OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

Master Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science (M.Sc.)

Analysis and Evaluation of Quantum Compilers

Student Name: Vincent Gierisch
Student Number: 3347546

Primary Supervising Professor: Prof. Dr. Wolfgang Mauerer
Secondary Supervising Professor: Prof. Dr. Kai Selgrad

Submission Date: May 15, 2024

Master Computer Science

OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

ERKLÄRUNG
ZUR MASTERARBEIT VON

Name:

Vorname:

Studiengang:

1. Mir ist bekannt, dass dieses Exemplar der Masterarbeit als Prüfungsleistung in das Eigentum der Ostbayerischen Technischen Hochschule Regensburg übergeht.

2. Ich erkläre hiermit, dass ich diese Masterarbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Regensburg, den

.....
Unterschrift

Diese Erklärung ist mit der Masterarbeit (eingeheftet) abzugeben.

Stand: 21.09.2018/Abt. III

Abstract

Quantum computers have the potential to solve specific tasks faster than the classical computational systems. There is currently a lot of effort to develop algorithms that take advantage of this type of quantum speedup. Today's quantum computers, often called noisy intermediate-scale quantum (NISQ) hardware, are prone to noise and have too few qubits for effective error correction. They also have restrictions on which operations can be applied to which qubits. Quantum compilers are necessary to abstract the constraints of NISQ devices.

This thesis presents an in-depth analysis and evaluation of quantum compilers. The study focuses on three commonly used quantum compilers: Qiskit, TKET, and BQSKit, each known for their unique approaches to compile and optimize quantum circuits. It is investigated how the quantum compilers behave depending on the chosen level of optimization. In order to address this question, an evaluation framework is developed which allows the evaluation of different quantum compilers by measuring circuit properties of the circuits compiled for defined backends. The evaluation is conducted with varying optimization levels of the compilers. It can be shown that the highest level of optimization does not necessary produce the best circuits in terms of circuit depth and overall gate count.

Acronyms

NISQ Noisy Intermediate-Scale Quantum

QAOA Quantum Approximate Optimization Algorithm

VQA Variational Quantum Algorithm

SABRE SWAP-based BidiREctional heuristic search algorithm

LEAP Larger Exploration by Approximate Prefixes

BQSKit Berkeley Quantum Synthesis Toolkit

PAS Permutation-aware synthesis

NISQ Noisy Intermediate-Scale Quantum

Contents

1 Introduction

The field of quantum computing originated from Richard Feynman's proposal in 1981 to harness the principle of quantum physics in order to construct computers capable of accelerating calculations to specific problems [1].
There are already quantum algorithms that in theory can solve certain problems more efficiently than classical computational systems. A representative of these is Shor's algorithm, which is designed to solve the integer factorization problem and the discrete logarithm problem [2]. Both problems can be solved by the algorithm in polynomial time when running on a quantum computer [2].
Today's quantum computers, are prone to noise and therefore introduce errors in the calculation. In addition to the erroneous calculations, they have also a very limited amount of available qubits. For both reasons, this devices are often referred to as Noisy Intermediate-Scale Quantum (NISQ). There are various physical implementations of NISQ devices and many providers of these systems, including IBM, Rigetti, and IonQ.

Since there is such a variety, there are also many different requirements for software that can run on it. Some of these constraints are a limited set of operations that a quantum computer is capable of executing as well as a constraint that some operations can only be applied to specific qubits.
To abstract the limitations of the quantum hardware and to provide the developer with the opportunity to concentrate more on the development of quantum algorithms, it is possible to use quantum compilers. A quantum compiler seeks to overcome the restraints described above by the modifying the algorithm to enable execution on specified hardware.

There are many different quantum compilers, all of which can handle many different types of quantum hardware. Some of these compilers are IBM's Qiskit, Google's Cirq, Rigetti's Quilc, Microsoft's Quantum Dev Kit, Quantinuum's TKET, and Lawrence Berkeley National Laboratory's BQSKit [3–8].
It is likely that the majority of developers are not fully aware of the inner workings of these quantum compilers. This is because their primary focus is on the development of quantum algorithms. Also the compiler providers tend to less comprehensive documentation.

This thesis provides an overview of the functionality of quantum compilers and performs an in-depth analysis of three selected compilers, namely Qiskit, TKET, and BQSKit. These compilers are subsequently evaluated individually and compared with each other, taking into account their parameters.

The remainder of this thesis is structured as follows: Section 2 gives a brief introduction to gate-based quantum computing. Section 3 provides an overview of the current physical realizations and limitations of quantum hardware. In Section 4 the general functionality of quantum compilers is described. Section 5 offers an in-depth analysis of three commonly used quantum compilers. In Section 6 describes the design and implementation of the evaluation framework. Section 7 presents the evaluation of these quantum compilers, accompanied by a discussion of the results. Finally, in Section 8, the results are summarized and potential future research directions are identified.

2 Background on Quantum Computing

This chapter aims to provide an overview of the basics of quantum computing. The Section 2.1 introduces the qubit and the concept of superposition. In Section 2.3 these concepts are expanded to multi-qubit systems. Subsequently, in Section 2.4, the circuit model together with quantum gates are introduced. The chapter concludes with the presentation of a family of quantum algorithms in Section 2.6 that will be used later in the evaluation.

2.1 Information on Qubits

Quantum computers use quantum effects like superposition and entanglement to speed up calculations. Classical computers use classical bits to store information. In contrast, in a quantum computer, information is stored in qubits. Unlike the classical bit, the qubit can be in a linear combination of states, often referred to as *superposition* [9].
The *Dirac notation*, sometimes referred to as *bra-ket notation*, is used to represent states in quantum mechanics and is denoted by $|\cdot\rangle$. In this representation, the *ket* $|\cdot\rangle$ corresponds to a column vector, while the *bra* $\langle\cdot|$ corresponds to the complex conjugate transpose of the ket. Equation (1) depicts the dirac notation.

$$\langle u| = (u_1^*, u_2^*, ..., u_n^*), |v\rangle = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \tag{1}$$

The superposition of a qubit can be written as a linear combination of $|0\rangle$ and $|1\rangle$ using the dirac notation,

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2}$$

,with $\alpha, \beta \in \mathbb{C}$. In Equation (2), α and β are called *amplitudes* and $|0\rangle$, $|1\rangle$ are known as *computational basis states*. They form an orthonormal basis for the vector space in which the representation of the qubit lies. When measuring the state of a qubit, the superposition collapses into one of the computational basis vectors. The obtained outcome is either $|0\rangle$ with a probability of $|\alpha|^2$ or $|1\rangle$ with a probability of $|\beta|^2$. To make sure that the probabilities add up to one, the amplitudes are normalized to satisfy the condition $|\alpha|^2 + |\beta|^2 = 1$.

2.2 Bloch Sphere

The state of a qubit can also be visualized by the *Bloch sphere*, which is a geometrical representation. Equation (2) can be reformulated as

$$|\psi\rangle = e^{i\gamma}(\cos\frac{\theta}{2}\,|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}\,|1\rangle) \tag{3}$$

,with $\theta, \varphi, \gamma \in \mathbb{R}$ [9].
The factor $e^{i\gamma}$ in front of the equation can be neglected as it is a *global phase* which has, in general, no observable consequences[9]. The trigonometric function's symmetry determines a value range $\theta \in [0, \pi]$ and $\varphi \in [0, 2\pi]$. A point on a three-dimensional sphere

is determined by the values of θ and φ, as illustrated in Figure 1. This sphere is commonly referred to as the Bloch sphere.

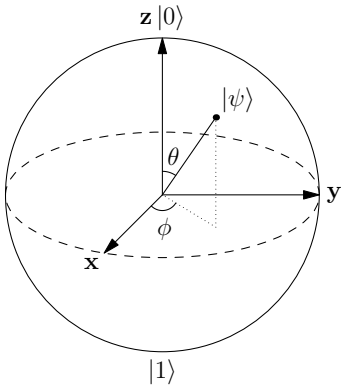


Figure 1: Illustration of the state of a single qubit in the Bloch sphere.

While the Bloch sphere can visualize the state of a single qubit, its not sufficient to graphical display the state of a multi-qubit system [9].

2.3 Multi-Qubit Systems

In a multi-qubit system, the individual qubits are grouped together in a *qubit register*. A two-qubit system, for example, has four computational basis states, namely $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$. The linear combination of these states, outlined in [9], is described as follows:

$$|\psi\rangle = \alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle + \alpha_{10}\,|10\rangle + \alpha_{11}\,|11\rangle \tag{4}$$

In Equation (4), $a_x \in \mathbb{C}$ with $x \in \{0,1\}^2$ are the amplitudes of each computational basis state. As in Equation (2) the amplitudes must satisfy the condition $\sum_{\{0,1\}^2}^{x} |a_x|^2 = 1$.
The state of a multi-qubit system with n qubits can also be described in a vector notation with a vector space of 2^n dimensions. Two vector spaces can be described by the *tensor product*. The elements of the two vector spaces V, W, are linear combinations of $|v\rangle \otimes |w\rangle$, with $|v\rangle \in V$, $|w\rangle \in W$. [9, 10]
Multiple states $|\psi_1\rangle, |\psi_2\rangle, ..., |\psi_n\rangle$ can be represented by one vector space through the use of the tensor product,

$$|\psi\rangle = |\psi_1, \psi_2, ..., \psi_n\rangle = \bigotimes_{i=1}^{n} |\psi_i\rangle\,. \tag{5}$$

The state of a quantum register consisting of n qubits is therefore described by a vector of a 2^n-dimensional vector space. This exponential growth in complexity is one of the indicators for a potential quantum advantage [9].

Another phenomenon that can occur in a multi-qubit system is entanglement. Entanglement occurs when two or more qubits become correlated in such a way that the state of one qubit cannot be described independently of the state of the other, regardless of the physical separation between them [9].
An example for such entangled quantum states are the four Bell states [9]. The first Bell state, namely $|\Phi^+\rangle$, is defined in Equation (6) [9].

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{6}$$

If a multi-qubit quantum system is prepared in state $|\Phi^+\rangle$ and one qubit is measured, both qubits enter the post-measuring state $|00\rangle$ or $|11\rangle$ with a probability of 0.5 respectively. Entanglement is a key ingredient in many quantum algorithms and another indicator for the potentials of quantum computing.

2.4 Circuit Models

The following section discusses the circuit model, which is a visual representation of quantum algorithms. It is also the main component, a quantum compiler acts on.

Deutsch introduced the quantum circuit model in 1989 [11]. The quantum circuit model is a description of a collection of qubits with gates that act on them in a fixed sequence. Similarly, a classical algorithm can be described as a sequence of operations that act on a given input. Given the input, these operations produce an output, which is the result of the algorithm. The circuit model shares similarities with this aspect. A quantum register, which is prepared in a specific state, serves as the input, while so called quantum gates act on the qubits. An output can be produced by measuring the qubits.
There are other rivals to the circuit model, like cluster state quantum computing and adiabatic quantum computing [10].

In the Deutsch circuit model, all qubits are organized as wires that run from left to right. Gates can act on qubits by placing them on the wires associated with the desired qubits.

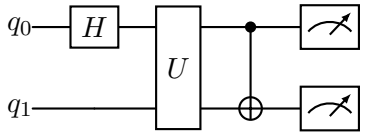


Figure 2: Example of a Quantum Circuit.

Figure 2 depicts an example of a quantum circuit, where a one-qubit gate H acts on qubit q_0, followed by the application of a two-qubit gate U to both qubits q_0 and q_1. Then, a $CNOT$ gate, another two-qubit gate, flips the target qubit q_1 based on the complex amplitude of the control qubit q_0, before the measurement of both qubits.

2.4.1 Quantum Gates

As previously stated, a gate acts on a quantum state and manipulates it. The gates of a quantum circuit can be represented by a unitary matrix. For example, the Pauli-X gate, which is the quantum equivalent of a NOT gate, rotates the quantum state around the x-axis. Its matrix representation is shown in Equation (7),

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}. \tag{7}$$

To apply the Pauli-X gate to the quantum state $|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$, the matrix is multiplied by the state vector described in Equation (8),

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}. \tag{8}$$

In the new state $|\psi'\rangle = \alpha\,|1\rangle + \beta\,|0\rangle$, the complex amplitudes of $|0\rangle$ and $|1\rangle$ have been swapped. In the analogy of the Bloch sphere, the state has been rotated around the x-axis by 180°.

One restriction on the matrix representing a quantum gate is that it must be *unitary*. For a matrix to be unitary, the condition

$$U^\dagger U = \mathbb{1}, \tag{9}$$

must be fulfilled. $U^\dagger$ denotes the transposed and complex conjugated of U. This constraint ensures that the condition $|\alpha|^2 + |\beta|^2 = 1$ is satisfied. [9]

Applying a gate to a quantum register works analogously. In Figure 3, a CNOT gate is applied to q_0 as the control qubit, and q_1 as the target qubit. A CNOT gate flips the target, based on the complex amplitude of the control qubit. The classical equivalent is an *XOR* operation with the notation $|A, B\rangle \rightarrow |A, A \oplus B\rangle$.

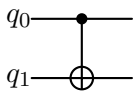


Figure 3: CNOT gate acting on q_0 and q_1.

Equation (10) depicts the matrix notation of the CNOT gate. Assuming the qubit register is initialized with $|\psi\rangle = |01\rangle$, the matrix vector representation is shown in Equation (11).

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{10}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \tag{11}$$

It can be seen that in the successor state $|\psi'\rangle = |11\rangle$, q_1 has been flipped, because q_0 is in the state $|1\rangle$.

The size of the matrix is determined by the number of qubits that the gate is acting on. A gate that operates on n qubits is represented by a $2^n \times 2^n$ matrix [11].

Another commonly used set of quantum gates are the so called *parametrized gates*. These kind of gates depend on one or more parameters [10]. A lot of quantum algorithms are using parametrized gates, especially *rotation gates*. The rotation gates represent a rotation along the x, y or z-axes of the Bloch sphere. Therefore, there are three rotation gates,

$$R_X(\theta) = \begin{bmatrix} \cos(\frac{\theta}{2}) & -i\sin(\frac{\theta}{2}) \\ -i\sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{bmatrix}, R_Y(\theta) = \begin{bmatrix} \cos(\frac{\theta}{2}) & -\sin(\frac{\theta}{2}) \\ \sin(\frac{\theta}{2}) & \cos(\frac{\theta}{2}) \end{bmatrix}, R_Z(\theta) = \begin{bmatrix} e^{-\frac{i\theta}{2}} & 0 \\ 0 & e^{\frac{i\theta}{2}} \end{bmatrix}, \tag{12}$$

where θ is the rotation angle [10]. With these three gates, every state of a single qubit can be produced.

The functionality of the three rotation gates can be reduced to one gate, namely the $U3$ gate. The matrix representation of the $U3$ gate is given by

$$U3(\theta, \phi, \lambda) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -e^{i\lambda}\sin(\frac{\theta}{2}) \\ e^{i\phi}\sin(\frac{\theta}{2}) & e^{i(\phi+\lambda)}\cos(\frac{\theta}{2}) \end{pmatrix}, \tag{13}$$

with θ, ϕ and λ as the rotation angles along the axes of the Bloch sphere [9]. Therefore the U3 gate implements an arbitrary single-qubit rotation.

2.4.2 Quantum Circuits

After examining the mathematical representation of quantum gates and their application to quantum states, the focus now shifts to the representation of quantum circuits.

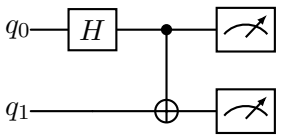


Figure 4: Quantum circuit producing a Bell state.

Figure 4 depicts a quantum circuit producing a Bell state (see Section 2.3). A Hadamard gate is applied to q_0 followed by a CNOT gate with q_0 as the control qubit and q_1 as the target qubit.
The Hadamard gate turns a state of $|0\rangle$ or $|1\rangle$ into a superposition of $|0\rangle$ and $|1\rangle$. More precisely, it transforms a state of $|0\rangle$ to the superposition $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. These two successor states are called $|+\rangle$ and $|-\rangle$ respectively. In the Bloch sphere analogy, the Hadamard gate is a combination of two rotation. First, it rotates the state vector about the y-axis by 90°, followed by a rotation about the x-axis by 180° [9].
The mathematical representation of a circuit can be described using matrix multiplication. The matrix for a calculation with multiple gates in parallel, results from the tensor product of the individual gates [12]. Equation (14) shows the calculation for the circuit, which Figure 4 depicts.

$$|\psi'\rangle = (CNOT)(H \otimes I_2)\,|\psi\rangle \tag{14}$$

The identity matrix is used for wires in the circuit that have no gate on this position, ensuring that all matrices have the same dimensions. It does not change the state of qubits in a noise free representation.

The matrix representation of the circuit shown in Figure 4 is therefore $(CNOT)(H \otimes I_2)$. Note that the size of the matrix increases exponentially with the number of qubits used

in the circuit.

An important characteristic factor of a circuit is the *circuit depth*. The circuit depth is defined as the number of gates that form the longest path from a qubit initialization to a measurement.

2.5 Quantum Programming Languages

Quantum programming languages usually consists of the definition of a quantum circuit, therefore, they represent the operations being applied to each of the qubits in the quantum computer [13]. There are also various graphical tools for defining a quantum circuit, however they become impracticable once the circuit reaches a certain size [13]. Quantum circuits can be described for example in Python using frameworks such as Qiskit, PyQuil or Cirq. These frameworks typically come with packages for describing quantum circuits as well as compiling and simulating them [3, 5].

Almost all frameworks are able to convert their representation into OpenQASM. OpenQASM is a programming language for describing quantum circuits and is becoming a de-facto standard for hardware-level quantum programming [14].

2.6 Quantum Algorithms

In this section a quantum algorithm and a specialization of it, namely the Variational Quantum Algorithm (VQA) and the Quantum Approximate Optimization Algorithm (QAOA) are described. These algorithms are used in this thesis for a more accurate comparison of various compiler settings.
The VQA and especially the QAOA are promising quantum algorithm. The QAOA aims to solve combinatorial optimization problems, which are typically NP-hard and therefore intractable to solve in the classic way.

2.6.1 Variational Quantum Algorithm

As previously stated, the QAOA belongs to the category of VQAs.
VQAs are quantum algorithms which use a hybrid approach of both classical computers and quantum computers to solve an optimization problem [15].
The algorithm uses a circuit, that consists of a sequence of parametrized gates (see Section 2.4.1). A cost function is established to quantify the objective of the optimization problem. It maps the parameters of the gates in the circuit to a real number and encodes the solution to the problem. The parameters are optimized in a hybrid quantum-classical loop.
Figure 5 depicts the hybrid quantum-classical loop. The quantum part of the loop estimates the cost function by probing the parametrized circuit with parameters from the previous optimization iteration, while a classical optimizer trains the circuit parameters based on the cost function.

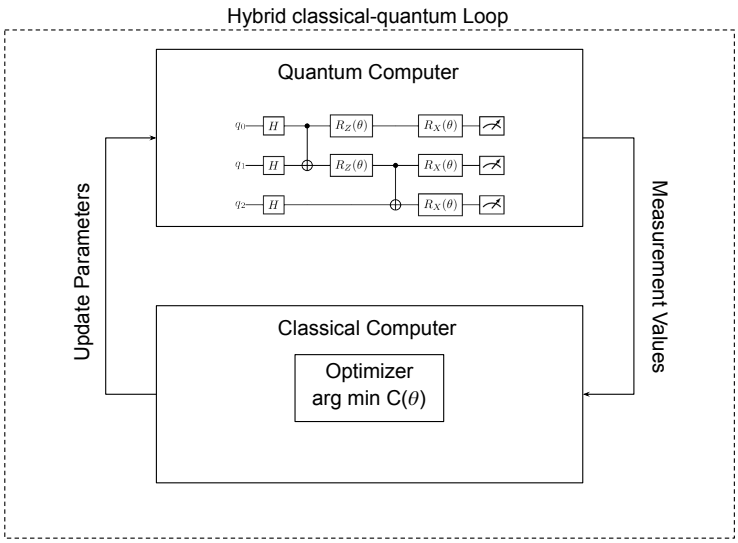


Figure 5: A VQA Hybrid classical-quantum Loop. Input to a VQA is the circuit with a set of parameters θ, as well as the cost function C(θ). The quantum computer executes the circuit while the classical computer uses the measurement values to formulate the cost function. A classical optimizer minimizes the cost function and updates the quantum circuit with the resulting parameters.
Source: Adapted from [15]

The output of a VQA can have multiple forms. A common way is to sample the quantum circuit once again with the resulting parameters of the hybrid quantum-classical loop. This would give a probability distribution of the possible algorithm result.

2.6.2 Quantum Approximate Optimization Algorithm

The QAOA is a promising approach for approximating solutions to combinatorial optimization problems, which was introduced by Farhi et al. in 2014 [16]. It has gained popularity in recent years as it is a candidate for working on noisy hardware [17].

The QAOA tries to find a solution by approximating the ground state of a cost Hamiltonian H_C by preparing a system with the ground state of a mixer Hamiltonian H_M. A Hamiltonian is an operator that represents the total energy of a quantum system. Here the ground state of the cost Hamiltonian H_C encodes the solution to the combinatorial optimization Problem. Typically, the mixer Hamiltonian H_M is a very simple Hamiltonian with the ground state already known. The system is then prepared with the ground state of H_M. According to the adiabatic theorem, when the system transits slowly enough from H_M to H_C it will end up in the more complex ground state of H_C. [16]

The mixer Hamiltonian H_M is often chosen to be $-\sum_{j=1}^{n} X^{(j)}$. Hence, the circuit is initialized with the mixers ground state $|s\rangle = |+\rangle^{\otimes n} = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n}$, where n is the number of qubits involved. In contrast, the cost Hamiltonian H_C depends on the specific problem. However, there are already predefined cost Hamiltonians in the form of Ising Hamiltonians for a lot of NP-complete and NP-hard problems [18].

The QAOA is constructed using an alternating approach of cost and mixer layers, labeled as $U_C(\gamma_k)$ and $U_M(\beta_k)$, where k is the k-th layer. The more layers a QAOA has, the more

accurately the result is approximated. The final state output of a QAOA circuit can be written in the form

$$|\psi(\gamma,\beta)\rangle = e^{-i\beta_k H_M} e^{-i\gamma_k H_C} ... e^{-i\beta_1 H_M} e^{-i\gamma_1 H_C} |s\rangle . \tag{15}$$

[16, 19] Using Equation (15), a circuit, depicted in Figure 6 can be created.

The set of parameters $\vec{\gamma}$, $\vec{\beta}$, are determined by minimizing a predefined cost function $C(x)$, in a manner of a VQA. The final solution of the QAOA is resolved by measuring the circuit a final time with the calculated set of parameters.

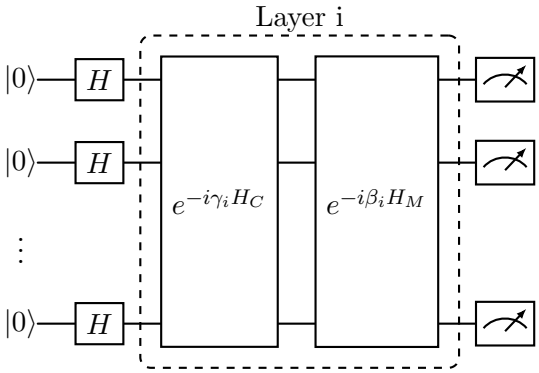


Figure 6: General QAOA circuit representation with mixer and cost Hamiltonian.

2.6.3 Max-cut Problem Overview

In this section the Max-cut problem is introduced. It will be used in Section 7.4 as a optimization problem for a QAOA to solve.

A commonly used combinatorial optimization problem for a QAOA to solve is the *Max-cut* problem. The problem is known to be NP-complete [20]. The objective of the Max-cut problem is to partition the nodes of a graph into two sets such that the number of edges between these two sets is as large as possible. Figure 7 depicts a graph describing the solution to the Max-cut problem.

