

Institute of Formal Methods in Computer Science

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master's Thesis

Exhaustive K-SWAP-Sequence Search for Qubit Mapping

Jakob Waibel

Course of Study: M.Sc. Informatik

Examiner: Prof. Dr. Stefan Funke

Supervisor: Prof. Dr. Stefan Funke,
Dr. Sebastian Brandhofer

Commenced: 2025-04-01

Completed: 2025-10-01

Abstract

Quantum computing promises to solve certain problems more efficiently than classical computation. Realizing this potential on noisy, intermediate-scale quantum (NISQ) devices remains challenging. Among other things, hardware connectivity constraints necessitate additional routing operations that degrade circuit fidelity. The qubit mapping problem – assigning logical qubits to physical qubits while minimizing inserted SWAP gates – is thus critical in quantum circuit transpilation. Exact methods scale poorly, and heuristic approaches like the SABRE algorithm, though efficient, rely on local decisions that can miss globally optimal solutions. In this thesis, we propose *K-SWAP SABRE* that integrates bounded exhaustive search over sequences of up to k SWAPs into the SABRE algorithm. A modified heuristic ensures global progress while maintaining comparability between explored branches. Experiments on benchmark circuits show that increasing k reduces required SWAP gates, especially on highly connected devices, though at the cost of longer runtime. Our results demonstrate that *K-SWAP SABRE* can improve heuristic qubit mapping quality without sacrificing scalability, offering a practical approach for mapping circuits to near-term and emerging quantum architectures.

Kurzfassung

Quantencomputing bietet das Potenzial, bestimmte Probleme effizienter zu lösen als klassische Computer. Die Umsetzung auf Noisy, Intermediate-scale Quantum (NISQ) Computern ist jedoch herausfordernd. Unter anderem, weil Hardware-Beschränkungen zusätzliche Routing-Operationen erfordern, die die Ergebnisqualität verringern. Das Qubit-Mapping – die Zuordnung logischer zu physischen Qubits bei minimalen SWAP-Gattern – ist daher ein entscheidender Schritt in der Transpilation von Quanten-Schaltungen. Exakte Methoden skalieren schlecht, während heuristische Ansätze wie SABRE effizient, aber lokal begrenzt sind und globale Optimierungen übersehen können. In dieser Arbeit wird *K-SWAP SABRE* vorgestellt, der begrenzte erschöpfende Suche über Sequenzen von bis zu k SWAP-Gattern in den SABRE Algorithmus integriert. Eine modifizierte Heuristik gewährleistet globalen Fortschritt, während die Vergleichbarkeit der Lösungszweige erhalten bleibt. Experimente auf Benchmark-Schaltungen zeigen, dass längere Sequenzen k die Anzahl benötigter SWAP-Gatter reduzieren können, insbesondere bei Quantencomputern mit hoher Konnektivität, jedoch auf Kosten längerer Laufzeiten. Die Ergebnisse verdeutlichen, dass *K-SWAP SABRE* die Qualität heuristischer Qubit-Zuordnung verbessern kann, ohne die Skalierbarkeit zu verlieren und einen praktikablen Ansatz für die Transpilation von Quanten-Schaltungen darstellt.

Contents

1	Introduction	9
1.1	Motivation and Objectives	9
1.2	Fundamentals of Quantum Computing	11
2	Related Work	16
2.1	Exact and Combinatorial Formulations	16
2.2	Layered Decomposition and Heuristic Algorithms	17
2.3	Benchmarks and Toolchain	18
3	The Qubit Mapping Problem	19
3.1	Preliminaries and Assumptions	19
3.2	Problem Definition	20
3.3	Review of the SABRE Algorithm	22
4	Exhaustive K-SWAP-Sequence Search	29
4.1	Algorithm Formulation	30
4.2	Heuristic Function Design	35
4.3	Algorithmic Complexity	42
5	Experimental Evaluation	44
5.1	Implementation	44
5.2	Benchmark Setup	45
5.3	Results and Discussion	47
6	Conclusion and Outlook	54
	Bibliography	56

