost function to estimate the cost of a swap gate. Available options are: *basic* and *lookahead* (used for the *Hardware-Aware Routing* algorithm). For a detailed explanation of this routing strategy, see Section 6.2. It is set to *basic* by default.

- –**lookaheadDepth:** the number of layers to use for the lookahead layer (used for the *Hardware-Aware Routing* and *Hardware-Aware RoutingSmart* algorithms). For a detailed explanation of these routing strategies, see Chapter 6. It is set to 20 by default.
- –**lookaheadWeight:** the weight parameter specifying the impact of the lookahead layer in the *lookahead* heuristic cost function (used for the *Hardware-Aware Routing* and *Hardware-Aware Routing Smart* algorithms). For a detailed explanation of these routing strategies, see Chapter 6. It is set to 0.5 by default.
- –**translateSwap:** if *True* the swap gates are translated based on the native gates available in the backend quantum technology (CX gates for superconducting and ion trap devices, CZ gates for NMR and quantum dot ones). They are not translated otherwise. It is set to *False* by default.
- –**CZtoRZZ:** if *True*, for NMR and quantum dot technologies, during the swap gate decomposition the CZ gates are decomposed according to Figure 6.6. They are not decomposed otherwise. It is set to *False* by default.
- –**CXtoRXX:** if *True*, for ion trap technology, during the swap gate decomposition the CX gates are decomposed according to Figure 6.14 (with the optional parameter v set to 1). They are not further decomposed otherwise. It is set to *False* by default.

Outputs: the routing step produces two outputs:

- **outputQasmFile:** the OpenQASM 2.0 description of the routed quantum circuit. The file is written inside the *outputQasmFolder* directory.
- **finalMapping:** the final logical to physical qubits mapping, reached dynamically starting from the initial mapping after all the swap gates added by the routing procedure are executed.

In Figure 5.12 is shown a schematic illustrating the inputs and outputs of the layout synthesis routing tool.

Chapter 6

Hardware-Aware routing strategies

As already explained in Section 1.2, today's quantum computing devices are labelled as **NISQ**, where the "N" stands for **noisy**. Indeed, the noise is limiting their computational capabilities. For this reason, it is essential that during the layout synthesis, not only the target coupling-constraint are satisfied, but this task must be accomplished side by side with an **optimization** on figure of merits such as: **number of additional gates**, **final quantum circuit fidelity**, **final quantum circuit execution time**.

The *BasicRouting* algorithm, presented in Section 5.2.3, is able to perform the routing step targeting any quantum technology, but its **simplistic nature** is not best suited for performing strong optimisations. Moreover, the algorithm is **suited for routing non-fully-connected topologies**: no modifications are performed when targeting a fully-connected device, because all of the two-qubits gates composing the input circuit are respecting the target coupling-constraint.

The need for a smarter routing heuristic for the presented work, combined with the necessity for NISQ devices of optimising as much as possible the previously cited main figure of merits, led to the implementation of two **hardware-aware routing strategies**. These algorithms were already underlined in the presentation of the *Router* class in Section 5.2.1, that is:

Hardware-aware routing: it is the implementation of the routing algorithm presented in [42]. The algorithm is suited for targeting non-fully-connected topologies, aiming at reaching a better optimisation in respect to the simplest *BasicRouting* algorithm.

Hardware-aware routing smart: it is a smart adaptation of the original routing algorithm presented in [42] suited for targeting fully-connected-topologies. This was required for solving the problem explained in the introduction, that

is, even if the coupling-contraints are satisfied, trying to optimise the final quantum circuit.

These strategies are labelled as **hardware-aware** because they are exploiting the provided **calibration data** of the target quantum computing device, in order to make smarter decisions during the routing process. The calibration data is a **table** containing the main properties regarding the nodes and the nodes' interactions of the target NISQ device. Table 6.1 is an example of the calibration data for the *ibmq_lima* superconducting device [13].

Backend name	ibmq_lima
Number of nodes	5
n_0 - n_1 CNOT error rate	4.772e-3
n_1 - n_0 CNOT error rate	4.772e-3
...	...
n_0 - n_1 CNOT gate time	305.778 ns
n_1 - n_0 CNOT gate time	341.333 ns
...	...
T1 n_0	123.54 µs
T2 n_0	165.6 µs
...	...

Table 6.1. In this table, a portion of the properties extrapolated from the *ibmq_lima* [13] superconducting device calibration data are presented.

The remaining of this chapter is organised as follows: Section 6.1 is devoted to explaining how a NISQ device is abstracted in the proposed layout synthesis library, Section 6.2 explains the implementation of the original *HaRoutingAlgorithm* that in the original paper [42] is suited for targeting superconducting quantum devices with a non-fully-connected topology; Section 6.3.2 explains the extension of the original algorithm for working with non-fully-connected topologies.

6.1 The Backend class

This section is devoted to explaining how a NISQ device is abstracted in the proposed layout synthesis library. Indeed, this is required not only for understanding the implementation of the hardware-aware routing strategies, but also for comprehending how the original algorithm [42], mainly intended for superconducting devices, **was adapted to target all the available quantum technologies in the presented library**. For this reason, it is mandatory to understand how a NISQ device is modelled.

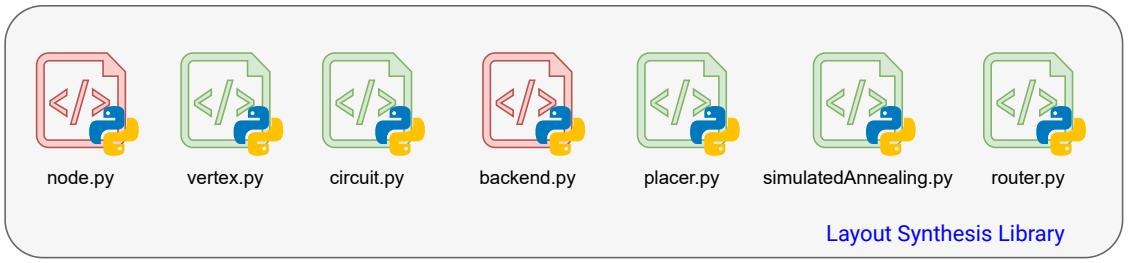

Figure 6.1. Representation of the python scripts composing the implemented layout synthesis library. The ones containing the *Backend* and *Node* classes are highlighted in red.

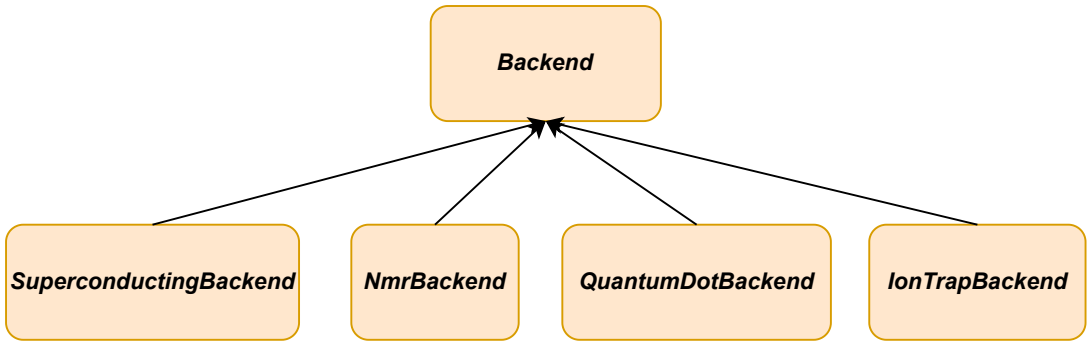

Figure 6.2. The *SuperconductingBackend*, *NmrBackend*, *QuantumDotBackend* and *IonTrapBackend* classes are all children of the *Backend* class

This abstraction happens using the ***Backend* class** inside the *backend.py* python script composing the library. It provides all the necessary functionalities required for obtaining the necessary backend information, both for checking the allowed two-qubits interactions, and also for extracting the **main hardware properties** (for example interaction fidelity and interaction execution time) required by the **hardware-aware routing algorithms** (Section 6.2 and Section 6.3.2) and **hardware-aware simulated annealing placement algorithm** (Section 4.3.4). Specifically, the *Backend* class (and thus also all the technology-specific *Backend* classes) provides to the *Placer* and *Router* classes the information on the allowed two-nodes interactions, to correctly perform the layout synthesis phase (see Section 3.1 for further details).

The *Backend* class is **mostly generic**. Indeed, it is independent of a specific quantum technology and can be used for modelling the allowed two-qubits interactions of a target quantum computing device, without error rates and gate times information. It is used for defining the main attributes and methods **inherited** by

technology-specific backend classes.
These backend classes are: ***SuperconductingBackend*** modelling superconducting NISQ devices, see Section 6.1.1 for details; ***NmrBackend*** modelling NMR NISQ devices, see Section 6.1.2 for details; modelling quantum dot NISQ devices, see Section 6.1.3 for details; ***IonTrapBackend*** modelling ion trap NISQ devices, see Section 6.1.4 for details.

The generic attributes defined for all the available quantum technologies are:

n_nodes: the number of nodes (physical qubits) composing the NISQ device.

nodes: list of the nodes composing the NISQ device. Each node is an instance of the ***Node*** class, used for modelling a physical qubit.

technology: string specifying the technology used for implementing the NISQ device (for generic backends, it is *None*).

The generic methods defined for all the available quantum technologies are:

drawCouplingGraph(): it draws the coupling-graph of the backend with Matplotlib [54].

getDistance(node1, node2): it extracts the shortest path length (number of edges) between two nodes in the coupling-graph.

isFullyConnected(): it checks if the NISQ device has a fully-connected topology (all interactions are allowed) or not.

6.1.1 Superconducting technology

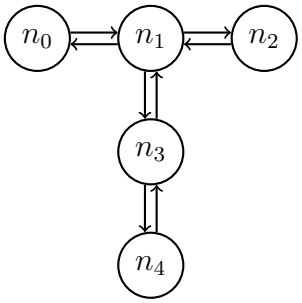

Figure 6.3. Directed coupling-graph representing the *ibmq_lima* IBMQ superconducting quantum device [13]. Each vertex of the graph represents a physical qubit. Each edge represents an allowed two-qubits interaction, and the graph is a directed one since the superconducting native two-qubit gate (the CX gate) is generally not symmetric.

```
[Basic]
technology = S
n_nodes = 5

[CXErrorRate]
0-1 = 0.005654368978596724
1-0 = 0.005654368978596724
1-2 = 0.006470157327009479
1-3 = 0.013136050396215682
2-1 = 0.006470157327009479
3-1 = 0.013136050396215682
3-4 = 0.01751329647388042
4-3 = 0.01751329647388042

[CXGateTime]
0-1 = 305.77777777777777e-9
1-0 = 341.3333333333333e-9
1-2 = 334.22222222222223e-9
1-3 = 497.7777777777778e-9
2-1 = 298.66666666666663e-9
3-1 = 462.2222222222222e-9
3-4 = 519.1111111111111e-9
4-3 = 483.5555555555555e-9
```

Figure 6.4. ibmq_lima.cfg configuration file modelling the *ibmq_lima* IBMQ superconducting quantum device [13]. All the provided gate times are expressed in ns.

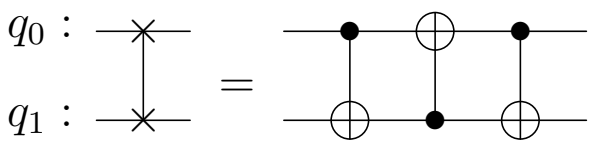

Figure 6.5. Swap gate decomposition using three consecutive CX gates.

General information regarding the implementation of a quantum computing device resorting to superconducting technology is provided in Section 1.4. NISQ devices implemented using this technology are generally **non-fully-connected**, for this reason the **layout synthesis procedure is mandatory** for solving the coupling-constraint.

The *SuperconductingBackend* class was implemented for modelling superconducting quantum computing devices. Being a *Backend*, it provides to the *Placer* and *Router* classes the information on the allowed two-nodes interactions, mandatory for performing the layout synthesis. Furthermore, **being a technology-specific *Backed* class**, it provides additional information such as the gate time and error rate for an allowed two-nodes interaction. This additional information is required by the *Hardware-Aware* and *Hardware-Aware smart* routing algorithms, for the pre-processing phase (see Section 6.2.1).
The main attributes composing this class are:

couplingGraph: it is a **directed graph** that is used for representing the coupling-graph of the architecture. The reason why a directed graph is required is that in superconducting technology, the error rate and gate time for a two-qubits interaction change when the control and target nodes are exchanged (the interaction is not symmetric). An example of coupling-graph representation for a superconducting device is the one depicted in Figure 6.3.

technology: string specifying the technology used for implementing the NISQ device. It is "S" for superconducting quantum computers.

The main methods offered by this class are:

getCXErrorRate(controlNode, targetNode): extracts the error rate for applying the quantum gate: "CX controlNode, targetNode". The CX gate for superconducting quantum devices is not symmetric, thus different error rates may be obtained if the nodes are inverted.

getCXGateTime(controlNode, targetNode): extracts the gate time for applying the quantum gate: "CX controlNode, targetNode". The CX gate for superconducting quantum devices is not symmetric, thus different gate times may be obtained if the nodes are inverted.

getSwapErrorRate(node1, node2): extracts the error rate for applying the swap gate to *node1* and *node2*.

The error rate of a swap gate is computed supposing that it is implemented as 3 consecutive CX gates, that are the native two-qubits gates available for superconducting technology. The decomposition of the swap gate using 3 CX gates is shown in Figure 6.5.
This error rate is thus dependent on the error rate for executing a CX between

the two nodes (considering that for superconducting devices, the CX gate is not symmetric). The symmetry of the swap gate is exploited, and the error rate is computed in the best case scenario (in accordance to [42, Sec. 3.A]). For best case scenario, it is intended:

$$SWAP_{errorRate} = 1 - \Big(CXsuccessRate_{node1-node2} \cdot CXsuccessRate_{node2-node1} \cdot \max\left(CXsuccessRate_{node1-node2}, CXsuccessRate_{node2-node1}\right) \Big) \quad (6.1)$$

getSwapGateTime(node1, node2): extracts the gate time for applying the swap gate to *node1* and *node2*.

The gate time of a swap gate is computed supposing that it is implemented as 3 consecutive CX gates, that are the native two-qubits gates available for superconducting technology. The decomposition of the swap gate using 3 CX gates is shown in Figure 6.5.
This gate time is thus dependent on the gate time for executing a CX between the two nodes (considering that for superconducting devices, the CX gate is not symmetric). The symmetry of the swap gate is exploited, and the gate time is computed in the best case scenario (the same computation done in [42, Sec. 3.A]). For best case scenario, it is intended:

$$SWAP_{gateTime} = CXgateTime_{node1-node2} + CXgateTime_{node2-node1} + \min\left(CXgateTime_{node1-node2}, CXgateTime_{node2-node1}\right) \quad (6.2)$$

Besides instantiating a superconducting backend class manually, by passing all the required parameters to its constructor, the layout synthesis library offers a method, ***fromConfigurationFile(cfgFilePath)***, allowing the **automatic instantiation** of a *SuperconductingBackend* reading a **configuration file (.cfg)**. Such file contains all the figure of merits required for the correct NISQ device modelling.

In Figure 6.4 it is reported the configuration file modelling the *ibmq_lima* IBMQ supercondicting quantum device. Each superconducting configuration file is composed of three sections:

Basic: it contains the basic details of the NISQ device: the technology ("S" for superconducting devices) and the number of nodes.

CXErrorRate: it contains the allowed two-qubits interactions, as well as the error rate for each one.

CXGateTime: it contains the allowed two-qubits interactions, as well as the gate time for each one. All the gate times inside this section are expressed in ns.

All the information inside Figure 6.4 regarding device topology, CX gate time and CX error rate were retrieved from [13]. Due to the open-source nature of the IBM quantum project, that guarantees a high availability of calibration data for their superconducting devices, the proposed approach for modelling this family of NISQ device is by directly using the available data. The same strategy is also used in the original implementation of the hardware-aware routing algorithm [55], where it is possible to specify the name of an IBM Quantum backend, and the calibration data are automatically retrieved.

6.1.2 NMR technology

General information regarding the implementation of a quantum computing device resorting to liquid-state NMR technology is provided in Section 1.5. NISQ devices implemented using this technology are generally **fully-connected**, for this reason **the layout synthesis procedure is not mandatory**, because all the interactions are allowed. However, recalling that for NMR NISQ devices, the gate times and error rates are dependent on the **J-coupling parameter** (which can be seen as an indicator of the interaction strength), a layout synthesis phase modifying the input quantum circuit, preferring the **strongest interactions** inside the molecule, could lead to an output quantum circuit with an optimal total gate time and total error rate.

The *NmrBackend* class was implemented for modelling NMR quantum computing devices. Being a *Backend*, it provides to the *Placer* and *Router* classes the information on the allowed two-nodes interactions, mandatory for performing the layout synthesis. Furthermore, **being a technology-specific *Backed* class**, it provides additional information such as the gate time and error rate for an allowed two-nodes interaction. This additional information is required by the *Hardware-Aware* and *Hardware-Aware smart* routing algorithms, for the pre-processing phase (see Section 6.2.1). It is similar to the *SuperconductingBackend* class, with the addition of modelling also the single-qubit gates times. This is a requirement for computing the swap gate time according to the decompositions of Figure 6.5 and Figure 6.7.
The main attributes composing this class are:

couplingGraph: it is an **undirected graph** that is used for representing the coupling-graph of the architecture. The reason why an undirected graph is sufficient to model a NMR quantum computing device is that the gate time and error rate of the native two-qubits interaction (Rzz gate) depend only on the J-coupling constant, that is fixed for every couple of nuclei inside the molecule. This graph specifies the allowed two-qubits interactions, while also storing the **J-coupling constant** between the interacting nodes as an attribute of the edges. An example of coupling-graph representation for a NMR device is the one depicted in Figure 6.9.

technology: string specifying the technology used for implementing the NISQ device. It is "M" for NMR quantum computers.

nodesEncoding: specifies which nuclei are used for encoding a node (for example, *nodesEncoding*[0] = "13C" means that n_0 is encoded using the Carbon-13 isotope). The allowed isotopes available in the current version of the library are Hydrogen-1 (1H), Carbon-13 (13C) and Fluorine-19 (19F).

Br: RF magnetic field amplitude in T.

The main methods offered by this class are:

getJcoupling(node1, node2): extracts the J-coupling constant for the *node1-node2* interaction.

getSingleQubitGateTime(node, theta, isRZ, isRZvirtual): extracts the gate time required for applying a Rx, Ry or Rz gate on the target node. Its inputs are:

- ***node:*** the node target of the single-qubit gate.
- ***theta:*** the angle of rotation.
- ***isRZ:*** *False* for Rx and Ry gates, *True* for Rz gates.
- ***isRZvirtual:*** *True* if the RZ gates are implemented virtually, *False* otherwise.

The gate time for Rx and Ry gates is computed with:

$$\tau = \frac{\theta}{\omega_{*,k}}, \tag{6.3}$$

where θ is the angle of ration and $\omega_{*,k}$ is the amplitude of the RF field computed as:

$$\omega_{*,k} = \gamma_n \cdot B_r, \tag{6.4}$$

where γ_n is the nuclear gyromagnetic ratio and B_r is the alternating magnetic field amplitude.

The Rz gate time is instead computed supposing the decomposition:

$$R_z(\alpha) = R_x(\frac{\pi}{2})R_y(\alpha)R_x(-\frac{\pi}{2}). \tag{6.5}$$

Thus, summing the gate times for the sequential Rx and Ry gates computed using Equation (6.3). If the Rz gates are instead implemented virtually, the gate time is 0 s.

getCZGateTime(controlNode, targetNode, isRZvirtual): extracts the gate time for applying the quantum gate: "CZ controlNode, targetNode". The CZ gate for NMR quantum devices is completely symmetric, thus the same gate times are obtained if the nodes are inverted.

The gate time of the CZ is computed supposing the gate is decomposed as shown in Figure 6.6.

The CZ gate time is thus computed summing:

- The gate time for applying a Rz gate to the control node. In case the Rz gates are implemented virtually, the gate time is considered 0 s. Otherwise, if the Rz gates are not implemented virtually, the gate time is computed resorting to the *getSingleQubitGateTime(node, theta, isRZ, isRZvirtual)* method.
- The gate time for applying a Rz gate to the target node. In case the Rz gates are implemented virtually, the gate time is considered 0 s. Otherwise, if the Rz gates are not implemented virtually, the gate time is computed resorting to the *getSingleQubitGateTime(node, theta, isRZ, isRZvirtual)* method.
- The time for applying the Rzz gate to the control and target node, computed as $\tau = \left|\frac{1}{2J}\right|$, where J is the J-coupling constant between the interacting nodes in Hz.

getCXGateTime(controlNode, targetNode, isRZvirtual): extracts the gate time for applying the quantum gate: "CX controlNode, targetNode". The CX gate for NMR quantum devices is not symmetric, thus different gate times may be obtained if the nodes are inverted.

The gate time of the CX is computed supposing the gate is decomposed as shown in Figure 6.7.

The CX gate time is thus computed summing:

- Two times the gate time for applying a Hadamard gate to the target node. There are two possible ways for computing it, depending on if the Rz gates are implemented virtually or not. In the first scenario, the gate is supposed to be decomposed according to Figure 6.10, in the latter scenario instead it is supposed to be decomposed as depicted in Figure 6.8.
- The gate time for applying a CZ gate, computed resorting to the *getCZGateTime(controlNode, targetNode, isRZvirtual)* method.

getSwapGateTime(node1, node2, isRZvirtual): extracts the gate time for applying the swap gate to *node1* and *node2*. The gate time of a swap gate is computed supposing that it is implemented as three consecutive CX gates.

The decomposition of the swap gate using three CX gates is shown in Figure 6.5.
This gate time is thus dependent on the gate time for executing a CX between the two nodes (considering that for NMR devices, the CX gate is not symmetric) computed resorting to the *getCXGateTime(controlNode, targetNode, isRZvirtual)* method. Indeed, the gate time is also dependent on if the Rz gates are implemented virtually or not since this changes the CX gate time. The symmetry of the swap gate is exploited, and the gate time is computed in the best case scenario (the same computation done in [42, Sec. 3.A]). For the best case scenario swap gate time computation, the Equation (6.2) is used.

getSwapErrorRate(node1, node2, isRZvirtual): extracts the error rate for applying the swap gate to *node1* and *node2*.

The error rate of a swap gate is computed supposing that it is implemented as three consecutive CX gates. The decomposition of the swap gate using three CX gates is shown in Figure 6.5.
The error is also dependent on if the Rz gates are implemented virtually or not. The symmetry of the swap gate is exploited, and the error rate is considered identical even if the interacting nodes are exchanged.

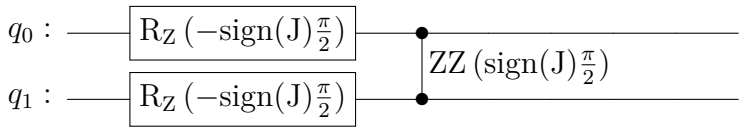

Figure 6.6. Implementation of the CZ gate for NMR and quantum dot quantum technologies, using the available native set of gates. J is the J-coupling or Exchange-Interaction (for quantum dots) constant between the nodes mapped to the logical qubits q_0 and q_1 .

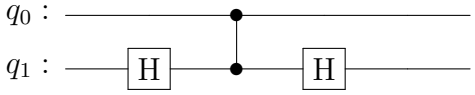

Figure 6.7. Implementation of the CX gate for NMR and quantum dot quantum technologies, decomposing it using H and CZ gates.

Besides instantiating a NMR backend class manually, by passing all the required

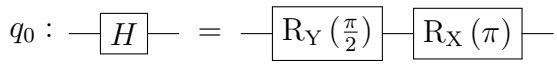

Figure 6.8. Decomposition of the Hadamard gate using Ry and Rx rotations. This decomposition is employed when the Rz gates are not implemented virtually, this is convenient remembering that in this case the Rz gate times are computed following the decomposition depicted in Equation (6.5).

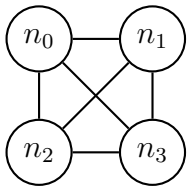

Figure 6.9. Undirected coupling-graph representing the *crotonic_acid* NMR quantum device [16, Sec. 8.3]. Each node represents a physical qubit and each edge represents an allowed two-qubits interaction. The architecture is fully-connected but each interacting has a different J-coupling constant.

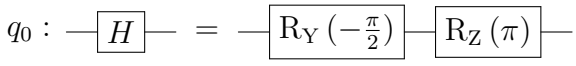

Figure 6.10. Decomposition of the Hadamard gate using Ry and Rz rotations. This decomposition is employed when the Rz gates are implemented virtually to exploit their 0 s gate times.

```
[Basic]
technology = M
n_nodes = 4
nodesEncoding = 13C, 13C, 13C, 13C
Br = 2.25e-6
minJ = 0.0

[JCoupling]
0-1 = 72.36
0-2 = 1.18
0-3 = 7.04
1-2 = 69.72
1-3 = 1.46
2-3 = 41.64

[SwapErrorRate_RzVirtual]
0-1 = 0.020083505167633464
0-2 = 0.21272011748545017
0-3 = 0.05981919466011165
1-2 = 0.017818414934632876
1-3 = 0.21582003282713824
2-3 = 0.01659076432867268

[SwapErrorRate_NotRzVirtual]
0-1 = 0.12461441845941845
0-2 = 0.39120315984675824
0-3 = 0.12999113996228395
1-2 = 0.1364459093362509
1-3 = 0.34847636244448155
2-3 = 0.07858987866826672
```

Figure 6.11. crotonic_acid.cfg configuration file modelling the *crotonic_acid* molecule. All the information regarding *n_nodes*, *nodesEncoding*, *Br* and the J-coupling constants, were retrieved from [16, Sec. 8.3]. The swap gate error rates, with the Rz gates implement virtually and not implemented virtually, were computed performing a simulation using MATLAB QuanTO [56].

parameters to its constructor, the layout synthesis library offers a method, ***fromConfigurationFile(cfgFilePath)***, allowing the **automatic instantiation** of a *NmrBackend* reading a **configuration file (.cfg)**. Such file contains all the figure of merits required for the correct NISQ device modelling.