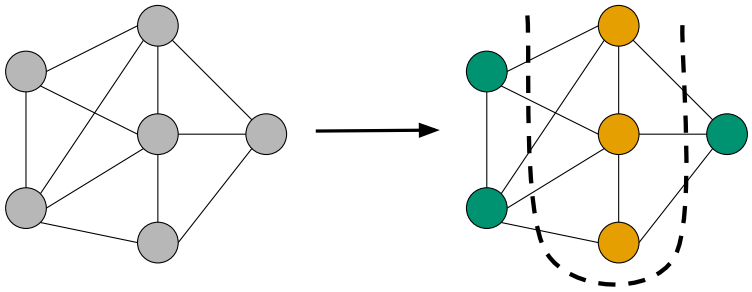


Figure 7: Graph to which the Max-cut Problem is applied. Left: A graph with six nodes and eleven edges. Each edge has the same weight. Right: Two sets of vertices. These are colored green and yellow, respectively. The edges between those two partitions build the cut set. The cut in this example has a size of eight.
Source: Adapted from [19]

Given a non-weighted graph $G = (V, E)$ where $V = (0, 1, \ldots, n)$ is a set of vertices and $E \subseteq (V \times V)$ a set of edges, the problem can be described by finding a subset $S \subseteq V$ such

that $\sum_{\substack{(u,v)\in E \\ u\in S, v\in N-S}} 1$ reaches a maximum among all possible subsets [20].

This can also be described as a quadratic binary program, with a variable $x_i \in \{-1, +1\}$ associated with every vertex $v_i \in V$. For an arbitrary subset S, $x_i = 1$ if $v_i \in S$ and -1 otherwise. The resulting equation is given by

$$max : \frac{1}{2} \sum_{(i,j)\in E} (1 - x_i x_j), \tag{16}$$

with $x_i \in \{-1, +1\}, \forall v_i \in V, \forall (i, j) \in E$. [21]

3 Quantum Hardware

This chapter presents an overview of the hardware approaches used to implement quantum computers, along with their limitations and errors. In Section 3.1 and Section 3.2 today's physical realizations of quantum hardware are presented. The Section 3.3 discusses the constraints and limitations these realizations have.

3.1 Noisy Intermediate-Scale Quantum Systems

In 2018 Preskill introduced the term Noisy Intermediate-Scale Quantum (NISQ). NISQ systems are the current generation of quantum computers. They have a limited number of qubits and high error rates. [22]
Noise is a major challenge for today's quantum computers. This noise is caused by the imperfection of qubits, which result in unstable states decaying over short periods. Also the gates used to manipulate the state of qubits introduce errors in the system. A detailed description of the faults in a gate or in a circuit can be found in chapter 3.3.3.
According to Preskill [22], "intermediate scale" refers to a number of qubits between 50 and a few hundred. The behavior of this amount of qubits cannot be simulated by classical computation systems. However, the number of qubits is too small for a full error correction [22].

3.2 Physical Realizations

Currently there are several promising hardware approaches for gate-based quantum computers. The following is a list of a selection of different physical realizations of a quantum computer, each with a brief description and their respective advantages and drawbacks.

Superconducting Transmon Qubits: Superconducting transmon qubits utilize macroscopic quantum phenomena. This quantum behavior is due to Cooper electron pairs that form in superconductors at extremely low temperatures.
This approach can be scaled effectively and the gate execution time is very fast. However, a qubit is only coupled to other qubits in its direct neighborhood. Also for the quantum phenomena to occur, extremely low temperatures are needed. [23]
Noise may be caused by imperfections in the building material as well as from external sources like heat [24].

Trapped Ion Qubits: In trapped ion qubits the quantum information is encoded in the electronic energy levels of ions which are suspended in a vacuum.
Great advantages are, that the overall system can operate at room temperature and that two-qubit operations are possible between any two qubits. The two main drawbacks of quantum computers using trapped ion qubits are that gate execution times are large compared to systems using transmon qubits, as well as its relatively poor scalability. [23]
Noise can occur by external electromagnetic fields disturbing the system [24].

Photonic Qubits: In a photonic quantum computer, a qubit is represented either as the polarization or location of a single photon.
The generation of a single qubit as well as the application of a single-qubit operation is relatively easy. However, multi-qubit operations are very inefficient and produce

a large resource overhead. Also the gate-set is very limited.
Noise in photonic quantum computers can occur due to photon loss, imperfect optical components or external environmental disturbances. [25]

Trapped ion quantum computers as well as quantum computers that are realized using a superconducting approach are the most common ones yet [23]. The noise as well as the other disadvantages associated with the physical implementations will persist in the NISQ era.

3.3 Quantum Hardware Constraints

The following chapter describes the constraints of a physical realization of a quantum computer in a more abstract way. A quantum compiler seeks to overcome these constraints.

Here a collection of hardware constrains is introduced, namely the limited connectivity of qubits, the support for only native gates and inaccuracies in the circuit introduced by noise. There are other sources of hardware constraints, including timing constraints and other vendor-specific constraints like frequency dependent limitations.

3.3.1 Limited Qubit Connectivity

One of the most crucial factors in quantum computing that affects the architecture and operational capabilities of quantum processors is the qubit connectivity.
A developer usually assumes an all-to-all connectivity. This would mean that every qubit can participate with any other qubit in a multi-qubit operation. Although an all-to-all connectivity layout would be preferable, this type of connectivity would introduce errors, especially in quantum computers using superconducting qubits, like chance of frequency collision, cross-talk, and spectator error [26]. Therefore, a compromise is used by limiting the possible interactions between qubit pairs.
The possible interactions between qubits are commonly specified by the vendor with a *connectivity graph* or *coupling map* [26, 27]. Qubits in the connectivity graph are often referred to as *physical qubits* [27]. Because they are real implementations in the physical realizations of a quantum computer.
Figure 8 depicts a connectivity graph of the IBM Quantum Falcon processor. This family of quantum processors was introduced in 2020 and consists of 27 superconducting qubits [26].

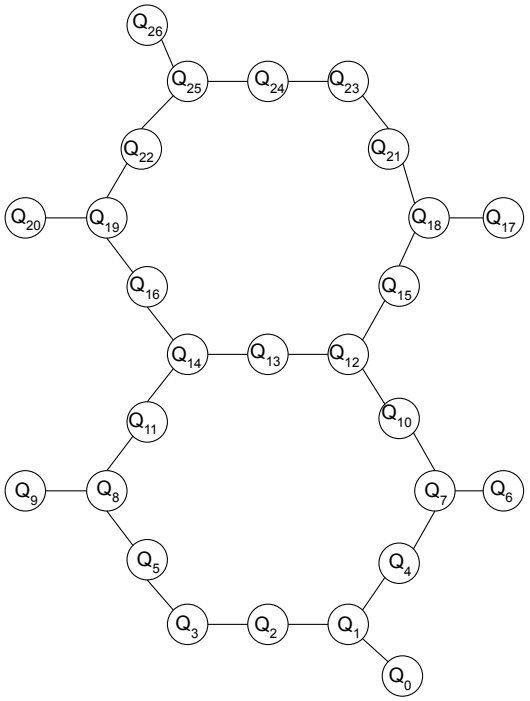


Figure 8: Connectivity graph of the IBM Quantum Falcon processor.

The way to interpret the connectivity graph is that all nodes represent a specific physical qubit. If two physical qubits share an edge in the connectivity graph, they can take part together in a multi-qubit operation.
With respect to the connectivity graph of the IBM Quantum Falcon processor (depicted in Figure 8), qubit Q_6 can just be part in a multi-qubit interaction with qubit Q_7. Therefore, qubit Q_6 can just perform a one-qubit or a two-qubit operation. Note that this connectivity graph has no directed edges, which means that the assignment of the target and control qubit in a multi-qubit operation is irrelevant.

3.3.2 Gate Sets

Every quantum programming language provides a large-scale collection of quantum gates to use in an arbitrary algorithm [28]. However, physical realizations of a quantum computer can just offer a limited set of gates. These gates are often referred to as *native gates* or *gate set*.

The native set of the IBM Quantum Falcon processor, for example, consists of the Controlled-X gate, the $\sqrt{X}$ gate and a parametrized Phase gate [26].
Quantum computers using trapped ion qubits also have a finite gate set. The gate set of IonQ Forte, a trapped ion quantum processor from IonQ, which was developed in 2022, contains four gates. These gates are, two parametrized single-qubit gates, namely the GPi gate as well as the Virtual Z gate, and two two-qubit gates, the Mølmer-Sørenson gate and the ZZ gate [29].

3.3.3 Errors in Quantum Systems

In all physical realizations of quantum computers, it cannot be assumed that the quantum system is closed. That is because the state of the quantum system is interfered by noise.

Specifically, the qubits and quantum gates are erroneous.
The states of the qubits are not stable, which means that it decays over a short period of time. This phenomenon is referred to as *decoherence* [9].
Similarly, the gates do not manipulate the state of the qubits the way they intended to. This results in a deviation from the expected result. This particular behavior is known as gate *fidelity* [9].

The decoherence as well as the gate fidelity depends on the specific platform. Indicators for decoherence and gate fidelity are typically published by the vendors themselves.
These indicators are analyzed in the remainder of this chapter.

Vendor	Platform	T_1	T_2	F_1	F_2
IBM	IMBQ-Sherbrooke	$271.55\mu s$	$176.81\mu s$	99.976%	99.249%
IonQ	Forte	$10s - 100s$	$1s$	99.98%	99.6%
Rigetti	ANKAA-2	$12.7\mu s$	$12.8\mu s$	99.614%	90.588%
Rigetti	ANKAA-9Q-1	$17.6\mu s$	$4.3\mu s$	99.3%	98.0%

Table 1: Indicators for decoherence and gate fidelity [30–32].

Table 1 contains the indicators for decoherence and gate fidelity for four selected quantum systems.
The *coherence* times T_1 and T_2 indicate the vulnerability of a qubit to noise.
T_1 is the average time in which a qubit can remain in its intended state [9]. That means that the qubit gradually transitions from the state $|1\rangle$ to $|0\rangle$, indicating a bit flip. This can be due to thermal relaxation because a qubit will slowly loose energy to its environment over time [9].
T_2 defines the average time it takes for the stability of the relative phase between states to undergo a phase flip [9]. This means, T_2 is the average duration before a superposition state, for example $|+\rangle$, transitions into an equal probability mixture of $|+\rangle$ and $|-\rangle$.

The gate *fidelities* F_1 and F_2 indicate the fidelity of a quantum operation, as quantum gates can never be implemented perfectly. The fidelity describes how "close" two quantum states are. These two states are the preferred output state and the real output state after a quantum gate operation [9]. F_1 characterizes the gate fidelity of one-qubit gates, whereas F_2 characterizes the gate fidelity for two-qubit gates.

The main differences in the vendor data in Table 1 is by far the coherence times T_1 and T_2. This is based on the physical realization on which the quantum computer is based on. The vendors IBM and Rigetti provide quantum computers using superconducting qubits, while the platform from IonQ is using trapped ions. The coherence times of IonQ's Forte platform is far longer than the coherence times of the other platforms. This is due to the trapped ions technology which has longer coherence times but, as stated in Section 3.2, also longer gate execution times [33].
Unfortunately, IBM does not provide data for the gate fidelities of IBMQ-Sherbrooke. However, they supply the mean average error for their ECR gate and their SX gate. As the ECR gate is their only two-qubit gate, this indicator can be used for F_2 [30]. The SX

gate is, however, not the only one-qubit gate in their gate set, so the mean SX-error can only be used for a rough estimation for F_1.
This is also the case for the quantum computers provided by Rigetti. Rigetti provides only F_2 for their CZ gate [32]. The fidelity F_2 for the XY gate, which is the only other two-qubit gate in their gate set, is unknown.

The total error of a quantum circuit depends on the number of and type of gates in it. With more gates acting on a qubit the probability for decoherence increases because each gate has its own operation duration. Also the fidelity of the whole quantum circuit shrinks as more gates are involved.
In summary, both the coherence time as well as the gate fidelity demonstrate the importance of keeping circuits as small as possible, as the probability for errors increases with the size of the circuit.

4 Functionality of Quantum Compilers

In Section 3.3 the constraints for physical realizations for quantum computers are described. A quantum compiler must take these constraints into account while compiling a quantum circuit to a specific platform. This chapter describes a general procedure for the compilation process.
In Section 4.1 a general description of the gate translation procedure is provided. Section 4.2 and Section 4.3 discuss the mapping and routing approaches of a compiler. Finally, Section 4.4 provides an overview of optimization techniques a quantum compiler possesses.

A quantum compiler receives the quantum circuit provided by the developer and a description of the backend on which the circuit will be executed. The basic description for a backend is made up of a coupling map and a gate set.
The task of the compiler is to transform the input circuit into a new quantum circuit which just contains gates from the gate set of the backend and has its multi-qubit operations just on qubits that share an edge on the coupling map. Some compilers additionally run optimizations to decrease the overall circuit depth, which reduces the impact of noise. In order to compile a circuit to the designated backend, the compiler chains multiple algorithms into a *compilation pipeline*.
There are four distinct tasks into which the compilation process can be subdivided: *Gate Translation*, *Initial Mapping*, *Routing*, and *Optimization*.

The following sections provide a detailed description of these tasks and discuss specific solution approaches. These approaches are used in the compilers that this thesis will focus on later.

4.1 Gate Translation

Like described in Section 3.3.2 the amount of different gates a backend usually provides is finite. The task of the compiler is to generate a new circuit, that is approximately identical to the input circuit, but only consists of gates that are in the gate set of the backend.

In classical computation, every arbitrary circuit can be produced by just using NAND gates. This means that, every arbitrary gate can be replaced with a combination of NAND gates. That is because this gate is one of the universal logic gates.

In quantum computing, a similar universality concept exists. The gate set a backend provides is usually a universal gate set. A set of gates is considered universal for quantum computation if any unitary operation can be approximated to any degree of accuracy by a quantum circuit [9] using only gates contained in this gate set. Every arbitrary unitary matrix U on an n qubit system can be decomposed into a product of two-level unitary matrices. A two-level unitary matrix can be implemented by using a single qubit unitary in combination with a CNOT gate. Gates in a universal gate set can approximate a single qubit unitary to an arbitrary precision. [9]

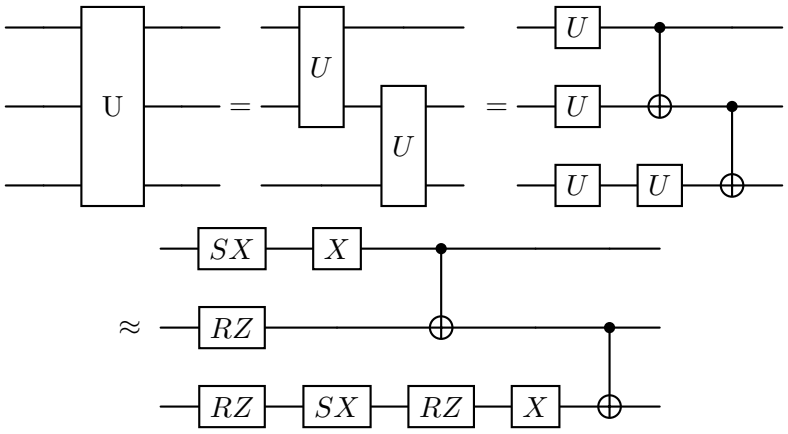


Figure 9: A general decomposition chain for a backend with a gate set consisting of SX, X, CNOT and RZ gates.

By using this decomposition chain, a compiler can break down any unitary into gates from the universal gate set of the backend. Such a decomposition chain for a backend with a gate set consisting of SX, X, CNOT and RZ gates is depicted in Figure 9. In addition to the goal of approximating the input circuit to a given degree, the circuit depth of the produced circuit should be as small as possible. That is due to the gate infidelities and the resulting overall circuit error, which is described in Section 3.3.3. A typical algorithm for the decomposition is the Solovay-Kitaev theorem that can approximate a unitary that acts on one qubit with single-qubit gates from a finite set [34]. Another approach would be the KAK decomposition, which decomposes a two-qubit unitary into a sequence of CNOT gates and one-qubit gates [35]. The solution algorithm for the gate translation of the individual compilers are explained in more detail in Section 5.

4.2 Initial Mapping

As already briefly mentioned in Section 3.3.1 the qubits in a backend, namely in the connectivity graph, are referred to as physical qubits [36]. The qubits in the circuit are referred to as *logical qubits* [36].
During the initial mapping stage, the compiler has to map the logical qubits to physical qubits prior to circuit execution [27]. This relabeling process does not introduce any additional gates to the circuit. In the literature, the initial mapping as well as the routing stage are often combined in one step, as the initial mapping is mandatory for the routing to take place [36, 37].

For example, a trivial initial mapping could be to map every logical qubit q_i to a physical qubit Q_i, i.e., $q_i \mapsto Q_i$. However, this specific mapping is not aware of the circuit it is for and is therefore probably not the best option, as an example in the following section shows.

It is not guaranteed that this mapping will satisfy the connectivity constraint for every circuit. But an advanced initial mapping strategic can reduce the additional gates which the routing stage otherwise must add.
The major challenge of the initial mapping is, that as the size of the connectivity map increases, it becomes increasingly difficult to find a perfect solution, as the mapping problem has been proven to be NP-complete [38].

This thesis describes initial mapping strategics of different compilers in Section 5.

4.3 Routing

As mentioned in Section 3.3.1 the qubits on a backend may have a limited connectivity. That means, the backend limits the possible interactions between qubit pairs. A multi-qubit operation just can be applied to qubits which share an edge on the connectivity graph. The quantum compiler has to place the multi-qubit operations on qubits that fulfill this condition.

For many circuits, an initial mapping is not sufficient to satisfy the connectivity constraints. Therefore, the mapping must be adjusted during the circuit execution.

In order to fulfill these constraints, the circuit must be adapted so that the qubits are remapped within the circuit. This can be achieved by *SWAP gates*. As the name suggests, the SWAP gates swaps the state of two qubits. Figure 10 depicts a SWAP gate and suggests a decomposition into three CNOT gates.

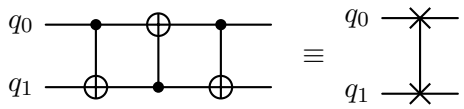


Figure 10: SWAP gate in combination with a decomposition into three CNOT gates.

As mentioned in Section 2.4.1 a CNOT operation has the notation $|A, B\rangle \rightarrow |A, A \oplus B\rangle$. Therefore, three CNOT operations, as depicted in Figure 10, operate on a two-qubit register, so that

$$\begin{aligned} |A, B\rangle &\rightarrow |A, A \oplus B\rangle \\ &\rightarrow |A \oplus (A \oplus B), A \oplus B\rangle = |B, A \oplus B\rangle \\ &\rightarrow |B, (A \oplus B) \oplus B\rangle = |B, A\rangle \,. \end{aligned} \tag{17}$$

It can be observed, that the three CNOT operations, depicted in Figure 10, and therefore the SWAP operation, indeed swap the quantum states of a two-qubit register. Before performing a multi-qubit operation, the compiler must add SWAP gates to generate a new mapping that satisfies the connectivity constraint, if the logical qubits are not already mapped to physical qubits that meet this requirement.
The primary goal of this stage is to identify paths in the connectivity graph connecting two physical qubits. Swap operations are subsequently implemented along the path.[27] This task is known to be NP-complete [39].

As with the gate translation task, the compiler should aim to minimize the number of added SWAP gates to reduce the overall circuit error.

The following example should clarify the connection of the role between the initial mapping and the routing stage. Figure 11 depicts a circuit going through the initial mapping and the routing stages of a quantum compiler.
Figure 11(a) shows the connectivity graph of a given backend. The circuit, depicted in

Figure 11(b), consists of four qubits and six CNOT gates. First, a trivial mapping is performed such that $q_i \mapsto Q_i$ with $i \in \{0, 1, 3\}$. With this mapping, the CNOT gate between q_1 and q_3 does not satisfy the connectivity constraints of the connectivity graph. Therefore, the compiler has to add a SWAP gate to create a new mapping. This new mapping now allows a two-qubit operation between q_1 and q_3, since q_1 is now mapped to Q_2. However with this new mapping, the upcoming CNOT operation between q_2 and q_3 is not allowed as the logical qubits q_2, q_3 are now mapped to Q_1, Q_3 respectively. That is why the compiler adds another SWAP operation to restore the mapping from the beginning. Now all remaining CNOT operations fulfill the connectivity constraint.
In Figure 11(c) the same circuit is depicted. However, it has a more advanced initial mapping. It can be seen that there is now no additional routing required by the compiler because every multi-qubit operation in the circuit satisfies the connectivity constraint.

This example should emphasize the importance of an advanced initial mapping strategic. The process of the mapping does not introduce additional gates to the circuit and therefore does not increase the overall circuit error.

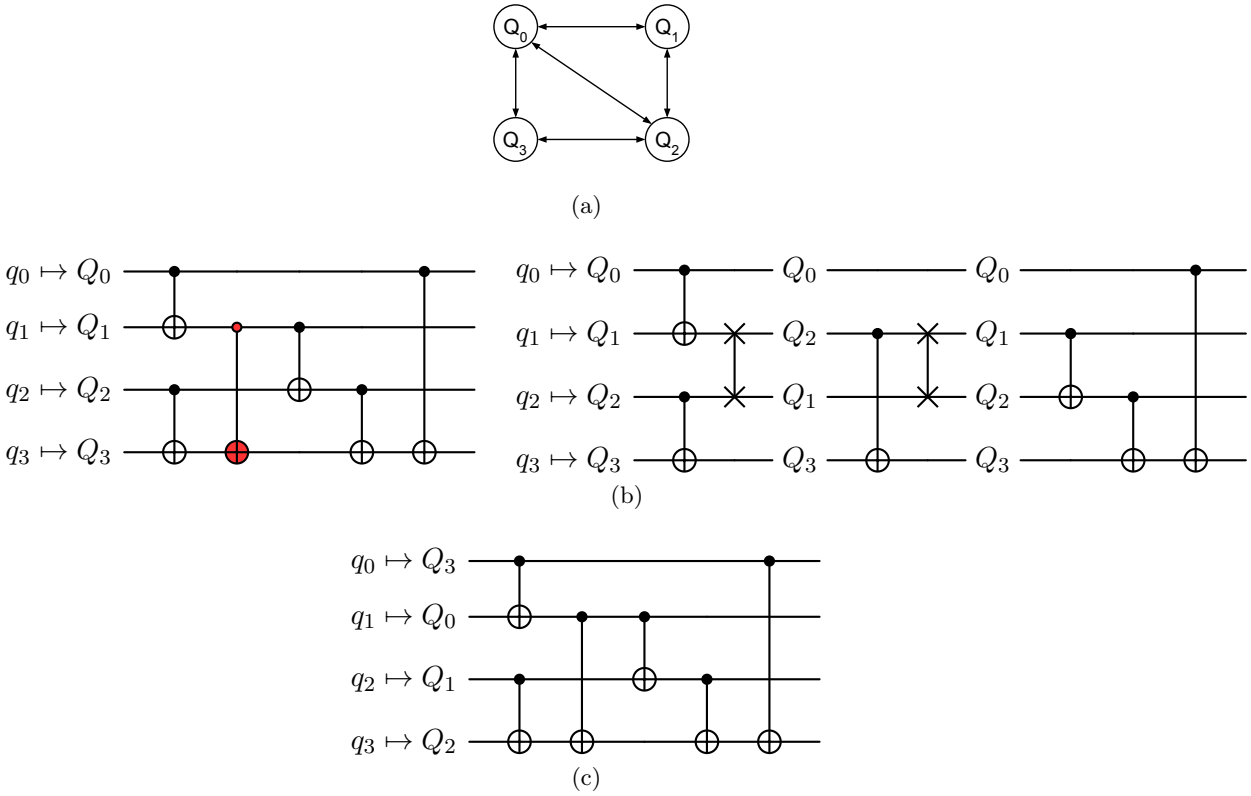


Figure 11: Example of an initial mapping with a subsequent routing.

The routing stage needs an initial mapping for the routing process. A good initial mapping can, of course, in some cases make the routing unnecessary.

This thesis describes routing strategics of different compilers in Section 5.

4.4 Optimization of Quantum Circuits

As stated in Section 3.3.3 the quantum operations are prone to noise and therefore possess a degree of infidelity. Thus, it is desirable to have a circuit with minimal gates and low

circuit depth.

Optimization is crucial at every stage of a quantum compiler. Especially since routing and initial mapping are NP-complete and therefore an exact solution is not feasible for large circuits [38, 39]. For this reason, optimization methods are used at these stages.
However, optimization can also be used as a separate stage. In this case, the entire circuit is analyzed and optimized.

The objective for a quantum compiler in the optimization stage is to reduce the overall circuit depth and the number of gates while maintaining the circuit's fidelity as much as possible.

There are mainly two possibilities to optimize a quantum circuit.
The first is through lossless optimization, achieved by removing unnecessary or redundant gates [3, 7]. For example, two CNOT gates, placed back-to-back, would cancel each other out. Therefore, the gates can be removed from the circuit.
The second possibility is an approximation of unitaries or a group of gates. This can lead to a higher infidelity of the circuit. However, this procedure can result in a lower circuit depth and therefore reduce the error [37].