List of Figures

1.1	A visualization of the Bloch sphere. In this case, a two-level quantum system is shown. State $ \psi\rangle$ is visualized as a linear combination of ground states $ 0\rangle$ and $ 1\rangle$ [Com09].	12
1.2	An example of a three qubit quantum circuit. Each horizontal line represents a qubit and the operations applied to it over time (left to right). The two boxes with an H label represent Hadamard gates. The three CNOT gates are represented by a control and target qubit connected by a vertical line. At the end of the circuit, a measurement is conducted on qubit q_2 . . .	13
1.3	Different examples of Quantum Processing Unit (QPU) topologies. The figure contains examples of a (a) linear topology, (b) square topology and (c) heavy-hex topology. The basic building blocks shown in the figure can be repeated to form larger coupling maps. Note that the square topology has to form a $N \times N$ grid to optimize the connectivity.	14
1.4	A single SWAP gate (left) is equivalent to three CNOT gates (right) when the circuit gets reduced to basis gates.	15
3.1	Comparison of the original logical circuit (3.1a) and the routed circuit (3.1b). The two boxed CNOT's in the left circuit are not directly routable under linear nearest-neighbor constraints (3.1c). Hence, a SWAP gate was inserted to successfully route the circuit.	20
3.2	Example of a 5-qubit quantum circuit composed out of 5 two-qubit gates. .	23
3.3	Example Directed Acyclic Graph (DAG) corresponding to the quantum circuit depicted in Figure 3.2. Furthermore, the front-layer and extended-set are highlighted.	24
4.1	Example circuit to illustrate the local heuristic of the classical SABRE algorithm. The left subfigure depicts a quantum circuit, while the right subfigure depicts the topology over time (top to bottom).	29
5.1	Distribution of test data in the HamLib <i>AbstractTranspile</i> suite [SMH+24]. Each point represents a single test circuit. The x-axis shows the number of qubits in the circuit, while the y-axis indicates the number of SWAP gates inserted by Qiskit's SABRE during routing.	47

5.2	Mean SWAP counts for <i>K-SWAP SABRE</i> (x-axis) vs Qiskit SABRE (y-axis) across topologies and sequence length k . Each point is the mean over $n=10$ independent runs. Color encodes number of qubits in the circuit. Both axes are log-scaled.	48
5.3	Distribution of relative SWAP improvement on linear, heavy-hex and square topology for $k \in \{1, 2, 3\}$. Violins show filtered distributions (Interquartile Range (IQR) rule).	50
5.4	Distribution of relative SWAP improvement of <i>K-SWAP SABRE</i> compared to Qiskit SABRE for circuits with fewer than 40 qubits on a square topology. Results shown for $k \in \{1, 2, 3, 4\}$	51
5.5	Analysis of individual circuit performance. We analyze the number of SWAPs, circuit depth and runtime across varying sequence lengths k	51

List of Listings

1.1	OpenQASM 2.0 representation of a quantum circuit.	13
-----	---	----

List of Algorithms

3.1	SABRE Routing: Run	25
4.1	K-SWAP SABRE Routing: Run	31
4.2	K-SWAP SABRE Routing: ChooseBestSwaps	33
4.3	K-SWAP SABRE Routing: ApplyPrefix	34

List of Abbreviations

APSP	All-Pair Shortest Path.	23
DAG	Directed Acyclic Graph.	4
DFS	Depth First Search.	27
IQR	Interquartile Range.	5
LDPC	Low-Density Parity-Check.	15
NISQ	Noisy, Intermediate-Scale Quantum.	9
QPU	Quantum Processing Unit.	4
SAT	Boolean Satisfiability Problem.	17
SDK	Software Development Kit.	10
SMT	Satisfiability Modulo Theories.	17

1 Introduction

Quantum computing promises fundamentally new ways to process information by harnessing the laws of quantum mechanics. For certain well-studied problems, quantum algorithms provably offer asymptotic speedups over the best-known classical algorithms: notable examples include Shor’s algorithm for integer factorization and Grover’s algorithm for unstructured search [DJ92; Gro96; Sho99]. At the same time, “quantum” has become a powerful cultural and marketing term, appearing frequently in product announcements and popular writing, so it is important to separate genuine scientific potential from hype.

The theoretical roots of the field reach back decades. Modern quantum mechanics was formulated in the early twentieth century, and the idea of using quantum systems to perform computation gained concrete form only in the late twentieth century. In 1981 Richard Feynman captured the key insight in one of the field’s most-cited remarks: “[...] nature isn’t classical, dammit, and if you want to make a simulation of nature, you’d better make it quantum mechanical [...]” [Fey18]. That observation both inspired and justified the study of quantum algorithms and quantum architectures.

1.1 Motivation and Objectives

From a systems perspective, today’s quantum processors are best described as Noisy, Intermediate-Scale Quantum (NISQ) devices: they make small- to medium-scale demonstrations possible, but they remain limited by decoherence and gate errors. Qubits, the quantum analogue of classical bits, are fragile two-level systems that can exist in superposition and become entangled. Quantum circuits are expressed as sequences of single-qubit and multi-qubit gates. In most physical devices, two-qubit gates can be implemented only between certain pairs of physical qubits determined by the hardware’s connectivity graph. This hardware constraint has direct consequences for mapping abstract quantum circuits to a physical device.