In Figure 6.11, is reported the configuration file modelling the *crotonic_acid* NMR quantum device, where all the molecule properties are retrieved from [16, Sec. 8.3].
Each NMR configuration file is composed of four sections:

Basic: it contains the basic details of the NISQ device:

- **technology**: "M" for NMR quantum computers.
- **n_nodes**: the number of nodes of the device.
- **nodesEncoding**: the isotopes used for encoding each node, starting from n up to n_{n_nodes}.
- **Br**: alternating magnetic field amplitude in T.
- **minJ**: the minimum J-coupling constant threshold in absolute value in Hz. All the interactions having a J-coupling constant such that $J < |minJ|$ are discarded since the interaction is considered too weak. This parameter can also be set to 0 Hz, thus accepting all the interactions.

JCoupling: it contains the allowed two-qubits interactions, as well as the J-coupling constant for each interaction in Hz.

SwapErrorRate_RzVirtual: it contains the swap gate error rates for all the valid two-nodes interactions, when the Rz gates are implemented virtually.

SwapErrorRate_NotRzVirtual: it contains the swap gate error rates for all the valid two-nodes interactions, when the Rz gates are not implemented virtually.

For computing the swap gate error rates of the *crotonic_acid* backend, reported in Figure 6.11, a python script was developed: *compute_NMR_Swap_Error_Rates.py*. For each pair of nodes of the backend (thus, for each possible swap gate), the script performs the following steps:

1. Construct a quantum circuit for generating a random superposition state between the two nodes to be swapped.
2. This quantum circuit is simulated using the **MATLAB QuanTO** [56] simulator, having the *crotonic_acid* as target backend. The output density matrix ρ' is retrieved from the simulator.
3. Construct the ideal swap gate unitary matrix U_{swap} and apply it to the density matrix ρ':

$$\rho = U_{swap} \cdot \rho' \cdot U_{swap}^{\dagger}$$

4. Add the swap gate decomposition to the quantum circuit generated from Step 1. This decomposition depends on if the Rz gates are implemented virtually or not: for the latter scenario, all the Rz gates are further decomposed to Rx and Ry rotations, according to Equation (6.5).

5. Simulate this new quantum circuit using the MATLAB QuanTO simulator. The output density matrix σ is retrieved from the simulator.

6. Compute the **fidelity** [5, Sec. 9.2.2] $F(\rho, \sigma)$, measuring the closeness between the two density matrices, as:

$$F(\rho, \sigma) = \mathrm{tr}\left\{\sqrt{\rho^{1/2}\sigma\rho^{1/2}}\right\}$$

7. The computed error rate for the current simulation is $1 - F(\rho, \sigma)$. The script repeats from Step 1 for a configurable number of simulation shots. The average measured error rate is the final one.

6.1.3 Quantum dots technology

General information regarding the implementation of a quantum computing device resorting to quantum dot technology is provided in Section 1.6. NISQ devices implemented using this technology are generally **non-fully-connected**, for this reason the **layout synthesis procedure is mandatory** for solving the coupling-constraint.

The *QuantumDotBackend* class was implemented for modelling quantum dot quantum computing devices. Being a *Backend*, it provides to the *Placer* and *Router* classes the information on the allowed two-nodes interactions, mandatory for performing the layout synthesis. Furthermore, **being a technology-specific *Backed* class**, it provides additional information such as the gate time and error rate for an allowed two-nodes interaction. This additional information is required by the *Hardware-Aware* and *Hardware-Aware smart* routing algorithms, for the pre-processing phase (see Section 6.2.1). It is similar to the *SuperconductingBackend* class, with the addition of modelling also the single-qubit gates times. This is a requirement for computing the swap gate time according to the decompositions of Figure 6.5 and Figure 6.7.
The main attributes composing this class are:

couplingGraph: it is an **undirected graph** that is used for representing the coupling-graph of the architecture. The reason why an undirected graph is sufficient to model a quantum dot quantum computing device is that the gate time and error rate of the native two-qubits interaction (Rzz gate) depend only on the Exchange-Interaction constant, that is fixed for every couple of nodes. This graph specifies the allowed two-qubits interactions, while also

storing the **Exchange-Interaction constant** between the interacting nodes as an attribute of the edges. An example of coupling-graph representation for a quantum dot device is the one depicted in Figure 6.12.

technology: string specifying the technology used for implementing the NISQ device. It is "Q" for quantum dot quantum computers.

RxyGateTimes_halfpi: the tuple of gate times in s for applying a Rx or Ry gate with an angle of $\frac{\pi}{2}$ to each node of the quantum dot backend.

The main methods offered by this class are:

getExchangeInteraction(node1, node2): extracts the Exchange-Interaction constant for the *node1-node2* interaction.

getSingleQubitGateTime(node, theta, isRZ, isRZvirtual): extracts the gate time required for applying a Rx, Ry or Rz gate on the target node. Its inputs are:

- ***node:*** the node target of the single-qubit gate.
- ***theta:*** the angle of rotation.
- ***isRZ:*** *False* for Rx and Ry gates, *True* for Rz gates.
- ***isRZvirtual:*** *True* if the RZ gates are implemented virtually, *False* otherwise.

The gate time for Rx and Ry gates is computed with:

$$\tau(\theta) = \left| \frac{\tau(Rxy(\frac{\pi}{2})) \cdot \theta}{\frac{\pi}{2}} \right|, \tag{6.6}$$

where $\tau(Rxy(\frac{\pi}{2}))$ is extracted from *RxyGateTimes_halfpi*.

The Rz gate time is instead computed supposing the decomposition of Equation (6.5). Thus, summing the gate times for the sequential Rx and Ry gates computed using Equation (6.6). If the Rz gates are instead implemented virtually, the gate time is 0 s.

getCZGateTime(controlNode, targetNode, isRZvirtual): extracts the gate time for applying the quantum gate: "CZ controlNode, targetNode". The CZ gate for quantum dot quantum devices is completely symmetric, thus the same gate times are obtained if the nodes are inverted.

The gate time of the CZ is computed supposing the gate is decomposed as shown in Figure 6.6.

The CZ gate time is thus computed summing:

- The gate time for applying a Rz gate to the control node. In case the Rz gates are implemented virtually, the gate time is considered 0 s. Otherwise, if the Rz gates are not implemented virtually, the gate time is computed resorting to the *getSingleQubitGateTime(node, theta, isRZ, isRZvirtual)* method.
- The gate time for applying a Rz gate to the target node. In case the Rz gates are implemented virtually, the gate time is considered 0 s. Otherwise, if the Rz gates are not implemented virtually, the gate time is computed resorting to the *getSingleQubitGateTime(node, theta, isRZ, isRZvirtual)* method.
- The time for applying the Rzz gate to the control and target node, computed as $\tau = \left|\frac{1}{2E}\right|$, where E is the Exchange-Interaction constant between the interacting nodes in Hz.

getCXGateTime(controlNode, targetNode, isRZvirtual): extracts the gate time for applying the quantum gate: "CX controlNode, targetNode". The CX gate for quantum dot quantum devices is not symmetric, thus different gate times may be obtained if the nodes are inverted.

The gate time of the CX is computed supposing the gate is decomposed as shown in Figure 6.7.

The CX gate time is thus computed summing:

- Two times the gate time for applying a Hadamard gate to the target node. There are two possible ways for computing it, depending on if the Rz gates are implemented virtually or not. In the first scenario, the gate is supposed to be decomposed according to Figure 6.10, in the latter scenario instead it is supposed to be decomposed as depicted in Figure 6.8.
- The gate time for applying a CZ gate, computed resorting to the *getCZGateTime(controlNode, targetNode, isRZvirtual)* method.

getSwapGateTime(node1, node2, isRZvirtual): extracts the gate time for applying the swap gate to *node1* and *node2*. The gate time of a swap gate is computed supposing that it is implemented as three consecutive CX gates. The decomposition of the swap gate using three CX gates is shown in Figure 6.5.
This gate time is thus dependent on the gate time for executing a CX between the two nodes (considering that for quantum dot devices, the CX gate is not symmetric) computed resorting to the *getCXGateTime(controlNode, targetNode, isRZvirtual)* method. Indeed, the gate time is also dependent on if the Rz gates are implemented virtually or not, since this changes the CX gate time. The symmetry of the swap gate is exploited, and the gate time is computed in

the best case scenario (the same computation done in [42, Sec. 3.A]). For the best case scenario swap gate time computation, the Equation (6.2) is used.

getSwapErrorRate(node1, node2, isRZvirtual): extracts the error rate for applying the swap gate to *node1* and *node2*.

The error rate of a swap gate is computed supposing that it is implemented as three consecutive CX gates. The decomposition of the swap gate using three CX gates is shown in Figure 6.5.
The error is also dependent on if the Rz gates are implemented virtually or not. The symmetry of the swap gate is exploited, and the error rate is considered identical even if the interacting nodes are exchanged.

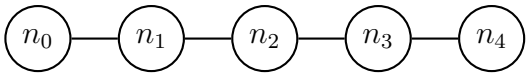

Figure 6.12. Undirected coupling-graph representing the *5nodes_quantum_dot* quantum device. Each node represents a physical qubit and each edge represents an allowed two-qubits interaction. The architecture's topology is linear.

Besides instantiating a quantum dot backend class manually, by passing all the required parameters to its constructor, the layout synthesis library offers a method, ***fromConfigurationFile(cfgFilePath)***, allowing the **automatic instantiation** of a *QuantumDotBackend* reading a **configuration file (.cfg)**. Such file contains all the figure of merits required for the correct quantum dot device modelling.

In Figure 6.13, is reported the configuration file modelling the *5nodes_quantum_dot* quantum device, constructed expanding the two-nodes device presented in [17]. Each quantum dot configuration file is composed of five sections:

Basic: it contains the basic details of the NISQ device:

- **technology**: "Q" for quantum dot quantum computers.
- **n_nodes**: the number of nodes of the device.
- **minExchangeInteraction**: the minimum Exchange-Interaction constant threshold in absolute value in Hz. All the interactions having an Exchange-Interaction constant such that $E < |minE|$ are discarded since the interaction is considered too weak. This parameter can also be set to 0 Hz, thus accepting all the interactions.

RxyGateTime_halfpi: it contains the gate times for applying a Rx or Ry gate to any node of the quantum computing device.

Exchange-Interaction: it contains the allowed two-qubits interactions, as well as the Exchange-Interaction constant for each interaction in Hz.

```
[Basic]
technology = Q
n_nodes = 5
minExchangeInteraction = 0.0

[RxyGateTime_halfpi]
0 = 250e-9
1 = 250e-9
2 = 250e-9
3 = 250e-9
4 = 250e-9

[ExchangeInteraction]
0-1 = 3.3333e6
1-2 = 3.0303e6
2-3 = 3.5211e6
3-4 = 2.7473e6

[SwapErrorRate_RzVirtual]
0-1 = 0.02846304062851257
1-2 = 0.031122655777865327
2-3 = 0.02945493745835359
3-4 = 0.02857003229001298

[SwapErrorRate_NotRzVirtual]
0-1 = 0.24301050424747428
1-2 = 0.264886331325479
2-3 = 0.26083669687701094
3-4 = 0.23818706397897216
```

Figure 6.13. *5nodes_quantum_dot.cfg* configuration file modelling the *5nodes_quantum_dot* quantum device depicted in Figure 6.12.

SwapErrorRate_RzVirtual: it contains the swap gate error rates for all the valid two-nodes interactions, when the Rz gates are implemented virtually.

SwapErrorRate_NotRzVirtual: it contains the swap gate error rates for all the valid two-nodes interactions, when the Rz gates are not implemented virtually.

For computing the swap gate error rates of the *5nodes_quantum_dot* backend, reported in Figure 6.13, a python script was developed: *compute_QuantumDot_Swap_Error_Rates.py*. It uses the **MATLAB QuanTO** simulator, following the same procedure explained in Section 6.1.2.

6.1.4 Ion Trap technology

General information regarding the implementation of a quantum computing device resorting to ion trap technology is provided in Section 1.7. In particular, the presented work aims at modelling a **single-trap** NISQ device with a linear chain of ions. Other architectures such as Quantum Charge Coupled Device [18, Sec. 1] are not part of the presented layout synthesis library. NISQ devices implemented using this technology are generally **fully-connected**, for this reason **the layout synthesis procedure is not mandatory**, because all the interactions are allowed. However, different pair of ions may have different gate times and error rates. A layout synthesis phase modifying the input quantum circuit, preferring the **better interactions**, could lead to an output quantum circuit with an optimal total gate time and total error rate.

The *IonTrapBackend* class was implemented for modelling ion trap quantum computing devices. Being a *Backend*, it provides to the *Placer* and *Router* classes the information on the allowed two-nodes interactions, mandatory for performing the layout synthesis. Furthermore, **being a technology-specific *Backed* class**, it provides additional information such as the gate time and error rate for an allowed two-nodes interaction. This additional information is required by the *Hardware-Aware* and *Hardware-Aware smart* routing algorithms, for the pre-processing phase (see Section 6.2.1). It is similar to the *SuperconductingBackend* class, with the addition of modelling also the single-qubit gates times. This is a requirement for computing the swap gate time according to the decompositions of Figure 6.5 and Figure 6.14.
The main attributes composing this class are:

couplingGraph: it is an **undirected graph** that is used for representing the coupling-graph of the architecture. The reason why an undirected graph is sufficient to model an ion trap quantum computing device is that the gate time and error rate, of the native two-qubits interaction (Rxx gate), are fixed for every couple of interacting nodes. This graph specifies the allowed two-qubits interactions, while also storing the **sign of the phase** χ (see Section 1.7

for further details), the **gate time of the Rxx gate** and the **error rate of the Rxx gate** between the interacting nodes as an attribute of the edges.

technology: string specifying the technology used for implementing the NISQ device. It is "I" for ion trap quantum computers.

RxyGateTimes_halfpi: the tuple of gate times in s for applying a Rx or Ry gate with an angle of $\frac{\pi}{2}$ to each node of the ion trap backend.

RxyErrorRates_halfpi: the tuple of error rates for applying a Rx or Ry gate with an angle of $\frac{\pi}{2}$ to each node of the ion trap backend.

The main methods offered by this class are:

getSignX(node1, node2): extracts the sign of the interaction parameter χ for the *node1-node2* interaction.

getSingleQubitGateTime(node, theta, isRZ, isRZvirtual): extracts the gate time required for applying a Rx, Ry or Rz gate on the target node. Its inputs are:

- ***node:*** the node target of the single-qubit gate.
- ***theta:*** the angle of rotation.
- ***isRZ:*** *False* for Rx and Ry gates, *True* for Rz gates.
- ***isRZvirtual:*** *True* if the RZ gates are implemented virtually, *False* otherwise.

The gate time for Rx and Ry gates is computed with:

$$\tau(\theta) = \left| \frac{\tau(Rxy(\frac{\pi}{2})) \cdot \theta}{\frac{\pi}{2}} \right|, \tag{6.7}$$

where $\tau(Rxy(\frac{\pi}{2}))$ is extracted from *RxyGateTimes_halfpi*.

The Rz gate time is instead computed supposing the decomposition of Equation (6.5). Thus, summing the gate times for the sequential Rx and Ry gates computed using Equation (6.7). If the Rz gates are instead implemented virtually, the gate time is 0 s.

getSingleQubitErrorRate(node, theta, isRZ, isRZvirtual): extracts the error rate for applying a Rx, Ry or Rz gate on the target node. Its inputs are:

- ***node:*** the node target of the single-qubit gate.
- ***theta:*** the angle of rotation.
- ***isRZ:*** *False* for Rx and Ry gates, *True* for Rz gates.

- ***isRZvirtual:*** *True* if the RZ gates are implemented virtually, *False* otherwise.

The error rate for Rx and Ry gates is computed with:

$$E(\theta) = \frac{E(Rxy(\frac{\pi}{2})) \cdot \theta}{\frac{\pi}{2}}, \tag{6.8}$$

where $E(Rxy(\frac{\pi}{2}))$ is extracted from *RxyErrorRates_halfpi*.

The Rz error rate is instead computed supposing the decomposition of Equation (6.5). Thus, according to:

$$E = 1 - RXY_{successRate}, \tag{6.9}$$

where $RXY_{successRate}$ is computed multiplying all the success rates of the sequential Rx and Ry gates of the Equation (6.5) decomposition. The success rate of a Rx or Ry gate is computed as $1 - E(Rxy(\frac{\pi}{2}))$.

If the Rz gates are instead implemented virtually, the error rate is 0.

getCXGateTime(controlNode, targetNode, isRZvirtual): extracts the gate time for applying the quantum gate: "CX controlNode, targetNode". The CX gate for ion trap quantum devices is not symmetric, thus different gate times may be obtained if the nodes are inverted.

The gate time of the CX is computed supposing the gate is decomposed as shown in Figure 6.14.

The CX gate time is thus computed summing:

- The gate time for applying a Ry gate to the control node, computed with the *getSingleQubitGateTime(node, theta, isRZ, isRZvirtual)* method.
- The gate time for applying a Rxx gate to the control and target node. This information is stored as an attribute of the edge connecting the interacting nodes in the coupling-graph.
- The gate time for applying a Rx gate to the control node, computed with the *getSingleQubitGateTime(node, theta, isRZ, isRZvirtual)* method.
- The gate time for applying a Rx gate to the target node, computed with the *getSingleQubitGateTime(node, theta, isRZ, isRZvirtual)* method.
- The gate time for applying a Ry gate to the control node, computed with the *getSingleQubitGateTime(node, theta, isRZ, isRZvirtual)* method.

getCXErrorRate(controlNode, targetNode, isRZvirtual): extracts the error rate for applying the quantum gate: "CX controlNode, targetNode". The

CX gate for ion trap quantum devices is not symmetric, thus different error rates may be obtained if the nodes are inverted.

The error rate of the CX is computed supposing the gate is decomposed as shown in Figure 6.14.

The CX error rate is thus computed as:

$$E = 1 - totSuccessRate, \tag{6.10}$$

where *totSuccessRate* is computed multiplying:

- The success rate for applying a Ry gate to the control node, computed with the *getSingleQubitErrorRate(node, theta, isRZ, isRZvirtual)* method.
- The success rate for applying a Rxx gate to the control and target node. The information on the error rate is stored as an attribute of the edge connecting the interacting nodes in the coupling-graph.
- The success rate for applying a Rx gate to the control node, computed with the *getSingleQubitErrorRate(node, theta, isRZ, isRZvirtual)* method.
- The success rate for applying a Rx gate to the target node, computed with the *getSingleQubitErrorRate(node, theta, isRZ, isRZvirtual)* method.
- The success rate for applying a Ry gate to the control node, computed with the *getSingleQubitErrorRate(node, theta, isRZ, isRZvirtual)* method.

getCZGateTime(controlNode, targetNode, isRZvirtual): extracts the gate time for applying the quantum gate: "CZ controlNode, targetNode". The CZ gate for ion trap quantum devices is not symmetric, thus different gate times may be obtained if the nodes are inverted.

The gate time of the CZ is computed supposing the gate is decomposed as shown in Figure 6.15.

The CZ gate time is thus computed summing:

- Two times the gate time for applying a Hadamard gate to the target node. There are two possible ways for computing it, depending on if the Rz gates are implemented virtually or not. In the first scenario, the gate is supposed to be decomposed according to Figure 6.10, in the latter scenario instead it is supposed to be decomposed as depicted in Figure 6.8.
- The gate time for applying a CX gate, computed resorting to the *getCX-GateTime(controlNode, targetNode, isRZvirtual)* method.

getSwapGateTime(node1, node2, isRZvirtual): extracts the gate time for applying the swap gate to *node1* and *node2*. The gate time of a swap gate is computed supposing that it is implemented as three consecutive CX gates.

The decomposition of the swap gate using three CX gates is shown in Figure 6.5.
This gate time is thus dependent on the gate time for executing a CX between the two nodes (considering that for ion trap devices, the CX gate is not symmetric) computed resorting to the *getCXGateTime(controlNode, targetNode, isRZvirtual)* method. Indeed, the gate time is also dependent on if the Rz gates are implemented virtually or not, since this changes the CX gate time. The symmetry of the swap gate is exploited, and the gate time is computed in the best case scenario (the same computation done in [42, Sec. 3.A]). For the best case scenario swap gate time computation, the Equation (6.2) is used.

getSwapErrorRate(node1, node2, isRZvirtual): extracts the error rate for applying the swap gate to *node1* and *node2*.

The error rate of a swap gate is computed supposing that it is implemented as three consecutive CX gates. The decomposition of the swap gate using three CX gates is shown in Figure 6.5.
This error rate is thus dependent on the error rate for executing a CX between the two nodes (considering that for ion trap devices, the CX gate is not symmetric) computed resorting to the *getCXErrorRate(controlNode, targetNode, isRZvirtual)* method. Indeed, the error rate is also dependent on if the Rz gates are implemented virtually or not, since this changes the CX gate time. The symmetry of the swap gate is exploited, and the gate time is computed in the best case scenario (the same computation done in [42, Sec. 3.A]). For the best case scenario swap gate time computation, the Equation (6.1) is used.

Besides instantiating an ion trap backend class manually, by passing all the required parameters to its constructor, the layout synthesis library offers a method, ***fromConfigurationFile(cfgFilePath)***, allowing the **automatic instantiation** of a *IonTrapBackend* reading a **configuration file (.cfg)**. Such file contains all the figure of merits required for the correct ion trap device modelling.
Each ion trap configuration file is composed of five sections:

Basic: it contains the basic details of the NISQ device:

- **technology**: "I" for ion trap quantum computers.
- **n_nodes**: the number of nodes of the device.
- **maxMsGateTime**: the threshold on the maximum execution time for the Mølmer–Sørensen gate (MS gate) [20]. All the interactions having an MS gate time such that $\tau_{MS} > maxMsGateTime$ are discarded, since the interaction is considered too weak.
- **maxMsErrorRate**: the threshold on the maximum error rate for the Mølmer–Sørensen gate (MS gate) [20]. All the interactions having an MS

error rate such that $E_{MS} > maxMsErrorRate$ are discarded, since the interaction is considered too weak.

RxyGateTime_halfpi: it contains the gate times for applying a Rx or Ry gate to any node of the quantum computing device.

RxyErrorRate_halfpi: it contains the error rates for applying a Rx or Ry gate to any node of the quantum computing device.

signX: it contains the allowed two-qubits interactions, as well as the sign of the phase parameter χ for each interaction (+1 or -1).

MsGateTime: it contains the gate time for applying the Mølmer–Sørensen gate (MS gate) [20] to each pair of interacting nodes.

MsErrorRate: it contains the error rate for applying the Mølmer–Sørensen gate (MS gate) [20] to each pair of interacting nodes.

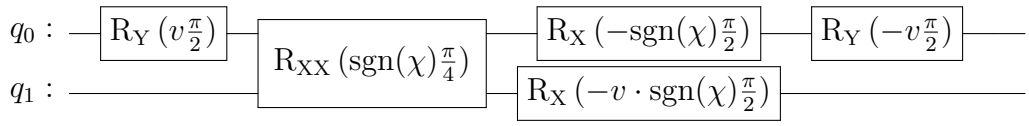

Figure 6.14. Decomposition of the CX gate using the native gates available for ion trap quantum technology, reported from [57]. The sign of the phase χ depends on the couple interacting ions. $v = \pm 1$ can be chosen arbitraly [57]. The Rxx gate is implemented using the Mølmer–Sørensen gate (MS gate) [20].

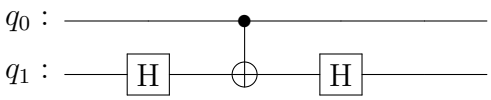

Figure 6.15. Implementation of the CZ gate for ion trap quantum technology, decomposing it using H and CX gates.

For the successive benchmarking phase of the layout synthesis library, a model for an ion trap device is required. Indeed, a python script was developed: *I_generate_backend_cfg.py*. It allows the generation of a configuration file for a 17-nodes ion trap device, having the MS gate times linearly dependent on the ions distances in the linear chain of ions. This model is proposed in [58]. In this article, the required gate times for executing an MS gate for some couple of ions at a fixed

distance are computed. After interpolating the gate times as a function of the ion distance, the linear relation obtained is:

$$MS_{gateTime} = 10 + 38d, \tag{6.11}$$

where $MS_{gateTime}$ is in [µs] and d is the ion's distance.

For the MS gate error rates computation, the model reported in [18, Sec. 7.C] was selected. In this model, the error rate of the MS gate has a linear dependency on the MS gate time (the greater the time of application of the two-qubits interaction, the greater the error rate):

$$MS_{gateFidelity} = 1 - \Gamma\tau - A(2\bar{n} + 1), \tag{6.12}$$

where $A \propto \frac{N_{ions}}{\ln(N_{ions})}$ is a scaling factor, $\bar{n}$ is the vibrational energy of the ion chain and Γ is the background heating rate of the trap in quanta/ s.

Being the second part of Equation (6.12) ($A(2\bar{n} + 1)$), used for modelling the decreasing of the fidelity during the shuttling operations (see Section 1.7), the scaling parameter A is set to 0 to neglect this effect. Furthermore, following the data provided in [58, Sec. 5], the background heating rate Γ is set to 25 quanta/ s. For $\tau(Rxy(\frac{\pi}{2}))$ and $E(Rxy(\frac{\pi}{2}))$ of respectively Equation (6.7) and Equation (6.8), the sample values reported in [59] are used.

6.2 Hardware-aware routing algorithm

The most performant algorithm implemented in the proposed layout synthesis library is the **Hardware-Aware Routing algorithm**. It is the implementation of the novel routing approach proposed in [42], solving the coupling-constraint for quantum computing devices with a **non-fully-connected topology**, and optimising the final quantum circuit gate time and fidelity **exploiting the calibration data** of the target device.

The father of this routing approach is the **SABRE heuristic** [32], explained in Section 5.1.3. The main modifications implemented by the authors of [42] are:

- Use the **calibration data** of the target device in order to perform the D matrix computation (see Section 5.1.3 for further details).
- If possible, substitute a problematic CX gate (not respecting the coupling-constraint) with a **bridge gate** (Figure 5.2) instead of adding a swap gate, avoid altering the current logical to physical qubits mapping, when this is not convenient.