This thesis describes optimizing strategics of different compilers in Section 5.

5 In-Depth Analysis of Quantum Compilers

This chapter presents a detailed analysis of three selected and commonly used quantum compilers, focusing on their compilation strategies.
One of the main contributions of this thesis is the thorough, and in-depth analysis of these compilers. Such an analysis of the functionality is important for several reasons. Firstly, it provides a basis for a fair comparison. Secondly, it gives developers a foundation for decision-making. This can help them decide which compiler to use for which specific purpose.
Although such an analysis is essential for the reasons mentioned above, it does not yet exist in this depth.

The chapter is structured as followed: In Section 5.1 an analysis of the BQSKit is provided. Section 5.2 gives an overview of the approaches the TKET compiler uses. Subsequently, Section 5.3 presents an analysis of the workflows of the Qiskit compiler.

There are several quantum compilers and frameworks that also contain compilers. These include Google's Cirq, Rigetti's Quilc, Microsoft's Quantum Dev Kit, Xanadu's PennyLane, IBM's Qiskit, Quantinuum's TKET, and Lawrence Berkeley National Laboratory's BQSKit.[3–8].
However, this thesis will limit the analysis and evaluation to only three compilers. These are Qiskit, TKET and BQSKit. Each of these compilers has a different approach.
Qiskit is an open-source quantum framework widely used in the field. It also has compilation capabilities [3]. TKET is an open-source, language-agnostic quantum compiler capable of generating code for a variety of NISQ devices [7]. BQSKit is a quantum compiler with its focus on circuit optimization [8].

As these compilers are all open-source, they are all updated frequently. Consequently, this analysis represents a snapshot of the current versions. The versions that are part of this thesis are Qiskit 1.0.0, TKET 1.25.0 and BQSKit 1.1.1.

5.1 Analysis of BQSKit

The Berkeley Quantum Synthesis Toolkit (BQSKit) is a quantum compiler framework which can compile quantum circuits for any backend [8]. It can compile a given circuit to any backend, as long as the gates from the universal gate set are known to BQSKit. It highly focuses on circuit optimization and uses a bottom-up synthesis approach for this [37]. The compilation process can be executed with four different levels of optimization [8].
The algorithms that are responsible for the compilation process are described in the following sections. The optimization of the system is mainly carried out by two algorithms, LEAP and QSearch. Since LEAP is simply a specification of QSearch, this text will focus on providing a more detailed description of QSearch.

5.1.1 QSearch

QSearch is a synthesis algorithm that can compile arbitrary unitaries into a sequence of two-qubit and single-qubit gates that are in the gate set. It is also aware of connectivity

constraints. [37]
Since each sequence of gates can be described as a unitary matrix, as described in Section 2.4.2, this algorithm can be used to compile any circuit onto an arbitrary backend.

Basically, QSearch spans a tree of possible solutions to synthesize a given unitary. It uses an A* algorithm to find the solution that is closest to the unitary. [37]

The A* algorithm is a graph traversal method for finding the shortest path between two points. The main advantage of the A* algorithm in contrast to other shortest path algorithms is the use of a heuristic function. This heuristic function estimates the cost from a given node to the goal, which helps the algorithm to prioritize nodes that are more likely to lead to the shortest path. [40]

Like stated above, the core of QSearch is a tree which is evolving during the algorithm. The root node describes a circuit structure with one U3 gate acting on each qubit separately. The algorithm expands the tree by placing successor nodes, a CNOT gate followed by one U3 gate on each of the two involved qubits, on each possible qubit line. In this process, the connectivity constraint is taken into account. An optimization function then determines the parameter of the U3 gates, by minimizing the distance between the target unitary U_{target} and each node. The A* algorithm selects the successor node that minimizes the estimated total cost of the path from start to finish. These costs $f(n)$ are specified by the amount of CNOT gates as well as the distance from the circuit to U_{target}. This process is repeated until the distance between U_{target} and a node meets a certain threshold. This node represents the synthesized circuit. [37]
Figure 12 depicts the evolution of the tree which is used in QSearch.

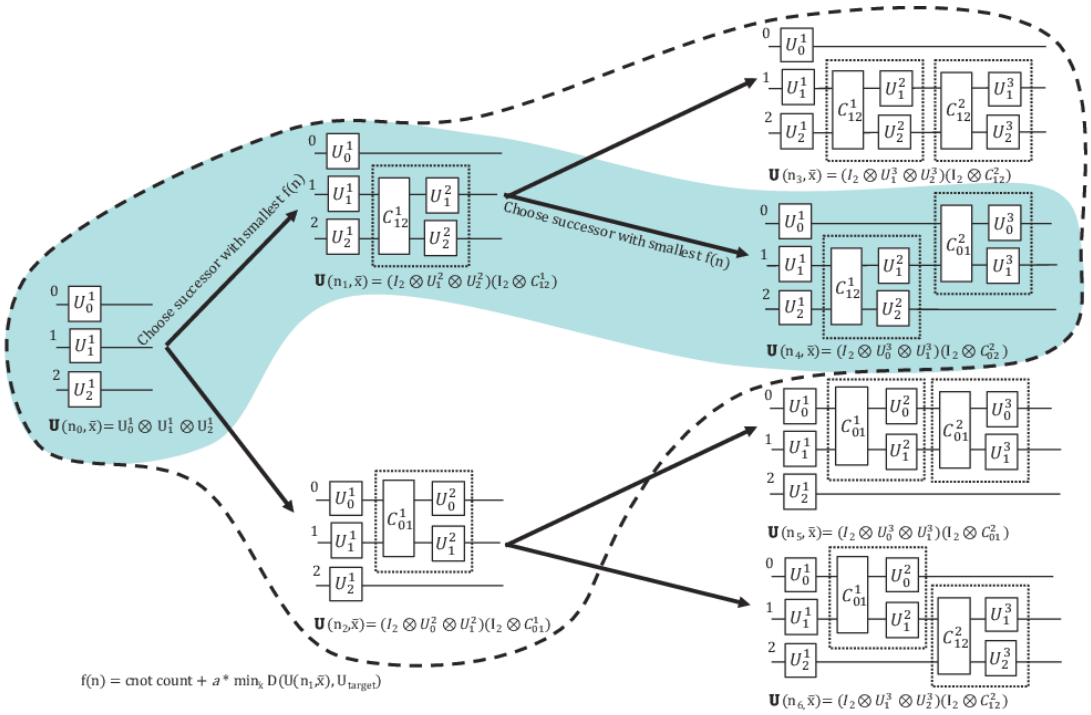


Figure 12: The evolution of the QSearch Tree. The tree is expanded from the root node, left in the picture, with successor nodes. A successor node consists of a CNOT gate and two U3 gates. The tree is expanded until the distance to U_{target} meets a certain threshold. The path on which the tree is expanded is chosen by a A* algorithm with the heuristic function $f(n)$. In this example, the algorithm selects the path highlighted in blue.
Source: [37]

Davis et al. also deliver a proof that this method works for any circuit that can be constructed with a finite number of CNOT and U3 gates [37]. While theoretically capable of solving for any circuit size, in practice, the scalability of QSearch is limited to four qubits [41]. This is, among other factors, due to the long backtracking chains [42]. Furthermore, as it converts the whole circuit to a single unitary, the complexity of QSearch is exponential to the number of qubits used in the circuit.

The QSearch algorithm can be considered both a routing algorithm (Section 4.3) and an optimization algorithm (Section 4.4).

5.1.2 Larger Exploration by Approximate Prefixes

Larger Exploration by Approximate Prefixes (LEAP) is a synthesis algorithm based on the QSearch framework, which is described in Section 5.1.1. It aims to improve the scalability and reduce the computational requirements. It achieves this by using an incremental approach that divides the synthesis process into segments and optimizes each one individually, which also reduces the frequency of backtracking. Therefore, it reduces complexity and search space by a divide-and-conquer method. [42]
Unlike QSearch, LEAP scales up to six qubits and according to Smith et al. LEAP can compile four qubit unitaries up to 59 times faster than QSearch, although it usually results in the same circuit depth [42].
However, it also considers the whole circuit as a unitary and therefore suffers from an exponential complexity just like QSearch.

5.1.3 SWAP-based BidiREctional Heuristic Search Algorithm

The SWAP-based BidiREctional heuristic search algorithm (SABRE) is an algorithm to approximate a solution for the mapping problem. The basic idea of SABRE is to get an initial mapping by using a reverse traversal technique. [36]

The core of SABRE is a SWAP-based heuristic search algorithm. This routing algorithm basically parses the circuit from left to right. If a two-qubit operation in the circuit does not fulfill the connectivity constraint, the routing algorithm finds a path in the connectivity graph with the help of a heuristic function. Based on this path, SWAP gates will be added to the circuit for the routing. [36]
The heuristic function is based on a Nearest Neighbor cost function, which prefers a route to the nearest physical qubit that satisfies the connectivity constraint [36].
To enable a look-ahead ability, not only the current two-qubit operation is considered, but also a set of future two-qubit operations. This ensures that routes are considered, which may not necessarily be the shortest path to physical qubits that allow for two-qubit operations. However, the sum of the routes for these two-qubit operations, together with upcoming ones, may be significantly lower. [36]
Besides the Nearest Neighbor cost function and the look-ahead ability, the heuristic function also introduces a decay effect, which penalizes routes that traverse qubits that were recently involved in another route. This decay effect makes sure that the heuristic search will tend to select SWAP operations that do not overlap, thus increasing the potential for parallelism in the resulting circuit. [36]

To generate an initial mapping, the SWAP-based heuristic search algorithm is performed twice. The two generated circuits are depicted in Figure 13. First, an initial mapping is randomly generated, followed by the application of a SWAP-based heuristic search to traverse the original circuit. This gives a final mapping, which is obtained by the remapping of the inserted SWAP operations. The final mapping obtained from this forward traversal is then used as the initial mapping in the subsequent reverse traversal. As all quantum operations, besides operations that collapse the quantum state, are reversible, a reverse circuit can be generated [10]. Figure 13 shows a reverse circuit. It is generated by inserting the CNOT operations present in the original circuit, parsed from right to left. One-qubit operations can be neglected here, as they do not affect the connectivity constraints. The same SWAP-based search is used with only the circuit reversed, and the original initial mapping is updated to the final mapping in the reverse traversal. This final mapping is the output from the SABRE algorithm. [36]

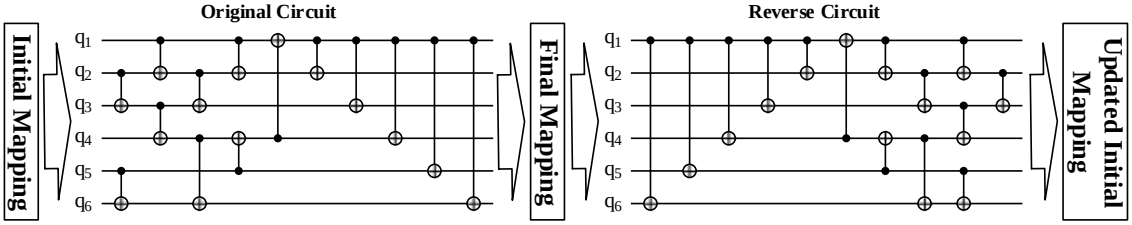


Figure 13: SABRE Reverse Traversal Technique. Produce an initial mapping by using a reverse traversal technique.
Source: [36]

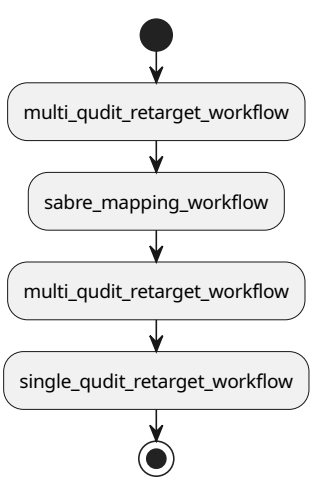
5.1.4 Permutation-Aware Synthesis

The Permutation-aware synthesis (PAS) is an extension to already described synthesis algorithms (see Section 5.1.1, Section 5.1.2).
The algorithm aims to find a permutation of the input and output qubits in order to change the unitary matrix. This can result in shorter circuits. [43]
However, as there are $n! \times n!$ permutations available, finding the best one is not feasible. Therefore, the already synthesized circuit is vertically partitioned into multiple blocks. The block is subsequently resynthesized for each possible unique permutation that also satisfies the connectivity constraint. [43]

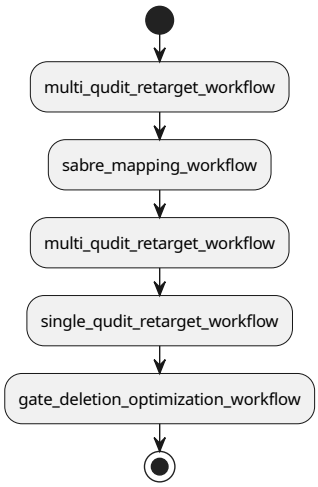
The permutation-aware mapping algorithm uses the concept of SABRE, but with a heuristic search algorithm that considers the previously found permutations. The routing algorithm, that is aware of permutations, inserts SWAP-gates based on the heuristic search algorithm. [43]

5.1.5 Compilation Pipeline

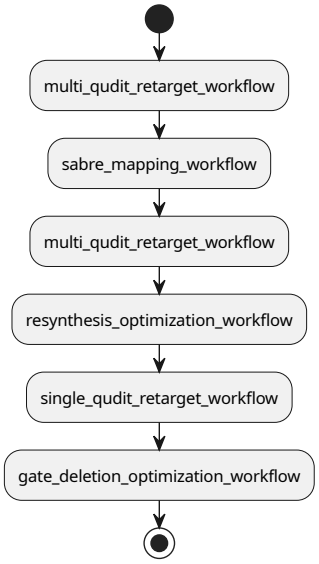
BQSKit has four compilation pipelines build in its framework. According to their documentation, the workflows have increasing optimization capabilities, which decreases the depth of the circuits [8].
This chapter provides a detailed description of the four workflows. These workflows will be evaluated in Section 7.1.3. Each workflow is a sequence of different subworkflows.
In the following section, each of the optimization workflows is divided into their subworkflows. Subsequently, each of these subworkflows is described in detail.
The information presented in this section is derived from the source code of BQSKit [8].



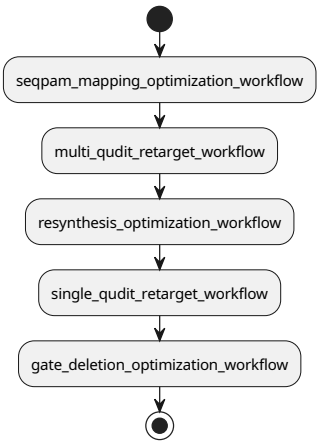
(a) BQSKit compilation workflow for optimization level 1.



(b) BQSKit compilation workflow for optimization level 2.



(c) BQSKit compilation workflow for optimization level 3.



(d) BQSKit compilation workflow for optimization level 4.

Figure 14: BQSKit compilation workflows for the four different optimization levels.

Figure 14(a) depicts the workflow for the first optimization degree. The first optimization level leads to a workflow that is divided into three distinct subworkflows, namely the *multi-qudit retarget workflow*, the *sabre-mapping workflow*, the *multi-qudit retarget workflow*, and the *single-qudit retarget workflow*.

Figure 15 shows the process of the multi-qudit retarget workflow. The objective of this workflow is to ensure that the circuit contains only multi-qubit gates that are also included in the gate set of the backend. The primary operations are conducted by the standard search synthesis workflow or, alternatively, by the AutRebase2QuditGatePass.

The standard search synthesis workflow executes the QSearch algorithm, described in Section 5.1.1 if the circuit has a width of three or less. For larger widths, the LEAP algorithm that is described in Section 5.1.2 is used. This differentiation is important for the scalability of the QSearch algorithm. Both algorithms synthesize the circuit so that it contains only gates from the backends gate set.

The AutoRebase2QuditGatePass substitutes the two-qubit operations in the circuit with gates from the gate set of the backend. This is achieved by an instantiation method that replaces each two-qubit gate with a collection of at most three quantum gates from the backends gate set [8].
The three algorithms lead to the same result, namely that all multi-qubit operations that remain within the circuit, are included in the gate set of the backend.

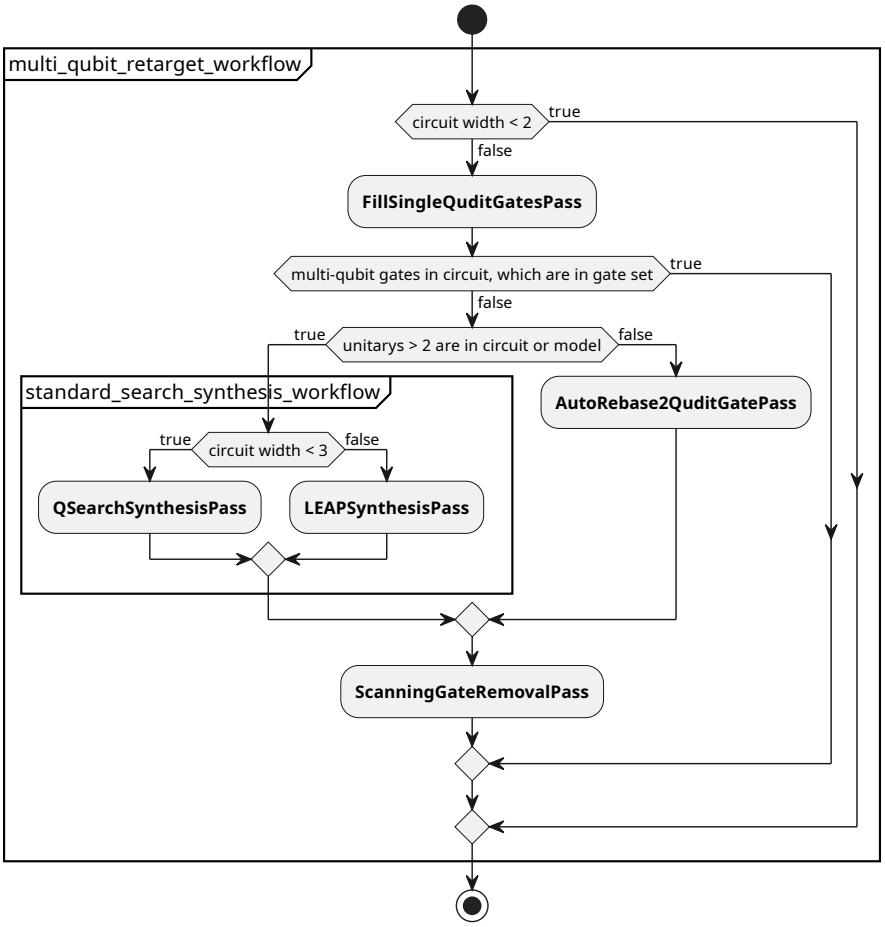


Figure 15: A flowchart of the multi-qudit retarget workflow used in BQSKit.

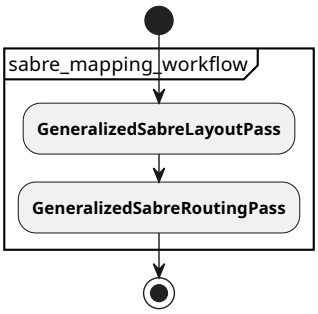


Figure 16: A flowchart of the SABRE mapping workflow used in BQSKit.

Figure 16 depicts the process of the sabre mapping workflow, which is performed after the multi-qudit retarget workflow. This workflow consists of the GeneralizedSabreLayoutPass and the GeneralizedSabreRoutingPass.
The GeneralizedSabreLayoutPass finds an initial mapping by executing the SABRE algo-

rithm that is described in Section 5.1.3. Subsequent after this, the GeneralizedSabreRoutingPass is performed, utilizing the SABRE heuristic to perform the routing and fulfill the connectivity constraint.

After the sabre mapping workflow, the compiler runs the multi-qudit retarget workflow again. This ensures that any SWAP operation that may have been introduced during the sabre mapping workflow are substituted with gates that are included in the gate set of the backend.

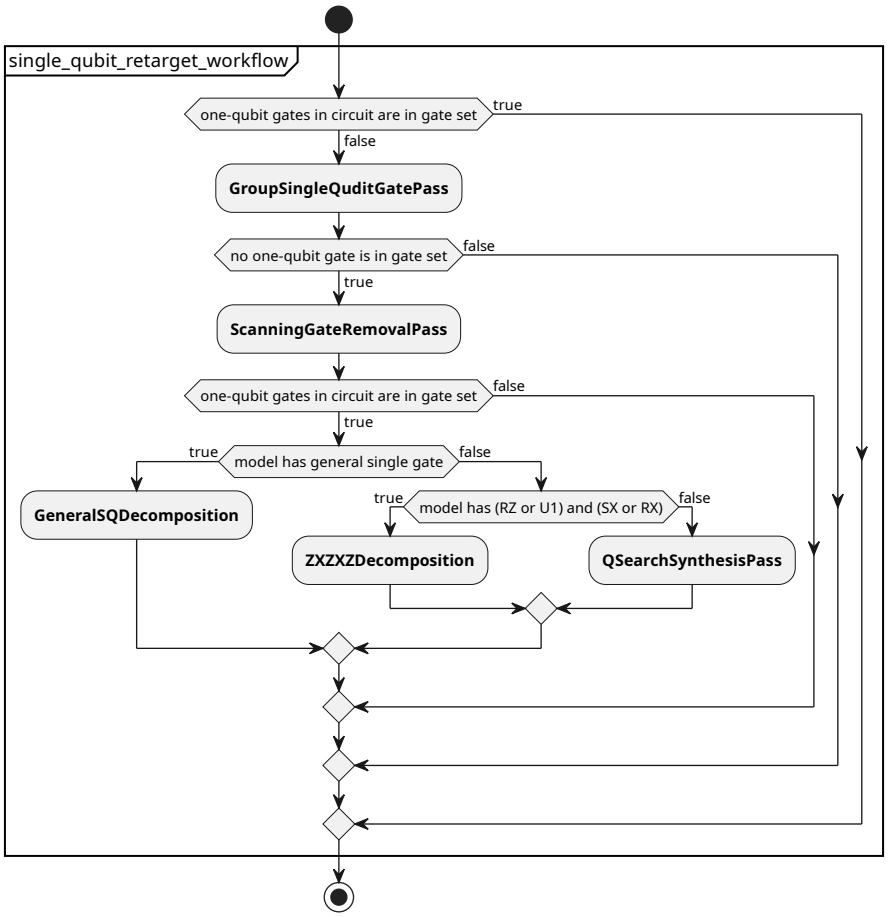


Figure 17: A flowchart of the single-qudit retarget workflow used in BQSKit.

The final process in the workflow generated for optimization level 1, is the single-qudit retarget workflow, which is depicted in Figure 17. In this process, all one-qubit gates are substituted with gates from the backends gate set.
The circuit is initially passed to the ScanningGateRemovalPass, where it is scanned from left to right. A one-qubit gate is removed, if the circuit's fidelity remains in a specified threshold after the removal. If this algorithm has successfully removed all the gates that are not in the native gate set, the workflow comes to an end. However, if there are still one-qubit gates that are not in the native gate set, these gates will be decomposed in the subsequent steps.
The single-qudit retarget workflow contains three decomposition algorithms. The GeneralSQDecomposition only works, if the native gate set includes a GeneralGate, which in BQSKit is a gate that parametrizes any unitary [8]. If the backend's gate set contains a

R_Z gate and either a $\sqrt{X}$ gate or a R_X gate the compiler runs the ZXZXZDecomposition task. This algorithm substitutes a one-qubit gate with a sequence of R_Z gates and R_X gates [8]. If the native gate set does not contain any of the above gates, the QSearch algorithm is applied. It synthesizes the circuit with gates from the gate set.
In either way, after the single-qudit retarget workflow, the circuit just contains gates that are included in the native gate set.

The workflow generated for the second optimization level is depicted in Figure 14(b).
It can be observed that the workflow differs only in the addition of the *gate-deletion optimization workflow* at the end of the process.

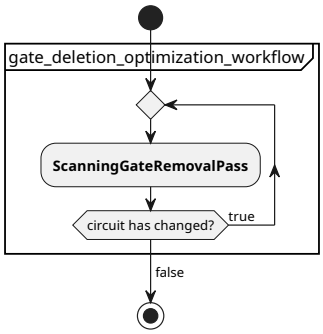


Figure 18: A flowchart of the gate-deletion optimization workflow used in BQSKit.