An essential task is therefore the *qubit mapping problem*: given a quantum circuit (a sequence of logical qubit operations) and a target device with limited connectivity, how should we map logical qubits to the device’s physical qubits so that every gate in the

circuit can be executed? Because two-qubit gates typically require the two participating qubits to be physically adjacent, a naive initial mapping will often be infeasible. Hence, the so-called transpiler must introduce additional operations, so-called SWAP gates, that move quantum states across the device in order to satisfy the adjacency constraints. Each SWAP gate increases the circuit length and the accumulated error. Consequently, minimizing the number of inserted SWAPs is crucial for improving the fidelity of execution on real hardware.

Solving the qubit mapping problem exactly is computationally demanding: exhaustive optimal solutions quickly become infeasible as circuit size and device size grow. Consequently, most practical transpilers rely on heuristic methods that trade optimality for scalability. Among these, the SABRE algorithm has gained wide adoption (e.g. in Qiskit, IBM’s Quantum Software Development Kit (SDK)), because it scales well and typically produces good mappings on moderate-size circuits [JTK+24]. SABRE makes routing decisions based primarily on the current frontier of the circuit. This local view is an effective way to keep the algorithm runtime low, but it can miss routing choices that are locally suboptimal but globally beneficial.

Therefore, this thesis is concerned with bridging the gap between scalable heuristics and more exhaustive local search: we ask whether bounded, targeted exploration of the SWAP-search space can be combined with an efficient heuristic to yield better mappings in practice. Concretely, we partition the routing problem into short sequences of up to k consecutive SWAPs and perform a limited exhaustive search within these small windows while otherwise following a heuristic strategy derived from SABRE. By enforcing measures that guarantee global progress and by designing a comparison metric between different branches of the local search, our approach retains the heuristic nature of SABRE while allowing the routing algorithm to consider decisions on a larger local horizon.

The main findings of this work are as follows. First, inserting a bounded exhaustive search of depth k into the heuristic routing loop reduces the number of SWAPs produced for a wide range of benchmark circuits. The reduction generally improves as k increases, at the cost of greater runtime. Second, the benefit of deeper local search depends on the connectivity of the quantum hardware: devices with increased connectivity allow larger improvements for the same search depth.

The remainder of this thesis is organized as follows. The rest of Chapter 1 introduces the fundamentals of quantum computing and the required terminology to understand this thesis. Chapter 2 reviews the related work and prior approaches to the qubit mapping problem. Chapter 3 defines the presumptions made, as well as the mapping and routing problem. We then conduct a review of the SABRE algorithm that this thesis builds upon. Chapter 4 describes our K -SWAP SABRE algorithm. Furthermore, we argue about the design of the used heuristic function and the specific mechanisms used to compare explored candidate sequences. We conclude the chapter by analyzing the algorithmic

complexity. Chapter 5 contains the experimental methodology and results. Finally, Chapter 6 concludes.

1.2 Fundamentals of Quantum Computing

In this section, we will briefly introduce the basics of quantum computing to equip the reader with enough background to follow the remainder of this thesis.

1.2.1 Quantum States

While in classical computing, information is represented in bits (0 and 1), quantum computing relies on *quantum bits*, short qubits ($|0\rangle$, $|1\rangle$), to represent information. The state of a quantum system is represented as a linear combination of the basis states $|0\rangle$ and $|1\rangle$. Equation 1.1 defines such a state $|\psi\rangle$. The Bra-ket notation is used to represent states in quantum mechanics.

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \quad \alpha, \beta \in \mathbb{C} \quad (1.1)$$

$|\psi\rangle$ represents a linear combination of ground states $|0\rangle$ and $|1\rangle$ with $\alpha, \beta \in \mathbb{C}$ as probability amplitudes with $|\alpha|^2 + |\beta|^2 = 1$. This linear combination of ground states is often referred to as superposition, as $|\psi\rangle$ can represent all 2^n possible states of n qubits at the same time. Since the concept of quantum computing is based on the foundation of quantum mechanics, measurement causes the superposition of Equation 1.1 to collapse. That is, the state falls into one of the two basis states $|0\rangle$ or $|1\rangle$ [NC10]. The probability of the quantum state collapsing into the basis state $|0\rangle$ is given by $|\alpha|^2$ while $|1\rangle$ occurs with probability $|\beta|^2$ [NC10].

In a two-dimensional complex space, a quantum state $|\psi\rangle$ can be represented as a vector with length 1. This complex space, as shown in Figure 1.1, is also commonly referred to as the Bloch sphere.

1.2.2 Quantum Circuits

When thinking of a the state of a qubit as a complex vector on the Bloch sphere, this state can be manipulated by applying unitary operations like e.g. rotations and reflections. These unitary operations form quantum logic gates. Gates can operate either on single qubits or on multiple qubits.