In the presented work, the routing algorithm reported in [42, Sec. 3] is integrated into the layout synthesis library. The original article is mainly targetting **IBM Quantum superconducting devices**. Indeed, the algorithm was adapted for

working also with NMR, quantum dot and ion trap NISQ devices with a **non-fully-connected topology**. To perform this adaption, a modelling of the swap gate time and error rate was necessary, **extrapolating the required information from the calibration data** of the target device. This is accomplished using the **technology-specific Backend classes** (*SuperconductingBackend*, *NmrBackend*, *QuantumDotBackend* and *IonTrapBackend*) as explained in Section 6.1.
With all the complexity of computing the swap gate time and error rate abstracted inside these classes, the core routing algorithm is the same **independently on the target quantum technology**.

6.2.1 Hardware-Aware routing algorithm - Preprocessing phase

The first phase of the Hardware-Aware routing algorithm is the **distance matrices generation** S, E and T, for computing the D matrix, that will be used by the routing phase for scoring all the candidate swap gates.

In Figure 6.16 it is depicted the generation process of the distance matrices. The steps performed for generating the S, E, T and D matrices are:

1. All the information retrieved from the calibration data is used for abstracting the target quantum computing device using a technology-specific *Backend* class, as explained in Section 6.1.

2. Initialise an empty undirected graph called G_{tmp}.

3. For each couple of connected nodes inside the target coupling graph, n_i and n_j, compute their swap gate time (t_{ij}) and swap error rate (e_{ij}). This is done using the methods offered by the technology-specific *Backend* classes, see Section 6.1 for further details.

4. Add an edge in the G_{tmp} graph between n_i and n_j, having t_{ij} and e_{ij} as edge weights.

5. Apply the **Floyd–Warshall algorithm** [52] to the G_{tmp} graph. This algorithm finds the **length of the shortest-path between any pair of nodes in the input graph**. The shortest path can be computed in two different modalities:

   - The shortest path between two nodes is the path having the **minimal total number of edges**.
   - The shortest path between two nodes is the path having the **minimal total sum of a specific weight**, for all the edges in the path.

Applying the Floyd–Warshall algorithm to G_{tmp} three times, with different modalities, it is possible to compute the S, E and T distance matrices as follows:

- **S matrix computation:** the shortest path between two nodes is the path having the minimal number of edges.
- **E matrix computation:** the shortest path between two nodes is the path having the minimal total sum of the e_{ij} weight.
- **T matrix computation:** the shortest path between two nodes is the path having the minimal total sum of the t_{ij} weight.

6. The final distance matrix D is computed as follows:

$$\alpha_1 \cdot S + \alpha_2 \cdot E + \alpha_3 \cdot T, \tag{6.13}$$

where α_1, α_2 and α_3 are **configurable parameters**.
Before computing the D matrix according to Equation (6.13), the matrices S, E and T are normalised, to have the same scale.

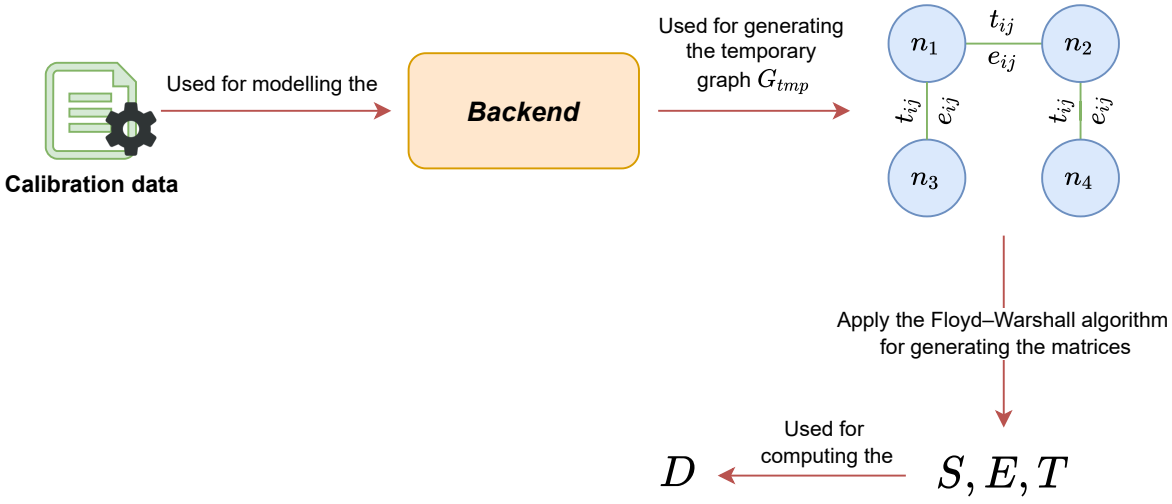

Figure 6.16. From the calibration data to the S, E, T and D matrices.

It is essential to know what each matrix represents, in order to understand the aim of the distance matrices computation:

- The matrix S is storing the **shortest path length** for any pair of nodes in the target quantum computing device. A generic entry $S[i][j]$ represents the **number of swap gates** required for applying a swap gate to n_i and n_j, respecting the target coupling-contraints.

- The matrix E is used for approximating the **error rate** for swapping any pair of nodes in the target quantum computing device. A generic entry $E[i][j]$ represents the **approximated total swap error rate** for applying a swap gate to n_i and n_j, respecting the target coupling-contraints.
- The matrix T is used for approximating the **gate time** for swapping any pair of nodes in the target quantum computing device. A generic entry $T[i][j]$ represents the **approximated total swap gate time** for applying a swap gate to n_i and n_j, respecting the target coupling-contraints.
- The matrix D is used for approximating the **distance** for swapping any pair of nodes in the target quantum computing device. For distance, it is intended the combination of number of swap gates, error rate and gate time. A generic entry $D[i][j]$ represents the **approximated distance** for applying a swap gate to n_i and n_j, respecting the target coupling-constraint.

6.2.2 Hardware-Aware routing algorithm - How the swap gates are scored

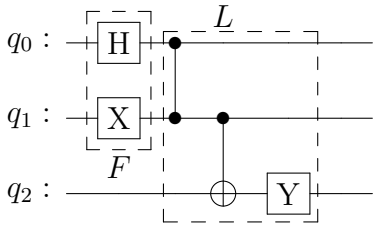

Figure 6.17. At the beginning, the output routed circuit is empty and the gates of the input circuit having no dependency constitute the front layer F. The look-ahead layer L contains some successors of the gates inside F. In this scenario, all the successors of the gates inside F constitute the look-ahead layer L.

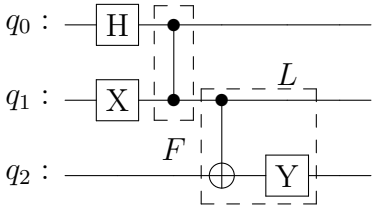

Figure 6.18. After the H and X single-qubit gates of the circuit depicted in Figure 6.17 are routed (added to the output routed circuit), they are removed from the front layer. The new front layer is now composed of only the CZ gate, since it is the only non-routed gate having no dependency. All the CZ gate successors constitute the look-ahead layer L.

The *Hardware-Aware routing algorithm*, exactly like the *SABRE* [32] routing algorithm, **generates a set of candidate swap gates that are scored to find the best one**. Before proceeding with the explanation of the core routing procedure, it is essential to explain how the candidate swap gates are scored.

The input quantum circuit is **divided into two layers** (set of gates): the **front layer** F and the **lookahead layer** L:

- The front layer F is the set of quantum gates, composing the circuit, that have no dependencies among them and on any other non-routed gate (a gate that was not already added to the output routed circuit). This means that all the gates in F can always be executed from a dependency perspective, even if this might not be the case from a hardware perspective (for example, if it is a two-qubits gate and the interacting nodes are not connected in the target coupling-graph). The front layer will constantly be updated during the routing process, removing the gates as soon as they are routed and inserting their successors. This is done in order to explore the quantum gates composing the circuit in topological order, respecting their dependencies.

- The look-ahead layer L is the set of some quantum gates, successors of the gates in F. The size of the look-ahead layer is **completely configurable**, allowing a **flexible depth of the look-ahead ability**.

An example of evolution of the F and L layer is reported in Figure 6.17 and Figure 6.18.

In order to **assign a score to a candidate swap gate**, bringing the logical to physical qubits mapping (when executed) from $\pi_{current}$ to π_{temp}, it is possible to use two heuristic cost functions:

The basic cost function H_{basic}:

$$H_{basic} = \sum_{gate \in F} D[\pi_{temp}(gate.q1)][\pi_{temp}(gate.q2)], \tag{6.14}$$

where:

- $D[\,][\,]$ is the distance matrix computed during the preprocessing phase.
- π_{temp} is the new logical to physical qubits mapping, supposing the candidate swap gate for which H_{basic} must be computed, is added to the quantum circuit and executed.
- $\pi_{temp}(gate.q1)$ and $\pi_{temp}(gate.q2)$ are the interacting nodes of $gate$ given the new mapping π_{temp}.

It is the simplest version of the heuristic, summing the **distances** between the interacting nodes for each gate inside the front layer. For distance between two nodes, it is intended the combination of swap gate number, swap error rate and swap gate time required for making the interaction compatible with the target hardware.

The look-ahead cost function $H_{lookahead}$:

$$H_{lookahead} = \frac{1}{|F|} \sum_{gate \in F} D[\pi_{temp}(gate.q1)][\pi_{temp}(gate.q2)] + W \cdot \frac{1}{|L|} \sum_{gate \in L} D[\pi_{temp}(gate.q1)][\pi_{temp}(gate.q2), \quad (6.15)$$

where:

- $D[\,][\,]$ is the distance matrix computed during the preprocessing phase.
- π_{temp} is the new logical to physical qubits mapping, supposing the candidate swap gate for which $H_{lookahead}$ must be computed, is added to the quantum circuit and executed.
- $\pi_{temp}(gate.q1)$ and $\pi_{temp}(gate.q2)$ are the interacting nodes of $gate$ given the new mapping π_{temp}.
- $|F|$ is the number of quantum gates inside the front layer F.
  $\frac{1}{|F|}$ is used to **normalise** the summation of the distances between the interacting nodes of the gates in F.
- $|L|$ is the number of quantum gates inside the **look-ahead layer** L. This set contains some successors of the gates in F, where the size of L is a tunable parameter. In this way, **the look-ahead ability is flexible**.
  $\frac{1}{|L|}$ is used to **normalise** the summation of the distances between the interacting nodes of the gates in L.
- W is the **look-ahead parameter**. It is a real value such that $0 \leq W < 1$ that is **used to reduce the importance of the look-ahead ability**.

It is the most complete version of the heuristic, summing the **distances** between the interacting nodes for each gate inside the front and look-ahead layer. For distance between two nodes, it is intended the combination of swap gate number, swap error rate and swap gate time required for making the interaction compatible with the target hardware.

The last requirement for evaluating a candidate swap gate is computing the **effect of the swap gate insertion to the gates composing the look-ahead layer**. Sometimes adding a swap gate might have a **negative impact on the look-ahead layer**, increasing the "distance" among the interacting nodes of the gates composing L. This effect is computed according to:

$$Effect = \sum_{gate \in L} D[\pi_{current}(gate.q1)][\pi_{current}(gate.q2)] - D[\pi_{temp}(gate.q1)][\pi_{temp}(gate.q2)], \quad (6.16)$$

where $\pi_{current}$ is the logical to physical qubits mapping before the candidate swap gate is added to the quantum circuit and executed.
If $Effect$ is negative, the candidate swap gate addition increases the "distance" between the interacting nodes of the look-ahead layer L.

6.2.3 Hardware-Aware routing algorithm - Routing phase

Figure 6.19 shows the flow-chart of the **core routing strategy** for the Hardware-Aware routing algorithm.

Its required inputs are:

- *placedCircuit:* the input quantum circuit of the routing procedure for which the placement was already performed.
- π_{init}**:** the initial mapping applied during the placement. It is required since during the routing phase, the current (considering all the swap gates added to the circuit) mapping between logical and physical qubits is tracked to output the final mapping π_{final} (the mapping at the end of the quantum circuit execution, before the measure operations).

The main steps followed by the presented strategy are:

1. The first step is an **initialisation phase**. The algorithm constructs step-by-step a new quantum circuit (***routedCircuit***), extracting gates from the placed circuit and inserting them in the new one, respecting their dependencies.
   During this initialisation, the *routedCircuit* is instantiated: at the beginning it contains no quantum gates; the quantum and classical registers size match the ones of the original circuit; the measures operations are the same as the placed circuit ones.
2. The **first layer of the input quantum circuit is extracted** (thus it is removed from the original circuit). All of its composing single-qubit gates and the two-qubits gates applied to nodes connected in the target coupling-graph are simply added to the output circuit.
3. If, at this step, the **first layer is not empty**, it means that it contains one or more two-qubits interactions that are not allowed on the target hardware. Therefore, the insertion of a swap gate (or a bridge gate, when possible) is mandatory for proceeding with the routing. The list ***swapCandidateList*** is constructed: it contains all the swap gates that can modify the logical to physical qubits mapping for at least one node associated with an interaction in the front layer F (without considering previously explored mappings).
4. Each candidate node inside *swapCandidateList* is scored using the H_{basic} (Equation (6.14)) or the $H_{lookahead}$ (Equation (6.15)) heuristic cost function (it is a

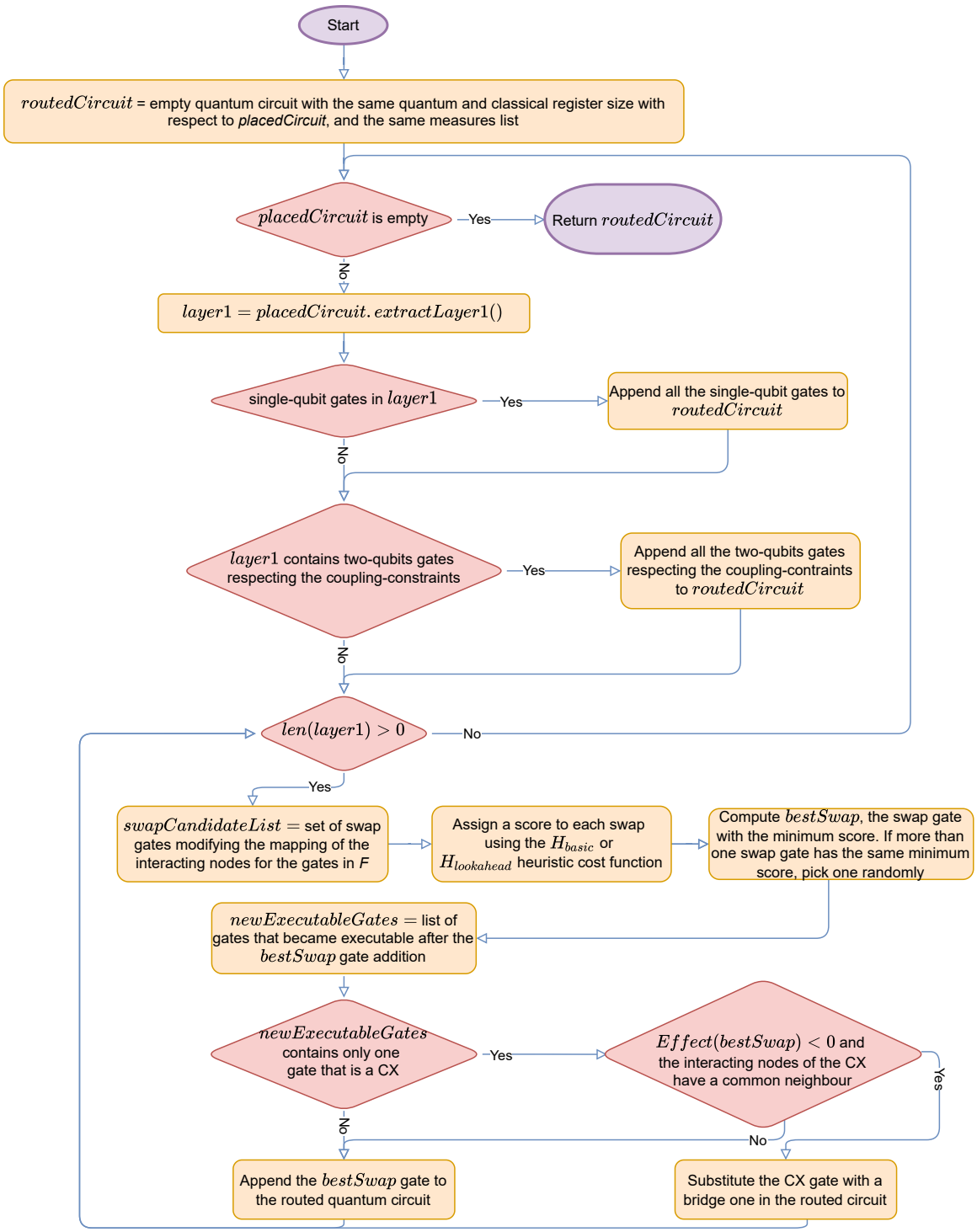

Figure 6.19. Flow chart of the Hardware-Aware routing algorithm.

configurable parameter). The best swap labelled as ***bestSwap*** is the one minimising the heuristic cost function. If there are more than one swap gates having the same minimal H, one is chosen randomly.

5. If the addition of the *bestSwap* to the *routedCircuit* allows one and only one gate in F to became executable:

   (a) Defining $\pi_{current} \rightarrow \pi_{new}$ the effect on the current mapping $\pi_{current}$ of the *bestSwap* addition. If the new executable gate is a CX gate acting on the logical qubits q_i and q_j, and if the length of the shortest path between the nodes $n_i = \pi_{new}(q_i)$ and $n_j = \pi_{new}(q_j)$ is exactly 2 (that is, if the two interacting nodes of the new executable gate have one common neighbour):

      - Compute the effect of the *bestSwap* addition using Equation (6.16). If the effect is negative, the CX gate is substituted with a bridge gate in the output routed circuit.
      - Otherwise, proceed to Step 6.

   (b) Else proceed to Step 6.

6. The *bestSwap* is added to the *routedCircuit*. The algorithm then repeats from Step 3 and continues until all the gates of the placed input circuit are added to the output routed circuit.

6.3 Routing techniques for fully-connected topologies

The routing phase explained in Chapter 5 and the procedures implemented in the proposed layout synthesis library, presented in Section 5.2.3 and Section 6.2, are related to NISQ devices having a non-fully-connected topology. Indeed, in these devices, the information on the allowed two-qubits interactions is **explicit**. However, for some quantum technologies having a **fully-connected topology**, such as NMR and ion-trap (single-trap devices, see Section 1.7), this information is **implicit** instead: all the interactions are **theoretically** allowed by the device, but in practice some interactions are **hard to implement**, and their fidelity and execution time would be unacceptable [33, Sec. 1]. Specifically, for NMR devices the hard to implement interactions are the ones acting on nuclei having a low J-coupling constant, while for ion trap devices, the problematic interactions are related to distant ions inside the ion trap (following the model presented in Section 6.1.4).
Therefore, even if routing techniques are not mandatory for fully-connected quantum computing devices, in practice this might be **essential to have meaningful results**. For this reason, **two hardware-aware approaches** to perform the routing phase targeting NISQ devices having a fully-connected topology are implemented in the proposed library:

1. Break the fully-connected topology, configuring a threshold on the minimum interaction strength accepted. This strategy is explained in Section 6.3.1.
2. The *Hardware-Aware smart* routing algorithm. This strategy is explained in Section 6.3.2.

The aim of both the two approaches is to avoid applying two-qubits gates on the weaker interactions of the target NISQ device.

6.3.1 Breaking the fully-connected topology with a minimum threshold

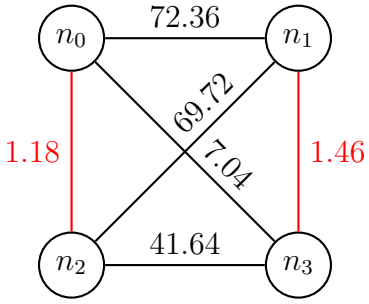

Figure 6.20. Undirected coupling-graph representing the *crotonic_acid* NMR quantum device [16, Sec. 8.3]. Each node represents a physical qubit and each edge represents an allowed two-qubits interaction. The architecture is fully-connected, but each interacting pair of nodes has a different J-coupling constant, shown in Hz. The weakest interactions are highlighted in red.

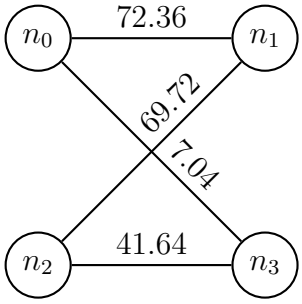

Figure 6.21. Representation of the *crotonic_acid* NMR quantum device [16, Sec. 8.3] shown in Figure 6.20 after configuring a minimum J-coupling threshold of 1.47 Hz. The weakest interactions are removed and the topology became non-fully-connected.

The first possibility offered by the proposed library, for routing a fully-connected quantum device, is to impose a **minimum threshold** on the **interaction strength**. It is essential to remark that the nature of the interaction strength parameter **depends on the modelled quantum technology**. For example, for NMR technology-specific *Backend* classes (see Section 6.1.2), it is possible to specify a **minimum J-coupling parameter**, since the strength (thus, execution time and error rate) of the interaction depends on the J-coupling constant (see Section 1.5 for further details). After breaking the fully-connected topology with a minimum threshold, it is possible to use the *Hardware-Aware* routing algorithm in the classical way (see Section 6.2). An example showing how it is possible to break the fully-connected topology for the *crotonic_acid* NISQ device is depicted in Figure 6.20

and Figure 6.21.

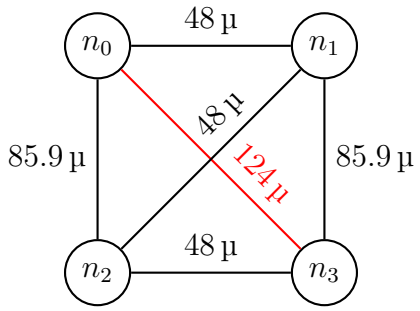

Figure 6.22. Undirected coupling-graph representing only 4 ions of the modelled ion trap backend explained in Section 6.1.4. Each node represents a physical qubit and each edge represents an allowed two-qubits interaction. The architecture is fully-connected, but each interacting pair of ions has a different MS gate time (and error rate, not depicted in this picture), shown in s. The weakest interaction is highlighted in red.

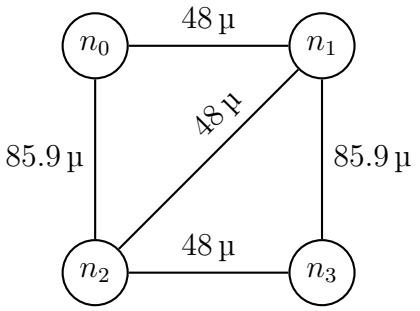

Figure 6.23. Representation of the ion trap backend coupling-graph shown in Figure 6.22 after configuring a maximum MS gate time of 86 µs. The weakest interaction is removed from the graph and the topology became non-fully-connected.

For ion trap technology-specific *Backend* classes, it is possible to break the fully-connected topology imposing a maximum gate time and error rate for the MS gate (as explained in Section 6.1.4). In particular, only the interactions having $\tau_{MS} \leq maxMsGateTime$ and $E_{MS} \leq maxMsErrorRate$ are considered allowed in the target hardware. An example showing how it is possible to break the fully-connected topology for an ion trap NISQ device is shown in Figure 6.22 and Figure 6.23.

6.3.2 Hardware-aware routing smart algorithm

For the presented work, an attempt to adapt routing techniques for NISQ devices having a fully-connected topology is made. Specifically, the original hardware-aware routing algorithm presented in Section 6.2 is expanded, making it compatible for targetting fully-connected quantum devices, aiming at **optimising the final quantum circuit execution time** (if this is feasible). The idea is to compute the best swap gate in accordance to the original algorithm, and then check if its addition (considering the overhead to perform it) could bring the two-qubits interactions towards the **stronger interacting nodes**, improving the final circuit execution time.

The pre-processing phase for computing the distance matrices remained unchanged in this adaptation (see Section 6.2.1 for further details). The modifications touched only the core routing strategy.
The flow-chart of the hardware-aware smart routing algorithm is shown in Figure 6.24. The main steps of the new algorithm are:

1. The first step is an **initialisation phase**. The algorithm constructs step-by-step a new quantum circuit (***routedCircuit***), extracting gates from the placed circuit and inserting them in the new one, respecting their dependencies.
   During this initialisation, the *routedCircuit* is instantiated: at the beginning it contains no quantum gates; the quantum and classical registers size match the ones of the original circuit; the measures operations are the same as the placed circuit ones.

2. The **first layer of the input quantum circuit is extracted** (thus it is removed from the original circuit). All of its composing single-qubit gates are simply added to the output circuit. It is essential to understand that all the remaining two-qubits gates in the first layer **are considered valid in hardware**, thus **the hardware-aware smart routing algorithm can target only NISQ devices with a fully-connected architecture**.

3. If, at this step, the **first layer is not empty**, it means that it contains one or more two-qubits interactions. The algorithm extracts the first gate inside the front layer, labelled as ***currGate***. The best swap gate ***bestSwap*** is computed using the same procedure explained in Section 6.2 (using the ***look-ahead heuristic***, *Equation* (6.15)). The time required for applying *bestSwap* is computed and labelled as ***bestSwapTime***.

4. At this stage, the algorithm checks wether it is more convenient to just add *currGate* to the routed circuit, or it might be favorable adding *bestSwap* before. To accomplish this, the approximated front and look-ahead layer execution time must be computed: one time considering adding *bestSwap* to the output circuit (***suppSwapTotalGateTime***), and a second time supposing not adding it, leaving the circuit as it is (***currTotalGateTime***).

   The approximated front and look-ahead total execution time is computed as following:

   $$\tau_{F,L} = \sum_{gate \in F} \tau_{gate} + \sum_{gate \in L} \tau_{gate}, \tag{6.17}$$

   where:

   - F and L are respectively the front and the look-ahead layer.
   - *gate* is a two-qubits gate inside F or L.