Figure 18 illustrates the process of the gate-deletion optimization workflow, which runs the ScanningGateRemovalPass until the circuit reaches a stable state, wherein no gates have been removed.
This workflow with the second optimization level produces circuits with at most the same number of gates or fewer than a circuit produced by the previous workflow.

Figure 14(c) depicts the workflow produced for the third optimization level. It introduces a new subworkflow, namely the *resynthesis-optimization workflow*.
The resynthesis-optimization workflow, illustrated in Figure 19, runs the standard search synthesis workflow, until the number of multi-qubit gates remains unaltered.

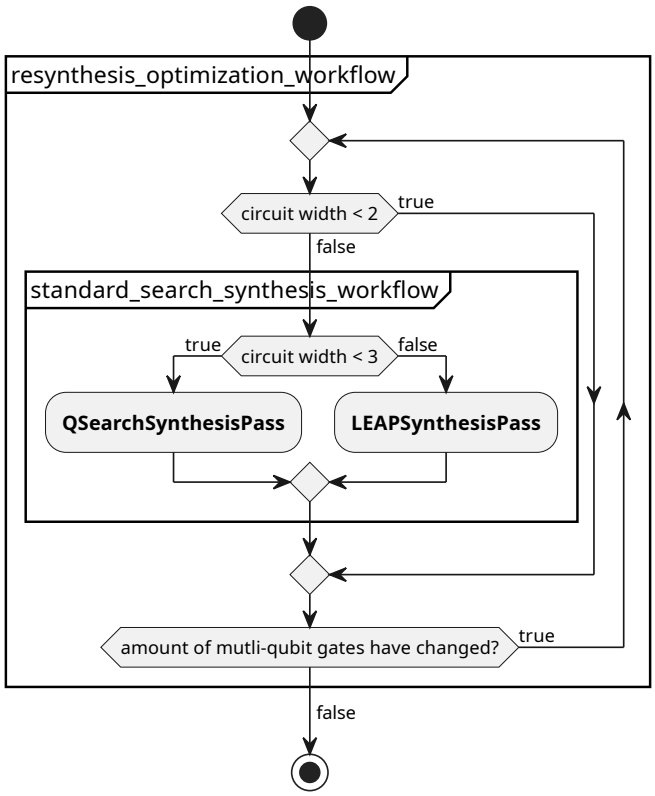


Figure 19: A flowchart of the resynthesis-optimization workflow utilized in BQSKit.

The last workflow generated for the third optimization level is depicted in Figure 14(d). Here, the initial mutli-qudit retarget workflow as well as sabre-mapping workflow are replaced by the *seqpam-mapping optimization workflow*.

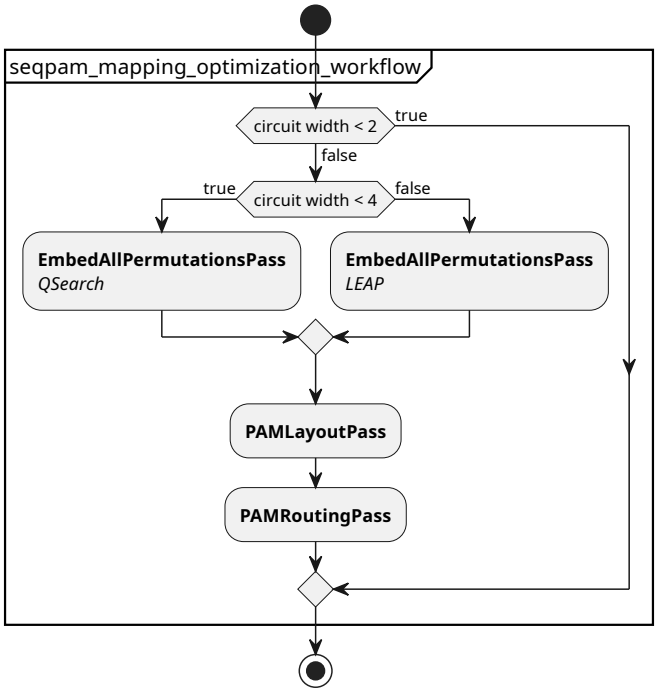


Figure 20: A flowchart of the seqpam-mapping optimization workflow used in BQSKit.

The seqpam-mapping optimization workflow uses the PAS algorithm, described in Section 5.1.4, for finding an initial mapping and subsequently generating a routing with a

potential lower two-qubit count.
For both tasks, the EmbedAllPermutationsPass initially generates permutations for either the QSearch or the LEAP algorithm, depending on the circuit width. Subsequently, the PAMLayoutPass and the PAMRoutingPass calculate the initial mapping and the routing, respectively.
The workflow generates circuits that satisfy the connectivity constraint and is expected to introduce fewer CNOT gates in the routing process than other algorithms such as SABRE [43].

All of these optimization workflows are evaluated in chapter Section 7.1.3.

5.2 Analysis of TKET

The TKET compiler, developed by Cambridge Quantum Computing, is a quantum compiler designed for NISQ devices. The compiler is retargetable and language agnostic. This means that it can parse input from a wide range of quantum software platforms and can generate circuits for many different quantum devices. [7]

The compilation process ca be split into two phases. A backend-independent optimization phase, with the objective to reduce the circuit depth, and a backend-dependent phase that prepares the circuit to be run on the backend. [7]

In contrast to BQSKit and Qiskit, TKET does not provide a general compilation routine, rather it defines a separate compilation pipelines for each supported backend. However, it is possible to define own compilation pipelines for custom backends. [7]
The chapter first covers the backend-dependent phase, consisting of the mapping and routing algorithm. Subsequently, the two main optimization techniques, namely the peephole optimization and the macroscopic optimization, used by TKET are described.

5.2.1 Graph Placement

TKET uses an initial mapping method called graph placement. It uses a heuristic which tries to maximize the number of two-qubit operations at the beginning of the circuit which can be performed without additional routing. [7]

To accomplish that, the mapping problem, which is described in Section 4.2, is cast as a subgraph isomorphism problem [7]. Given two graphs G_1 and G_2 the subgraph isomorphism problem is the problem of finding a subgraph in G_2 that is *isomorphic* to G_1. Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there exists a bijective function $f : V_1 \mapsto V_2$ such that iff $(u, v) \in E_1$, then $(f(u), f(v)) \in E_2$. [44]

For this task, the circuit is represented as an interaction graph G_I. Each logical qubit is identified by a vertex and a multi-qubit operation is depicted as an edge between the vertices that represents the qubits the operation acts on. [7]
The graph placement searches for a subgraph isomorphism between G_I and the connectivity graph G_C from the backend. If no isomorphism can be found, an edge from G_I is removed. Edges that belong to operations that are scheduled for a later point in the circuit are more likely to be removed. Each vertex that does not have an edge anymore

is removed from G_I. The algorithm repeats this process until a subgraph isomorphism is found. The founded subgraph serves as the initial mapping.[7]

5.2.2 Routing Approach

The routing algorithm from TKET, begins with an initial partial placement, which is identified by the graph placement algorithm. It then proceeds to add SWAP operations until the connectivity constraint is fulfilled.
Like SABRE, the routing algorithm makes use of a nearest neighbor heuristic. In addition to that, if this heuristic produces multiple routes with an equal ranking, it generates a tree with a branch for each possible route. With the help of this tree, the costs for future routes are calculated, until a winning route is identified. [7]

5.2.3 Peephole Optimization

The peephole optimization uses a sliding window which traverses the circuit and searches for small patterns that are replaceable with equivalent substructures. Substructures that are eligible for this replacement have a lower gate count or contribute to a smaller circuit depth. The peephole optimization is generic and therefore not adjustable for a specific application. [7]

In TKET, the gate set of the circuit is converted to a set of *Clifford gates* [7]. This set includes the Hadamard gate, the CNOT gate and the $R_Z(\frac{\pi}{2})$ gate [10]. A variety of simplification techniques are available for circuits consisting of Clifford gates [7, 45]. Figure 21 depicts two example circuits constructed with Clifford gates that can be reduced to equivalent circuits containing fewer CNOT gates.

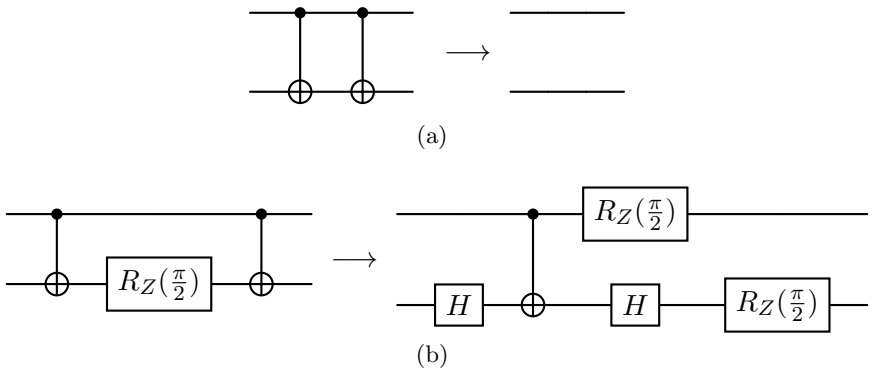


Figure 21: Simplification techniques for Clifford gates. Both circuit snippets (left) can be reduced to structures with a lower CNOT gates count (right).
Source: Adapted from [7]

In addition to the Clifford simplification approach, TKET also uses the sliding window to identify long sequences of gates over one or two qubits. The compiler performs a KAK or a Euler decomposition if the resulting sequence helps to reduce the CNOT gate count.[7] The advantage of the decomposition algorithms is that they define an upper bound on the resulting gate count [35].

Both methods, which use the sliding window of the peephole optimization, have the potential to reduce the number of CNOT gates.

5.2.4 Macroscopic Optimization

In contrast to the peephole optimization, the macroscopic optimization aims to identify high-level structures and subcircuits within the circuit. This optimization method depends on the specific quantum algorithm which the circuit implements. The components of the quantum algorithm are analyzed, and an attempt is made to substitute these components with equivalent structures that either have fewer CNOT gates by default or structures that can be more effectively optimized by the peephole optimization. [7]

The macroscopic optimization method requires a circuit representation which labels separate components in the quantum algorithm [7]. In the conception chapter of this thesis, OpenQASM is used as the main representation of the circuit. As OpenQASM is a low-level description of quantum circuits with a primary focus on gate-level operations, the macroscopic optimization is not able to identify high-level structures in the circuit.

5.2.5 Compilation Pipeline

As stated in Section 5.2, TKET does not provide a general compilation pipeline. Each backend, which is supported by the compiler, has a separate defined compilation routine. In Section 6.5, a general compiler pipeline using the algorithms provided by TKET is designed and discussed.

5.3 Analysis of Qiskit

Qiskit is an open-source quantum computing framework developed by IBM. The framework is designed to operate with a quantum computer at the application, circuit, and pulse levels, respectively. It contains multiple tools for circuit compilation, building applications, simulating quantum hardware and for quantum hardware calibration. Although it is primarily developed to operate with the quantum computers provided by IBM, it can also be utilized to design algorithms that can be executed on any quantum hardware that supports a Qiskit input or a circuit representation in OpenQASM.

5.3.1 Overview of the Transpiler

The compilation tool as defined in Section 4 is called *transpiler* in Qiskit. As with BQSKit, the transpiler is capable of generating circuits for an arbitrary backend that is defined by a connectivity graph and a gate set. There is already a pre-built compilation pipeline with four different optimization levels. These levels are numbered sequentially from 0 to 3. It uses already discussed algorithms like SABRE (Section 5.1.3) and the subgraph isomorphism placement algorithm (Section 5.2.1). This compilation pipeline is described in depth in this chapter.

For each optimization level, the pre-built compilation pipeline defines a different workflow. However, these workflows can be divided into eight distinct stages, which are as follows:

Pre-Init stage: Checks if the circuit contains instructions that are not supported.

Init stage: High-level structures in the circuit are decomposed. All unitaries that remain in the circuit, act on no more than qubits.

Layout stage: Logical qubits are mapped to the backends physical qubits.

Routing stage: SWAP gates are added to fulfill the connectivity constraint.

Translation stage: All gates in the circuit are translated to gates from the gate set of the backend.

Pre-Optimizing stage: Prepares the circuit for the optimization.

Optimization stage: Optimization methods are applied in order to reduce the sources of noise-related errors.

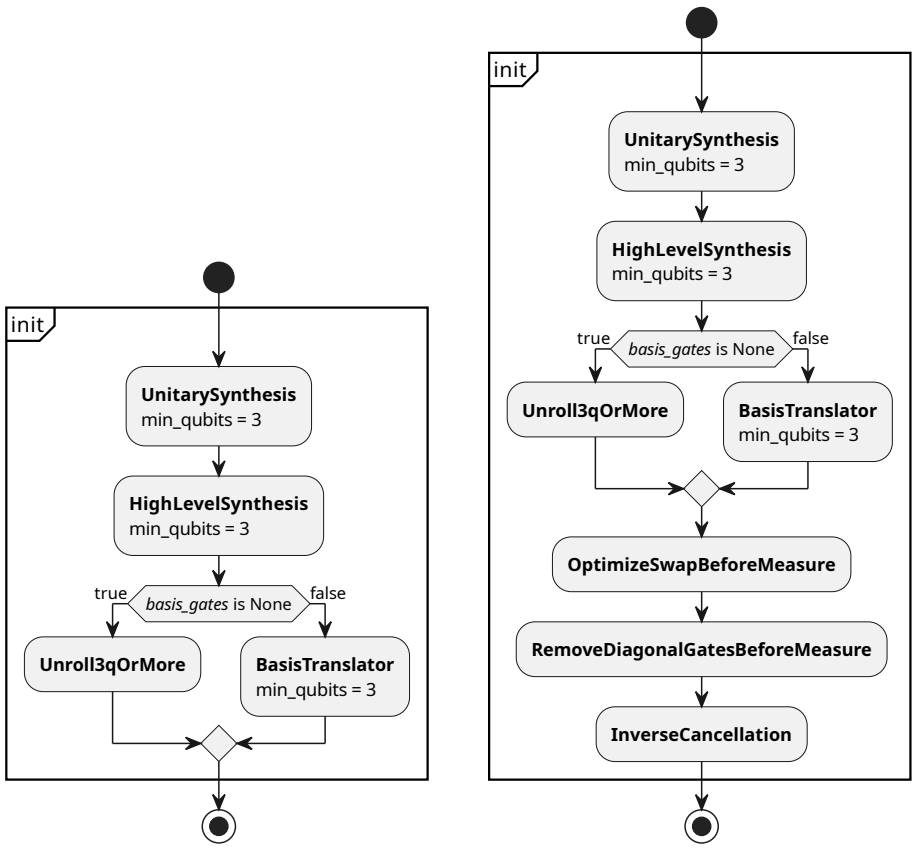
Scheduling stage: Improves the scheduling of the circuit on the backend.

Each of these stages contains multiple algorithms that are applied to the circuit. However, it is possible to replace each of these stages with a self-defined stage.
The following is a description of all the stages for each level of optimization.

5.3.2 Compilation Pipeline

The information presented in this section is derived from the source code of Qiskit [3].
As stated, the pre-init stage checks the circuit for flow control instructions which are not supported by Qiskit. Therefore, the stage does not result in any circuit modifications at any optimization level. Consequently, it is not addressed further in this context.

The initial stage ensures that all unitaries that work on more than two qubits are decomposed or in the native gate set. This process is the same for the first three optimization levels. In the fourth level, further optimization passes are appended.

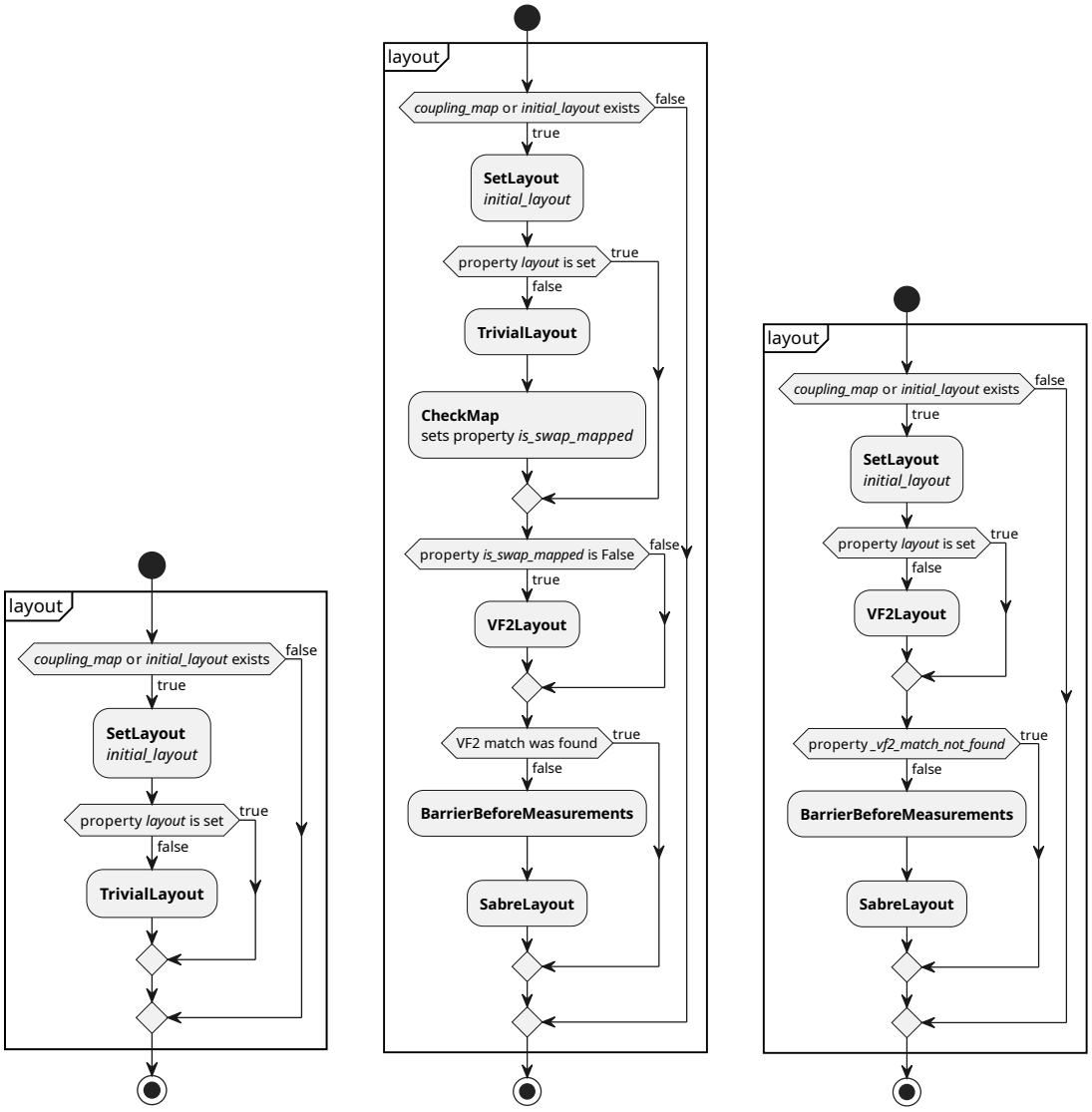


(a) Flowchart of Init-stage in the first three optimization levels.

(b) Flowchart of Init-stage in the fourth optimization level.

Figure 22: The workflow of the init stage for all four optimization levels. For the first three levels on the left and for the fourth optimization level on the right.

Figure 22 illustrates the initial stage of the Qiskit compiler. The *UnitarySynthesis* pass at the beginning of the workflow synthesizes unitaries. The gates to which a unitary is synthesized are part of the native gate set.
Subsequently, higher-level objects as well as custom definitions are unrolled in the *HighLevelSynthesis* pass. Should a gate set exist, the *BasisTranslator* pass translates all involved gates according to an equivalence library. If there are no native gates, the *Unroll3qOrMore* pass recursively expands gates that act on more than two qubits until the circuit contains gates with two-qubit and one-qubit gates.
All of these passes just act on objects in the circuit that act on three or more qubits.
In addition to that, the workflow for the fourth optimization level appends three different optimization passes, namely *OptimizeSwapBeforeMeasure*, *RemoveDiagonalGatesBeforeMeasure*, *InverseCancellation*. These processes remove unnecessary and redundant gates and perform a relabeling on the qubits if necessary.
All of these two workflows decompose unitaries that act on more than three qubits. However, the workflow for the fourth optimization level has the potential to reduce the gate count.



(a) Flowchart of layout stage for optimization level 0.

(b) Flowchart of layout stage for optimization level 1.

(c) Flowchart of layout stage for optimization level 2-3.

Figure 23: The workflow of the layout stage that is responsible for determine and set the initial mapping.

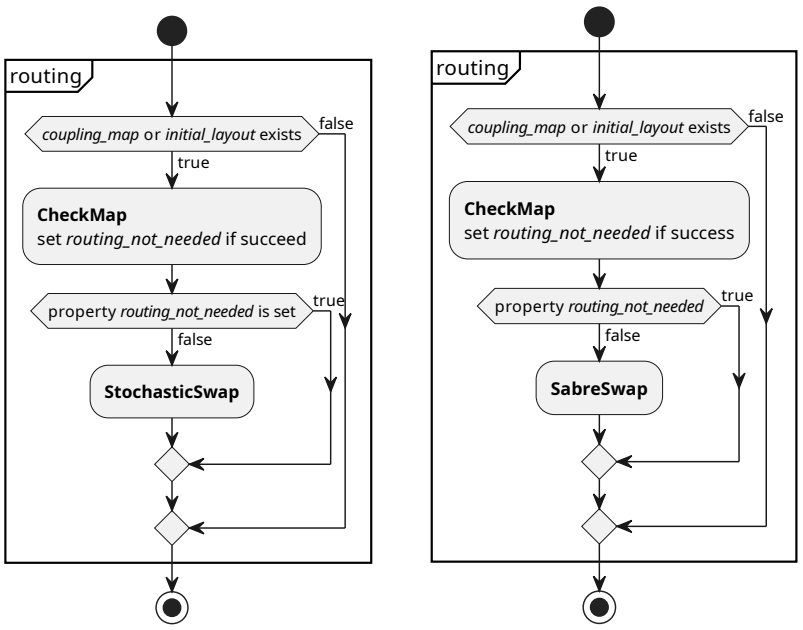
The workflows for the layout stage is depicted in Figure 23. There are three distinct workflows, as the optimization levels 2 and 3 share the same.

All three workflows require a coupling map, or a predefined initial layout, to begin. If a predefined layout exists, it is set by the *SetLayout* pass, and the routine would then conclude without further processing.

The workflow for the first optimization level runs the *TrivialLayout* pass to identify an initial mapping. It maps each logical qubit q_x to a physical qubit Q_x, $q_x \mapsto Q_x$. This mapping process is not aware of the circuit and does not aim to improve the subsequent routing routine.

The second optimization level, depicted in Figure 23(b), also initially runs the TrivialLayout pass. However, subsequent to this, it checks whether the mapping already satisfies the connectivity constraint. This ensures that the *VF2Layout* pass is only executed when necessary. The VF2Layout pass is similar to the graph placement algorithm described in Section 5.2.1 in TKET, with the exception that edges from the circuit graph are not

removed in the event that no result is obtained. Should a mapping be retrieved by the VF2Layout pass, the workflow will conclude. Furthermore, an additional routing is not required as this layout would fulfill the connectivity constraints. If, however, no mapping is found, the workflow will proceed to the *SabreLayout* pass, which is described in Section 5.1.3.
The layout workflow for the optimization levels 2 and 3, depicted in Figure 23(c), is nearly identical to the workflow with optimization level 1. The only distinction is that the TrivialLayout pass is not executed in advance, and therefore it begins the mapping process with the VF2Layout pass.



(a) Flowchart of routing stage for optimization level 0.

(b) Flowchart of routing stage for optimization level 1, 2 and 3.

Figure 24: The workflow of the routing stage that is responsible for inserting SWAP gates to satisfy the connectivity constraint.

Figure 24 depicts the workflow for the routing stage. Both workflows start by checking if the routing is necessary. This is performed by the *CheckMap* pass, which determines if all multi-qubit operations fulfill the connectivity constraint. Should the routing be necessary, the workflow for the first optimization level adds the *StochasticSwap* pass. This pass uses a randomized algorithm to insert SWAP gates. The remaining optimization levels apply the *SabreSwap* pass, a SWAP-based heuristic search algorithm that is described in Section 5.1.3. Consequently, it is possible, that a more optimal result may be obtained at a higher optimization level. However, the algorithm may require more time.