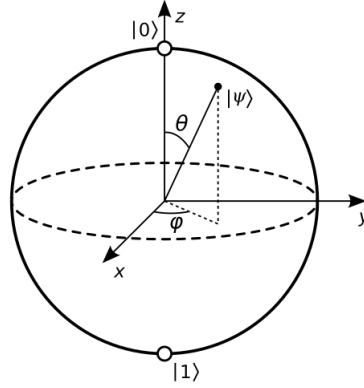


Figure 1.1: A visualization of the Bloch sphere. In this case, a two-level quantum system is shown. State $|\psi\rangle$ is visualized as a linear combination of ground states $|0\rangle$ and $|1\rangle$ [Com09].

Common single-qubit operations include the Hadamard gate (H) that transforms the state of a qubit from one of the basis states $|0\rangle$ and $|1\rangle$ into an equal superposition, the Pauli X gate that transforms a state $\alpha|0\rangle + \beta|1\rangle$ into $\alpha|1\rangle + \beta|0\rangle$ or the Pauli Z gate that changes the sign of $|1\rangle$ to $-|1\rangle$ and vice versa [NC10].

All of the beforementioned gates can be represented as unitary matrices as shown in Equation 1.2.

$$H \equiv \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Z \equiv \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad (1.2)$$

In addition to single-qubit gates, there are multi-qubit gates that act on two or more qubits. One example is the CNOT gate. Like other multi-qubit gates, it is composed of control and target qubits. The CNOT gate flips the target qubit if and only if the control qubit is $|1\rangle$. In visualizations of quantum circuits, control bits are indicated by $\bullet$ while target bits are indicated by $\oplus$.

Applying a sequence of gates on a number of logical qubits then forms a quantum circuit. In the following, we distinguish logical qubits – the qubits of the quantum circuit – and physical qubits – the qubits on the quantum hardware. Figure 1.2 shows a three qubit quantum circuit where each horizontal line represents a logical qubit and the operations applied to it over time (left to right). The gates in this circuit are Hadamard gates, depicted as boxes with an H label, and CNOT gates represented by a control and target qubit connected by a vertical line.

Starting from the left, a CNOT is applied on qubits q_0 and q_1 . Independently, a Hadamard gate is applied to q_2 . After executing the previous two operations, the precedence

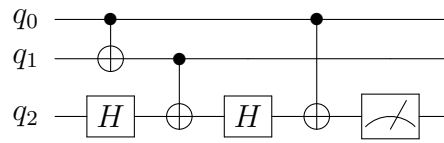


Figure 1.2: An example of a three qubit quantum circuit. Each horizontal line represents a qubit and the operations applied to it over time (left to right). The two boxes with an H label represent Hadamard gates. The three CNOT gates are represented by a control and target qubit connected by a vertical line. At the end of the circuit, a measurement is conducted on qubit q_2 .

constraints for the CNOT on q_1 and q_2 are resolved and the gate can be executed. That is, the unitary transformation can be applied to the state of the qubit. Afterwards, the second Hadamard gate on q_2 can be applied. The last gate executed is another CNOT that operates on qubits q_0 and q_2 . After all gates have been executed, a measurement is conducted on qubit q_2 . Unlike all other gates, a measurement is not reversible, which is why a measurement is not considered a gate. As previously mentioned, a measurement causes the state to collapse into either $|0\rangle$ or $|1\rangle$ [NC10].

While the circuit diagrams as the one in Figure 1.2 are humanly parse-able, another standard is required to efficiently describe circuits in software. Therefore, quantum assembly languages like OpenQASM 2.0 and OpenQASM 3.0 were developed [CBSG17; CJA+22]. The following short example in Listing 1.1 shows the quantum circuit from Figure 1.2 described in OpenQASM 2.0.

Listing 1.1 OpenQASM 2.0 representation of a quantum circuit.

```
OPENQASM 2.0;
include "qelib1.inc";

qreg q[4];
creg c[4];
cx q[0], q[1];
h q[2];
cx q[1], q[2];
h q[2];
cx q[0], q[2];
measure q[2] -> c[2];
```

1.2.3 Quantum Hardware

After introducing the theoretical concept of a quantum circuit, we still require hardware to execute it on. The implementation of a quantum computer can be based on a variety of technological concepts – all building on top of quantum mechanics. For example, it can be based on superconductors, ion traps, neutral atoms or quantum dot technologies. This thesis is targeting the connectivity constraints of IBM’s commercially available quantum computers based on superconductors [KYG+07].