- τ_{gate} is the *gate* execution time. This time is different when computing *currTotalGateTime* or *suppSwapTotalGateTime*. In the first case the interacting nodes of *gate* are the ones of the original circuit, in the latter these interacting nodes might change as a consequence of the *bestSwap* addition. The gate execution time is computed using the methods offered by the technology-specific classes abstractions (see Section 6.1), with two required approximantions:
  (a) For superconducting quantum devices, not modelling the time required for executing a single-qubit gate (see Section 6.1.1), in case *gate* is a CZ gate, its gate time is approximated to the gate time required for executing a CX to the same target nodes as *gate*.
  (b) If a two-qubits interaction is not a CX or CZ gate (even if this is the expected scenario, since it is usual to perform the layout synthesis before the CX or CZ gate decomposition):
    – For superconducting and ion trap devices, all the non-CX and non-CZ gates are considered CX gates for the gate time computation.
    – For NMR and quantum dot devices, all the non-CX and non-CZ gates are considered CZ gates for the gate time computation.

  When computing the *SuppSwapTotalGateTime*, *bestSwapTime* is added to $\tau_{F,L}$, taking in consideration the overhead of the swap gate addition.

  The single-qubit gate times are not considered in Equation (6.17). This is because these times are usually similar regardless of the target node. Therefore, the heuristic aims at minimising the total execution time considering only the two-qubits interactions.

5. If *SuppSwapTotalGateTime* is lower than *currTotalGateTime*, the *bestSwap* is added to the routed circuit, otherwise *gate* is added. The algorithm then repeats from Step 3 and continues until all the gates of the placed input circuit are added to the output routed circuit.

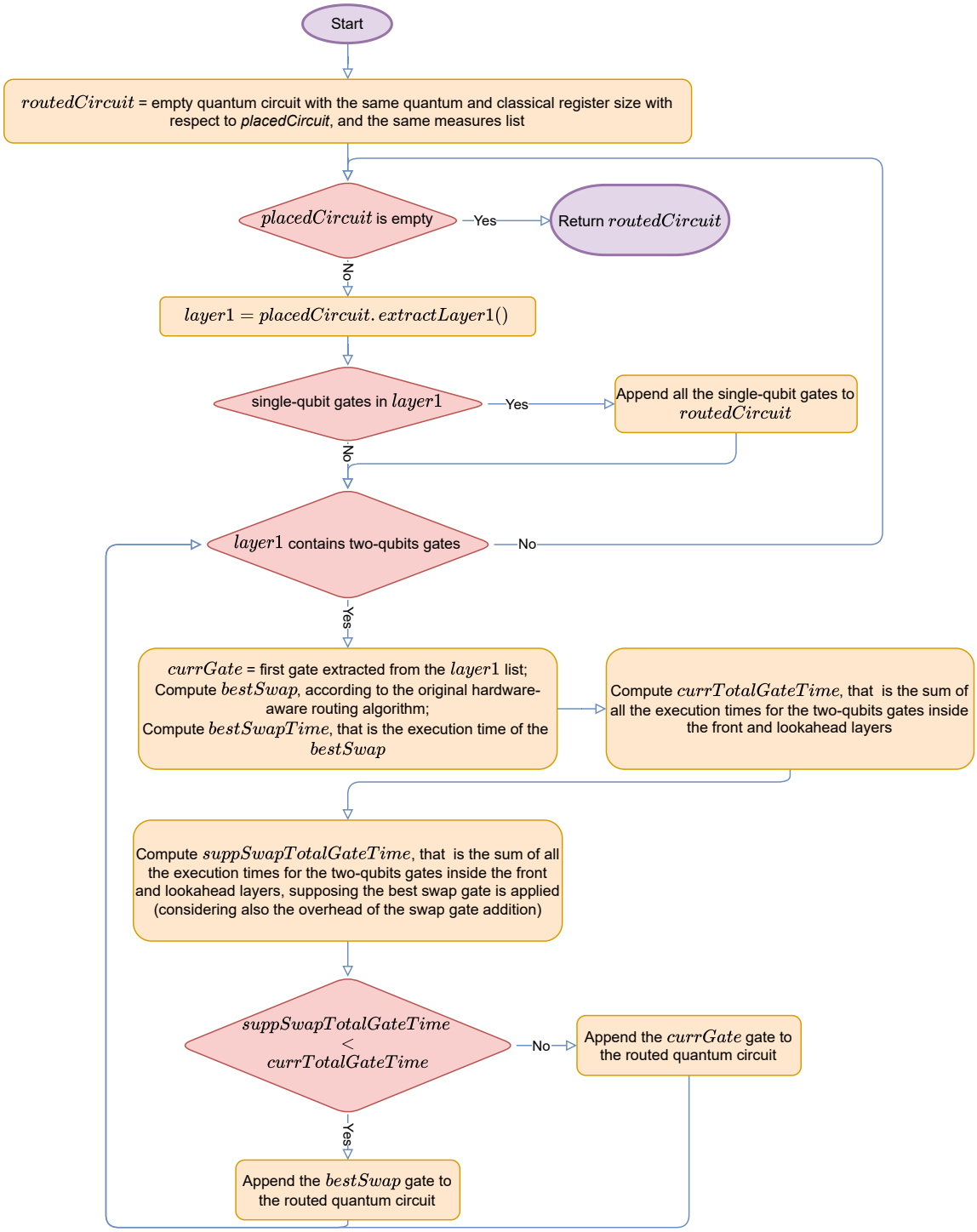

Figure 6.24. Flow chart of the Hardware-Aware smart routing algorithm.

Part III

Evaluation of results and conclusions

Chapter 7

Benchmarking

After the development of the proposed layout synthesis library, it was essential to **assess the correctness** of the heuristics implementation and **evaluate the performance** of the included strategies against the state-of-the-art, for some selected figure of merits. For the latter task, the implemented strategies were benchmarked against **IBM's Qiskit** and **Cambridge Quantum Computing's t|ket⟩**, the two leading quantum compilation toolchains at the time of writing. Therefore, a series of Python scripts are developed to accomplish these two tasks. This chapter is devoted to explaining the verification and benchmarking procedures, and presenting the evaluated results. Specifically, the chapter is structured as follows: in Section 7.1 an in-depth explanation of the testing procedures is presented; in Section 7.2 the functional checking results are underlined, to validate all the implemented heuristics; in Section 7.3 the results of the benchmarking phase for the implemented placement heuristics are presented; in Section 7.5 and Section 7.4 the results of the benchmarking phase for the implemented routing heuristics are presented.

7.1 General information

7.1.1 Tested circuits

A **set of quantum circuits description** written in the OpenQASM 2.0 language was constructed in order to perform the functional equivalence checking and the benchmarking tests. The same circuits used in [15, Ch. 5] are selected, in particular they are retrieved from [60] and [61].
Following the approach used by [32, Sec. 5] and [42, Sec. 4], the quantum circuits used for the benchmarking phase are **divided into three sets**: **small-sized**, **medium-sized** and **large-sized**, depending on the size of the quantum register. Furthermore, only circuits with **fewer than 35000 quantum gates** are chosen.

The only modifications performed to the selected circuits is to rename the quantum and classical register to respectively "q" and "c" (convention required by the proposed library), and to add the measurement operations in some circuits where this was missing (measuring the final state in the classical register is mandatory for performing the functional equivalence checking).

To emulate the expected use-case for the presented layout synthesis library (that is, a subsequent step to the logic synthesis phase, before decomposing the two-qubits interactions), all the circuits were **rebased** using the **technology's legal set of gates** as defined in [15, Ch. 5]. Specifically:

- For **superconducting technology** the gate basis is: $\{U_1, U_2, U_3, CX\}$
- For **NMR and quantum dots technology** the gate basis is: $\{R_X, R_Y, R_Z, CZ\}$
- For **trapped ions technology** the gate basis is: $\{R_X, R_Y, R_Z, CX\}$

To perform the rebasing task, the **Qiskit Terra Transpiler** [22] was employed (**Qiskit Terra libraries version 0.18.3**), configuring an **optimisation level of 0**, to avoid any further optimisation.

All the tests evaluated in this chapter **were performed using as input quantum circuits these rebased circuits**. Because of the limited number of physical qubits for the NMR and quantum dots considered backends, **only the small-sized circuits set was utilised for these technologies**. Specifically, the NMR backend was tested only with the small-sized circuits with at most four qubits, and the quantum dots backend was tested only with the small-sized circuits with at most five qubits.

7.1.2 Backend configuration files used for testing

The proposed layout synthesis library requires a backend configuration file containing all the information on the allowed two-qubit interactions and quantum gates features (used only for the hardware-aware placement and routing strategies). Therefore, **four configuration files were used for the testing phase**, one for each target technology available. Specifically:

- For **superconducting technology**, the *ibmq_toronto.cfg* backend configuration file was developed, modelling the mocked version of the **ibmq_toronto** 27-qubits superconducting device [62].
- For **NMR technology**, the *crotonic_acid.cfg* backend configuration file was developed (shown in Figure 6.11), modelling the **crotonic acid** 4-qubits NMR quantum device [16, Sec. 8.3].
- For **quantum dots technology**, the *5nodes_quantum_dot.cfg* backend configuration file was developed (shown in Figure 6.13), modelling the 5-qubits

quantum dots device presented in Section 6.1.3, constructed expanding the data available in [17].

- For **trapped ions technology**, the *ion_trap_experimental.cfg* backend configuration file was developed, modelling the 17-qubits trapped ions device presented in Section 6.1.4

7.1.3 Functional equivalence - methodology

It is mandatory that all the implemented heuristics do not alter the **functional behaviour** of the final quantum circuit. Therefore, since in the implemented library the classical bits where each **logical qubit** is measured remain constant, the frequency of measurement for each eigenstate of the output circuit **must be similar** to the input one (before performing the placement and the routing). Thus, the functional equivalence checking task can be accomplished resorting to a **discrete probability distributions divergence computation**.
An alternative verification strategy is to compute the fidelity [5, Sec. 9.2.2] between the input and output circuit's quantum states, before the measures operations are performed, and to assess that the fidelity is close to one. To accomplish this, the state vector of the output circuit **must be reordered** (based on the swap gates applied) before computing the fidelity. This strategy is more precise, but the computational complexity for both the simulation and the state vector reordering is not negligible. Indeed, it was performed only for small quantum circuits and small NISQ devices during the development phase. In this chapter, only the results of the functional equivalence checking resorting to a discrete probability distribution divergence computation are presented.

Specifically, the main steps performed during the functional equivalence checking are:

1. The functional equivalence checking ensures the **closeness** between the *input_circuit* and the *ouput_circuit*, that are respectively:
   - The input quantum circuit, before applying any layout synthesis transformations (placement, routing or both).
   - The quantum circuit output of the placement or routing step (or both).
2. Both the *input_circuit* and the *ouput_circuit* are simulated employing the **Qiskit Aer QASM simulator (Qiskit Aer libraries version 0.9.1)** [63], using 20000 simulation shots. Indeed, this simulator returns the frequency of each measured eigenstate, that is, a discrete probability distribution.
3. The **divergence** between these two distributions is computed according to **three different metrics**:

Kullback Leibler divergence (KLD): it estimates the "difference" between the two distributions [64]. Therefore, to prove the functional equivalence, **a KLD close to zero must be obtained**. The KLD is computed as:

$$D_{KL}(P\|Q) = \sum_{x\in\chi} P(x)\log\left(\frac{P(x)}{Q(x)}\right), \tag{7.1}$$

where P and Q are the two discrete probability distributions.

The KLD can be computed only for discrete probability distributions having a **common support**. Indeed, for the KLD computation only the common support is used, and all the non-common measures are randomly distributed to the common ones (with uniform probability).

Hellinger fidelity: it measures the "closeness" between the two distributions in percentage (Hellinger fidelity $\in$ [0,1]). This fidelity is computed resorting to the **Qiskit Terra API**, in order to have a metric that is not directly developed (to further prove the functional equivalence). This fidelity, in the official documentation [65], is defined as: $(1-H^2)^2$, where H is the **Hellinger distance** between the two distributions computed as:

$$H(P,Q) = \frac{1}{\sqrt{2}}\sqrt{\sum_{i=1}^{k}(\sqrt{p_i}-\sqrt{q_i})^2}, \tag{7.2}$$

where $P = (p_1,\ldots,p_k)$ and $Q = (q_1,\ldots,q_k)$ are the two discrete probability distributions.

Since this metric measures how much the two discrete probability distributions are "close", **a Hellinger fidelity near one must be obtained**.

Fidelity (approximated): it measures the closeness between the two discrete probability distributions in percentage (fidelity approximated $\in$ [0,1]). It is labelled as "approximated" since it is computed as the **fidelity** [5, Sec. 9.2.2] between the two **reconstructed state vectors**, using only the frequency of measurement of each egeinstate, obtained from the Qiskit Aer QASM simulator.

For the fidelity (approximated) computation only the **common support** of the two discrete probability distributions is used. Indeed, if after the simulations the two supports have at least one non-common element, the following steps are performed:

(a) The common support is extracted, and the percentage of non-common measures is computed for both the original and the transformed discrete probability distributions.

(b) For the fidelity (approximated) computation only the common support is used, and all the non-common measures are randomly distributed

to the common ones (with uniform probability). If the percentage of non-common measures is greater than 0.05 (for the input or the output distribution), **the fidelity is set to zero** (due to the high percentage of non-common measures).

4. For the two discrete probability distributions (and thus, quantum circuits) to be equivalent, all the three different metrics explained before must not exceed some **defined thresholds**:

   - For the KLD, the **maximum acceptable threshold** is set to 0.05.
   - For the Hellinger fidelity, the **minimum acceptable threshold** is set to 0.95.
   - For the fidelity (approximated), the **minimum acceptable threshold** is set to 0.95.

7.1.4 Coupling-constraint checking

Following the definition of the routing step (Definition 5.0.1), it is essential that after applying each implemented routing heuristic, the two-qubit interactions composing the output circuit **satisfy the coupling-constraint of the target NISQ device**. Therefore, also a coupling-constraint check is performed for all the implemented routing strategy. It is accomplished by counting the number of non-hardware compliant interactions and making sure that they are equal to zero.

For all the available routing techniques in the proposed layout synthesis library, for all the target technologies and for all the testing circuits, the number of non-allowed interactions was **always zero**, thus **validating the routing heuristics implementation**.

7.1.5 Benchmarking phase

In order to evaluate the performance of the implemented heuristics and to compare them against Qiskit and t|ket⟩, a benchmarking phase is essential. Thus, a set of **quantum circuit metrics** was defined in order to evaluate and compare the different layout synthesis strategies. Different metrics are used for the placement and routing steps, and also for different target hardware topologies (fully-connected and non-fully-connected), ensuring a fair comparison.

Placement step

After the placement step, the logical qubits are mapped to the physical ones, but the placed quantum circuit is probably not executable in hardware yet. Indeed, an evaluation of the total execution time and error rate is not possible.
The selected metric in order to compare the quality of the different implemented

placement procedures is the **number of non-executable two-qubit gates**. Placement algorithms that allow the most two-qubit interactions to be executed without any further swap gate addition (during the subsequent routing phase) are preferred.

This metric of evaluation is used for all the implemented placement procedures: *Trivial Mapping*, *Simulated Annealing Dense Mapping* and *Simulated Annealing Hardware-Aware Mapping*. This comparison is only valid for the non-fully-connected hardware technology (superconducting and quantum dots), since it would be unfair to compare it to the fully-connected topologies of NMR and trapped ions devices. Indeed, for the latter a different strategy is used, explained later in this section.

All the implemented placement strategies cited before are compared (for the selected metric) against the Qiskit placement strategies.

Placement and routing steps - non-fully-connected technologies

Since the routing step follows the placement, it could not be evaluated alone. Therefore, the **complete layout synthesis**, with different combinations of available placement and routing strategies, is evaluated.

The selected metrics for performing the benchmarking are the following:

***swapCount*:** the number of swap gates added in the final quantum circuit description.

***CF_circuitDepth*:** the circuit depth compression factor between the input circuit (*rebased_circuit*) and the output one (*placed_routed_circuit*). It is one of the metrics used in [66, Sec. 2.3], computed as:

$$CF\_circuitDepth = \frac{\text{circuit depth}(rebased\_circuit)}{\text{circuit depth}(placed\_routed\_circuit)} \tag{7.3}$$

The circuit depth is defined as the number of layers of the quantum circuit.

***CF_gateCount*:** the circuit gate count compression factor between the input circuit (*rebased_circuit*) and the output one (*placed_routed_circuit*). It is one of the metrics used in [66, Sec. 2.3], computed as:

$$CF\_gateCount = \frac{\text{gate count}(rebased\_circuit)}{\text{gate count}(placed\_routed\_circuit)} \tag{7.4}$$

For the gate count, both the single-qubit and two-qubit gates are considered.

***totalExecutionTime*:** the approximated total execution time of the final quantum circuit. The methods available in the technology-specific *Backend* classes (see Section 6.1) are used to perform this computation. Moreover, the following additional information is used:

- For the superconducting backend, the U_2 gate time is extrapolated from the *FakeToronto* [62] system properties (the X gate time). The U_1 gate times are set to 0 when the R_z gates are implemented virtually, otherwise they are supposed to be implemented using a U_3 gate. For the latter, the gate time is set to $\tau(U_3) = 2 \cdot \tau(U2)$.

circuit cost function C**:** the overall cost of the output quantum circuit, defined by a single metric. It is one of the metrics used in [66, Sec. 2.3], computed as:

$$C = -D \log K - \sum_i \log {F_i}^{1q} - \sum_j \log {F_j}^{2q}, \tag{7.5}$$

where:

- D is the output circuit depth (the number of layers).
- K is a constant penalisation factor. Different K values are used for the four different backends, ensuring that: ${F_{avg}}^{1q} < K < {F_{avg}}^{2q}$ (the same strategy adopted in [66, Sec. 2.3]). All the values of K used for performing the benchmarks are underlined in Table 7.4.
- ${F_i}^{1q}$ is the i-th single-qubit gate fidelity, computed as $1 - gate\_error\_rate$. This error rate is computed as:
  - For the superconducting backend, the U_2 error rates are extrapolated from the *FakeToronto* [62] system properties (the X gate error). The U_1 error rates are set to 0 when the R_z gates are implemented virtually, otherwise they are supposed to be implemented using a U_3 gate. For the latter, the error rate is set to $E(U_3) = 2 \cdot E(U_2)$.
  - For the quantum dots backend, the $R_x(\frac{\pi}{2})$ error rates are computed performing a simulation using **MATLAB QuanTO** [56], as explained in Section 6.1.2. The $R_x(\theta)$ error rates are computed as:

$$E(R_x(\theta)) = \frac{E(Rx(\frac{\pi}{2})) \cdot \theta}{\frac{\pi}{2}} \tag{7.6}$$

    The R_y error rates are supposed equal to the R_x ones. The R_z gates if not implemented virtually are supposed to be decomposed according to Equation (6.5).
- ${F_j}^{2q}$ is the j-th two-qubit gate fidelity, computed as $1 - gate\_error\_rate$. This error rate is computed as:
  - For the superconducting backend, the *getCXErrorRate* method of the *SuperConductingBackend* class is used (see Section 6.1.1).
  - For the quantum dots backend, the CZ error rates are computed performing a simulation using **MATLAB QuanTO** [56], as explained in Section 6.1.2.

Different combinations of the implemented placement and routing strategies cited before are compared (for the selected metrics) against the Qiskit and t|ket⟩ ones.

Placement and routing steps - fully-connected technologies

For fully-connected technologies (NMR and trapped ions), it would not be fair to compare the number of swap gates added during the layout synthesis phase. The reason for this is that, theoretically, no swap gates are required to satisfy the coupling-constraint of the target NISQ device, but they might be added to achieve a more optimised final quantum circuit.

The two metrics used for benchmarking the layout synthesis of the fully-connected technologies are:

***totalExecutionTime*:** the approximated total execution time of the final quantum circuit. The methods available in the technology-specific *Backend* classes (see Section 6.1) are used to perform this computation.

***circuit cost function* C:** the overall cost of the output quantum circuit, defined by a single metric. It is one of the metrics used in [66, Sec. 2.3], computed as:

$$C = -D \log K - \sum_i \log {F_i}^{1q} - \sum_j \log {F_j}^{2q}, \tag{7.7}$$

where:

- D is the output circuit depth (the number of layers).
- K is a constant penalisation factor. Different K values are used for the four different backends, ensuring that: ${F_{avg}}^{1q} < K < {F_{avg}}^{2q}$ (the same strategy adopted in [66, Sec. 2.3]). All the values of K used for performing the benchmarks are underlined in Table 7.4.
- ${F_i}^{1q}$ is the i-th single-qubit gate fidelity, computed as $1 - gate\_error\_rate$. This error rate is computed as following:
  - For the trapped ions backend, the *getSingleQubitErrorRate* method of the *IonTrapBackend* class is used (see Section 6.1.4).
  - For the NMR backend, the $R_x(\frac{\pi}{2})$ error rates are computed performing a simulation using **MATLAB QuanTO** [56], as explained in Section 6.1.2. The $R_x(\theta)$ error rates are computed using Equation (7.6) The R_y error rates are supposed equal to the R_x ones. The R_z gates if not implemented virtually are supposed to be decomposed according to Equation (6.5)
- ${F_j}^{2q}$ is the j-th two-qubit gate fidelity, computed as $1 - gate\_error\_rate$. This error rate is computed as following:

- For the trapped ions backend, the *getCXErrorRate* method of the *IonTrapBackend* class is used (see Section 6.1.4).
- For the NMR backend, the CZ error rates are computed performing a simulation using **MATLAB QuanTO** [56], as explained in Section 6.1.2.

For the comparison (using the selected metrics), the routing algorithms for fully-connected topologies, presented in Sections 6.3.1 and 6.3.2 are evaluated and compared to performing only the placement step, to assess if these implemented solutions could add further optimisation. Furthermore, the comparison is also done against the Qiskit and t|ket⟩ placement and routing strategies.

Qiskit placement and routing

All the quantum circuits used for the benchmarking of the proposed layout synthesis library were also placed and routed using the **Qiskit compilation toolchain**, in order to perform a comparison.
The applied Qiskit's placement strategies are:

- *DenseLayout* [37].
- *SabreLayout* [39].
- *NoiseAdaptiveLayout* [38].

The applied Qiskit's routing strategies are:

- *SabreSwap* [67].
- *StochasticSwap* [49].

All these placement and routing strategies are well explained in Section 4.1.1 and Section 5.1.1.

All the four target backends presented in Section 7.1.2 were modelled using the **Qiskit Terra API**. Specifically, for each backend, two data-structures are required:

- The ***CouplingMap*** representing the allowed two-qubit interactions.
- The ***BackendProperties*** representing the information on the quantum gates features. This data structure is required for the *NoiseAdaptiveLayout* and *DenseLayout* placement strategies. Specifically, the latter was tested in two modalities: without using the *BackendProperties* (hardware-unaware), or with this additional information (hardware-aware).

The *BackendProperties* for each tested backend were constructed as follows:

- For the superconducting backend, they are extracted from the *FakeToronto* [62] class instance.

- For the trapped ions backend, they were constructed using the available **CX error rates**.

- For the NMR and quantum dot backends, they were constructed using the available **CZ error rates**. Since the *NoiseAdaptiveLayout* placement algorithm requires the CX error rates, to perform a fair comparison the CZ error rates in the BackendProperties were represented as CX error rates.

- For the NMR, quantum dots and trapped ions backends, the readout error rates information was not used, as this was not available. Furthermore, this information is not an optimisation metric considered during the layout synthesis phase by the implemented heuristics.

t|ket⟩ placement and routing

All the quantum circuits used for the benchmarking of the proposed layout synthesis library were also placed and routed using the t|ket⟩ compilation toolchain, in order to perform a comparison. Specifically the **pytket library version 1.3.0** [21, Sec. 1.4] was utilised.

The applied t|ket⟩'s placement strategies are:

- *LinePlacement* [11, Sec. 3.2]

- *GraphPlacement* [21, Sec. 7.1]

- *NoiseAwarePlacement* [21, Sec. 9.2]

All these placement strategies are well explained in Section 4.1.2.

All the four target backends presented in Section 7.1.2 were modelled using the **pytket API**. Specifically, for each backend, two data-structures are required:

- An ***Architecture*** object containing the list of the allowed nodes interactions only.

- A **dictionary** specifying the two-qubits error rates. In particular, this data-structure is required only for the *NoiseAwarePlacement*, being the only t|ket⟩'s hardware-aware strategy tested. This placement heuristic can also use the single-qubit and readout error rates. During the benchmarking phase, the algorithm was tested using two modalities: one in which the single-qubit error rates were not passed, and another in which this additional information was also passed. The readout error rates information was never used instead.

The t|ket⟩'s routing strategy applied is the default one, using the **lexicographical comparison**, presented in Section 5.1.2.

Since the t|ket⟩'s placement strategies may produce a *partial mapping*, **the placement alone could not be used for comparison**. The results of the complete layout synthesis procedure, using all the combinations of the previously cited placement and routing strategy, are presented in this chapter.

7.2 Functional equivalence - evaluation of results

Figures 7.1 to 7.5 underline the correctness of the implemented heuristics. Specifically, for the majority of the circuits under test, both the fidelity (approximated) and the Hellinger fidelity are very close to one. Moreover, also the KLD is near zero for the majority of the tested quantum circuits. The circuits with the highest KLD (also with the lowest Hellinger and approximated fidelity) are the ones of the medium-sized set (the results are similar independently of the target technology), and the *ising_n10.qasm* circuit is the "worst" one. Indeed, not only is this result expected due to the large number of measured eigenstate in this circuit, but also the fidelities and KLD do not exceed the acceptable thresholds, still validating the proposed layout synthesis library.

All of the implemented heuristics were subject to functional equivalence checking, using different parameter configuration. In this section, the results are reported only for some combination of placement and routing heuristics, and for a specific configuration. The results are independent of the heuristic applied and are **more related to the quantum circuit set**, proving even more strongly their correctness.