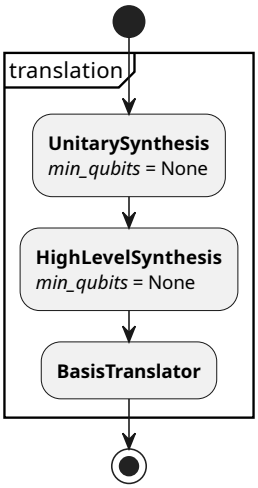


Figure 25: The workflow of the translation stage.

The workflow for the routing stage is depicted in Figure 25 and remains consistent across all four optimization levels. It is similar to the initial stage workflow of the first three optimization levels. Three passes are executed in sequence, namely the UnitarySynthesis pass, the HighLevelSynthesis pass and the BasisTranslator pass. The BasisTranslator pass decomposes all quantum gates in the circuit to gates that are in the native gate set. This is achieved through the use of an equivalence library. This library contains decomposition rules of known gates. In these rules, the basic translator pass searches for a path from a native gate to the gate of the circuit to be decomposed. Once the workflow has been completed, all gates that are included in the circuit are part of the gate set of the back-end. Furthermore, it also satisfies the connectivity constraint, as it does not introduce new multi-qubit operations on qubits that are mapped to non-connected physical qubits.

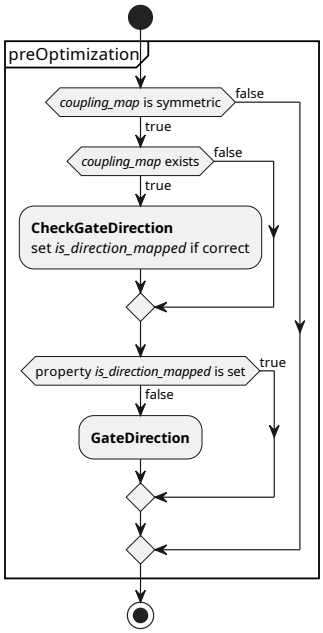


Figure 26: The workflow of the pre-optimization stage.

Figure 26 depicts the workflow of the pre-optimization stage, which remains consistent across all optimization levels. The pre-optimization process requires a symmetric connec-

tivity graph to run. As stated in Section 3.3.1, the connectivity graph can be represented as a directed graph. This means that it also restricts which physical qubit must serve as the control qubit and which should be the target qubit in a multi-qubit operation. To check if the circuit fulfills this restriction, the *CheckGateDirection* pass is run. Should the circuit not satisfy this condition, the pass *GateDirection* is executed. This pass modifies multi-qubit operations in order to align with the direction of the coupling graph. An example of such a modification illustrated in Figure 27. Here, a Controlled-X gate with q_0 as control and q_1 as target bit is modified so that target and control qubits are flipped.

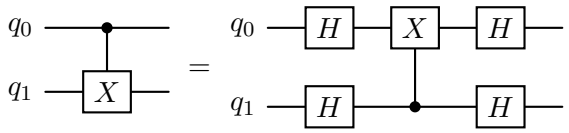
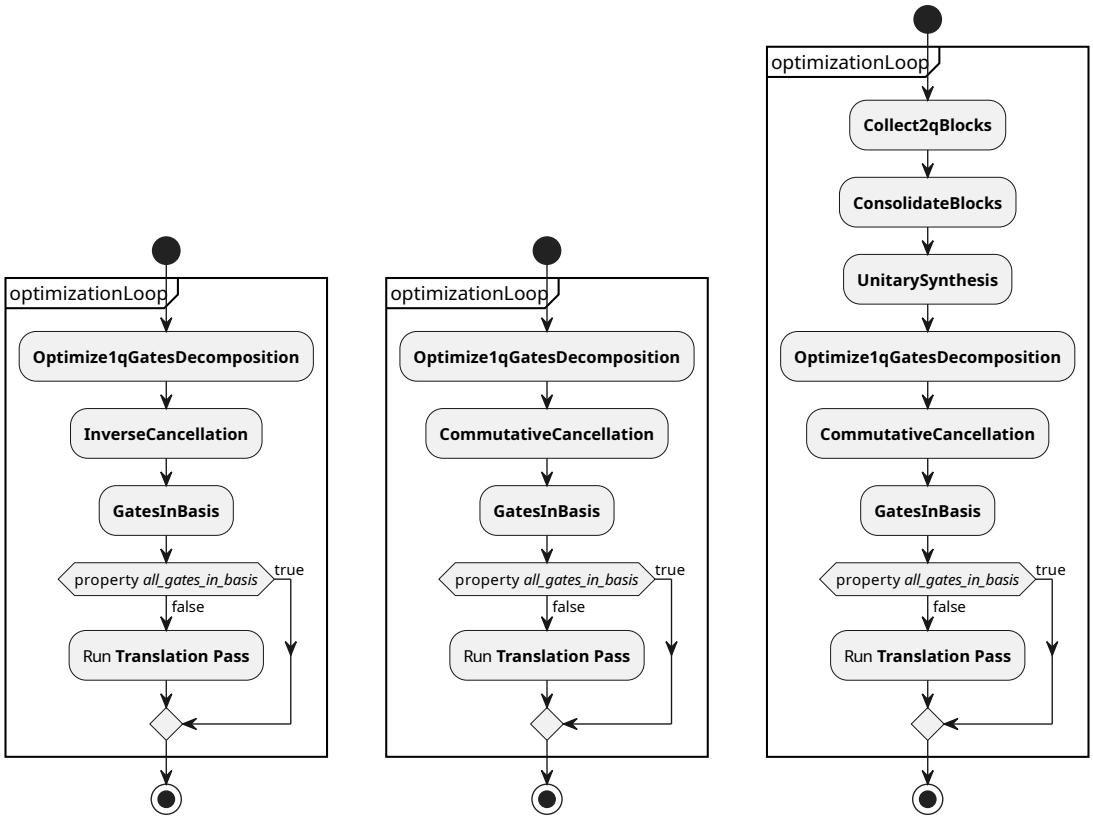


Figure 27: Direction change of a Controlled-X gate.



(a) Optimization workflow for level 1.

(b) Optimization workflow for level 2.

(c) Optimization workflow for level 3.

Figure 28: The optimization loops in Qiskit for three different levels of optimization.

Figure 28 shows the three different optimization workflows that exist in Qiskit. These workflows run all run in a loop, which is referred to as optimization loop in this section. First, the process within the loop is described for each optimization level. Subsequently, the termination conditions are explained.

As the compiler for optimization level does not do any optimization in this stage, it is not listed in Figure 28.

The optimization loop for level 1, depicted in Figure 28(a) consists of two different passes,

namely the *Optimize1qGatesDecomposition* pass and the *InverseCancellation* pass. The Optimize1qGatesDecomposition pass synthesizes chains of one-qubit gates using the Euler decomposition and combines them into a single gate. In the InverseCancellation pass, gates that are the inverse of each other and occur back-to-back are removed. Figure 21(a) shows that a two CNOT gates that are back-to-back can be removed. Following these two passes, the compiler employs the GatesInBasis pass, to ascertain whether all gates within the circuit remain within the gate set of the backend. If this is not the case, the translation stage is executed once more. This is important, as the Optimize1qGatesDecomposition may potentially produce gates that are not necessary within the gate set.
For the optimization level 2 the optimization loop is nearly identical to the previous one. Except that the InverseCancellation pass is substituted with the *CommutativeCancellation* pass. This pass not only considers gates that are back-to-back, but also applies commutation rules to the circuit in order to move gates and then remove the gates that are inverse to each other.
The optimization loop for optimization level 3 is structured like the one for optimization level 2. However, three additional gates are run in advanced, namely the *Collect2qBlocks* pass, the *ConsolidateBlocks* pass and the *UnitarySynthesis* pass. The Collect2qBlocks pass partitions the circuit into several blocks that contain at least one two-qubit operation. Subsequently, these blocks are rewritten as unitaries in the ConsolidateBlocks pass. In the UnitarySynthesis pass, these unitaries are approximated to sequences of gates from the gate set. The level of approximation can be set by the attribute *approximation_degree*.
The optimization loop for optimization level 1 and 2 terminates once the circuit depth remains unchanged in two consecutive iterations.
The termination condition in optimization level 3 is more advanced. There, the compiler searches for a local minimum in terms of the circuit depth and breaks the loop if it is found.

The workflow for the scheduling stage is not described in depth, as it does not contribute to the circuit properties that are evaluated in section Section 7. However, a flowchart of this stage is depicted in Appendix A.

6 Design and Implementation of the Evaluation Framework

This chapter offers an overview of the evaluation framework which is used in Section 7. In Section 6.1 the properties of the compilation workflow which the framework measures are presented. Section 6.2 and Section 6.3 provide an overview of the circuits and backends that are used in the evaluation. In Section 6.4 the workflow of the evaluation framework is presented. Section 6.5 the compilation pipeline for TKET that is used during the evaluation is introduced. The Section 6.6 describes the implementation of the Echoed Cross-Resonance gate in BQSKit. Finally, Section 6.7 provides an overview of another compilation workflow that is used in the evaluation.

6.1 Description of the Analyzed Properties

The evaluation framework in this thesis runs the compilation pipeline from the three different quantum compilers, that are described in Section 5, with respect to different backends. Subsequently, the resulting circuits are analyzed and multiple properties of them are measured.

The compilers allow the user to adjust multiple parameters for the compilation process. However, the evaluation in this thesis if focused on the optimization level and the degree of approximation.

The impact of the optimization level on the compilation workflow in Qiskit and BQSKit is analyzed in depth in Section 5. The same impact for TKET is described in Section 6.5 as this compilation pipeline is part of the implementation.

The approximation degree serves as a value that determines the extent to which the synthesis algorithms may approximate the circuits. It should be noted that such an approximation degree is only present in Qiskit and BQSKit, as TKET does not focus on synthesis. The exact definition of this value differs between Qiskit and BQSKit.
In Qiskit, the degree of approximation is only vaguely defined as a heuristic dial between one and zero. One represents no approximation, while zero represents maximal approximation. The degree is only used in the UnitarySynthesis pass.[3]
The approximation degree in BQSKit is also a value between one and zero. It is defined as the maximum distance between the target and the synthesized unitary that is permitted. This distance is based on the *Hilbert-Schmidt inner product*, which uses the mathematical relationship described in Equation (9). The Hilbert-Schmidt inner product is defined as

$$\langle U, U_S \rangle_{HS} = tr(U^\dagger U_S), \tag{18}$$

with U to be the unitary to synthesize and U_S the resulting unitary after the synthesis process. The trace tr of a matrix is defined as the sum of its diagonal elements. [8, 9, 37]

The circuit properties that are measured for the evaluation are the depth of the circuit, the amount of the one-qubit gates and the amount of the two-qubit gates.
As stated in Section 2.4.2, the circuit depth is defined as the longest route in the circuit, consisting of gates, from the initialization of a qubit until a measurement. The compilation process aims to decrease the circuit depth because the probability for decoherence increases

with more gates acting on a qubit. Therefore, it is preferable to have a circuit with a lower circuit depth.
The amount of gates can also be used to estimate the fidelity of the circuit, as the fidelity tends to decrease with the number of gates that are involved in the operation. The number of gates is classified as one-qubit gates and two-qubit gates, with the latter typically possessing a much higher error rate. This value is therefore calculated separately. In addition to that, the time that the compilation process is needed is also measured, as this value can also be important in deciding which compiler to choose.

6.2 Description of the Analyzed Circuits

The evaluation is conducted on two selected circuits. The first circuit is a parametrized circuit with a high degree of expressibility that can therefore be used in a variety VQAs [46]. The second circuit is a QAOA ansatz that encodes the max-cut problem that is described in Section 2.6.3.

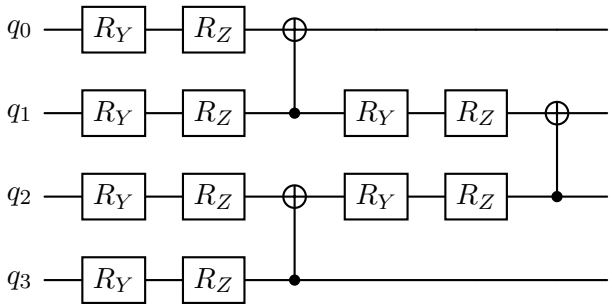


Figure 29: A single layer of the VQA circuit, which is proposed by Sim et al. for its high expressibility and representational quality of a solution space.
Source: Adapted from [46]

Figure 29 illustrates a single layer of a circuit that can be used in VQAs. One layer consists of twelve one-qubit gates, each of which is parametrized, and three CNOT gates that act on four logical qubits. To generate a circuit, multiple layers a put back-to-back. The number of layers in a circuit can be increased to enhance its expressibility [46]. However, this also results in a greater circuit depth.
The circuit used for the evaluation consists of five layers, a configuration that, according to Sim et al., represents a good trade-off between expressibility and circuit depth [46]. For the remainder of this thesis, the circuit will be referred to as the VQA circuit.

As previously stated, the second circuit is a circuit constructed from a QAOA for the purpose of approximating the max-cut problem. For this construction, a cost Hamiltonian H_C is needed.
The cost Hamiltonian H_C for the max-cut problem can be defined by taking Equation (16) into account. Here, the binary variable $x_i \in \{+1, -1\}$ is mapped to the Pauli Z operator. The Hamiltonian H_C is therefore defined as

$$H_C = -\frac{1}{2} \sum_{(i,j)\in E} (I - Z_i Z_j) \tag{19}$$

with I representing the identity operator and Z_i is the Pauli Z operator. [21]

Following the proposal in Section 2.6.2, the mixer Hamiltonian H_M is chosen to be

$$H_M = -\sum_{j \in V} X_j, \tag{20}$$

with X_i is the Pauli X operator. As already stated in Section 2.6.2 the initialize state of the circuit can chosen to be $|+\rangle^{\otimes n}$ as this state corresponds to the lowest energy state of the mixer Hamiltonian H_M [21]. As quantum circuits are typically initialized with $|0\rangle^{\otimes n}$ the described state can by achieved by applying a Hadamard gate to each qubit involved. To construct the unitaries for the mixing layers $U_C(\gamma_k)$ and the cost layers $U_M(\beta_k)$ the following transformations can be used:

$$U_C(\gamma_k) = e^{-i\gamma_k H_C} = \prod_{i=1, j<i}^{n} R_{Z_i Z_j}(2\gamma_k), \tag{21}$$

$$U_M(\beta_k) = e^{-i\beta_k H_M} = \prod_{i=1}^{n} R_{X_i}(2\beta_k), \tag{22}$$

with R_{X_i}, $R_{Z_i Z_j}$ being parametrized rotational gates. [21]
The circuit is constructed by placing $U_C(\gamma_k)$ and $U_M(\beta_k)$ back-to-back in multiple layers, as shown in Figure 6. For the purpose of the evaluation, the amount of layers is set to eight. The resulting circuit will be referenced throughout the remainder of this thesis as the QAOA circuit.

The graphs for the Max-cut problem are generated using the *gnm_random_graph* function from the Python library *NetworkX*. The function receives a fixed seed for reproducibility and generates a graph with five vertices and ten edges.

6.3 Description of the Backends

In the evaluation, circuits are compiled to be run on different backends. Namely, these backends are IBMQ-Sherbrooke, Rigetti ANKAA-2, Rigetti ANKAA-9Q-1 and self defined backend with a linear nearest neighbor physical qubit connectivity. Each backend is only defined by its connectivity graph and the gate set.

Backend	#Qubits	Gate Set	C
IMBQ-Sherbrooke	127	$\{ECR, R_Z, SX, X\}$	0.89%
Rigetti ANKAA-2	84	$\{R_X, R_Z, CPhase, CZ, iSWAP\}$	4.27%
Rigetti ANKAA-9Q-1	9	$\{R_X, R_Z, CPhase, CZ, iSWAP\}$	33.33%
Linear-Nearest-Neighbour	5	$\{R_Z, S_X, X, CX\}$	40.00%

Table 2: Properties of the backends used in the evaluation.

The properties of the four backends are presented in Table 2. C describes the connectivity of the qubits in the connectivity graph. It is calculated by $C = N_C/N_{C,max}$, where N_C is the amount of connections between pairs of qubits, and $N_{C,max}$ is the maximal possible number of connections [47]. $N_{C,max}$ is therefore given by $(n(n-1))/2$, where n is the number of qubits.
There are three distinct gate sets. The IBMQ-Sherbrooke has 127 qubits and its gate set

consists of the two-qubit Echoed Cross-Resonance gate, the R_Z gate, the $\sqrt{X}$ gate and the Pauli-X gate. A graphical representation of the connectivity map of this quantum backend can be found in Appendix B.
The two backends from Rigetti share a common gate set. This set includes a R_X gate, a R_Z gate, a CPhase gate, a CZ gate, and an iSWAP gate. For the self defined backend with five qubits, which is referred to as Linear-Nearest-Neighbour backend, the gate set consists of the R_Z gate, the S_X gate, the X gate, and the CX gate. The connectivity graph of this backend has a linear topology and is depicted in Appendix B.

6.4 Description of the General Workflow

The evaluation framework, written in Python 3.11, takes an arbitrary quantum circuit as its input. Subsequently, it compiles these circuits to the already mentioned backends. Finally, the properties, described in Section 6.1, are measured.

As there are certain parameters that must be set for an evaluation run, these can be outsourced to a configuration file, which is referred to as a job configuration file.

```
1  name = VQACircuitEvaluation
2  circuitType = qasm
3  circuit = circuits/circuitQaoa.qasm
4  compilers = qiskit tket bqskit
5  approximationDegree = 1.0 0.9
6  optiLvl = 0 1 2 3
7
8  path = plots/tmp/
9  useUUID = True
10 parameterTypes = bound
11
12 backends = IBMQ127 Rigetti84 Rigetti9 LNN
```

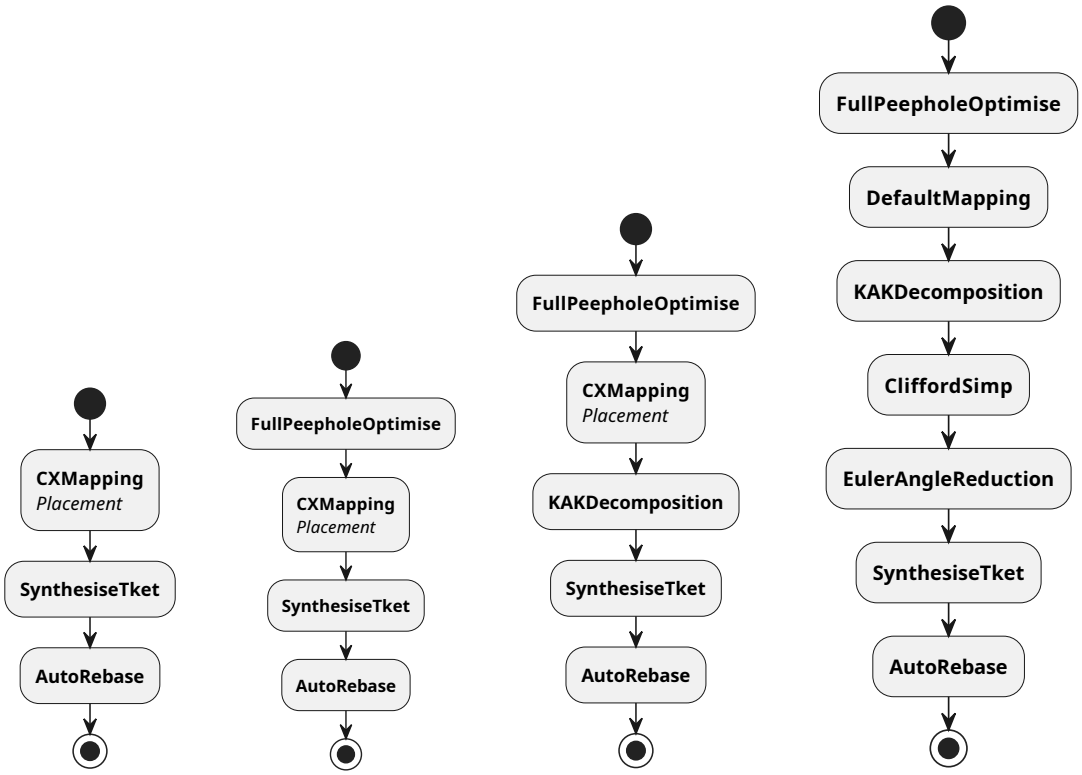
Listing 1: Job configuration file for an evaluation run.

In Listing 1, a job configuration file is illustrated. There are several configuration values to point out here. The *compilers* value specifies on which compilers the evaluation should be run. With the parameters *approximationDegree* and *optiLvl* the approximation degree and the optimization level of the compilers can be set. The values for the parameter approximationDegree may be selected from the interval $[0, 1]$. Similarly, the values for the parameter optiLvl may be chosen from $\{0, 1, 2, 3\}$.
It is important to note that the approximationDegree values as well as the optimization level have been standardized to align with the requirements of Qiskit. This means that an approximation Degree of 0.0 signifies the maximum possible approximation, whereas a degree of 1.0 represents no approximation. In BQSKit the definition is reversed, and therefore is adjusted during runtime. As optimization level, BQSKit accepts values from $\{1, 2, 3, 4\}$, which is therefore also adjusted.
The value *backends* sets the backends on which the circuits should be compiled.

The workflow starts with a job description file, which is depicted in Listing 1. Subsequently, the circuit to evaluate is imported by reading in a text file on the specified path. The circuit files are pre generated and are in the OpenQASM3 format. This format supports free parameters, which makes it comfortable to use with VQA circuits [28]. In the framework, the circuit is represented as a Qiskit circuit object, as this facilitates conversion to the BQSKit and TKET circuit representation. The parameter for the parametrized gates are randomly chosen from the interval $[0, 2\pi]$ and subsequently assigned to the circuit. The circuit is then compiled to each specified backend with each selected compiler and for each combination of the defined parameters. Following the compilation process, each circuit is subjected to a series of checks to ensure that it meets the specified connectivity constraints. Additionally, it is verified that each gate within the circuit is included in the gate set of the corresponding backend. Finally, the resulting circuit properties, defined in Section 6.1 are measured. The compilation time is measured by the difference of the system time before and after the compilation. This is only an estimate of the actual time because other processes may be running on the system. The measurement values, together with the compiled circuits, are stored for future use.

6.5 TKET Compilation Pipeline

As mentioned in Section 5.2 TKET does not provide a general compilation pipeline for self defined backend, but it has defined compilation workflows for various quantum devices. Therefore, a part of this thesis is the design and implementation of a general compilation pipeline for the TKET compiler. The compilation pipeline was designed with the TKET compilation flow for IBM and Rigetti quantum devices as a point of reference [7].



(a) TKET compilation workflow for level 0.

(b) TKET compilation workflow for level 1.

(c) TKET compilation workflow for level 2.

(d) TKET compilation workflow for level 3.

Figure 30: The four distinct compilation workflows for the different levels of optimization in TKET.

As in the case of Qiskit and BQSKit, four distinct optimization levels are implemented, namely 0, 1, 2 and 3. Figure 30 depicts the four different workflows that are generated for these levels of optimization.

The workflow for the first optimization level, illustrated in Figure 30(a), contains only three passes, namely *CXMapping*, *SynthesiseTket*, and *AutoRebase*. The CXMapping pass is responsible for the initial mapping and the routing by inserting CNOT gates. In order to perform this function, it requires a Placement pass to establish an initial mapping, as well as a description of the connectivity graph. In this case, the Placement pass performs the bare minimum by mapping $q_x \mapsto Q_x$ for each logical qubit q_x. Section 5.2.2 describes the routing that takes place in the CXMapping pass.
The SynthesiseTket pass together with the AutoRebase pass translate all gates in the circuits to gates that are included in the gate set of the backend. For this task, first, all two-qubit gates are synthesized to gates from the native gate set. Subsequently, all single-qubit gates are substituted by the three rotation gates $R_Z(\alpha)$, $R_X(\beta)$, $R_Y(\gamma)$. These rotation gates are then replaced by gates from a subset of the native gate set.

For the optimization level 1, the workflow is expanded by the *FullPeepholeOptimise* pass. This pass is described in depth in Section 5.2.3. It is important that this pass is executed before the routing, as the FullPeepholeOptimise pass can introduce new multi-qubit gates.

The compilation pipeline for level 2, depicted in Figure 30(c), adds the *KAKDecomposition* pass after the CXMapping pass. This pass, designed for optimization, performs a KAK decomposition on a chain of gates, if the result has a lower gate count. It is configured to not be able to introduce new multi-qubit gates. Consequently, this pass can be implemented subsequent to the mapping and routing processes.