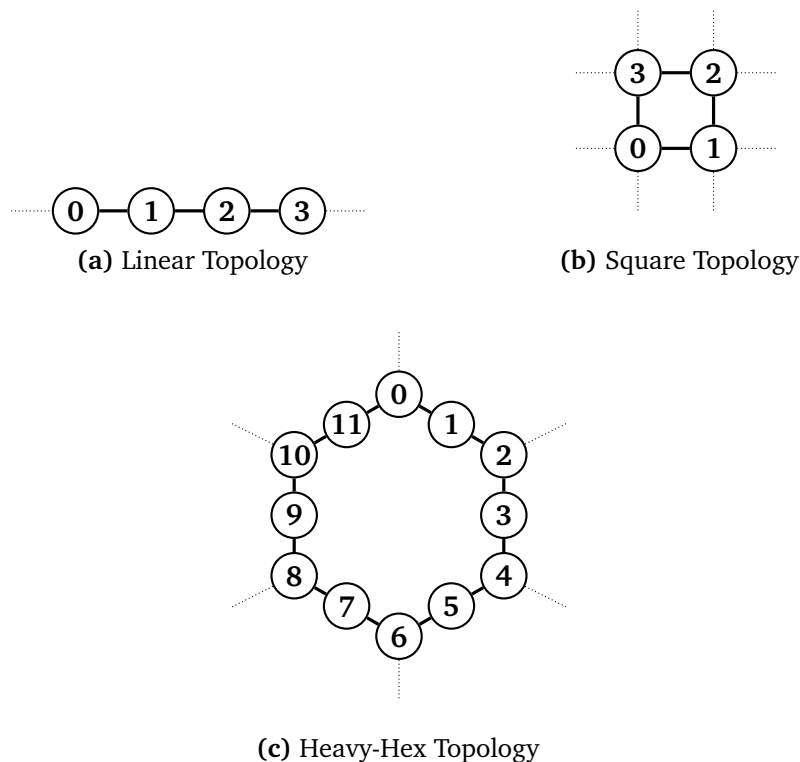


Figure 1.3: Different examples of QPU topologies. The figure contains examples of a (a) linear topology, (b) square topology and (c) heavy-hex topology. The basic building blocks shown in the figure can be repeated to form larger coupling maps. Note that the square topology has to form a $N \times N$ grid to optimize the connectivity.

So far, we assumed that we can execute multi-qubit gates on arbitrary logical qubits in the circuit. However, the actual physical qubits on the QPU need to be physically adjacent in order to execute a quantum gate. They are usually not connected using an all-to-all topology. This is due to the static characteristic of superconducting QPU’s that prohibits moving around physical qubits. Coupling all qubits on a superconducting QPU requires

long-range coupling which increases the error-rate due to crosstalk as the number of connections increases. Some examples of topologies can be observed in Figure 1.3. Each topology comes with different advantages regarding error-rate and qubit connectivity. Among them are the linear topology (Figure 1.3a), the square topology (Figure 1.3b) and the heavy-hex topology (Figure 1.3c). These are the topologies (coupling maps) that will be considered in scope for this thesis. Each topology is constructed out of the basic building blocks depicted in Figure 1.3. Multiple basic blocks can be connected as suggested by the dashed edges. Note that for connectivity reasons, the square topology is usually constructed out of the basic building blocks such that the final topology resembles a $N \times N$ grid. Currently, the square topology depicted in Figure 1.3b is the most promising topology to IBM since it can be efficiently simulated, requires less depth to represent the same connectivity and is a prerequisite for implementing quantum error correction Low-Density Parity-Check (LDPC) codes as introduced by Breuckmann and Eberhardt [BE21].

Now, in order to be able to execute a quantum circuit on superconducting hardware, a series of transformations is required. This process is referred to as transpilation and includes stages like the mapping of the circuit onto the physical device or the transformation of the gates in the circuits into the subset of basis gates the quantum hardware supports. The transpilation stage this thesis is concerned with is the routing (qubit mapping) stage. In this stage, the logical qubits ($q_0, \dots, q_n$) of the circuit are mapped onto the physical qubits ($Q_0, \dots, Q_n$) of the QPU. This process can not necessarily guarantee that all logical qubits involved in two qubit operations are actually mapped onto adjacent physical qubits. This is achieved by augmenting the circuit with so-called SWAP gates.

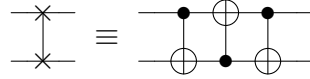


Figure 1.4: A single SWAP gate (left) is equivalent to three CNOT gates (right) when the circuit gets reduced to basis gates.