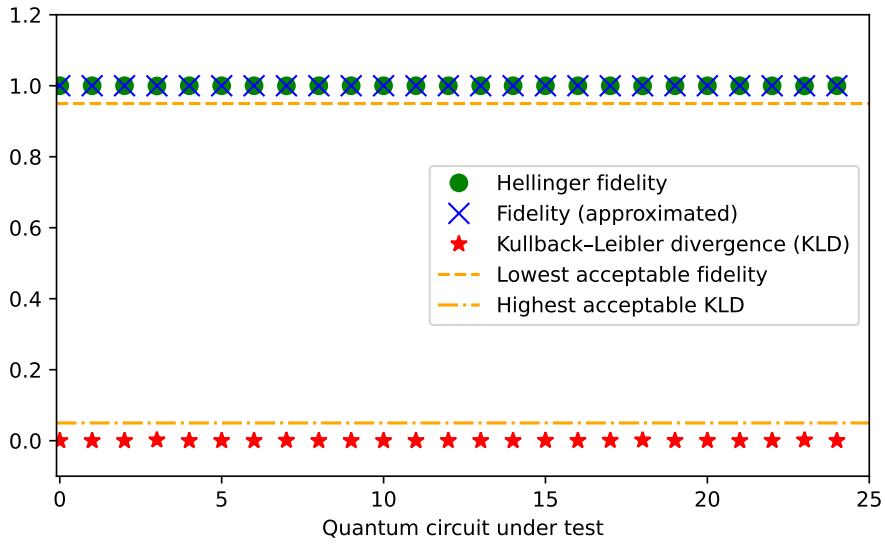

Figure 7.1. Results of the functional equivalence checking test for the *SimulatedAnnealingHardwareAwareMapping* placement strategy. The quantum circuits under test are the small-sized circuits, and the target technology is trapped ions.

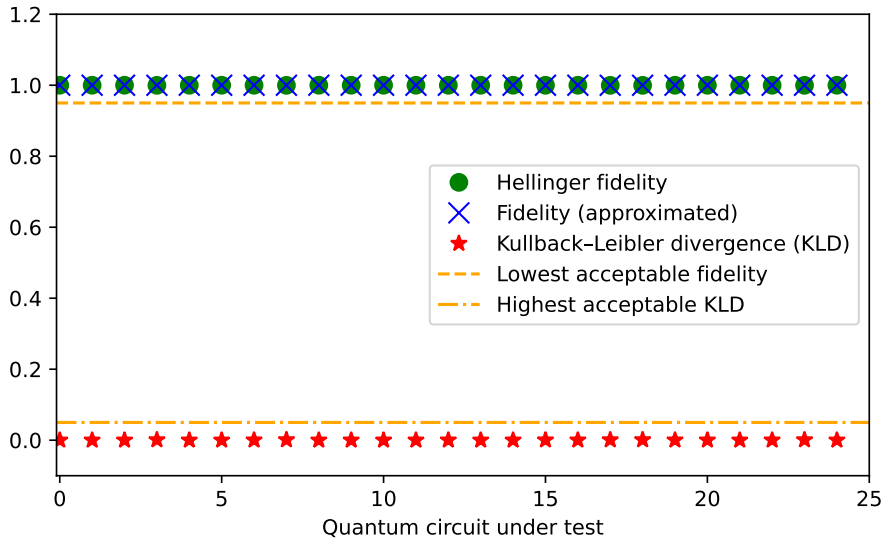

Figure 7.2. Results of the functional equivalence checking test for the *TrivialMapping* placement strategy combined with the *BasicRouting* routing strategy. The quantum circuits under test are the small-sized circuits, and the target technology is superconducting.

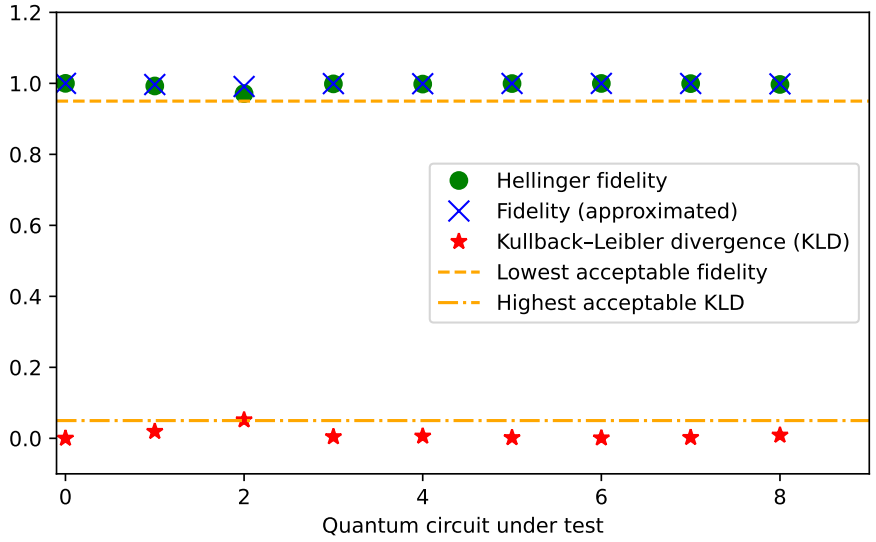

Figure 7.3. Results of the functional equivalence checking test for the *TrivialMapping* placement strategy combined with the *HardwareAwareRouting* routing strategy. The quantum circuits under test are the medium-sized circuits, and the target technology is superconducting.

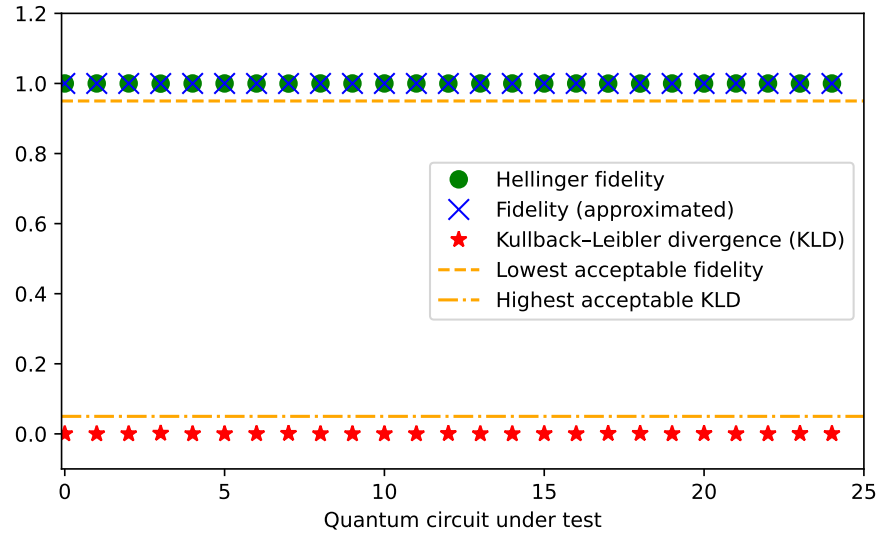

Figure 7.4. Results of the functional equivalence checking test for the *SimulatedAnnealingDenseMapping* placement strategy combined with the *HardwareAwareRouting* routing strategy. The quantum circuits under test are the small-sized circuits, and the target technology is quantum dots.

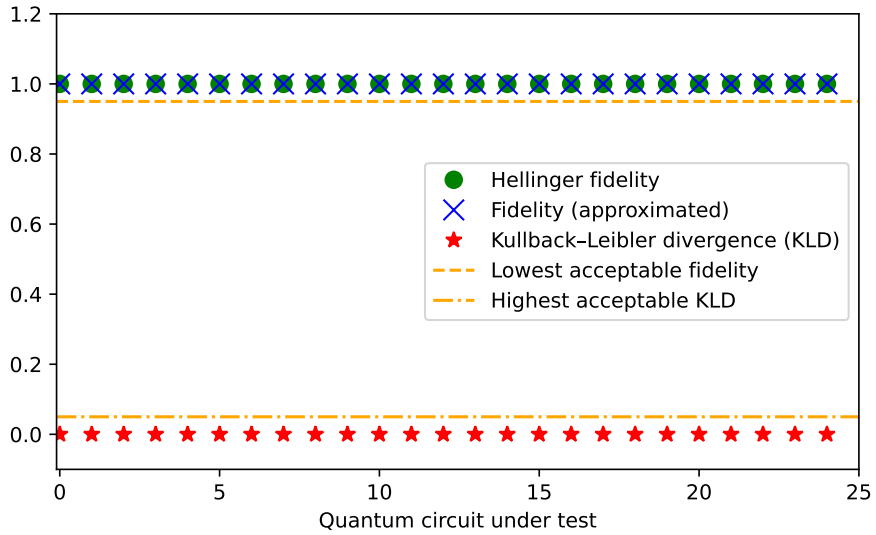

Figure 7.5. Results of the functional equivalence checking test for the *SimulatedAnnealingHardwareAwareMapping* placement strategy combined with the *HardwareAwareRouting* routing strategy. The quantum circuits under test are the large-sized circuits (only the first 25 circuits are shown), and the target technology is superconducting.

Circuit under test name	Number
adder_n4.qasm	0
basis_change_n3.qasm	1
basis_trotter_n4.qasm	2
bell_n4.qasm	3
cat_state_n4.qasm	4
deutsch_n2.qasm	5
dnn_n2.qasm	6
error_correctiond3_n5.qasm	7
fredkin_n3.qasm	8
grover_n2.qasm	9
hs4_n4.qasm	10
iswap_n2.qasm	11
linearsolver_n3.qasm	12
lpn_n5.qasm	13
pea_n5.qasm	14
qaoa_n3.qasm	15
qec_en_n5.qasm	16
qft_n4.qasm	17
qrng_n4.qasm	18
quantumwalks_n2.qasm	19
teleportation_n3.qasm	20
toffoli_n3.qasm	21
variational_n4.qasm	22
vqe_uccsd_n4.qasm	23
wstate_n3.qasm	24

Table 7.1. Small-sized quantum circuits set.

Circuit under test name	Number
adder_n10.qasm	0
dnn_n8.qasm	1
ising_n10.qasm	2
qaoa_n6.qasm	3
qpe_n9.qasm	4
sat_n11.qasm	5
simon_n6.qasm	6
vqe_uccsd_n6.qasm	7
vqe_uccsd_n8.qasm	8

Table 7.2. Medium-sized quantum circuits set.

Circuit under test name	Number
0410184_169.qasm	0
3_17_13.qasm	1
4_49_16.qasm	2
4gt11_84.qasm	3
4gt12-v1_89.qasm	4
4gt13-v1_93.qasm	5
4gt4-v1_74.qasm	6
4gt5_77.qasm	7
4mod5-bdd_287.qasm	8
4mod5-v1_24.qasm	9
4mod7-v1_96.qasm	10
9symml_195.qasm	11
adr4_197.qasm	12
aj-e11_165.qasm	13
alu-bdd_288.qasm	14
alu-v4_37.qasm	15
bv_n14.qasm	16
C17_204.qasm	17
clip_206.qasm	18
cm152a_212.qasm	19
cm42a_207.qasm	20
cm82a_208.qasm	21
cm85a_209.qasm	22
cnt3-5_180.qasm	23
co14_215.qasm	24

Table 7.3. Large-sized quantum circuits set (only the first 25 circuits are shown).

Quantum technology	K
Superconducting	0.9892
NMR	0.9893
Quantum dots	0.9994
Trapped ions	0.9789

Table 7.4. K penalisation factor used for the *circuit cost function C* computation.

7.3 Benchmarking results - Placement

In this section, the results of the benchmarking phase for the placement step alone are reported, shown in Figures 7.6 to 7.9. Specifically, in each figure, the **average number of non-allowed two-qubit interactions** (considering all the quantum circuits in each set) is plotted. All the results presented in this chapter suppose that the R_z **gates are implemented virtually**, since this is the typical condition when a quantum circuit in a real NISQ device is executed. Indeed, this information is used for the error rates computation in all the hardware-aware placement strategies (both the ones implemented in the presented library and the tested Qiskit's heuristics).
For the *Simulated Annealing Hardware-Aware Mapping* strategy, in the previously cited figures the configured coefficients are also indicated, in the form: $S\alpha_1 E\alpha_2 T\alpha_3$ (see Section 6.2.1 for more details about these coefficients).

7.3.1 Small-sized circuits set

Figure 7.6 underlines that for the small-sized quantum circuits set and for the superconducting technology, all the implemented heuristics **outperform the Qiskit's ones**, with a number of non-allowed interactions sensibly reduced. In particular, the implemented methods obtain an improvement of roughly 25% with respect to the IBM's compiler.
It was unexpected that the *TrivialMapping* strategy, being the simplest one, could have better results, for the small-sized circuits and superconducting technology, when compared to the smarter Qiskit's strategies. These results could be related to the fact that the metrics considered for the optimisation by the smart algorithms are slightly different and more complex than the one used for comparison in this section. Therefore, a placement which is worse than others from the number of non-allowed two-qubit interactions point of view, can be optimal considering other figure of merits, such as the number of swap gates required in the routing stage. For example, the aim of the *Qiskit SabreLayout* is the minimisation of the total number of swap gates added for the complete SABRE layout synthesis procedure. It is possible that to a higher number of non-allowed interactions obtained after the initial mapping, corresponds a lower number of swap gates added during the routing phase. Furthermore, the goal of the *Qiskit NoiseAdaptiveLayout* is more related to the error rates minimisation rather than the non-allowed interactions minimisation. The slight discrepancy between the comparison metric and the one used for the optimisation process of each algorithm is probably more evident on small circuits, where the solution space is reduced, allowing the trivial algorithm to have a good chance of delivering an optimal or suboptimal placement.
The situation is different for the small-sized circuits set for the quantum dots backend, as shown in Figure 7.9. Specifically, the *Qiskit Dense Layout* and the *Qiskit*

SabreLayout have **similar results to the ones offered by the proposed library**. The *Qiskit NoiseAdaptiveLayout* is instead the worst strategy for maximising the allowed two-qubit interactions.
Summarising the results for the small-sized circuits set, **the optimal placement strategy is the implemented *Simulated Annealing Hardware-Aware Mapping***, showing for both superconducting and quantum dots technology the maximum reduction of non-allowed two-qubit interactions.

7.3.2 Medium-sized circuits set

For the medium-sized circuits set, shown in Figure 7.7, the results are instead different. Specifically, the *Qiskit SabreLayout* and the *Qiskit NoiseAdaptiveLayout* show the **best results for reducing the non-allowed interactions**. Furthermore and surprisingly, for the other heuristics, the results show that the simplest implemented strategy (the *TrivialMapping*) has better results than all the others smarter algorithms. As for the small-sized circuits set, the explanation to this strange behaviour could be related to a different optimisation metric sought by these strategies.

7.3.3 Large-sized circuits set

For the large-sized circuits set, presented in Figure 7.8, **all the placement strategies have comparable results**. The *Simulated Annealing Hardware-Aware Mapping* and the *Qiskit SabreLayout* still show a slight improvement. Indeed, the good results of the SABRE reverse traversal mapping were expected (based on the published results in the associated literature), and the similarities with the implemented heuristic is encouraging for validating the proposed methodology.

7.3.4 Review of the results

In all the figures shown it is possible to notice that the implemented *TrivialMapping* and *Simulated Annealing Dense Layout* have identical results. The reason for this is that the latter strategy starts its iterative search using the trivial mapping as starting configuration. Indeed, for both the superconducting and the quantum dots backend, **the connectivity of this trivial mapping corresponds to the maximum one**, thus the two heuristics return the same mapping.

Summarising the presented evaluations, the implemented *Simulated Annealing Hardware-Aware Mapping* **seems to slightly outperforms** all the other heuristics for all the tested technologies and circuits sets, except for the medium-sized one.

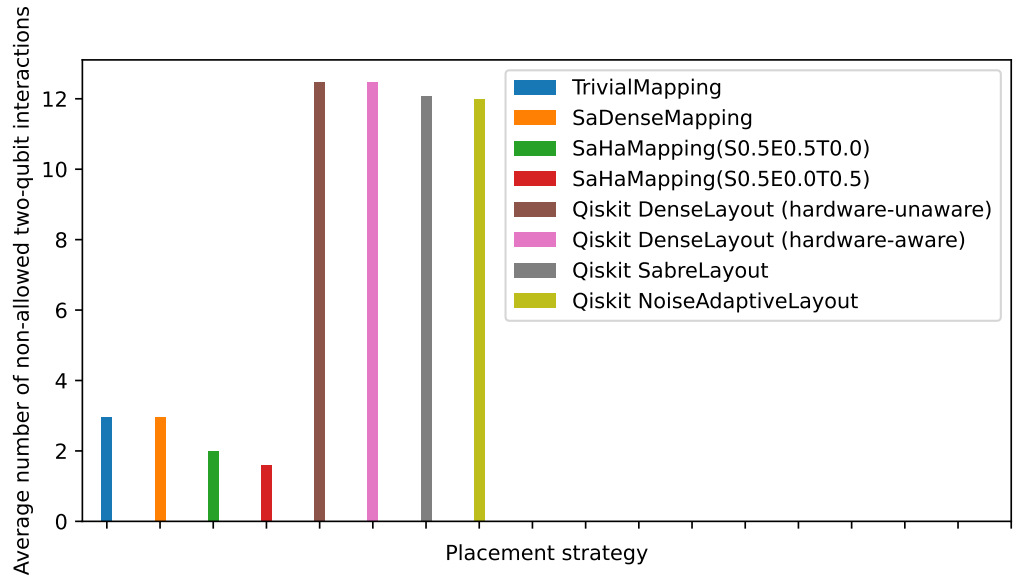

Figure 7.6. Results of the benchmarking test for the placement strategies. The quantum circuits under test are the small-sized circuits, and the target technology is superconducting.

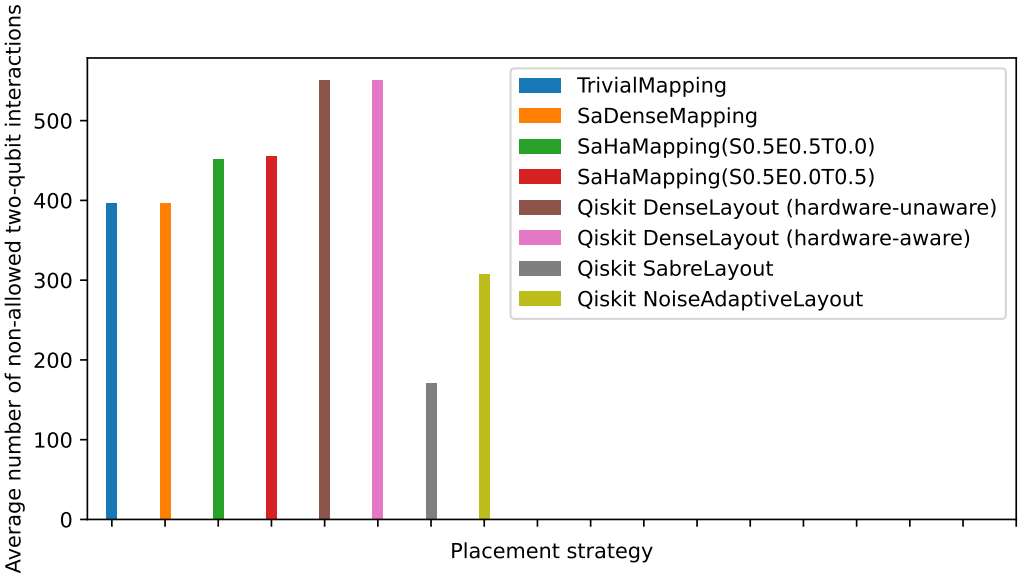

Figure 7.7. Results of the benchmarking test for the placement strategies. The quantum circuits under test are the medium-sized circuits, and the target technology is superconducting.

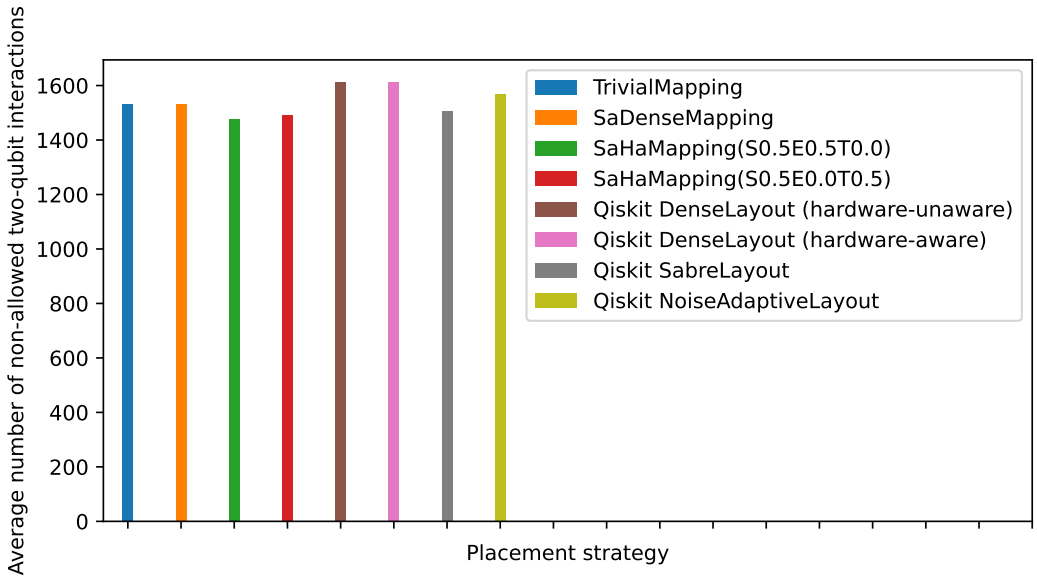

Figure 7.8. Results of the benchmarking test for the placement strategies. The quantum circuits under test are the large-sized circuits, and the target technology is superconducting.

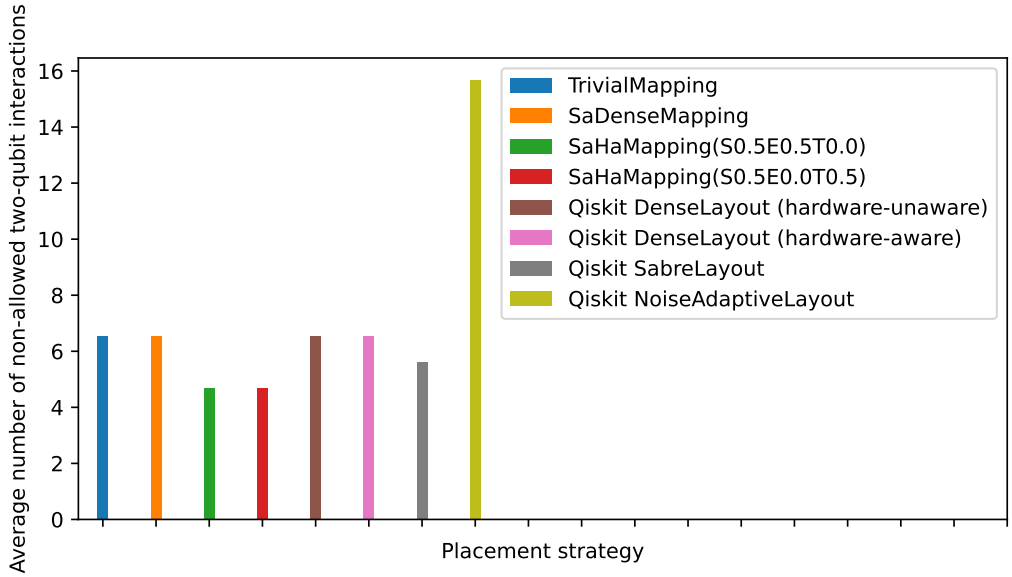

Figure 7.9. Results of the benchmarking test for the placement strategies. The quantum circuits under test are the small-sized circuits, and the target technology is quantum dots.

7.4 Benchmarking results - Placement and Routing - non-fully-connected technologies

In this section, the results of the benchmarking phase for the placement and routing step (the **complete layout synthesis**) of the non-fully-connected technologies (superconducting and quantum dots) are reported, shown in Figures 7.10 to 7.21. Specifically, for each technology, the **average number of swap gates added**, the **average execution time** and the **average circuit cost function** C are plotted. All the results presented in this section suppose that the R_z **gates are implemented virtually**, since this is the typical condition when executing a quantum circuit in a real NISQ device. Indeed, this information is used for the error rates computation in all the hardware-aware placement and routing strategies (both the ones implemented in the presented library and the tested Qiskit's and t|ket⟩'s heuristics).
For the *Simulated Annealing Hardware-Aware Mapping* and the *Hardware-Aware Routing* strategies, in the previously cited figures, the configured coefficients are also indicated, in the form: $S\alpha_1 E\alpha_2 T\alpha_3$ (see Section 6.2.1 for more details about these coefficients).

7.4.1 Small-sized circuits set

For the small-sized quantum circuits and for the superconducting technology, as it is possible to notice in Figures 7.10 to 7.12, the *Hardware-Aware Routing* algorithm **always outperforms all the Qiskit's strategies** (in all the three compared metrics), with results **comparable** to the ones obtained by the t|ket⟩'s compiler. Moreover, the combination of the implemented *Simulated Annealing Hardware-Aware Mapping* and *Hardware-Aware Routing* shows a **slight improvement** even compared to the t|ket⟩'s strategies, in terms of minimum average value of swap gates added, execution time and circuit cost function C.
An unexpected result noticeable in these plots is that the simplest implemented layout synthesis strategy (*Trivial Mapping* plus *Basic Routing*) shows results **not so far with respect to the smarter ones**, and sometimes **even better** than the Qiskit's heuristics. However, remembering that all of these strategies are heuristics solutions and not exact algorithms, the outcomes are reasonable. Furthermore, because the tested circuits are short, the number of feasible solutions (layouts) might be restricted, with only a little margin for improvement.

For the small-sized circuits set and for the quantum dots technology, shown in Figures 7.19 to 7.21, all the heuristics implemented for the proposed library, the Qiskit'ones and t|ket⟩'ones seem to have **consistent results**. In particular, the execution time and circuit cost function C are almost identical independently on the applied layout synthesis strategies. It might be that due to the tested device coupling-graph (linear topology), for the tried circuits, the possible improvements

in terms of execution time and error rate are quite limited, independently on the adopted strategy.