The compilation workflow for the last optimization level, namely level 3, is depicted in Figure 30(d). This workflow introduces three new passes, namely the *DefaultMapping* pass, the *CliffordSimp* pass and the *EulerAngleReduction*. The DefaultMapping pass replaces the CXMapping pass, that is used in the first three workflows. In contrast to the naive initial mapping approach utilized by the CXMapping pass, the DefaultMapping pass employs the graph placement algorithm that is described in Section 5.2.1. This method results in a longer computation time, yet it may potentially result in a reduced gate count due to the potential for fewer SWAP operations introduced by the routing method. The CliffordSimp optimization pass, performs a series of rewrite rules with the objective of simplifying Clifford gate sequences. This procedure is similar to the method described in Section 5.2.3. The other added optimization pass, namely the EulerAngleReduction pass, employs the Euler angle decomposition to shorten sequences of P and Q rotations, where $P, Q \in \{R_X, R_Y, R_Z\}$. Chains of $P - Q$ and $Q - P$ are decomposed to $P - Q - P$ triples. In this workflow, P is set to R_Y and Q is set to R_Z as sequences of these gates may appear in the VQA circuit. Both optimization passes are configured to prohibit the introduction of new two-qubit gates, which allows them to be placed subsequent to the routing pass.

It can be assumed that the four workflows will result in improved circuit properties, specifically in terms of the number of gates and circuit depth, as the degree of optimization increases.

6.6 BQSKit Echoed Cross-Resonance Gate

In order to compile a circuit to a given backend, a compiler needs the backend's connectivity graph and its gate set. All gates from the backends native gate set must be familiar to the compiler.

Since BQSKit has its first release in October 2021, it is still quite new [8]. Therefore, the toolkit is only aware of a limited amount of quantum gates.

As mentioned in Section 6.3, the IBMQ-Sherbrooke consists of the ECR gate, the R_Z gate, the $\sqrt{X}$ gate and the X gate. BQSKit is familiar with the R_Z gate, the $\sqrt{X}$ gate and the Pauli-X gate. However, it does not support the ECR gate. Such a support is important so that BQSKit is able to produce circuits that only have the ECR gate as its two-qubit gate. There is already an open request in their open-source community for the support of the ECR gate, but it is not implemented yet.

The Echoed Cross-Resonance gate is a two-qubit gate that implements $\frac{1}{\sqrt{2}}(IX - XY)$ [48].

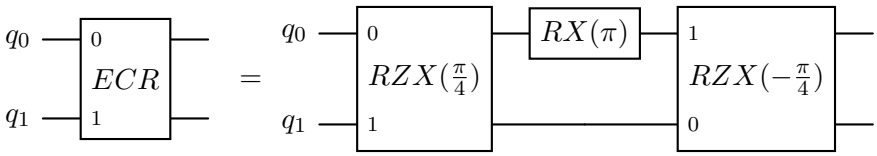


Figure 31: Decomposition of the Echoed Cross-Resonance gate. The Echoed Cross-Resonance gate can be decompose into two RZX gates and a single RX gate.
Source: Adapted from [48]

Figure 31 shows a possible decomposition of the Echoed Cross-Resonance gate into two RZX gates and a single RX gate with fixed parameters. The unitary matrix representation of the ECR gate can be written as

$$ECR_{q_0,q_1} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 1 & 0 & i \\ 1 & 0 & -i & 0 \\ 0 & i & 0 & 1 \\ -i & 0 & 1 & 0 \end{bmatrix}. \tag{23}$$

It is important to note, that the matrix representation in Equation (23) as well as the decomposition suggestion, depicted in Figure 31, are constructed under the convention, that higher qubit indices are more significant, also known as little-endian convention [48].

A definition of a new gate in BQSKit is composed of two parts, the unitary matrix of the gate, which is already defined in Equation (23), and its OpenQASM definition. For the unitary matrix the Equation (23) can be used. The OpenQASM definition can be derived from the Figure 31. As BQSKit also uses the little-endian ordering according to the maintainers, the decomposition and the unitary matrix representation do not need to be adjusted.

```
1  gate rzx(param0) q0,q1 {
2      h q1;
3      cx q0,q1;
4      rz(param0) q1;
5      cx q0,q1;
6      h q1;
7  }
8  gate ecr q0,q1 {
9      rzx(pi/4) q0,q1;
10     x q0;
11     rzx(-pi/4) q0,q1;
12 }
```

Listing 2: An OpenQASM definition for the ECR gate.

The OpenQASM definition for the ECR gate, together with the RZX gate, is depicted in Listing 2. It is important to also define the RZX gate, as this gate is not in the standard gate library of OpenQASM [28]. This definition together with the unitary matrix makes up the gate definition for the ECR gate in BQSKit.

In order to contribute the support for the ECR gate in the BQSKit toolkit, a pull request has been created, which is currently under review at the time of submission of this thesis.

6.7 Naive Compilation Pipeline

In order to provide a more comprehensive illustration of the quality of the individual compilers under evaluation, a compilation pipeline is constructed that performs the minimum necessary operations. Such a compiler helps to visualize how the individual workflows of the other compilers optimize the circuit. The compilation pipeline, referred to as naive compiler, uses different passes from the Qiskit framework and is depicted in Figure 32.

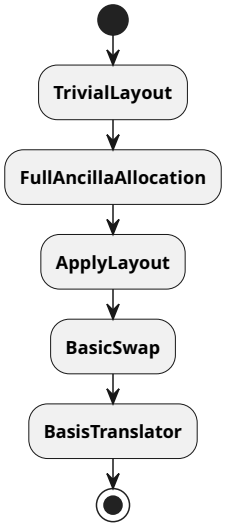


Figure 32: Compilation workflow of the naive compiler.

These compilation passes from Qiskit that form the pipeline of the naive compiler are the TrivialLayout pass, the FullAncillaAllocation pass, the ApplyLayout pass, the BasicSwap pass and the BasisTranslator pass. The TrivialLayout is also used for the Qiskit compilation workflows for the first two optimization levels and is therefore described in Section 5.3.
Both, the FullAncillaAllocation pass and the ApplyLayout pass ensure that the layout which is identified by the TrivialLayout pass is applied to the circuit.
The BasicSwap pass ensures that the connectivity constraint of the backend is fulfilled by introducing SWAP operations into the circuit. To accomplish this, the pass searches for two-qubit operations in the circuit. If a found operation cannot be executed due to the connectivity graph, the pass calculates the shortest path from the physical target qubit to the physical control qubit. Subsequently, SWAP operations are inserted along this path.
In order to ensure that all gates in the circuit are also in the gate set of the backend, the compilation pipeline contains the BasisTranslator pass. This pass is also used by the default compilation workflow in Qiskit throughout all its optimization levels and is therefore described in Section 5.3. [3]

This workflow produces circuits that are compatible with a specified backend without performing any optimizations.

7 Evaluation of the Quantum Compilers

This chapter presents the outcome of the evaluation and offers potential explanations for them.
Section 7.1 provides a comparison of the levels of optimization for each compiler separately. This is extended in Section 7.2, here the compilers are compared directly with each other. The Section 7.3 evaluates the time that the compilers take for their task. In Section 7.4 the effects of the approximation degree, a parameter in Qiskit and BQSKit, are evaluated.

7.1 Optimization Levels Comparison

All compilers have the potential to produce circuits with fewer gate count and less circuit depth with a rising optimization level. In order to test this hypothesis, all three compilers were instructed to compile the two prepared test circuits to the four introduced backends for all four levels of optimization. All other parameters for the compilation process are set to their default values.

The plots presented in this chapter, visualize the compilation and optimization behavior of the individual compilers. For each compiler and circuit, there are three distinct plots. These plots illustrate the circuit depth, the number of one-qubit gates, and the amount of two-qubit gates, for the circuit after the compilation process.
On the x-axis, the four optimization levels are presented for each backend to which the circuit is compiled. The backends are labeled as *LNN* for the self defined Linear-Nearest-Neighbour, *IBMQ128* for the IBMQ-Sherbrooke and *Rigetti9*, and *Rigetti84* for the backends from Rigetti with nine and 84 qubits respectively.

A black solid line represents the circuit properties which the circuit posses before the compilation process. Although it is possible that a compiler could produce a circuit with fewer gates due to the optimization, compiling a circuit to a backend with a finite gate set, and a connectivity graph will often result in an equal or increased number of gates.

The orange dashed line illustrates the circuit values from the circuit that the naive compilation pipeline, described in Section 6.7, has compiled.

7.1.1 Qiskit Optimization Levels Comparison

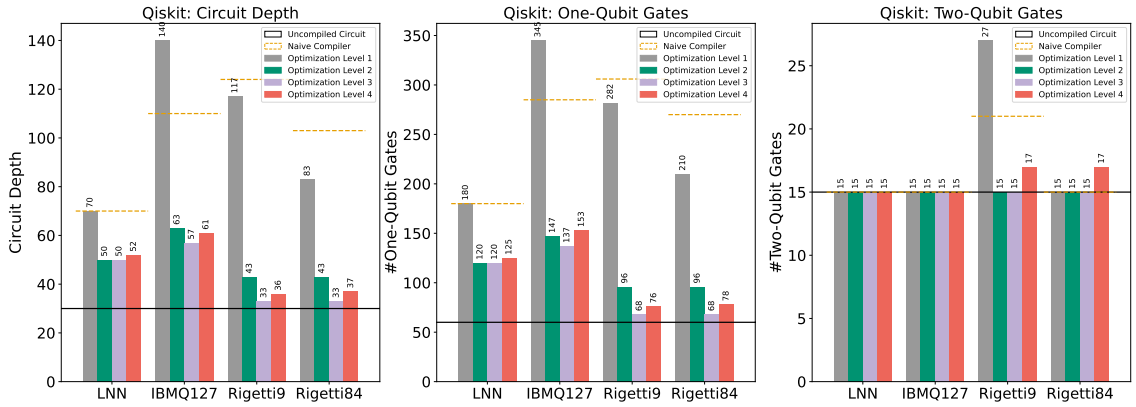


Figure 33: Comparison of the four optimization levels in Qiskit for the VQA circuit.

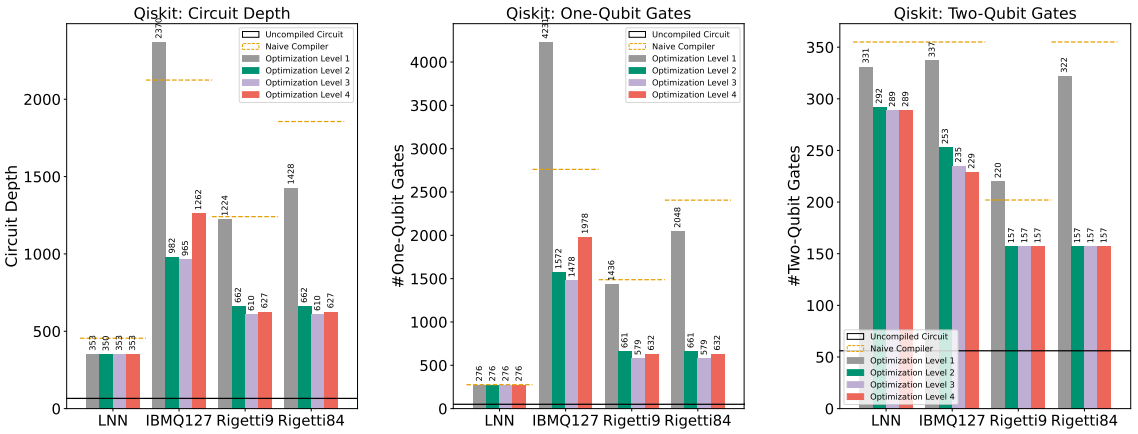


Figure 34: Comparison of the four optimization levels in Qiskit for the QAOA circuit.

Figure 33 depicts the plot that compares the circuit depth and the amount of one-qubit and two-qubit gates of the compiled VQA circuit.
It can be observed that the optimization 0 indeed produces the highest circuit properties. In certain instances the circuit properties, specifically the circuit depth and the number of one-qubit gates of the generated circuit for IBMQ127 and the two-qubit gates of the circuit for Rigetti9, are even higher than those of the circuit produced by the naive compiler. This can be an indicator that the StochasticSwap pass from the workflow for optimization level 0 in Qiskit introduces more SWAP operations than the BasicSwap pass of the naive compiler. Both passes are required to add SWAP operations, as the TrivialLayout does not provide a layout that satisfies the connectivity constraint for Rigetti9.
The compilation workflows for level 1 and 2 produce similar circuit properties, although optimization level 2 tends to produce more favorable results. This could be due to the slightly improved optimization loop present in this level.
The process of compiling a circuit at level 3 often results in the creation of properties that are inferior to those produced by the compiler at optimization level 2. Once more, this discrepancy may be attributed to the different optimization loops, as this is the only significant difference between these two levels.

All workflows introduce a significantly greater number of gates during the compilation of the QAOA circuit, in comparison to the VQA circuit. This can be due to the fact, that the QAOA circuit does not have a linear topology regarding the two-qubit gates like the VQA circuit. As all of the available four backends have a subgraph that also has a linear topology, adding SWAP operations is not mandatory while compiling the VQA circuit. However, as the topology, that the two-qubit operations span in the QAOA circuit, has a more arbitrary shape, the compiler has to add additional SWAP operations in order to satisfy the connectivity constraint of the backend.
The plot for the circuit properties of the compiled QAOA circuit, depicted in Figure 34, shows a very similar result. Similarly, the workflow for optimization level 0 produces the least favorable circuit properties, which may be attributed to the previously outlined reasons.
The workflows for optimizations 1 and 2 result again in comparable circuit properties, with the latter workflow yielding to slightly better outcomes.
The workflow for optimization level 3 also produces slightly less favorable results of the QAOA circuit than the workflow for optimization level 2.

Note that the results for optimization level 3 can be improved by adjusting the approximation degree parameter. This adjusting is discussed in Section 7.4.

7.1.2 TKET Optimization Levels Comparison

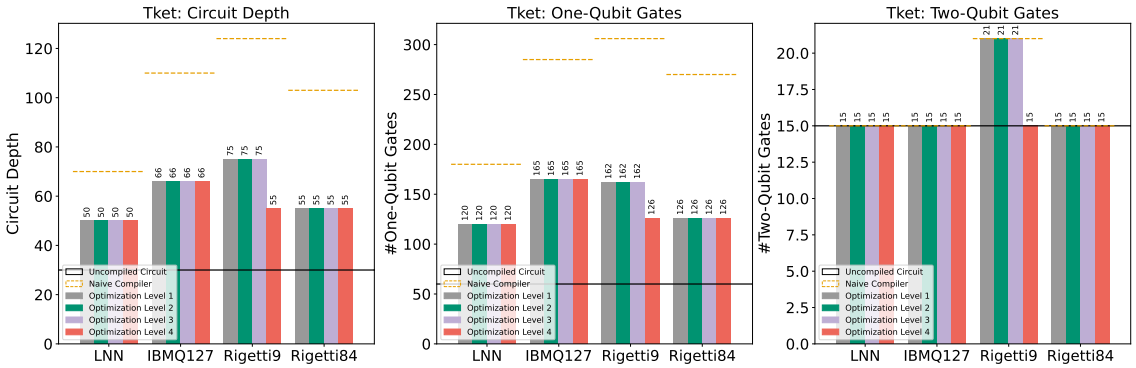


Figure 35: Comparison of the four optimization levels in TKET for the VQA circuit.

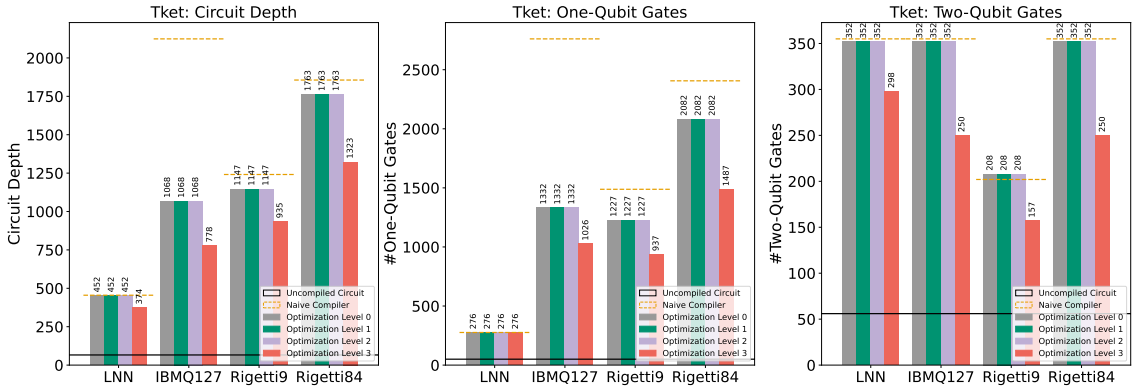


Figure 36: Comparison of the four optimization levels in TKET for the QAOA circuit.

The plots, depicted in Figure 35 and Figure 36, respectively, compare the optimization levels based on the circuit characteristics for the VQA circuit and the QAOA circuit that are compiled by TKET.
Like in the plots in the previous chapter, a large difference can be seen in the added gates between the QAOA circuit and the VQA circuit. This is most likely due to the same reason that is described above.

For both circuit types, the first three optimization levels produce the same circuit attributes. This is an indicator that the optimization passes that are part of the compilation workflows cannot remove any gates. For instance, the FullPeepholeOptimise algorithm is designed to identify and replace known circuit structures. A similar approach is employed by the CliffordSimp pass and the EulerAngleReduction. Obviously, no such structures can be identified by TKET to replace.
The TKET workflow for optimization level 3 is able to produce circuits with better properties in some cases. Given that this occurs in the case of the VQA circuit compiled for the Rigetti9 backend, it is reasonable to assume that this improvement is triggered by the graph placement algorithm. This statement is supported by the two-qubit gate count in

the compiled QAOA circuit for all four backends. As previously stated, the compiler is required to add more SWAP operations, as it is constrained by the design of the circuit. An advanced mapping approach, such as the graph placement algorithm, could potentially reduce the necessity for these operations.

7.1.3 BQSKit Optimization Levels Comparison

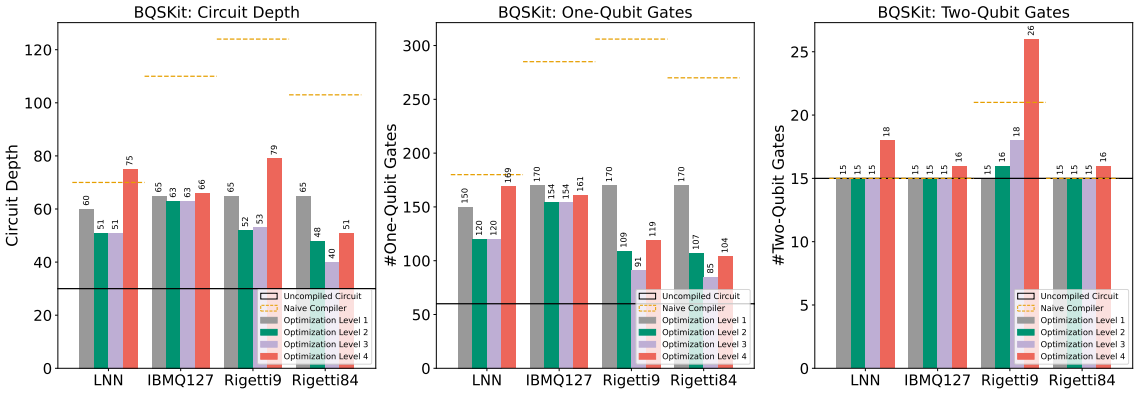


Figure 37: Comparison of the four optimization levels in BQSKit for the VQA circuit.

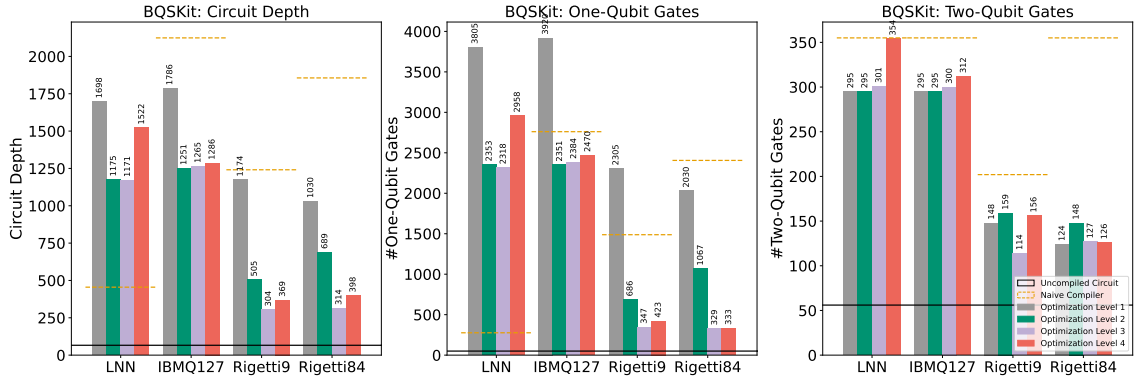


Figure 38: Comparison of the four optimization levels in BQSKit for the QAOA circuit.

The diagrams shown in Figure 37 and Figure 38 illustrate the comparison of the four optimization levels based on the circuit properties for the VQA circuit and the QAOA circuit, respectively, both compiled for the four backends by BQSKit.
The discrepancy between the circuit types regarding their properties are also present when compiling with BQSKit.

It can be observed that circuits that have been compiled with BQSKit do not necessarily have better properties with increasing optimization level. In the majority of cases, the first level of optimization results in worse circuit characteristics, especially the amount of one-qubit gates and the consequential increase in the circuit depth, for both the VQA circuit and the QAOA circuit in comparison to the following levels. One potential explanation for this behavior could be the absence of the gate_deletion_optimization_workflow in the pipeline for the first optimization level.
The workflow for the fourth optimization level also produces VQA and QAOA circuits with an increased number of gates, or in some cases only as many gates as the workflows

for optimization levels 2 and 3. An explanation for this phenomenon may be found in the differing mapping algorithms employed by the workflows, namely the SABRE and the PAS algorithm.

Interestingly, the amount of one-qubit gates, and consequently the circuit depth, of the QAOA circuit compiled for the LNN backend exceeds that of the naive compiler by orders of magnitude. This could be due to the fact, that the QSearch and LEAP algorithm, used by workflows for all optimization levels, synthesize the circuit by introducing gates from the gate set in layers to an empty circuit. The translation of gates, with the help of an equivalence library, seems to produce better solutions in this case in terms of the number of one-qubit gates. Furthermore, the gates, particularly the R_{ZZ} used in the QAOA circuit, can be effectively substituted by this method through the use of R_Z and CNOT gates. Both gates are part of the native gate set of the LNN backend.

7.2 Compiler Comparison

In order to compare the different compilers, the QAOA circuit is compiled for the IBMQ127 and the Rigetti9 backend with all four optimization levels. The backends are selected based on the observation that the QAOA circuits, produced for Rigetti9 and Rigetti84 have similar circuit properties when compiled by the three compilers. The QAOA circuits, compiled for the LNN backend, have a strong deviation when compiled with BQSKit. The possible reason for this is described in Section 7.1.3.
The following plots visualize the differences between the four compilers regarding the properties of their compiled circuits. On the x-axis, the four distinct optimization levels are presented. It is important to note that optimization levels for BQSKit are increased by one to maintain a uniform presentation. The bars, each color represents a different compiler, show the circuit depth and the amount of one-qubit and two-qubit gates respectively. Additionally, the bars are grouped by the four optimization levels.

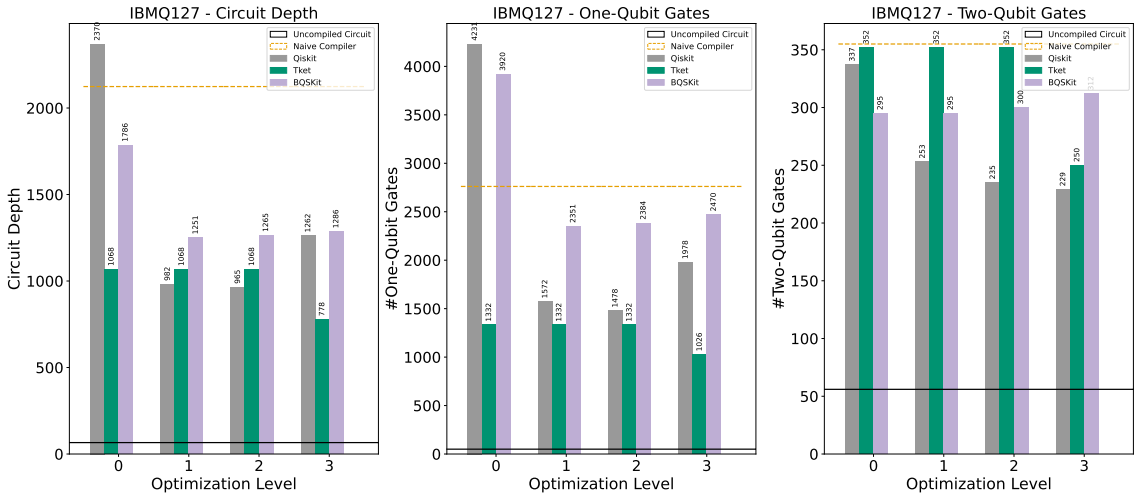


Figure 39: Comparison of the three compilers based on the compiled QAOA circuit for the IBMQ127 backend.