A SWAP gate exchanges the state of two adjacent physical qubits. This allows us to move a logical qubit arbitrarily around the coupling map by swapping the state of physical qubits along the way. However, inserting a SWAP gate into the circuit at transpilation comes with a cost. Each operation applied to the state of a qubit introduces noise and thereby increases the error-rate. As can be seen in Figure 1.4, each SWAP gate is constructed out of three CNOT gates [NC10]. Hence, it is important to minimize the number of SWAPs inserted during transpilation.

2 Related Work

This thesis builds upon a broad spectrum of research addressing the qubit mapping problem – the task of transforming a quantum circuit into an equivalent hardware-constrained circuit that respects the limited connectivity of the QPU. Previous work on the qubit mapping problem can be categorized into two distinct methods: exact formulations that reduce the mapping problem to known combinatorial or mathematical programming problems to seek optimal solutions for small problem instances (Section 2.1), and heuristic approaches that trade this optimality for scalability as the problem size increases (Section 2.2). Furthermore, this thesis benefits from previous contributions focused on improving the benchmarking capabilities and toolchain for quantum circuits (Section 2.3).

2.1 Exact and Combinatorial Formulations

Several contributions have formalized the qubit mapping problem as an instance of classical combinatorial problems, clarifying its complexity and thereby informing exact-solution strategies. Siraichi et al. show how the problem can be modeled as a combination of token swapping (NP-hard) and subgraph isomorphism (NP-complete). To achieve this, they developed their *Bounded Mapping Tree* algorithm that utilizes dynamic programming to partition the qubit mapping problem and evaluate the partitions with subgraph isomorphisms. The subsolutions are then combined by a transformation equivalent to token swapping [SSCP18].

Tan and Cong developed an optimal layout synthesizer based on spacetime-based variable encoding of the problem. Their formulation yields an exact solver that explores a larger solution space than earlier exact techniques while reducing space and time complexity exponentially compared to related approaches. Furthermore, they developed a second, approximate but near-optimal synthesizer based on the same encoding strategy [TC20].

Complementing combinatorial exactness, Steinberg et al. made a theoretical contribution in which they model a quantum circuit and the QPU as density matrices that are modified by a composition of quantum operations. By applying this methodology, they show how a minimal SWAP solution corresponds to the minimization of distances between quantum

states, so-called lightcone bounds. Furthermore, they also provide an initial-placement technique based on graph similarity [SBS+24].

Brandhofer et al. extend exact formulations of the mapping problem by incorporating hardware features beyond fixed connectivity. For neutral-atom architectures, where qubits can be physically displaced, they augment the coupling graph with *movement edges* and encode the joint choice between SWAPs and movements as an Satisfiability Modulo Theories (SMT) problem, yielding shallower and more reliable circuits [BPB21]. In complementary work, they model mid-circuit resets as resource-reuse operations: qubits can transition between *in use* and *free*, and Boolean Satisfiability Problem (SAT)/SMT constraints enforce consistent reuse. By accounting for hardware-specific reset fidelities, they demonstrate significant fidelity gains (up to $4.3\times$) compared to SWAP-only mapping [BPK23]. These works show how exact solvers can flexibly encode hardware primitives such as qubit movement and reuse, broadening the scope of optimal mapping formulations.

2.2 Layered Decomposition and Heuristic Algorithms

Zulehner et al. introduced an approach that decomposes the circuit into layers of disjoint two-qubit gates and inserts so-called permutation layers composed of SWAP gates to transform between different layer mappings. Their method relies on A^* to route the permutation layers. The algorithm is further optimized using a lookahead scheme and a initial mapping strategy to improve the routing quality. It is especially effective when a circuit can be partitioned into many small parallel branches [ZPW18].

Heuristic, scalable methods dominate practical usecases for near-term hardware. Among them, SABRE, a SWAP-based bidirectional heuristic search algorithm by Li et al. became the defacto standard of industrial transpilation practices. In their work, they made the observation that mapping decisions should be primarily based on the qubits that are able to execute next, that is, the qubits have no unexecuted predecessor gates. Based on this, they developed a bidirectional initial mapping strategy, as well a distance-based heuristic with an optional decay mechanism to reduce the depth of the resulting circuit. With this, they managed to execute the mapping and routing process in polynomial runtime [LDX19].

There have been multiple follow-up publications refining SABRE’s heuristic and robustness. Among them, LightSABRE by Zou et al. augments the SABRE algorithm with relative scoring that manages to reduce the scoring cost from $\mathcal{O}(n)$ to $\mathcal{O}(1)$, support for control flow operations and improved runtime performance with concurrent trials and additional heuristic refinements. They also provide a circuit-breaker mechanism that allows for

making progress since the original heuristic is prone to getting stuck at local minima [ZTH+24].

Since this thesis builds on top of the SABRE algorithm and the extensions introduced by the LightSABRE paper, the technical details are explained in more depth as part of Chapter 3.