7.4.2 Medium-sized circuits set

The results of the medium-sized circuits set for the superconducting technology, shown in Figures 7.13 to 7.15, underline that the optimal swap gate count and execution time is strongly related to the routing strategy, and almost independent of the initial mapping applied. Specifically, the worst results are obtained with the implemented *Basic Routing* and the *Qiskit StochasticSwap*. Indeed, these outcomes affirms the stronger optimisation power of the other tested heuristics. The best results (for these two metrics) are obtained with all the t|ket⟩'s strategies and with the combination of *Qiskit SabreSwap* and *Qiskit SabreRouting* (that is, the complete SABRE layout synthesis).
The implemented *Hardware-Aware Routing* algorithm shows a slightly worse optimisation compared to *SabreRouting*, for all the three considered metrics. Being SABRE the father of the implemented *Hardware-Aware Routing* algorithm, the similarity between the obtained results was expected, even if in this particular scenario, the latter obtained an inferior optimisation.

7.4.3 Large-sized circuits set

For the large-sized circuits set and for the superconducting technology, shown in Figures 7.16 to 7.18, the results are **consistent for all the three compared metrics**. Specifically, the **worst results** are obtained with the *Basic Routing* strategy (independently of the placement heuristic used) and all the combinations of the Qiskit's tested heuristics. The **best results** are obtained instead with the implemented *Simulated Annealing Hardware-Aware Mapping* and *Hardware-Aware Routing* and all the tested t|ket⟩'s strategies. Indeed, the *Hardware-Aware Routing* has almost identical results to the t|ket⟩'s *LexiRouteRouting* regarding swap gate count and execution time, showing even better results for the circuit cost function C.
The obtained findings agree with the expectations, demonstrating that the *Hardware-Aware Routing*, by leveraging calibration data knowledge, outperforms its father algorithm (the complete SABRE layout synthesis) in all the three metrics.

7.4.4 Review of the results

Summarising the obtained results, on average the combination of *Simulated Annealing Hardware-Aware Mapping* and the *Hardware-Aware Routing* outperforms the *SABRE* routing strategy, even if the latter has some better results in specific tested scenarios (like for the medium-sized circuits set).

It is surprising that the t|ket⟩'s *GraphPlacement* plus *LexiRouteRouting* strategy, being hardware-unaware and thus **without the information on the quantum gates features**, have comparable results to the implemented hardware-aware layout synthesis procedures. Nevertheless, the advantage of the Cambridge Quantum's compiler in terms of computational capabilities, being completely developed in C++, is most probably exploited to obtain these good results.

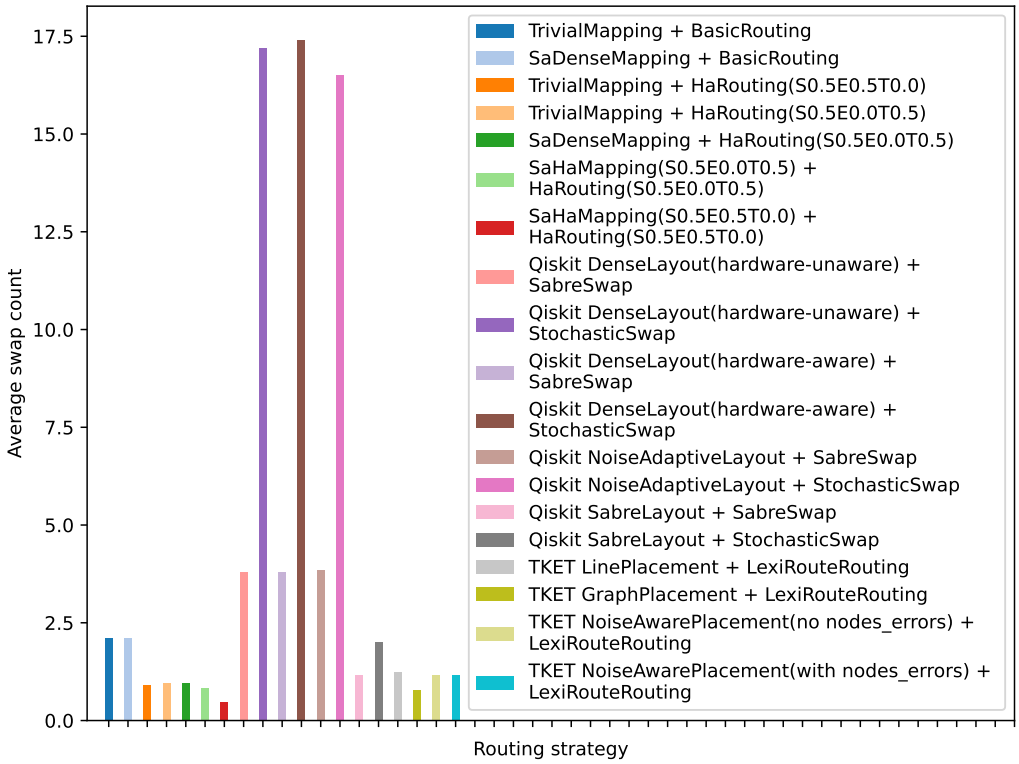

Figure 7.10. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the small-sized circuits, and the target technology is superconducting.

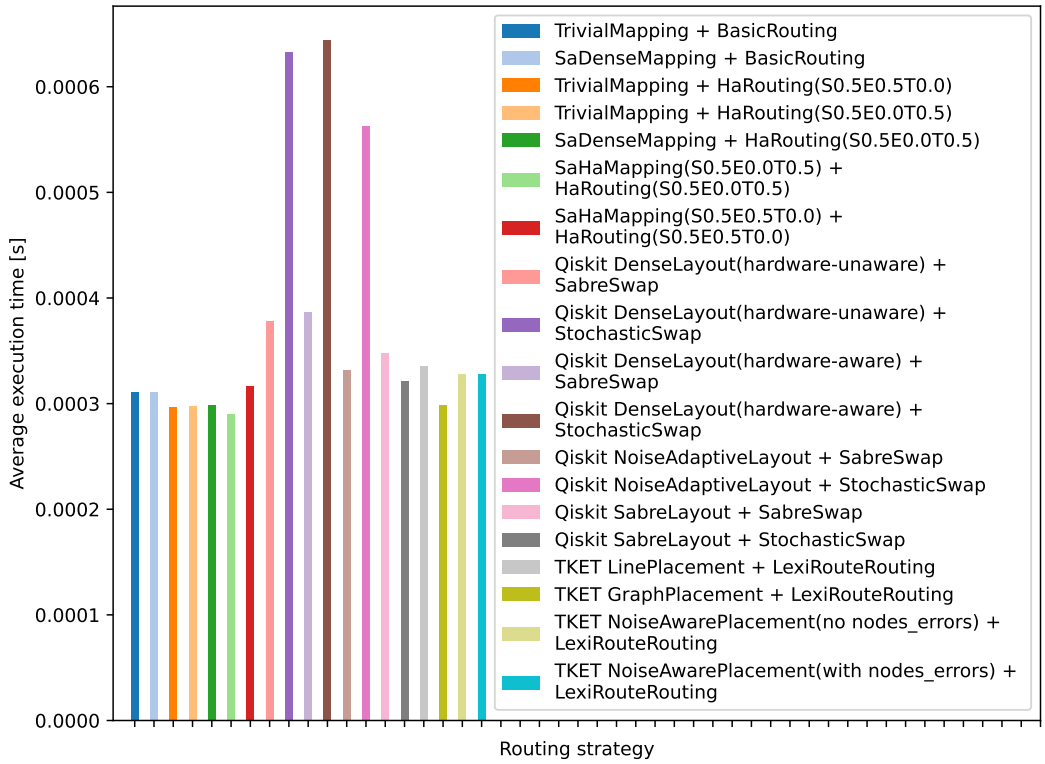

Figure 7.11. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the small-sized circuits, and the target technology is superconducting.

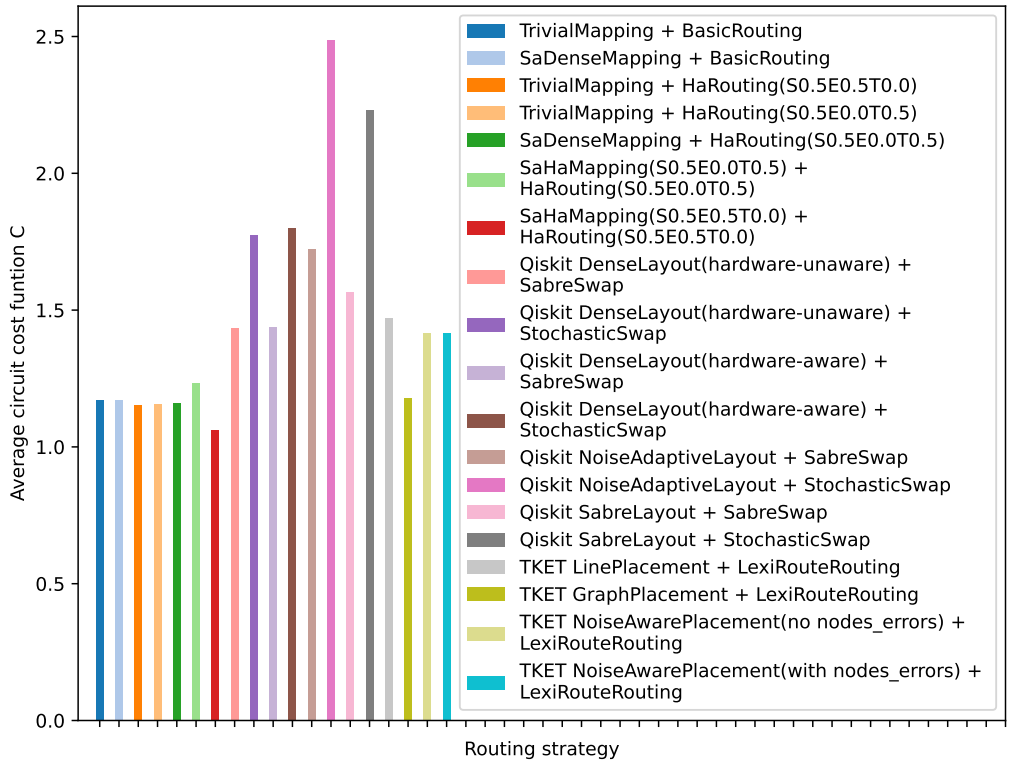

Figure 7.12. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the small-sized circuits, and the target technology is superconducting.

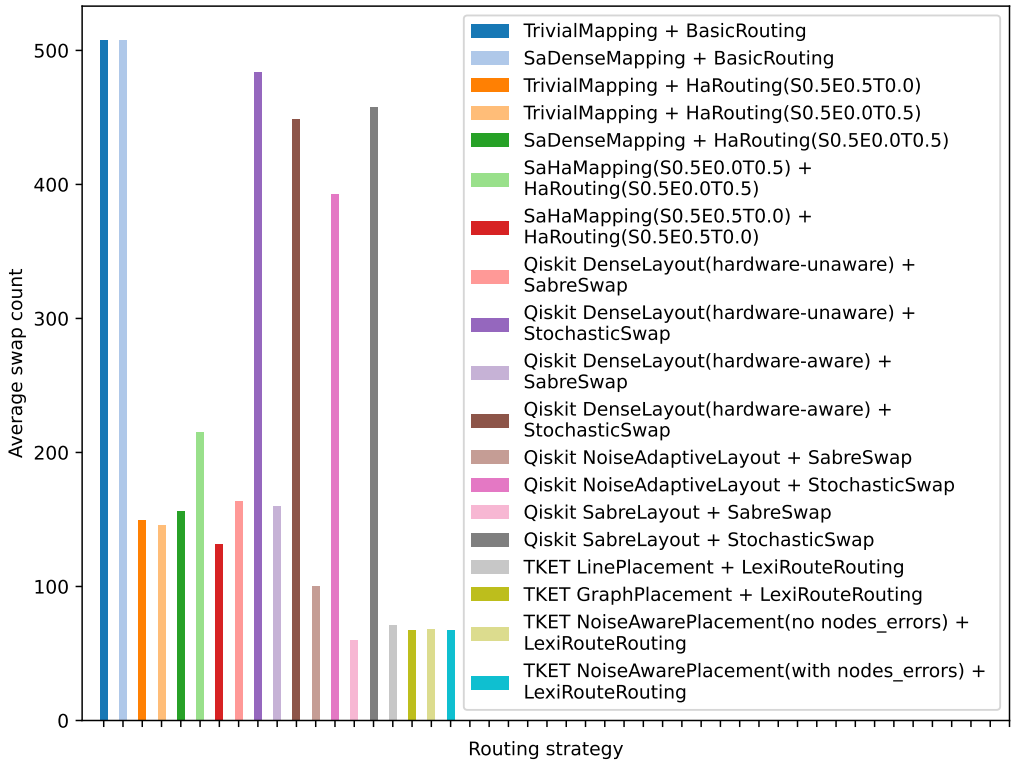

Figure 7.13. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the medium-sized circuits, and the target technology is superconducting.

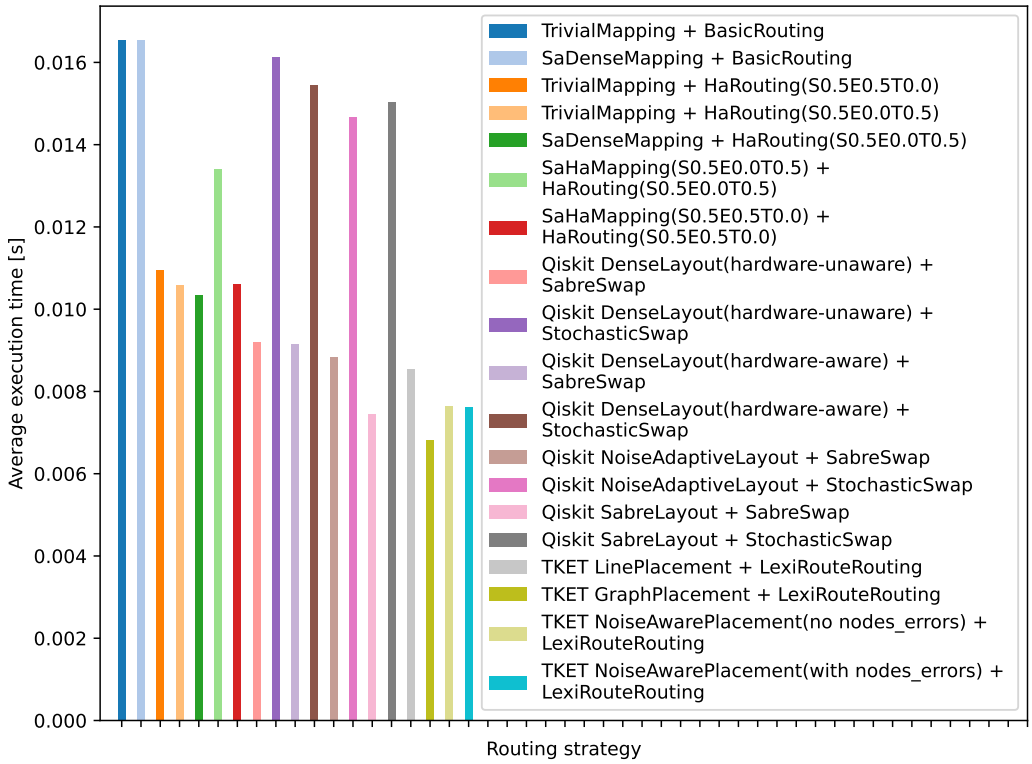

Figure 7.14. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the medium-sized circuits, and the target technology is superconducting.

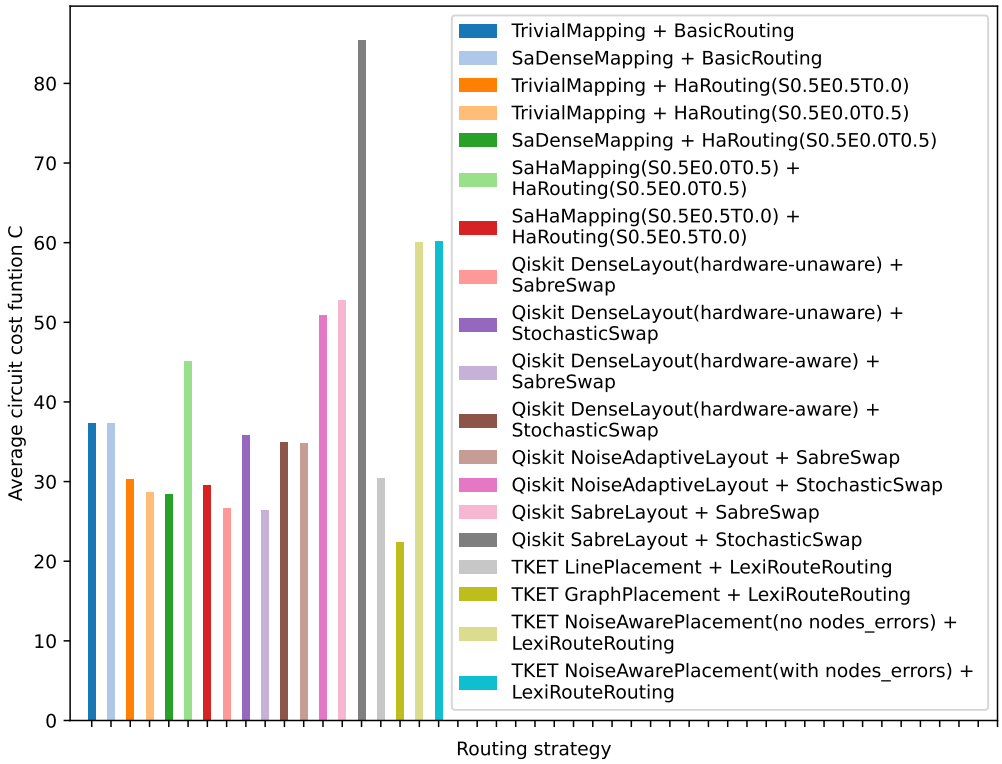

Figure 7.15. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the medium-sized circuits, and the target technology is superconducting.

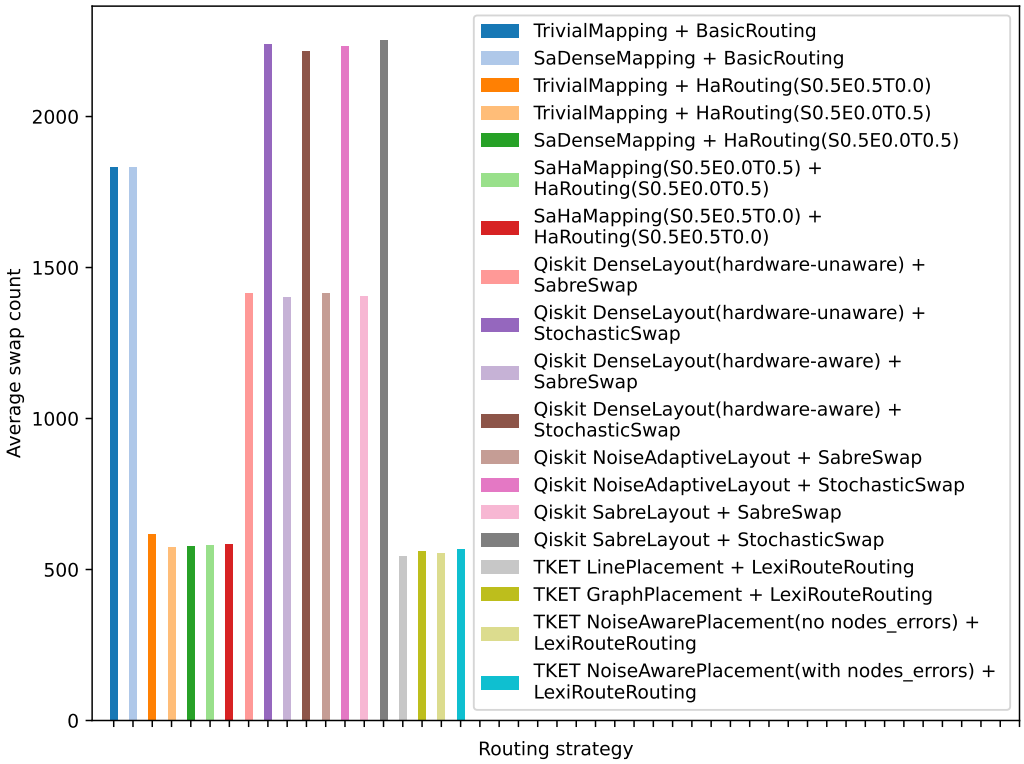

Figure 7.16. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the large-sized circuits, and the target technology is superconducting.

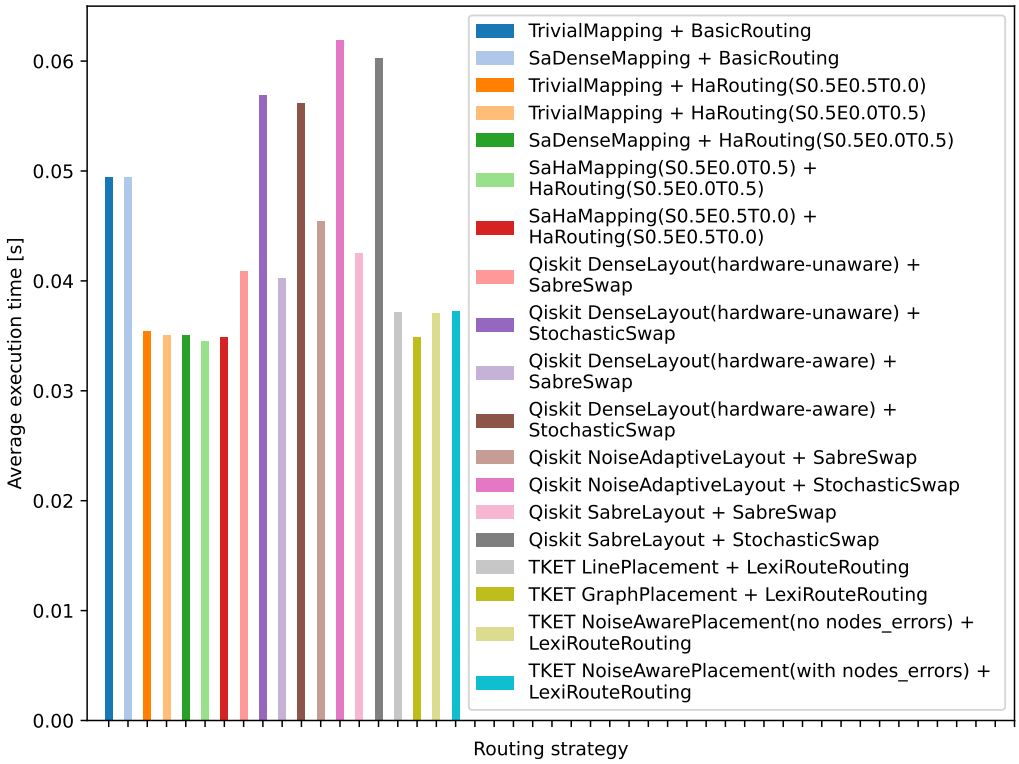

Figure 7.17. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the large-sized circuits, and the target technology is superconducting.

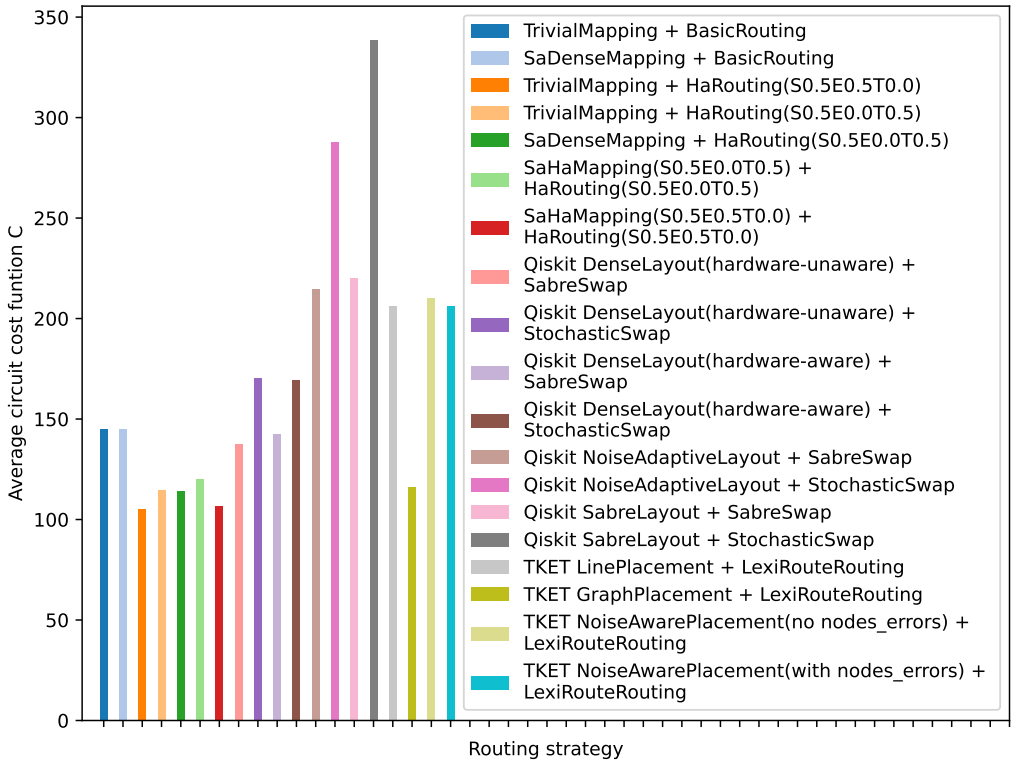

Figure 7.18. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the large-sized circuits, and the target technology is superconducting.