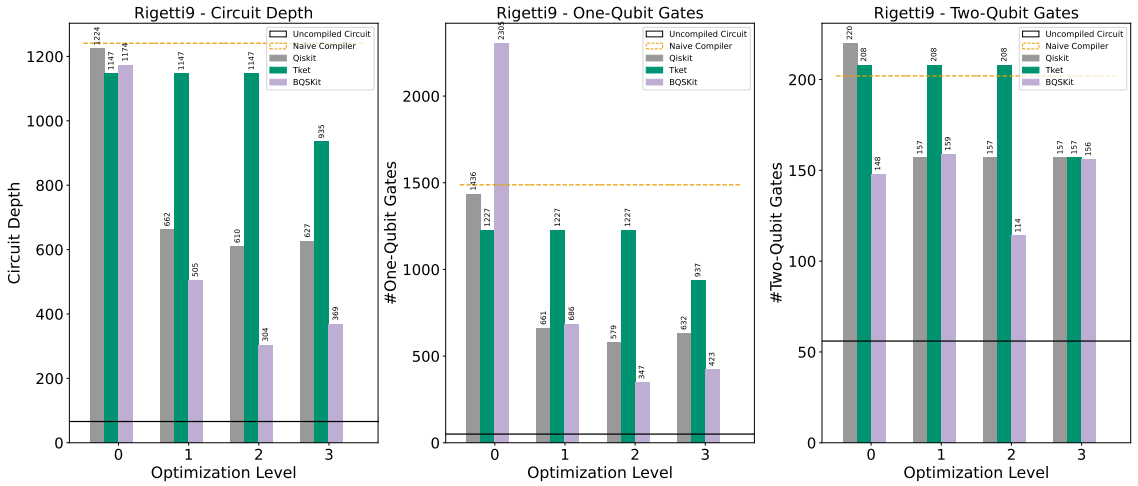


Figure 40: Comparison of the three compilers based on the compiled QAOA circuit for the Rigetti9 backend.

Figure 39 and Figure 40 depict the differences in the produced circuits of the compilers for each optimization level on the backends IBMQ127 and Rigetti9 respectively.
It can be seen that, for both backends, there is no clear preference for a particular compiler over the other compilers for this circuit. As each compiler has advantages and disadvantages regarding the circuit properties associated with each backend.
For the IBMQ127 backend, it appears that the circuits produced by Qiskit and TKET are more preferable in terms of circuit depth and one-qubit gate count. However, the characteristics of the circuits produced by BQSKit seem to be better when compiled for the Rigetti9 backend. At least for the last three levels of optimization.

It appears that the selection of the backend on which the quantum application is to be executed is at least as important as the choice of the compiler itself.

7.3 Compile Time Comparison

For a comparison of the time the compilation process takes, the compilation duration of each compiler is measured for each optimization level. Like mentioned in Section 6.4 the time is measured by a difference of the system times. Therefore, this is only an estimation of the actual compile time, especially for short periods. As the results for the VQA circuit and the QAOA circuits are similar, this discussion will focus on the time required for the QAOA circuit to be compiled.

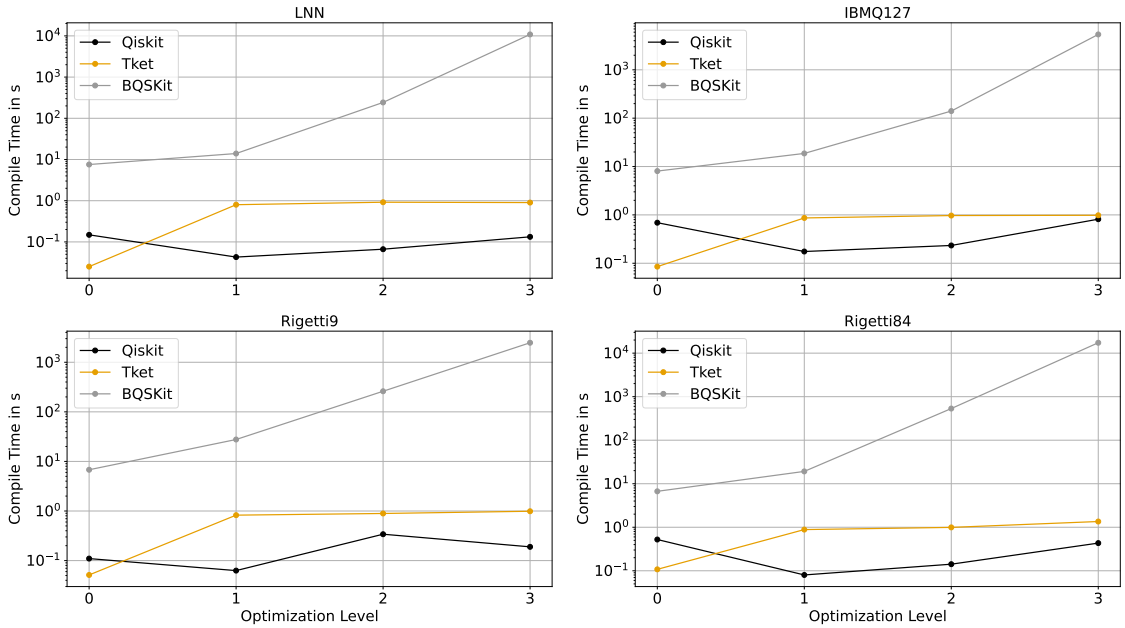


Figure 41: Comparison of the compile times for the QAOA circuit.

Figure 41 depicts the time the compilers need, to compile the QAOA circuit onto the four different backends. Each plot represents a different backend, with the lines representing the individual compilers. The optimization levels are x-axis values, while the time in seconds is y-axis values. It is important to note that the values on the y-axis are expressed by a logarithmic scale.

TKET and Qiskit posses similar compile times. It appears that Qiskit requires a longer compilation time when the optimization level is set to 0. This could be an indicator that the StochasticSwap pass requires more time than the routing approaches employed in other optimization levels.
The compilation times for the optimization levels 1, 2, and 3 in TKET are nearly identical. The duration for the first level of optimization is significantly shorter. This could be due to the FullPeepholeOptimise pass, that is only absent in this level.
The compilation duration in BQSKit posses almost an exponential increase with increasing level of optimization. This may be due to the additional synthesis passes or to the parameters for QSearch and LEAP, which change with each optimization level. The last optimization level also makes use of the PAS algorithm, which also probably leads to a longer computational time, as it checks all available permutations.

7.4 Effects of the Approximation Degree

In order to evaluate the behavior of compilers when the approximation degree is changed, the QAOA circuit is compiled for the IBMQ127 backend. As the TKET compilation pipeline, introduced in Section 6.5, does not provide an approximation degree, this part of the evaluation is focused on the Qiskit and BQSKit compiler. The circuits have been compiled by Qiskit with an optimization level of three and by BQSKit with the second level of optimization. The backend and circuit are chosen in a representative way, as the compiler's behavior is similar for other circuits and backends. The line in the plot, depicted in Figure 42, represent the circuit properties of the produced circuits with varying

approximation degree. On the x-axis the value of the approximation degree is placed. Both plots follow the Qiskit convention that an approximation degree of one equates to no approximation, while zero represents maximal approximation.

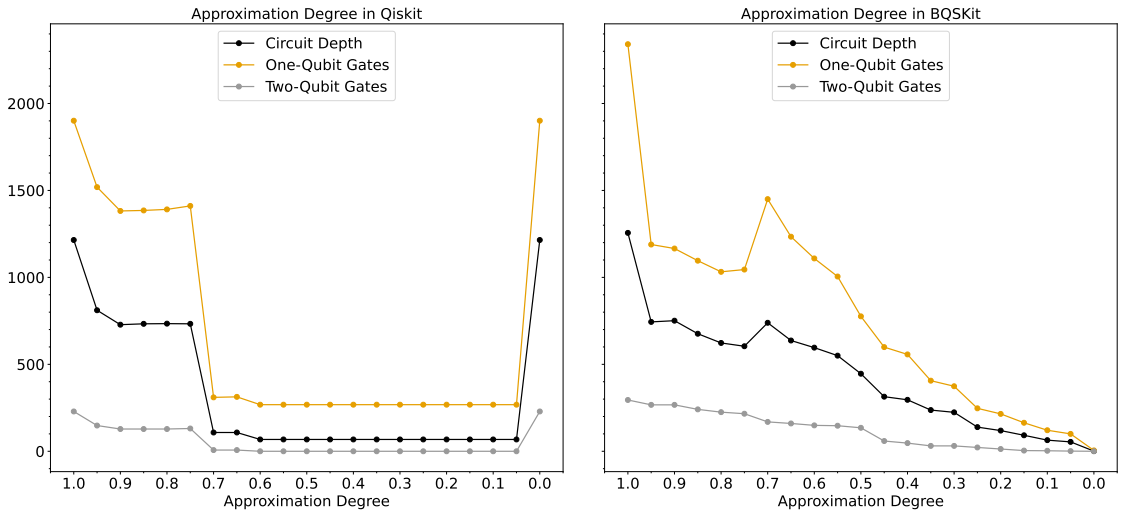


Figure 42: Evaluation of the Approximation Degree. The QAOA is compiled for the IBMQ127 backend with the Qiskit and the BQSKit compiler.

The plots, illustrated in Figure 42, show the circuit properties of the resulting circuits from the Qiskit and BQSKit compilation workflow.
The progression of the one-qubit and two-qubit counts from the circuits produced by Qiskit show a comparable pattern. The gate count initially falls rapidly in the interval between 1.0 and 0.9 degrees of approximation. After this decline, the circuit properties remain steady from 0.9 to 0.75. Between 0.7 and 0.05 the amount of gates reaches a minimum. In this range, there are almost none two-qubit gates in the circuit. It is noteworthy that, with an approximation degree of 0.0 the number of gates in the circuit increases until it reaches the same level as without approximation.
In BQSKit the amount of one-qubit gates in the resulting QAOA circuit decreases rapidly in between 1.0 and 0.9. Besides this fall, the gate count of the one-qubit gates and two-qubit gates in the circuit further declines linearly until the circuit contains no more gates at an approximation degree of 0.0.

It can be observed that BQSKit already can compensate for its relative weaknesses in comparison to Qiskit with a small degree of approximation. As the circuit posses less gates, up to zero, with more approximation, it can be assumed that the circuit's fidelity also decreases. However, in the present of noise, a reduction in the number of gates could result in a lower overall error rate.

In order to test this hypothesis, the QAOA, constructed in Section 6.2, is executed by searching for parameters that minimize the expected energy of the cost Hamiltonian H_C. For the minimization, the SciPy's COBYLA optimizer is used. The backend on which the circuit is run with 1024 shots, posses a noise model provided by Qiskit that simulates the noise present at the backend [3]. To generate a solution to the Max-cut problem, the circuit is executed one final time with these parameters. The histograms depicting this solution can be found in Appendix C. In the histogram, every digit in the bit-strings represents one vertex in the graph. The two values zero and one defines the partition to

which the node belongs.

The plots, depicted in Figure 43, show the expectation value per iteration of the optimizer. Each plot was produced using the Qiskit compiler with optimization level 3 and an approximation degree $d \in \{1.0, 0.95, 0.9, 0.85, ..., 0.5\}$. It is important to note that the y-axis which represents the expectation value has a logarithmic scale.

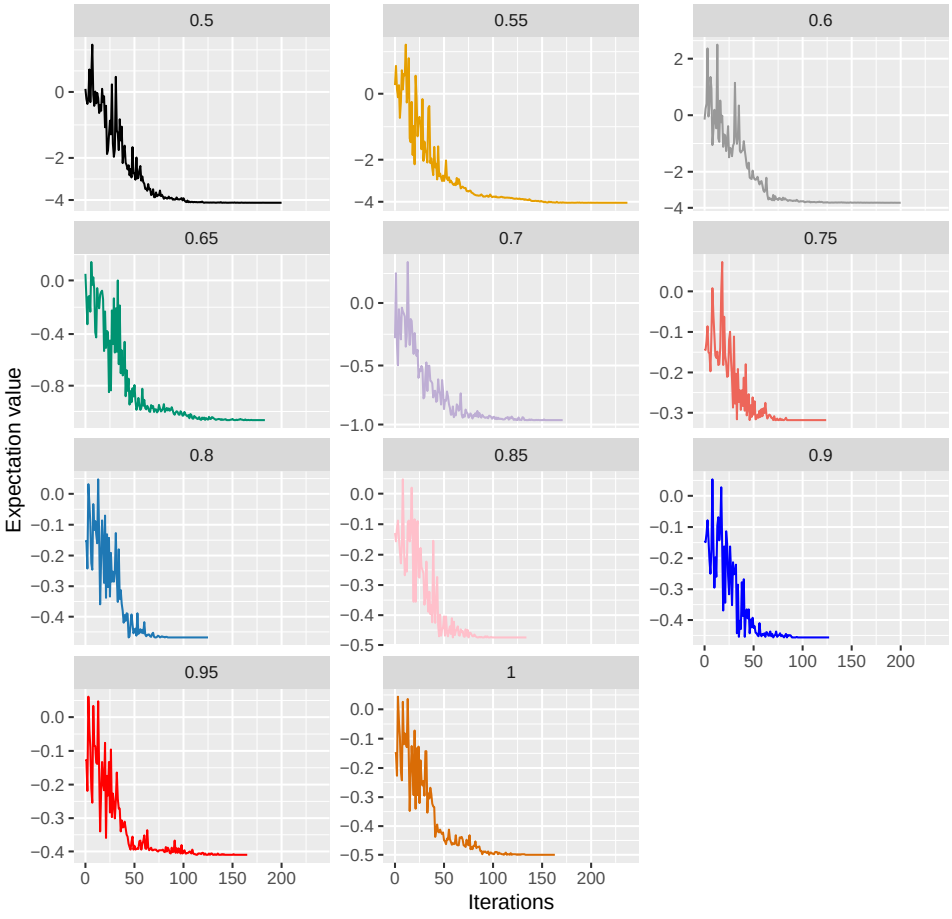


Figure 43: Evaluation of the Approximation Degree. The plot depicts the optimization process of the parameters of the QAOA circuit compiled by Qiskit with varying degrees of approximation.

It can be observed that the minimal expectation value remains relatively consistent up to an approximation degree of 0.75. This behavior also matches the situation in Figure 42. With a further decreasing approximation degree, the minimal expectation value sinks until it reaches a limit at a degree of 0.6.
It appears that with a further approximation degree, the optimizer is capable of finding parameters that result in an expected value that is significantly reduced. However, when examining the histograms generated using these parameters, that the results do appear to be invalid for the Max-cut problem. This is because a valid solution bit-string for the Max-cut problem should always have a second bit-string that is inverse.
Despite the existence of noise, the variation in the approximation degree within the interval 1.0 to 0.75 does not appear to have a significant impact on the derived solutions. This may be attributed to the limited size of the QAOA, made up of only five qubits. Consequently, the circuit may be too small to be able to detect a difference.

8 Conclusion

To conclude this thesis, this section provides an insight into possible future research areas, as well as a summary of the topics and findings that are covered in this thesis.

8.1 Future Works

During the analysis and evaluation of the three compilers, some points were noticed that are worth investigating in the future.
As the separate stages, namely mapping, routing, optimization, and translation, can almost be considered independent of one another, the evaluation should be extended by examining each stage in isolation. This could help to identify favorable approaches for each stage, leading to the development of a more effective workflow.
As observed in the evaluation, the selection of the backend can be more important than the choice of the optimization level. It would be beneficial to investigate which coupling map and which gate set are more favorable for which circuit.
During the evaluation, it was noticed that the optimization stages of the different compilers can produce circuits with better properties when the parameters of parametrized gates are assigned, as opposed to being unassigned. This may prove beneficial in a hybrid classical-quantum loop in the context of a VQA, as it offers the possibility of producing circuits that are better optimized and thus more resistant to noise.
All these research opportunities can contribute to the development of better compilation techniques, which would lead to circuits that are more resistant to noise.

8.2 Summary

This thesis provided an overview of the limitations of current quantum computers, and presents techniques for abstracting and overcoming these limitations.
The in-depth analysis of BQSKit, TKET, and Qiskit describes the workflow of each level of optimization in detail. This analysis also serves to find possible reasons for the behavior of the compilers during the evaluation. The evaluation framework, that was developed during this thesis, enables a fair comparison of the compilers by providing them with the same circuit to be compiled on defined backends. In this development, a contribution was made to BQSKit by providing a gate definition that was not included yet. The evaluation pointed out that the highest level of optimization does not necessary lead to the most favorable circuit properties. In most cases, the third level performed better in terms of circuit depth and gate count. Qiskit and BQSKit can further reduce the number of gates, with a greater degree of approximation. However, this can lead to a high infidelity of the circuit. It is not possible to make a general recommendation for a specific compiler, as each of them has its weaknesses on different backends.

Effective compilation and especially optimization is paramount in the NISQ era, as hardware limitations and error rates present significant challenges. Both processes are of particular importance for enabling more accurate and efficient execution of quantum algorithms, despite the constraints of current quantum technology.

Appendices

A Flowchart of Qiskit's Scheduling Stage

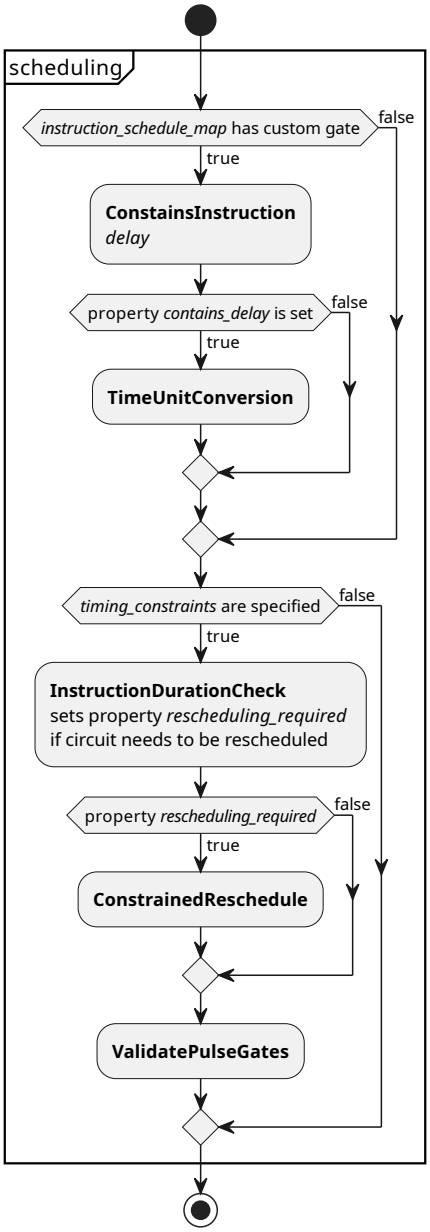


Figure 44: Scheduling stage of Qiskit

B Connectivity Graphs of Backends

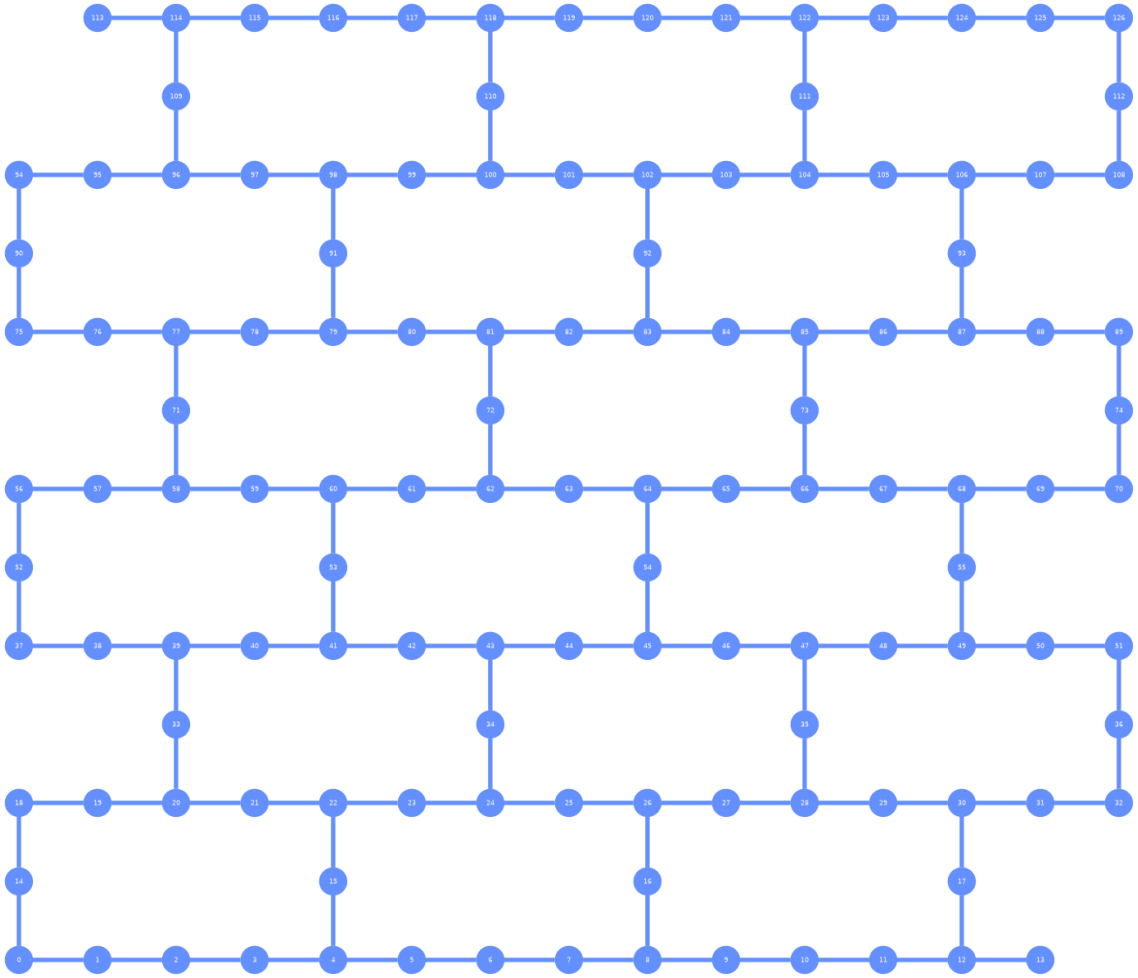


Figure 45: Connectivity graph of IBMQ-Sherbrooke

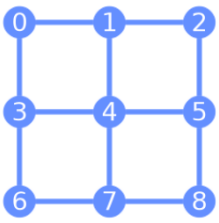


Figure 46: Connectivity graph of Rigetti ANKAA-9Q-1

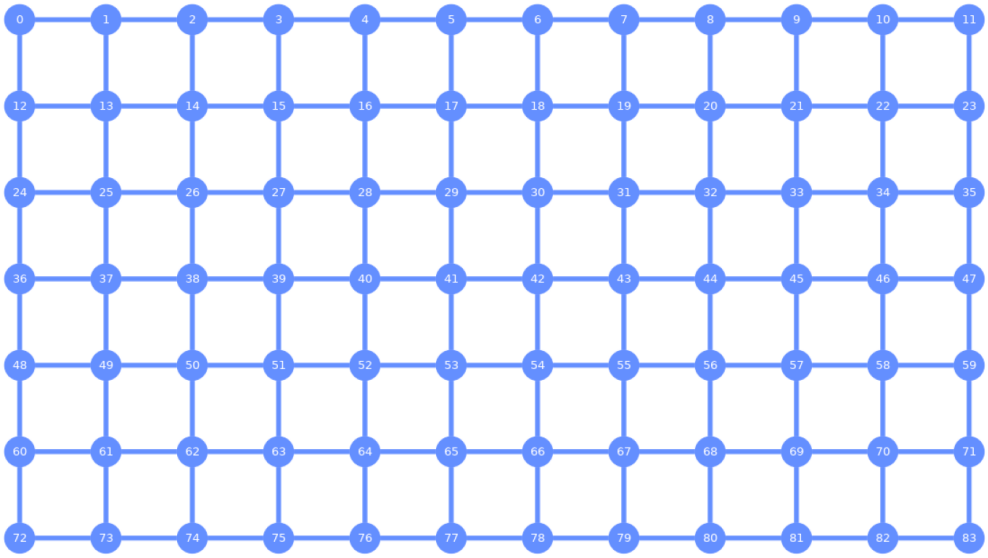
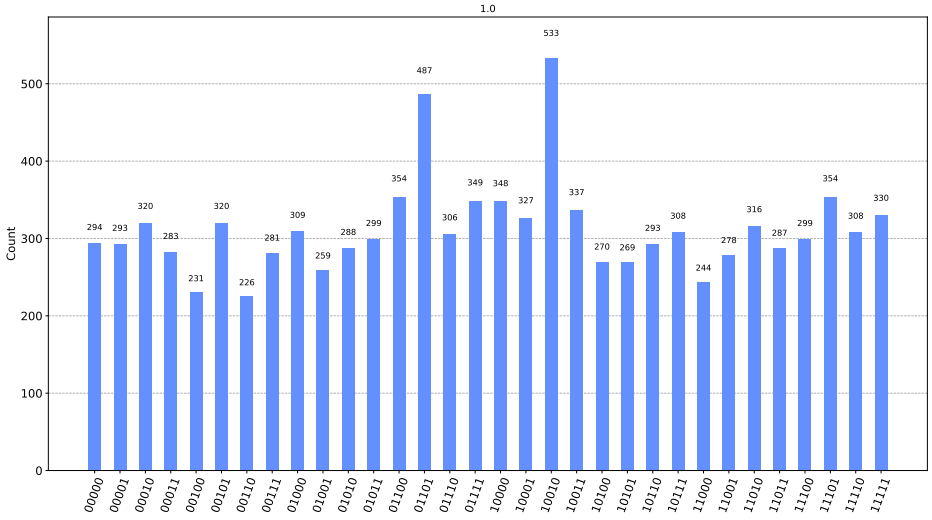