2.3 Benchmarks and Toolchain

Work on the qubit mapping algorithm is tightly coupled with benchmarking frameworks and open toolchains. Qiskit and the OpenQASM standards provide widely used integration [CBSG17; JTK+24]. Qiskit is IBM’s SDK to construct and transpile quantum circuits, as well as to interface with IBM’s quantum hardware. Its open nature made it a widely adopted project that enables research in the field of quantum information [JTK+24].

On the evaluation side, several benchmarking suites have emerged over time. *SupermarQ* introduced feature-based, application-level benchmarks by applying traditional benchmarking techniques to the quantum domain. Among other things, Tomesh et al. define several feature vectors to characterize the structure of a quantum circuit and ensure that the benchmark suite is representative of realistic quantum workloads [TGO+22]. More recently, Nation et al. developed *Benchpress*, a large suite encompassing more than 1000 circuits with up to 930 qubits and 10^6 two-qubit operations. In the scope of their work, they developed a common interface to test different implementations, natively supporting the integration of custom transpiler implementations [NSB+25].

3 The Qubit Mapping Problem

In this section, we will first lay out the core assumptions for the problem in scope of this thesis. Afterwards, we will define the qubit mapping problem in detail. Furthermore, we will discuss the technical details of the SABRE algorithm that this thesis builds upon.

3.1 Preliminaries and Assumptions

No three- or more qubit gates. This thesis is scoped to only consider one or two-qubit gates. This limitation does not change the semantics of the quantum circuit as any gate operating on more than two qubits can be unrolled until it is composed out of one and two-qubit gates only. While all examples in this thesis are using CNOT gates, it is important to note that the algorithms presented support routing arbitrary two-qubit gates.

No barriers. Barriers are a way to partition a circuit into multiple segments that will be treated independently of each other. Ignoring barriers changes the semantics of a circuit in that routing decisions will not be made independently for the corresponding segments. However, this allows for routing the circuit in a global scope.

No control flow. Other implementations like e.g. Qiskit allow for control flow operations like conditionals and loops. Since implementing control flow introduces unnecessary complexity that needs to be addressed in dedicated edge-cases, these operations will be omitted as part of this thesis. However, enabling control flow (e.g. loops or conditionals) does not fundamentally alter the routing semantics. Each branch or loop block can be treated as a subcircuit and recursively apply the same SWAP-routing logic, while restoring the layout at the end of a block. Thus omitting explicit control flow support does not limit the algorithm's expressiveness.

Optimize number of SWAPs. The design of a heuristic function for an algorithm depends on what the overall optimization goal is. In the scope of this thesis, we will focus on minimizing the number of SWAPs that are inserted in the routing process. Other valid

optimization goals include minimizing the depth (critical-path length), error rate or execution time of a circuit.

Static initial mapping. The original SABRE paper by Li et al. proposed a bidirectional technique to determine an initial mapping of the logical qubits from the circuit to the physical qubits on the QPU [LDX19]. This circuit will reuse that mapping technique as is. Choosing a trivial mapping (e.g. $q_0 \mapsto Q_0$) often comes with the downside of many SWAPs being inserted at the beginning to form a funnel. This is due to the distance-based heuristic that tries to minimize the distance between logical qubits in the circuit.

3.2 Problem Definition

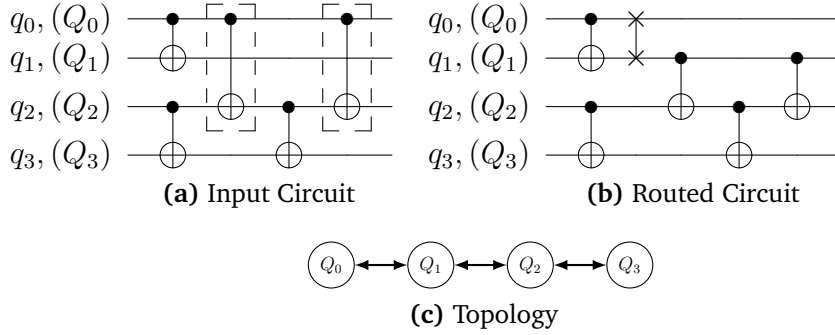


Figure 3.1: Comparison of the original logical circuit (3.1a) and the routed circuit (3.1b). The two boxed CNOT's in the left circuit are not directly routable under linear nearest-neighbor constraints (3.1c). Hence, a SWAP gate was inserted to successfully route the circuit.