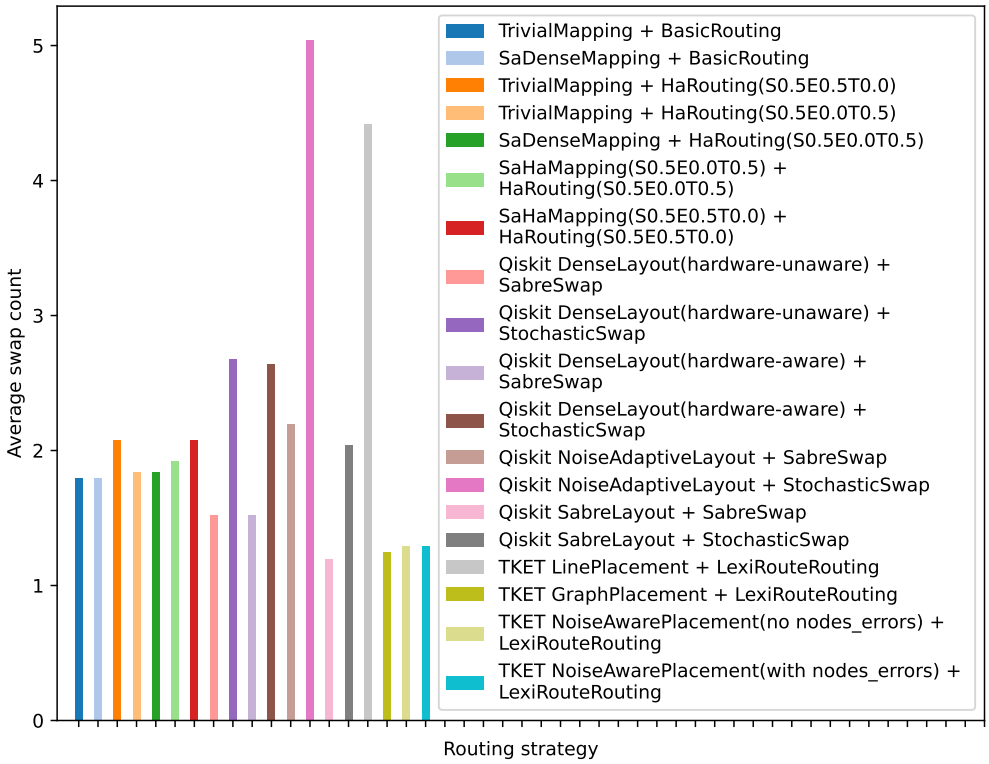

Figure 7.19. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the small-sized circuits, and the target technology is quantum dots.

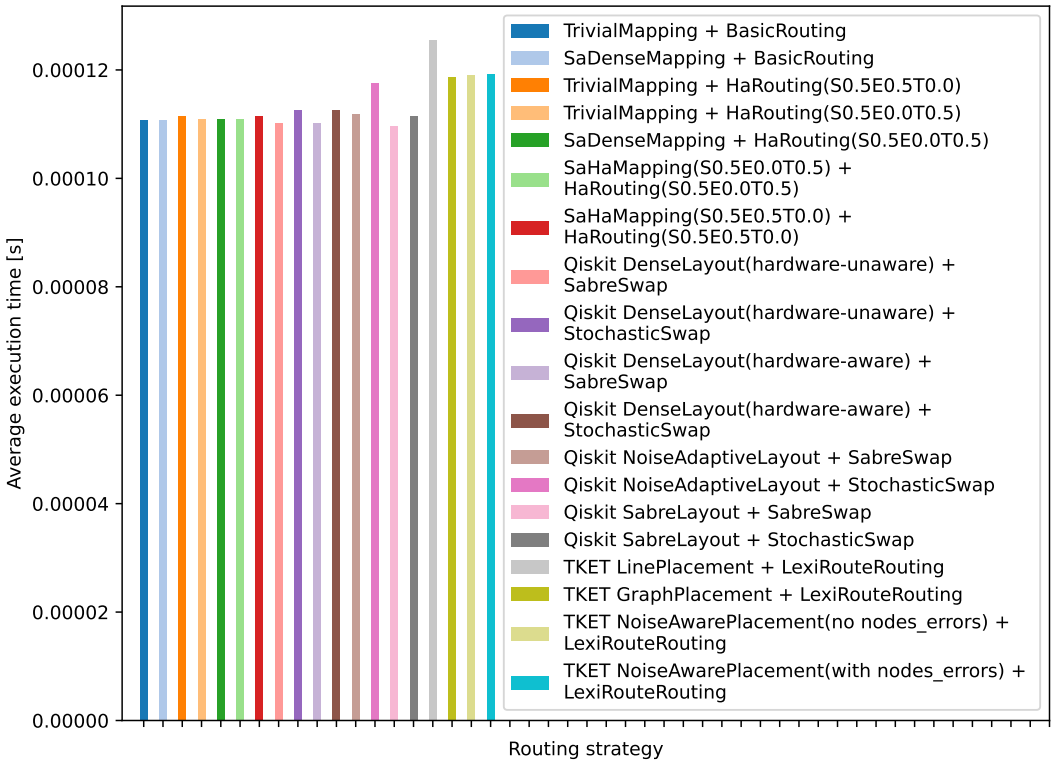

Figure 7.20. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the small-sized circuits, and the target technology is quantum dots.

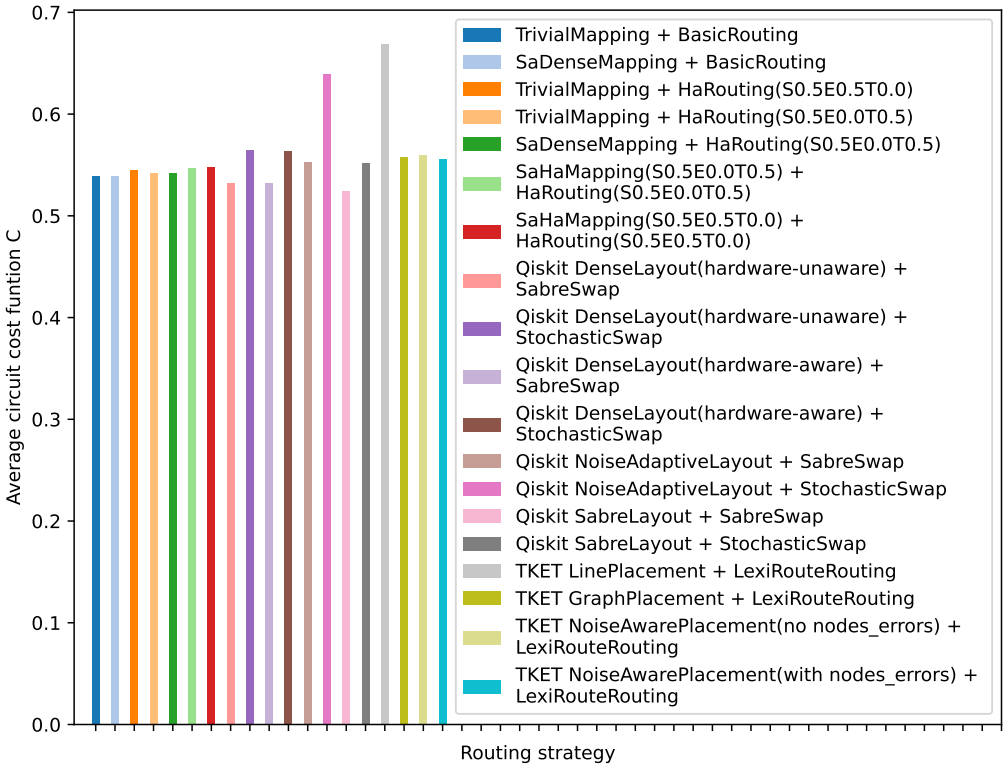

Figure 7.21. Results of the benchmarking test for the placement and routing strategies. The quantum circuits under test are the small-sized circuits, and the target technology is quantum dots.

7.5 Benchmarking results - Placement and Routing - fully-connected technologies

In this section, the results of the benchmarking phase for the placement and routing step (the **complete layout synthesis**) of the fully-connected technologies (NMR and trapped ions) are reported, shown in Figures 7.22 to 7.29. Specifically, for each technology, the **average execution time** and the **average circuit cost function** C are plotted. All the results presented in this section suppose that the R_z **gates are implemented virtually**, since this is the typical condition when executing a quantum circuit in a real NISQ device. Indeed, this information is used for the error rates computation in all the hardware-aware placement and routing strategies (both the ones implemented in the presented library and the tested Qiskit's and t|ket⟩'s heuristics). Moreover, for benchmarking the *Hardware-Aware Routing* strategy, the connectivity of the two fully-connected backends is broken (as explained in Section 6.3.1). In details, for the NMR backend a minimum J-coupling threshold of 1.47 Hz is set, and for the trapped ions backend the maximum MS gate time and error rate are set to respectively 504 µs and 0.0126.
For the *Simulated Annealing Hardware-Aware Mapping* and the *Hardware-Aware Routing Smart* strategies, in the previously cited figures, the configured coefficients are also indicated, in the form: $S\alpha_1 E\alpha_2 T\alpha_3$ (see Section 6.2.1 for more details about these coefficients).

7.5.1 Small-sized and medium-sized circuits set

For the small-sized and medium-sized circuits set, for both NMR and trapped ions, the results seems very similar. Indeed, all the implemented strategies seems to have the best optimisation in terms of execution time. The *Hardware-Aware Routing Smart* algorithm has, for some circuits, obtained a slightly powerful optimisation, but the average is not significantly affected.

In order to benchmark the strength of the implemented routing strategies for fully-connected technologies, and compare the results with the strategy of simply not adding any swap gates, Tables 7.5 to 7.7 show the results of respectively: *TrivialMapping* alone, *TrivialMapping* plus the *Hardware-Aware Routing* strategy, *TrivialMapping* plus the *Hardware-Aware Routing Smart* strategy. All of these strategies were tested considering the NMR backend.
The results underline that all the circuits for which *Hardware-Aware Routing* and *Hardware-Aware Routing Smart* does not add any swap gates (swap count is equal to zero) have the same execution times as when *TrivialMapping* was used alone. On the other hand, for circuits in which these strategies adds swap gates, the execution times are always reduced. **These results demonstrate that even if the technologies are fully-connected, by using smarter routing strategies some optimisations are still reachable**. The obtained results are not a surprise

for the *Hardware-Aware Routing Smart* algorithm, since it explicitly checks that the overhead of inserting a swap gate is repaid, but still the average are important due to its heuristic nature.

7.5.2 Large-sized circuits set

Remarkable results can be observed in Figure 7.28. There, it is noticeable that the implemented *Hardware-Aware Routing Smart* algorithm improves the average total execution time, **reaching the best results** against all the other heuristics, both the implemented ones and the Qiskit's and t|ket⟩'s solutions.
These promising results underline that for the fully-connected technologies in which the strength of the two-qubit interactions varies (meaning different execution time and/or error rate), using a smart algorithm that prefers the stronger interactions could lead to a better optimised final quantum circuit.

7.5.3 Review of results

For all the tested circuits sets and for both the two fully-connected technologies, only the implemented routing strategies adds some swap gates, aiming to reach a stronger optimisation. The Qiskit's and t|ket⟩'s heuristics only modified the quantum circuit during the placement step. Indeed, all of these major compilers assumes that being the devices fully-connected, the routing procedure does not require performing any modification on the quantum circuit.

The implemented routing strategies for the fully-connected devices try a different approach. Instead of accepting the quantum circuit as executable, an attempt to optimise the final execution time is made. Specifically, the available information on the gate features are exploited, checking if it might be worth to pay the overhead of inserting swap gates to avoid the weaker interactions.
Extremely good results are obtained using the *Hardware-Aware Routing Smart* algorithm, that inserts swap gates to bring the interactions towards the stronger interacting nodes. The obtained outcomes underline that in the worst scenario, this algorithm does not modify at all the quantum circuit's main figure of merits. In the best scenario, however, the execution time is minimised. The obtained results strongly suggests that more research on the routing procedures for fully-connected devices must be explored.

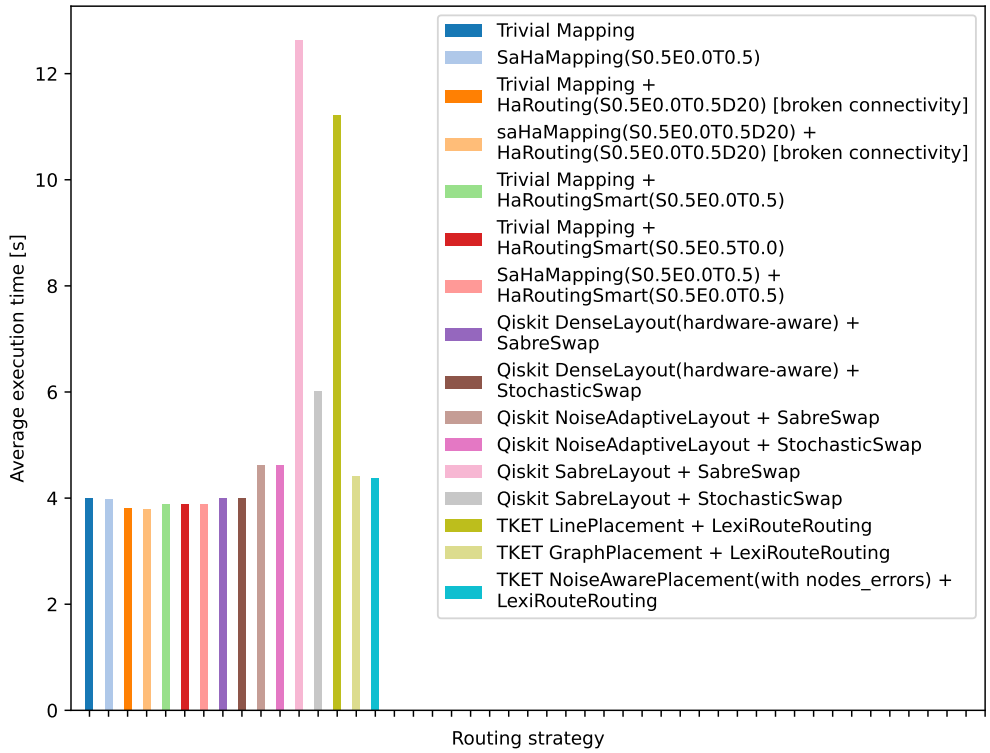

Figure 7.22. Results of the benchmarking test for the placement and routing strategies of the fully-connected technologies. The quantum circuits under test are the small-sized circuits, and the target technology is NMR.

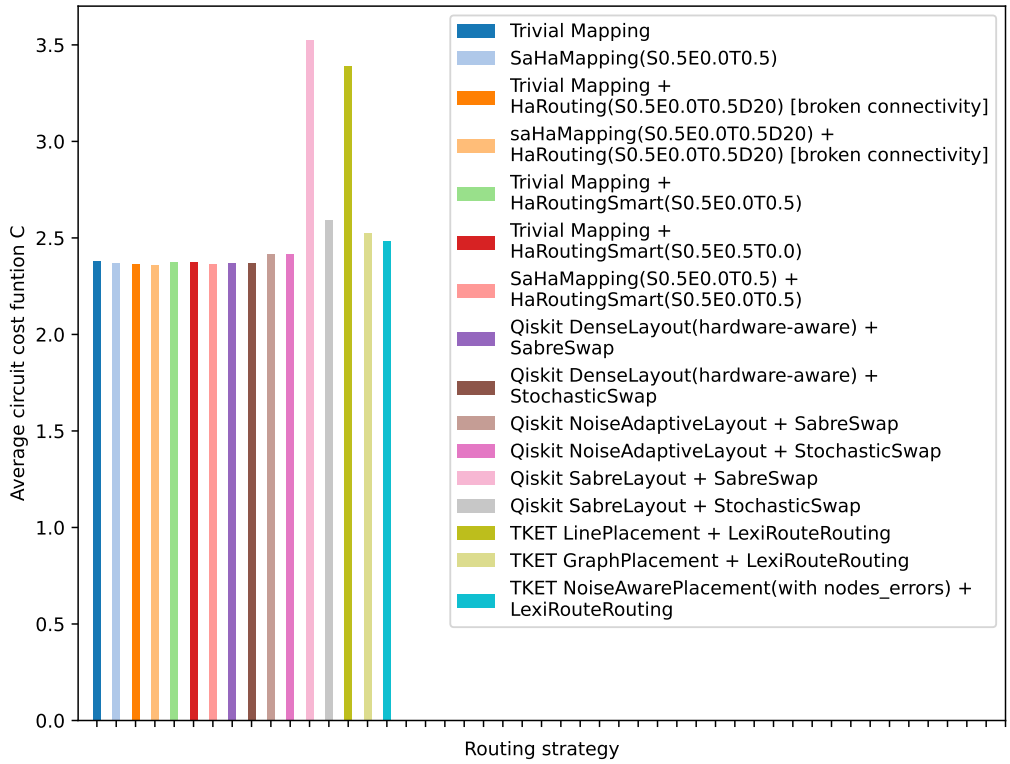

Figure 7.23. Results of the benchmarking test for the placement and routing strategies of the fully-connected technologies. The quantum circuits under test are the small-sized circuits, and the target technology is NMR.

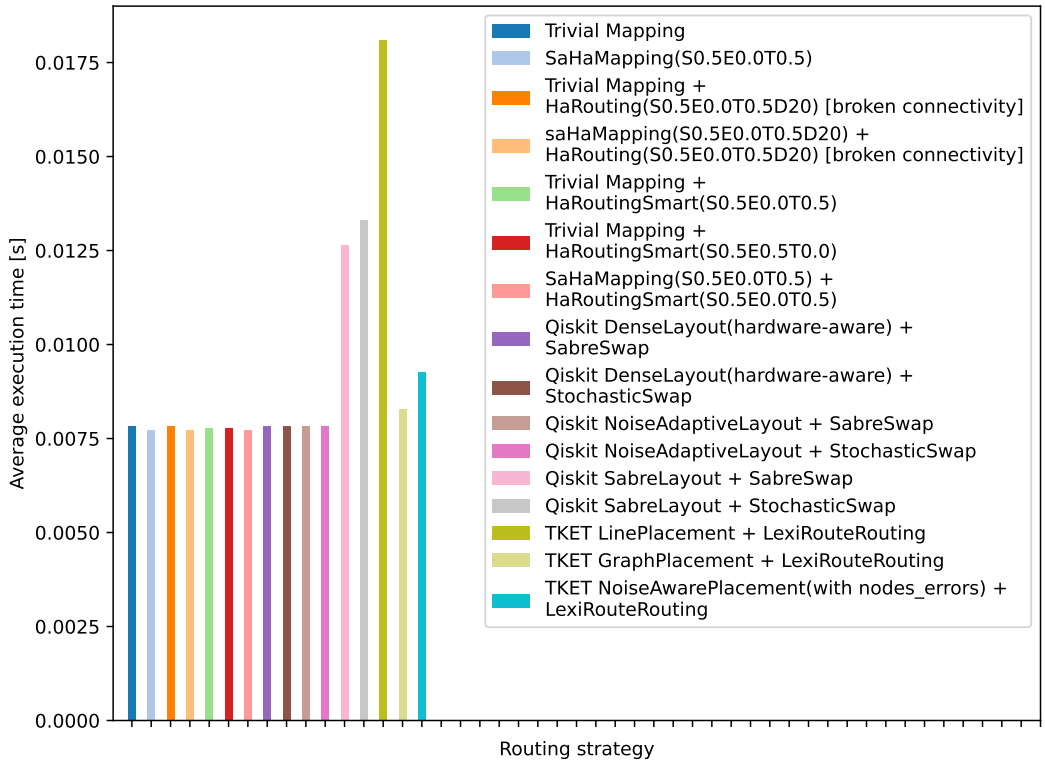

Figure 7.24. Results of the benchmarking test for the placement and routing strategies of the fully-connected technologies. The quantum circuits under test are the small-sized circuits, and the target technology is trapped ions.

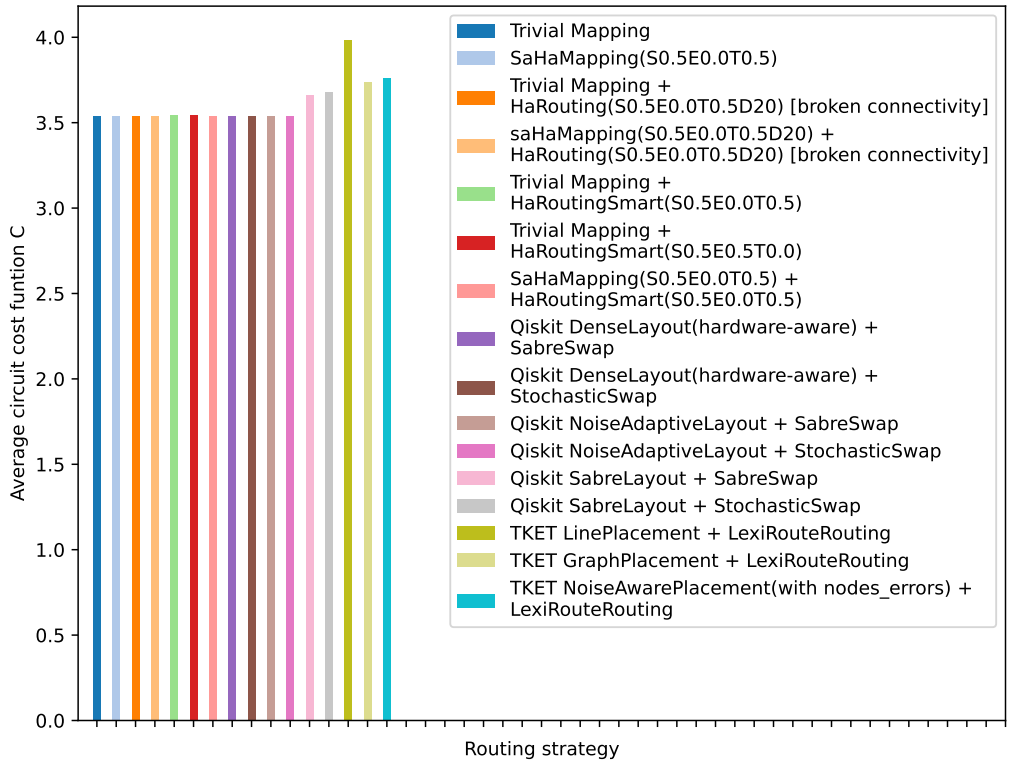

Figure 7.25. Results of the benchmarking test for the placement and routing strategies of the fully-connected technologies. The quantum circuits under test are the small-sized circuits, and the target technology is trapped ions.

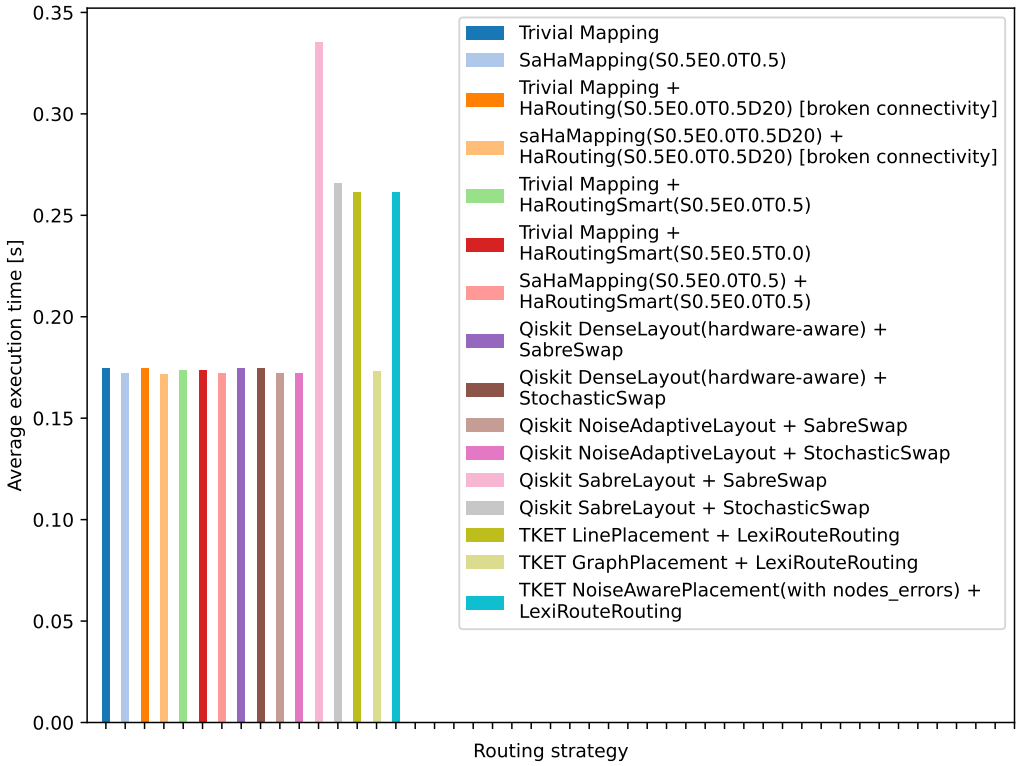

Figure 7.26. Results of the benchmarking test for the placement and routing strategies of the fully-connected technologies. The quantum circuits under test are the medium-sized circuits, and the target technology is trapped ions.

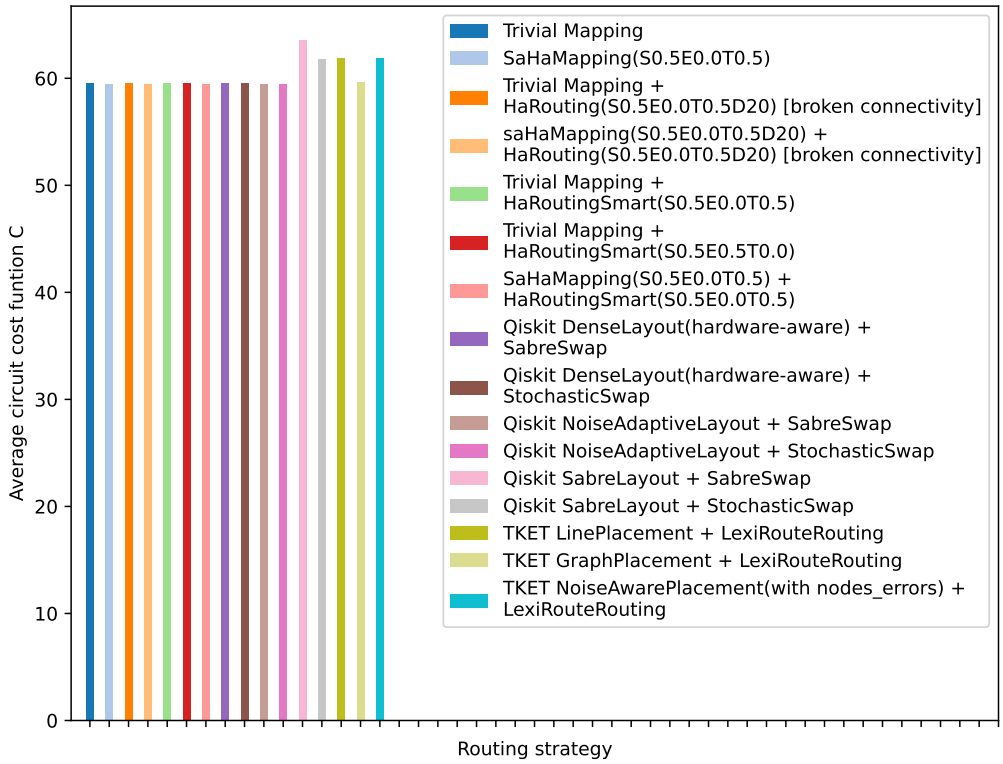

Figure 7.27. Results of the benchmarking test for the placement and routing strategies of the fully-connected technologies. The quantum circuits under test are the medium-sized circuits, and the target technology is trapped ions.