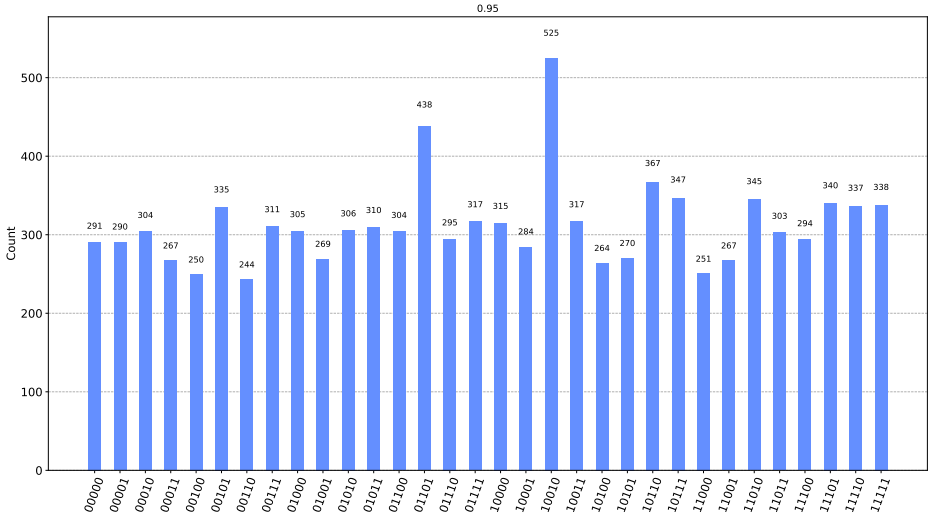
Figure 47: Connectivity graph of Rigetti ANKAA-2

Figure 48: Connectivity graph of Linear-Nearest-Neighbour

C Results of the Quantum Approximation Optimization Algorithm

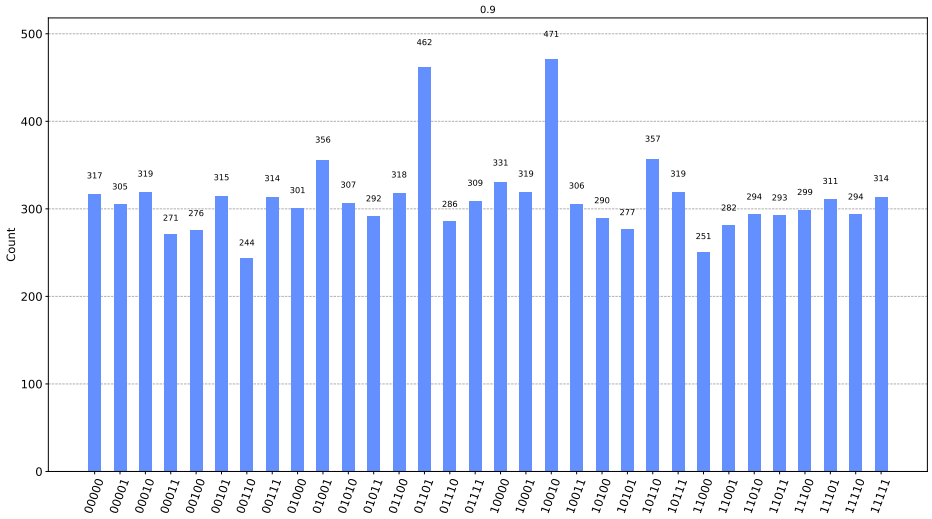


(a) Histogram encoding the solution to the Max-cut problem with approximation degree 1.0

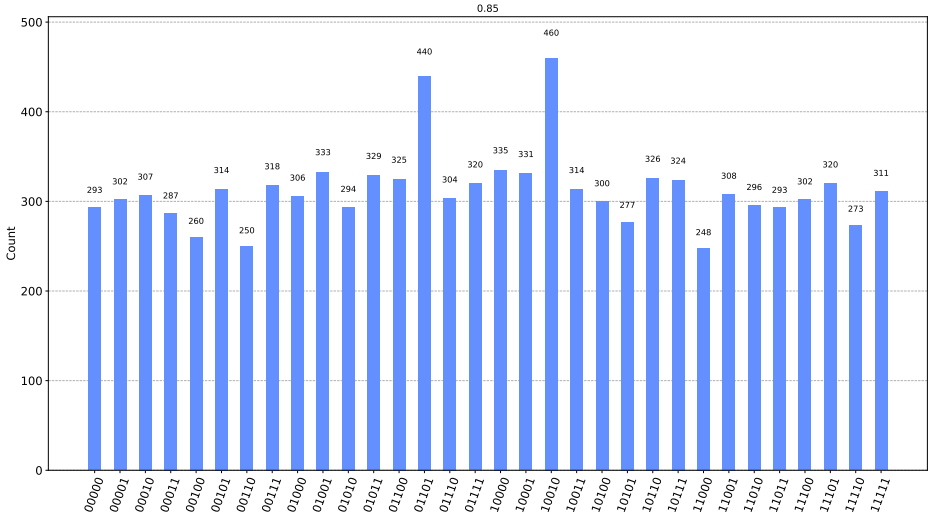


(b) Histogram encoding the solution to the Max-cut problem with approximation degree 0.95

Figure 49: Histograms encoding the solution to the Max-cut problem with different approximation degrees

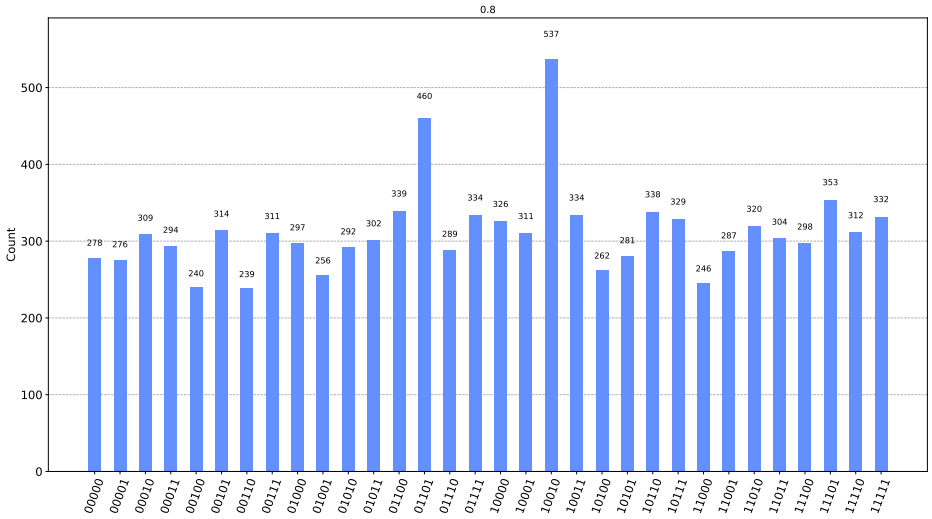


(a) Histogram encoding the solution to the Max-cut problem with approximation degree 0.90

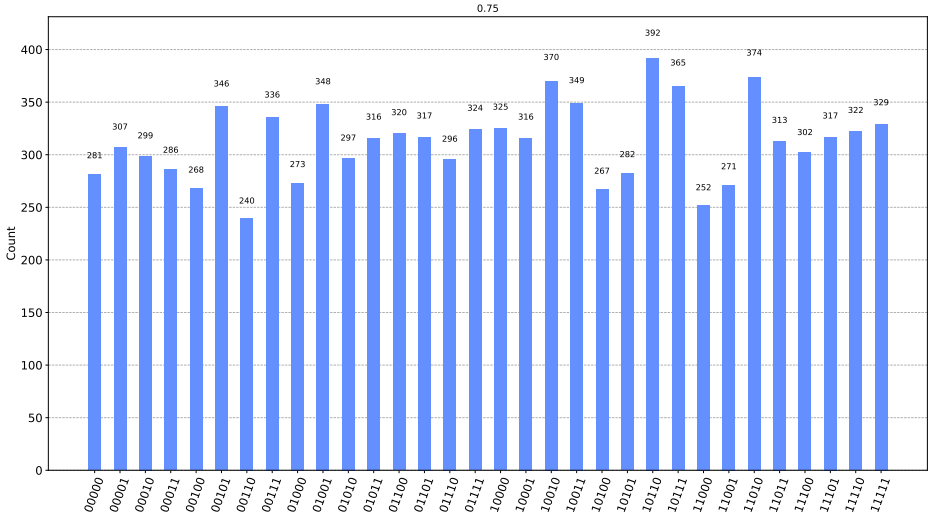


(b) Histogram encoding the solution to the Max-cut problem with approximation degree 0.85

Figure 50: Histograms encoding the solution to the Max-cut problem with different approximation degrees

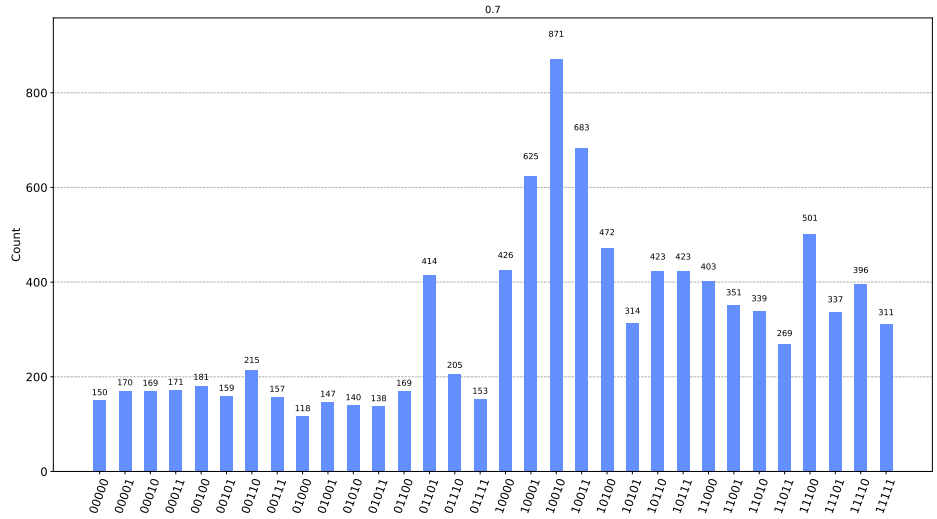


(a) Histogram encoding the solution to the Max-cut problem with approximation degree 0.80

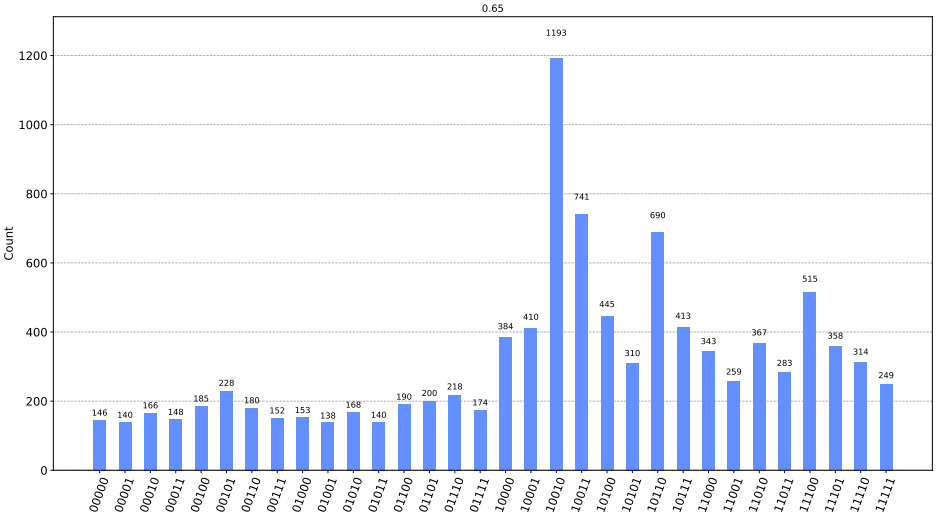


(b) Histogram encoding the solution to the Max-cut problem with approximation degree 0.75

Figure 51: Histograms encoding the solution to the Max-cut problem with different approximation degrees

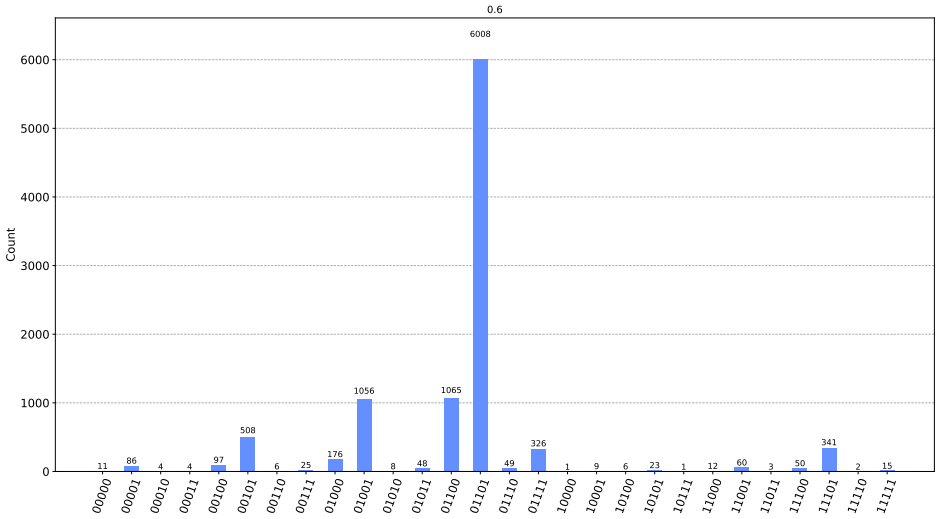


(a) Histogram encoding the solution to the Max-cut problem with approximation degree 0.70



(b) Histogram encoding the solution to the Max-cut problem with approximation degree 0.65

Figure 52: Histograms encoding the solution to the Max-cut problem with different approximation degrees



(a) Histogram encoding the solution to the Max-cut problem with approximation degree 0.60

Figure 53: Histograms encoding the solution to the Max-cut problem with different approximation degrees

References

[1] John Preskill. "Quantum computing 40 years later". In: *Feynman Lectures on Computation*. CRC Press, 2023, pp. 193–244.

[2] P.W. Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, pp. 124–134. DOI: 10.1109/SFCS.1994.365700.

[3] Qiskit contributors. *Qiskit: An Open-source Framework for Quantum Computing*. 2023. DOI: 10.5281/zenodo.2573505.

[4] Cirq Developers. *Cirq*. Version v1.3.0. Dec. 2023. DOI: 10.5281/zenodo.10247207. URL: https://doi.org/10.5281/zenodo.10247207.

[5] Robert S. Smith, Michael J. Curtis, and William J. Zeng. *A Practical Quantum Instruction Set Architecture*. 2016. URL: http://arxiv.org/abs/1608.03355.

[6] Microsoft. *Azure Quantum Development Kit*. URL: https://github.com/microsoft/qsharp.

[7] Seyon Sivarajah et al. "Tket: a retargetable compiler for NISQ devices". In: *Quantum Science and Technology* 6.1 (Nov. 2020), p. 014003. ISSN: 2058-9565. DOI: 10.1088/2058-9565/ab8e92. URL: http://dx.doi.org/10.1088/2058-9565/ab8e92.

[8] Ed Younis et al. *Berkeley Quantum Synthesis Toolkit (BQSKit) v1*. Apr. 2021. DOI: 10.11578/dc.20210603.2. URL: https://www.osti.gov/biblio/1785933.

[9] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[10] Eleanor G Rieffel and Wolfgang H Polak. *Quantum computing: A gentle introduction*. MIT press, 2011.

[11] David Deutsch. "Quantum computational networks". In: *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 425 (1989), pp. 73–90.

[12] Matthias Homeister. *Quantum Computing verstehen*. Springer, 2008.

[13] Manuel A Serrano et al. "Quantum software components and platforms: Overview and quality assessment". In: *ACM Computing Surveys* 55.8 (2022), pp. 1–31.

[14] Tao Yue et al. "Challenges and Opportunities in Quantum Software Architecture". In: *Recent Trends in Software Architecture* (2023), pp. 45–52.

[15] María Cerezo et al. "Variational quantum algorithms". In: *Nature Reviews Physics* 3 (2020), pp. 625–644.

[16] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. "A quantum approximate optimization algorithm". In: *arXiv preprint arXiv:1411.4028* (2014).

[17] Cheng Xue et al. "Effects of quantum noise on quantum approximate optimization algorithm". In: *Chinese Physics Letters* 38.3 (2021), p. 030302.

[18] Andrew Lucas. "Ising formulations of many NP problems". In: *Frontiers in physics* 2 (2014), p. 74887.

[19] Kostas Blekos et al. "A review on Quantum Approximate Optimization Algorithm and its variants". In: *Physics Reports* 1068 (June 2024), pp. 1–66. ISSN: 0370-1573. DOI: 10.1016/j.physrep.2024.03.002. URL: http://dx.doi.org/10.1016/j.physrep.2024.03.002.

[20] Michael R Garey, David S Johnson, and Larry Stockmeyer. "Some simplified NP-complete problems". In: *Proceedings of the sixth annual ACM symposium on Theory of computing*. 1974, pp. 47–63.

[21] Wilson R. M. Rabelo, Sandra D. Prado, and Leonardo G. Brunnet. *A QAOA approach with fake devices: A case study for the maximum cut in ring graphs*. 2024. arXiv: 2404.03501 [quant-ph].

[22] John Preskill. "Quantum Computing in the NISQ era and beyond". In: *Quantum* (2018). DOI: 10.22331/q-2018-08-06-79.

[23] Travis S Humble et al. "Quantum computing circuits and devices". In: *IEEE Design & Test* 36.3 (2019), pp. 69–94.

[24] Nathalie P De Leon et al. "Materials challenges and opportunities for quantum computing hardware". In: *Science* 372.6539 (2021), eabb2823.

[25] Sergei Slussarenko and Geoff J Pryde. "Photonic quantum information processing: A concise review". In: *Applied Physics Reviews* 6.4 (2019).

[26] Petar Jurcevic et al. "Demonstration of quantum volume 64 on a superconducting quantum computing system". In: *Quantum Science and Technology* 6.2 (2021), p. 025020.

[27] Carmen G. Almudéver et al. "Realizing Quantum Algorithms on Real Quantum Computing Devices". In: *2020 Design, Automation & Test in Europe Conference & Exhibition, DATE 2020, Grenoble, France, March 9-13, 2020*. IEEE, 2020, pp. 864–872. DOI: 10.23919/DATE48585.2020.9116240. URL: https://doi.org/10.23919/DATE48585.2020.9116240.

[28] Andrew Cross et al. "OpenQASM 3: A Broader and Deeper Quantum Assembly Language". In: *ACM Transactions on Quantum Computing* 3.3 (Sept. 2022), pp. 1–50. ISSN: 2643-6817. DOI: 10.1145/3505636. URL: http://dx.doi.org/10.1145/3505636.

[29] *Getting started with Native Gates*. [Online; Accessed April 3, 2024]. Jan. 2024. URL: https://ionq.com/docs/getting-started-with-native-gates.

[30] *IBM Quantum Platform - IBM Sherbrooke*. [Online; Accessed April 3, 2024]. Apr. 2024. URL: https://quantum.ibm.com/services/resources?system=ibm_sherbrooke.

[31] *IonQ Forte*. [Online; Accessed April 3, 2024]. Apr. 2024. URL: https://ionq.com/quantum-systems/forte.

[32] *Rigetti Systems*. [Online; Accessed April 3, 2024]. Apr. 2024. URL: https://qcs.rigetti.com/qpus.

[33] Christopher Monroe and Jungsang Kim. "Scaling the ion trap quantum processor". In: *Science* 339.6124 (2013), pp. 1164–1169.

[34] A. Yu Kitaev. "Quantum computations: algorithms and error correction". In: *Russian Mathematical Surveys* 52.6 (Dec. 1997), pp. 1191–1249.

[35] Robert R Tucci. "An introduction to Cartan's KAK decomposition for QC programmers". In: *arXiv preprint quant-ph/0507171* (2005).

[36] 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*. 2019, pp. 1001–1014.

[37] Marc G Davis et al. "Towards optimal topology aware quantum circuit synthesis". In: *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE. 2020, pp. 223–234.

[38] Marcos Yukio Siraichi et al. "Qubit allocation". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 2018, pp. 113–125.

[39] Alexandru Paler, Alwin Zulehner, and Robert Wille. "NISQ circuit compilation is the travelling salesman problem on a torus". In: *Quantum Science Technology* 6 (2020).

[40] Peter E Hart, Nils J Nilsson, and Bertram Raphael. "A formal basis for the heuristic determination of minimum cost paths". In: *IEEE transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[41] Ed Younis et al. "Qfast: Conflating search and numerical optimization for scalable quantum circuit synthesis". In: *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE. 2021, pp. 232–243.

[42] Ethan Smith et al. "Leap: Scaling numerical optimization based synthesis using an incremental approach". In: *ACM Transactions on Quantum Computing* 4.1 (2023), pp. 1–23.

[43] Ji Liu et al. "Tackling the qubit mapping problem with permutation-aware synthesis". In: *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. Vol. 1. IEEE. 2023, pp. 745–756.

[44] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022.

[45] Andrew Fagan and Ross Duncan. "Optimising Clifford Circuits with Quantomatic". In: *Electronic Proceedings in Theoretical Computer Science* 287 (Jan. 2019), pp. 85–105. ISSN: 2075-2180. DOI: 10.4204/eptcs.287.5. URL: http://dx.doi.org/10.4204/EPTCS.287.5.

[46] Sukin Sim, Peter D. Johnson, and Alán Aspuru-Guzik. "Expressibility and Entangling Capability of Parameterized Quantum Circuits for Hybrid Quantum-Classical Algorithms". In: *Advanced Quantum Technologies* 2.12 (Oct. 2019). ISSN: 2511-9044. DOI: 10.1002/qute.201900070. URL: http://dx.doi.org/10.1002/qute.201900070.

[47] Karen Wintersperger, Hila Safi, and Wolfgang Mauerer. *QPU-System Co-Design for Quantum HPC Accelerators*. 2022. arXiv: 2208.11449 [cs.AR].

[48] *IBM Quantum Documentation ECR Gate*. [Online; Accessed April 28, 2024]. URL: https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.ECRGate.

List of Figures

List of Tables