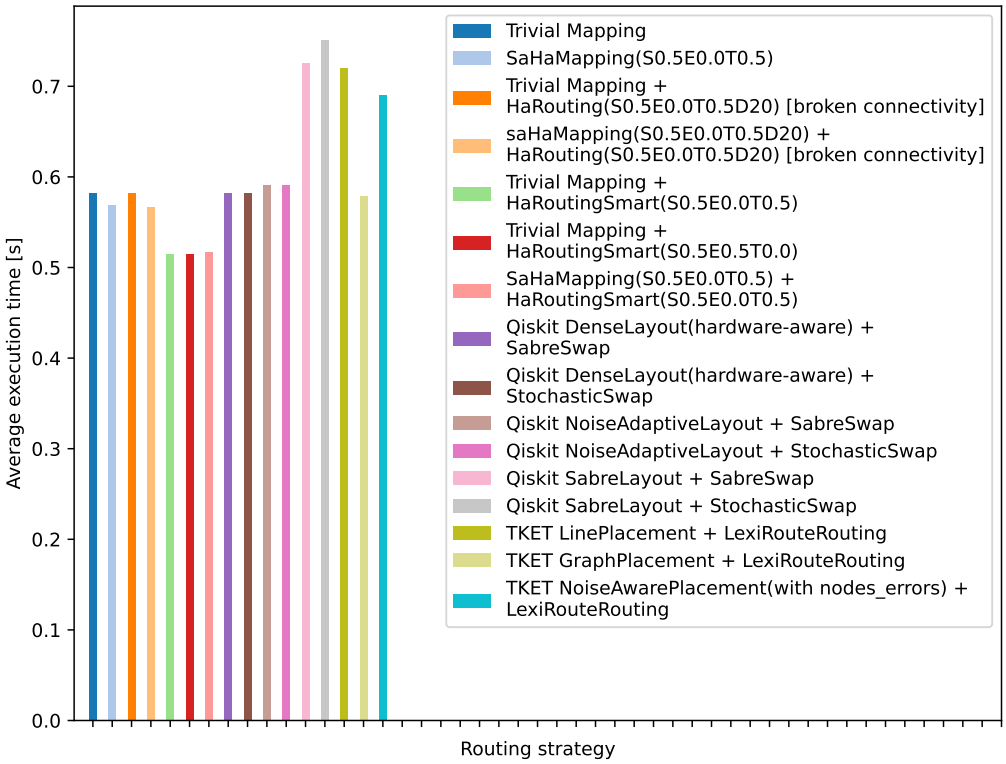

Figure 7.28. Results of the benchmarking test for the placement and routing strategies of the fully-connected technologies. The quantum circuits under test are the large-sized circuits, and the target technology is trapped ions.

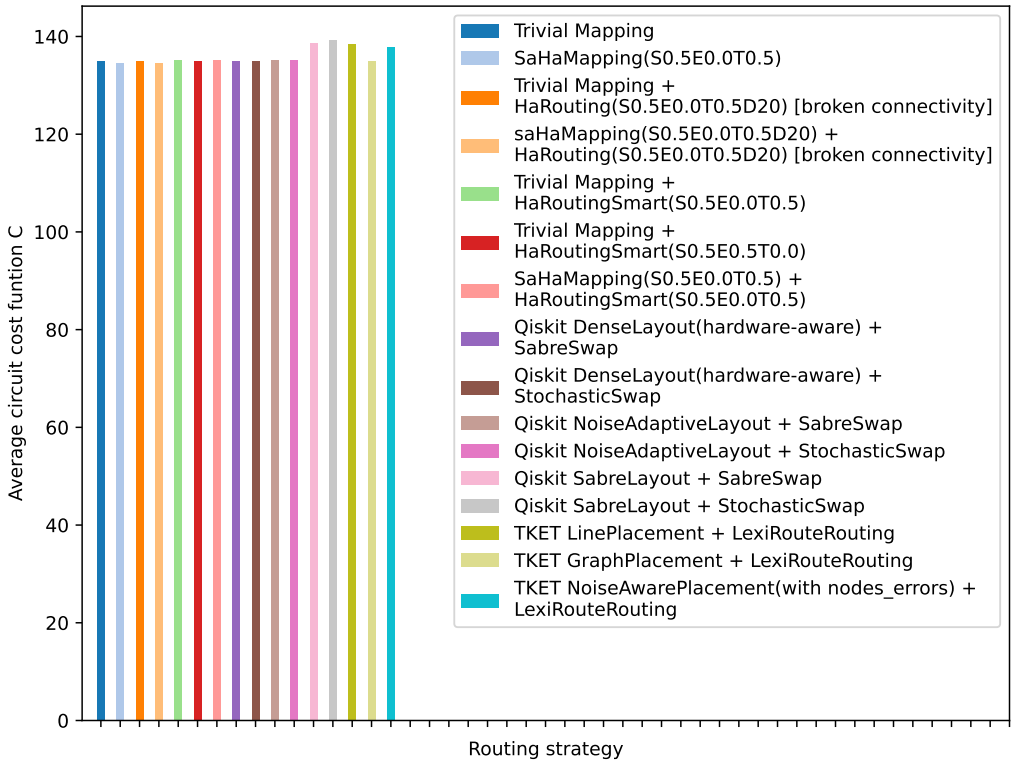

Figure 7.29. Results of the benchmarking test for the placement and routing strategies of the fully-connected technologies. The quantum circuits under test are the large-sized circuits, and the target technology is trapped ions.

	TrivialMapping	
Circuit name	**Swap count**	**Execution time [s]**
adder_n4.qasm	0	0.944301
basis_change_n3.qasm	0	0.547967
basis_trotter_n4.qasm	0	52.276457
bell_n4.qasm	**0**	**1.295001**
cat_state_n4.qasm	0	0.243986
deutsch_n2.qasm	0	0.183303
dnn_n2.qasm	0	4.857846
fredkin_n3.qasm	**0**	**1.491774**
grover_n2.qasm	0	0.532622
hs4_n4.qasm	0	0.992432
iswap_n2.qasm	0	0.283597
linearsolver_n3.qasm	0	0.712982
qaoa_n3.qasm	**0**	**1.376494**
qft_n4.qasm	**0**	**2.639705**
qrng_n4.qasm	0	0.124513
quantumwalks_n2.qasm	0	1.525190
teleportation_n3.qasm	0	0.263107
toffoli_n3.qasm	**0**	**1.352919**
variational_n4.qasm	0	1.419673
vqe_uccsd_n4.qasm	0	9.143029
wstate_n3.qasm	**0**	**1.695306**

Table 7.5. Results of the benchmarking test for the *TrivialMapping* strategy of the fully-connected technologies. The quantum circuits under test are the small-sized circuits, and the target technology is NMR.

	TrivialMapping + HaRouting(S0.5E0.0T0.5)	
Circuit name	**Swap count**	**Execution time** [s]
adder_n4.qasm	0	0.944301
basis_change_n3.qasm	0	0.547967
basis_trotter_n4.qasm	0	52.276457
bell_n4.qasm	**1**	**0.961430**
cat_state_n4.qasm	0	0.243986
deutsch_n2.qasm	0	0.183303
dnn_n2.qasm	0	4.857846
fredkin_n3.qasm	**3**	**0.909449**
grover_n2.qasm	0	0.532622
hs4_n4.qasm	0	0.992432
iswap_n2.qasm	0	0.283597
linearsolver_n3.qasm	0	0.712982
qaoa_n3.qasm	**2**	**0.710398**
qft_n4.qasm	**2**	**1.494051**
qrng_n4.qasm	0	0.124513
quantumwalks_n2.qasm	0	1.525190
teleportation_n3.qasm	0	0.263107
toffoli_n3.qasm	**3**	**0.768762**
variational_n4.qasm	0	1.419673
vqe_uccsd_n4.qasm	0	9.143029
wstate_n3.qasm	**3**	**1.111149**

Table 7.6. Results of the benchmarking test for the *TrivialMapping* plus *Hardware-Aware Routing* strategy of the fully-connected technologies. The quantum circuits under test are the small-sized circuits, and the target technology is NMR (with broken connectivity).

	TrivialMapping + HaRoutingSmart(S0.5E0.0T0.5)	
Circuit name	**Swap count**	**Execution time [s]**
adder_n4.qasm	0	0.944301
basis_change_n3.qasm	0	0.547967
basis_trotter_n4.qasm	0	52.276457
bell_n4.qasm	**1**	**0.961430**
cat_state_n4.qasm	0	0.243986
deutsch_n2.qasm	0	0.183303
dnn_n2.qasm	0	4.857846
fredkin_n3.qasm	**1**	**1.158726**
grover_n2.qasm	0	0.532622
hs4_n4.qasm	0	0.992432
iswap_n2.qasm	0	0.283597
linearsolver_n3.qasm	0	0.712982
qaoa_n3.qasm	**0**	**1.376494**
qft_n4.qasm	**3**	**1.395679**
qrng_n4.qasm	0	0.124513
quantumwalks_n2.qasm	0	1.525190
teleportation_n3.qasm	0	0.263107
toffoli_n3.qasm	**2**	**1.103118**
variational_n4.qasm	0	1.419673
vqe_uccsd_n4.qasm	0	9.143029
wstate_n3.qasm	**2**	**1.445767**

Table 7.7. Results of the benchmarking test for the *TrivialMapping* plus *Hardware-Aware Routing Smart* strategy of the fully-connected technologies. The quantum circuits under test are the small-sized circuits, and the target technology is NMR.

Chapter 8

Conclusions

The target of this thesis was the development of a layout synthesis library, completely independent of any quantum compilation framework, to perform the placement and routing steps in order to bring an abstract quantum circuit description into a physical one, executable on quantum computing hardware.

One point must be emphasised. The proposed library was not developed to replace nor competing with the state-of-the-art compilers in terms of optimisation strength. The key component of the proposed work is flexibility. The library provides classes and methods to rapidly and efficiently implement layout synthesis procedures, in order to assess their potential. It was implemented for simplifying the research activities in this fascinating field, and not for competing with other toolchains.
Heuristic solutions were explored and incorporated inside this work, with the confidence and hopefulness that scalability will be the main concern for NISQ devices in the near future.

The developed library is compatible with superconducting, quantum dots, NMR and trapped ions technologies, where each target device is defined by a backend configuration file in which also the gate features information can be included. For computing this calibration data, different strategies were adopted. For superconducting devices, this information is directly retrieved from the IBM Quantum project. For NMR and quantum dots backends, the error rates were computed by using the MATLAB QuanTO simulator of VLSI Lab of Politecnico di Torino. For trapped ions instead, the gate features computation was performed by using a model found in the associated literature.

The most powerful strategies developed for the proposed library are all based on the novel hardware-aware approach, exploiting the quantum gates features to make smarter decisions. These clever procedures were tested for all the current state-of-the-art quantum technologies, demonstrating that satisfactory optimisation results could be obtained. Indeed, the optimisation power that the proposed library strategies can reach is comparable to the consolidated Qiskit and t|ket⟩

compilers, validating even more the proposed approaches.

Moreover, during the production of this work a research effort was made, to evaluate if cleverer routing procedures should be tried even for fully-connected technologies. The results of this study are remarkable: when the gate features depend on the interacting nodes, even if the target device has no connectivity limitations, still swap gates addition could be beneficial for the final quantum circuits main figure of merits.
However, the obtained outcomes also underline the limitation of such heuristic solutions, which cannot always make the optimal decision in every step of the iterative exploration process.

Following the completion of the library's development, the work was integrated inside the VLSI quantum circuits compilation toolchain, completing the ambitious project started by M. Avitabile in 2021 of realising a full and independent quantum compiler. All the previous advice proposed for the placement and routing integration was followed, maintaining the original philosophy of allowing an easy expansion.
Although the library and the toolchain are complete, there is still room for improvement. First and foremost, to enhance the results in terms of compilation time, the available placement and routing procedures should be implemented using a compiled and performant language, such as C++, instead of an interpreted one. In this way, the library would transform into a shell around the core layout synthesis implementation. Maybe in this way, also computationally heavier solutions might be explored.
Secondly, following the advancements in quantum computing hardware, the existing quantum technologies gate characteristics modelling may be enhanced, while also adding additional future devised quantum device implementations. For example, the support for trapped ions technology could be expanded, by allowing the abstraction of more recent architectures such as the Quantum Charge Coupled Device (QCCD). In this way the "classical" routing approach for this technology, implemented through ion shuttling, can be integrated.
Furthermore, an automatic tool for the generation of the backend configuration file could be produced for all the supported technologies. Last but not the least, the proposed library could always be expanded, exploring future publications on the layout synthesis problem.

The contemporary NISQ devices appear to be highly promising for converting the theoretical quantum advantage into a practical one. Indeed, the main objective is proving that the "bet" of the quantum model of computation is actually worth it. To contribute to this incredible achievement, a quantum compilation toolchain is essential, in order to effectively execute a quantum circuit and make it produce meaningful results. The hope is that the produced work will constantly be expanded by future researchers to continuously improve (at least at a circuital level) the

execution of quantum algorithms on real device, with the hope that will better solve problems of interest for Society and Industries.

Bibliography

[1] Travis S. Humble, Himanshu Thapliyal, Edgard Munoz-Coreas, Fahd A. Mohiyaddin, and Ryan S. Bennink. Quantum Computing Circuits and Devices, April 2018. http://arxiv.org/abs/1804.10648.

[2] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Santa Fe, NM, USA, 1994. IEEE Comput. Soc. Press. https://doi.org/10.1109/SFCS.1994.365700.

[3] Lov K. Grover. A fast quantum mechanical algorithm for database search, November 1996. http://arxiv.org/abs/quant-ph/9605043.

[4] Noson S. Yanofsky and Mirco A. Mannucci. *Quantum computing for computer scientists*. New York : Cambridge University Press, Cambridge, 2008. OCLC: ocn212859032.

[5] Michael A. Nielsen and Isaac L. Chuang. *Quantum computation and quantum information*. Cambridge University Press, Cambridge ; New York, 10th anniversary ed edition, 2010.

[6] Bochen Tan and Jason Cong. Optimality Study of Existing Quantum Computing Layout Synthesis Tools. *IEEE Transactions on Computers*, 70(9):1363–1373, September 2021. https://doi.org/10.1109/TC.2020.3009140.

[7] Bochen Tan and Jason Cong. Optimal Layout Synthesis for Quantum Computing. 2020. https://doi.org/10.48550/ARXIV.2007.15671.

[8] David P. DiVincenzo. The Physical Implementation of Quantum Computation. *Fortschritte der Physik*, 48(9-11):771–783, September 2000. https://doi.org/10.1002/1521-3978(200009)48:9/11<771::aid-prop771>3.0.co;2-e.

[9] John Preskill. Quantum Computing in the NISQ era and beyond. 2018. https://doi.org/10.48550/ARXIV.1801.00862.

[10] Prakash Murali, Jonathan M. Baker, Ali Javadi Abhari, Frederic T. Chong, and Margaret Martonosi. Noise-Adaptive Compiler Mappings for Noisy

Intermediate-Scale Quantum Computers, January 2019. http://arxiv.org/abs/1901.11054.

[11] Alexander Cowtan, Silas Dilkes, Ross Duncan, Alexandre Krajenbrink, Will Simmons, and Seyon Sivarajah. On the Qubit Routing Problem. page 32 pages, 2019. https://doi.org/10.4230/LIPICS.TQC.2019.5.

[12] Alwin Zulehner and Robert Wille. *Introducing design automation for quantum computing*. Springer, Cham, 2020. https://public.ebookcentral.proquest.com/choice/publicfullrecord.aspx?p=6166904.

[13] 5-qubit backend: IBM Q team, "IBM Q Lima backend specification V1.0.36", 2022. Retrieved from https://quantum-computing.ibm.com/.

[14] IBM Quantum, 2022. https://quantum-computing.ibm.com/.

[15] Manfredi Avitabile. Proposal for a multi-technology, template-based quantum circuits compilation toolchain. Master's thesis, Politecnico di Torino, 2021. https://webthesis.biblio.polito.it/19223/1/tesi.pdf.

[16] Mario Simoni. Modelling Molecular Technologies for Nuclear Magnetic Resonance Quantum Computing. Master's thesis, Politecnico di Torino, April 2020. https://webthesis.biblio.polito.it/14446/1/tesi.pdf.

[17] Davide Costa. Definition of compact models for the simulation of spin qubits in semiconductor quantum dots. Master's thesis, Politecnico di Torino, April 2022. https://webthesis.biblio.polito.it/22822/1/tesi.pdf.

[18] Prakash Murali, Dripto M. Debroy, Kenneth R. Brown, and Margaret Martonosi. Architecting Noisy Intermediate-Scale Trapped Ion Quantum Computers. 2020. https://doi.org/10.48550/ARXIV.2004.04706.

[19] Colin D. Bruzewicz, John Chiaverini, Robert McConnell, and Jeremy M. Sage. Trapped-ion quantum computing: Progress and challenges. *Applied Physics Reviews*, 6(2):021314, June 2019. https://doi.org/10.1063/1.5088164.

[20] Klaus Mølmer and Anders Sørensen. Multiparticle Entanglement of Hot Trapped Ions. *Physical Review Letters*, 82(9):1835–1838, March 1999. https://doi.org/10.1103/PhysRevLett.82.1835.

[21] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Science and Technology*, 6(1):014003, January 2021. https://doi.org/10.1088/2058-9565/ab8e92.

[22] Transpiler (qiskit.transpiler) — Qiskit 0.36.1 documentation, 2022. https://qiskit.org/documentation/apidoc/transpiler.html.

[23] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintao Pereira. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 113–125, Vienna Austria, February 2018. ACM. https://doi.org/10.1145/3168822.

[24] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language, July 2017. http://arxiv.org/abs/1707.03429.

[25] Sergei Bravyi and Alexei Kitaev. Universal Quantum Computation with ideal Clifford gates and noisy ancillas. *Physical Review A*, 71(2):022316, February 2005. https://doi.org/10.1103/PhysRevA.71.022316.

[26] Abraham Héctor, AduOffei, Yunus Ismail, Akhalwaya, Aleksandrowicz Gadi, Alexander Thomas, Alexandrowics Gadi, Arbel Eli, Asfaw Abraham, et al. Qiskit: An Open-source Framework for Quantum Computing, 2021. 10.5281/zenodo.2573505.

[27] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, September 2003. https://doi.org/10.1145/937503.937505.

[28] Jakob Puchinger and Günther R. Raidl. Combining Metaheuristics and Exact Algorithms in Combinatorial Optimization: A Survey and Classification. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, José Mira, and José R. Álvarez, editors, *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*, volume 3562, pages 41–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. https://doi.org/10.1007/11499305_5.

[29] George Bernard Dantzig. *Linear programming and extensions.* Princeton landmarks in mathematics and physics. Princeton Univ. Press, Princeton, NJ, 11. printing, 1. paperback printing edition, 1998.

[30] John D. C. Little, Katta G. Murty, Dura W. Sweeney, and Caroline Karel. An Algorithm for the Traveling Salesman Problem. *Operations Research*, 11(6):972–989, December 1963. https://doi.org/10.1287/opre.11.6.972.

[31] Richard Bellman. *Dynamic programming.* Dover Publications, Mineola, N.Y, dover ed edition, 2003.

[32] Gushu Li, Yufei Ding, and Yuan 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*, pages 1001–1014, Providence RI USA, April 2019. ACM. https://doi.org/10.1145/3297858.3304023.

[33] Dmitri Maslov, Sean M. Falconer, and Michele Mosca. Quantum Circuit Placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(4):752–763, April 2008. https://doi.org/10.1109/TCAD.2008.917562.

[34] Alwin Zulehner, Alexandru Paler, and Robert Wille. Efficient mapping of quantum circuits to the IBM QX architectures. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1135–1138, Dresden, Germany, March 2018. IEEE. https://doi.org/10.23919/DATE.2018.8342181.

[35] Janusz Kusyk, Samah M. Saeed, and Muharrem Umit Uyar. Survey on Quantum Circuit Compilation for Noisy Intermediate-Scale Quantum Computers: Artificial Intelligence to Heuristics. *IEEE Transactions on Quantum Engineering*, 2:1–16, 2021. https://doi.org/10.1109/TQE.2021.3068355.

[36] TrivialLayout — Qiskit 0.36.1 documentation, 2022. https://tinyurl.com/2p8rpz7w.

[37] DenseLayout — Qiskit 0.36.1 documentation, 2022. https://tinyurl.com/msjvd6c3.

[38] NoiseAdaptiveLayout — Qiskit 0.37.0 documentation, 2022. https://tinyurl.com/3329bv6p.

[39] SabreLayout — Qiskit 0.36.1 documentation, 2022. https://tinyurl.com/558h9njd.

[40] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983. https://doi.org/10.1126/science.220.4598.671.

[41] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, January 1986. https://doi.org/10.1016/0305-0548(86)90048-1.

[42] Siyuan Niu, Adrien Suau, Gabriel Staffelbach, and Aida Todri-Sanial. A Hardware-Aware Heuristic for the Qubit Mapping Problem in the NISQ Era. *IEEE Transactions on Quantum Engineering*, 1:1–14, 2020. https://doi.org/10.1109/TQE.2020.3026544.

[43] Robert Wille, Stefan Hillmich, and Lukas Burgholzer. Efficient and Correct Compilation of Quantum Circuits. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, Seville, Spain, October 2020. IEEE. https://doi.org/10.1109/ISCAS45731.2020.9180791.

[44] Debjyoti Bhattacharjee and Anupam Chattopadhyay. Depth-Optimal Quantum Circuit Placement for Arbitrary Topologies. *arXiv:1703.08540 [quant-ph]*, March 2017. http://arxiv.org/abs/1703.08540.

[45] Kyle E. C. Booth, Minh Do, J. Christopher Beck, Eleanor Rieffel, Davide Venturelli, and Jeremy Frank. Comparing and Integrating Constraint Programming and Temporal Planning for Quantum Circuit Compilation. *arXiv:1803.06775 [quant-ph]*, March 2018. http://arxiv.org/abs/1803.06775.

[46] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. Mapping Quantum Circuits to IBM QX Architectures Using the Minimal Number of SWAP and H Operations. *arXiv:1907.02026 [quant-ph]*, July 2019. http://arxiv.org/abs/1907.02026.

[47] Robert Wille, Aaron Lye, and Rolf Drechsler. Optimal SWAP gate insertion for nearest neighbor quantum circuits. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 489–494, Singapore, January 2014. IEEE. https://doi.org/10.1109/ASPDAC.2014.6742939.

[48] BasicSwap — Qiskit 0.36.1 documentation, 2022. https://tinyurl.com/mv8y2tx9.

[49] StochasticSwap — Qiskit 0.36.1 documentation, 2022. https://tinyurl.com/yjn8xd93.

[50] LookaheadSwap — Qiskit 0.36.1 documentation, 2022. https://tinyurl.com/bdh2bsdm.

[51] Sven Jandura. Improving a Quantum Compiler, September 2018. https://medium.com/qiskit/improving-a-quantum-compiler-48410d7a7084.

[52] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, June 1962. https://doi.org/10.1145/367766.368168.

[53] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, December 1959. https://doi.org/10.1007/BF01386390.

[54] John D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. https://doi.org/10.1109/MCSE.2007.55.

[55] Siyuan NIU. HA algorithm, March 2022. original-date: 2020-06-28T18:58:42Z https://github.com/peachnuts/HA.

[56] Giovanni Amedeo Cirillo. *Engineering quantum computing technologies: from compact modelling to applications*. PhD thesis, Politecnico di Torino, doctorate program in Electrical, Electronics and Communications Engineering, 2022.

[57] Dmitri Maslov. Basic circuit compilation techniques for an ion-trap quantum machine. *New Journal of Physics*, 19(2):023035, February 2017. https://doi.org/10.1088/1367-2630/aa5e47.

[58] Colin J. Trout, Muyuan Li, Mauricio Gutierrez, Yukai Wu, Sheng-Tao Wang, Luming Duan, and Kenneth R. Brown. Simulating the performance of a distance-3 surface code in a linear ion trap. *New Journal of Physics*, 20(4):043038, April 2018. https://doi.org/10.1088/1367-2630/aab341.

[59] IonQ API Calibrations | Cirq, 2022. https://tinyurl.com/fbb3pjju.

[60] JKU IIC circuits repository. https://tinyurl.com/ywtf6xnv.

[61] Ang Li. QASMBench circuits repository, June 2022. https://github.com/pnnl/qasmbench.

[62] FakeToronto — Qiskit 0.37.0 documentation. https://tinyurl.com/yckyxxdf.

[63] QasmSimulator — Qiskit 0.37.0 documentation, 2022. https://tinyurl.com/3wmk4z9n.

[64] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, March 1951. https://doi.org/10.1214/aoms/1177729694.

[65] qiskit.quantum_info.hellinger_fidelity — Qiskit 0.36.2 documentation. https://tinyurl.com/mr27xdmc.

[66] Y. Kharkov, A. Ivanova, E. Mikhantiev, and A. Kotelnikov. Arline Benchmarks: Automated Benchmarking Platform for Quantum Compilers, February 2022. http://arxiv.org/abs/2202.14025.

[67] SabreSwap — Qiskit 0.37.0 documentation, 2022. https://tinyurl.com/4tkb5c83.