The qubit mapping problem can be nicely motivated using a simple circuit as shown in Figure 3.1a. Suppose the circuit is running on a QPU with the linear topology displayed in 3.1c. Recall that quantum gates can only be executed once the physical qubits it operates on are physically adjacent. At the beginning, gates can only be executed on physical qubit pairs $\{Q_0, Q_1\}$, $\{Q_1, Q_2\}$ and $\{Q_2, Q_3\}$. Now assume that we want to execute the 4-qubit circuit from Figure 3.1a on this topology.

We apply the following trivial initial mapping from logical to physical qubits: $\{q_0 \mapsto Q_0, q_1 \mapsto Q_1, q_2 \mapsto Q_2, q_3 \mapsto Q_3\}$. The circuit we want to execute contains 5 CNOT gates operating on different qubits. The first two CNOT gates can be directly executed on the given topology without the need to insert any SWAP gates. After the first two gates have been executed, we observe that the third gate from the left (marked with a dashed box) does not operate on a valid physical qubit pair as we defined above. To

solve this, we need to insert a SWAP gate to exchange the state of two adjacent physical qubits that will make the gates executable. The two options here are either to apply a SWAP gate on $\{Q_0, Q_1\}$ or on $\{Q_1, Q_2\}$. When we apply the SWAP $\{Q_0, Q_1\}$, the layout changes to $\{q_1 \mapsto Q_0, q_0 \mapsto Q_1, q_2 \mapsto Q_2, q_3 \mapsto Q_3\}$. Based on that, we observe that all following gates can be executed without the need to insert any additional SWAP gates. In contrast to that, inserting a SWAP on $\{Q_1, Q_2\}$ would change the layout to $\{q_0 \mapsto Q_0, q_2 \mapsto Q_1, q_1 \mapsto Q_2, q_3 \mapsto Q_3\}$. With this layout, the fourth CNOT gate that follows on the current one operates on two qubits whose operands are not adjacent anymore. Since we want to minimize the number of SWAP gates that we need to insert, we pick the first swap candidate operating on $\{Q_0, Q_1\}$. For larger problem instances, we have to keep in mind that a lot of complexity of the problem stems from the fact that applying a SWAP gate to make a gate executable might increase distances between qubits that need to be adjacent for subsequent two-qubit gates.

An alternative solution to this specific circuit can be found when trying to apply a different initial mapping before executing the circuit. For this example, there exists an initial mapping that results in all gates being executable from the start without the need to insert any SWAP gates: $\{q_1 \mapsto Q_0, q_0 \mapsto Q_1, q_2 \mapsto Q_2, q_3 \mapsto Q_3\}$. This initial mapping is equivalent to the mapping we get from inserting the first SWAP gate from the previous solution.

Now, after having motivated the problem using an example, let us continue by formally defining the qubit mapping problem before we continue to propose possible solutions.

Definition 3.2.1 (Qubit Mapping Problem) *Given a DAG $G(V, E)$ that represents the two-qubit gates and their precedence constraints and the coupling map (topology) $G(V, E)$ of a QPU, find an initial mapping π such that all two-qubit gates can be executed on adjacent physical qubits once their precedence constraints have been satisfied.*

In 2016, Siraichi et al. proved that the qubit mapping problem is NP-Complete [SSCP18] based on a reduction from subgraph isomorphisms [Coo23].

With increasing problem sizes, finding a valid initial mapping to solve the qubit mapping problem is rarely possible. In practice, it is therefore commonly extended by a sub-problem, the so-called swap minimization problem [SSCP18]. It focuses on augmenting the circuit with the minimum number of SWAP gates required such that the result is a valid mapping for all two-qubit gates.

By showing that that the swap minimization problem involves solving the classical token swapping problem which has been shown to be NP-hard [YDI+15], Siraichi et al. proved that the swap minimization problem is NP-complete [SSCP18]. As mentioned in Section 3.1, this is the part of the extended qubit mapping problem this thesis tries to improve upon. It can be formally defined as follows.

Definition 3.2.2 (Swap Minimization Problem) *Given a DAG $G(V, E)$ that represents the two-qubit gate precedence relations in a quantum circuit, the current layout π and a coupling map $G(V, E)$, augment the circuit by inserting SWAP gates such that all precedence constraints are satisfied and the number of inserted SWAP gates is minimized.*

Note that in the following, we will access the distances of two physical qubits on the coupling map by the function $d(q_a, q_b)$ which returns the distances between two physical qubits the logical qubits q_a and q_b are currently mapped to.

3.3 Review of the SABRE Algorithm

Now that we formally defined the qubit mapping problem, we will discuss how the SABRE algorithm, originally designed by Li et al. [LDX19], solves this problem in detail. Furthermore, we consider the improvements found by Zou et al. [ZTH+24]. The remainder of this thesis then builds upon the SABRE algorithm and the concepts explained in this section.

SABRE resembles a heuristic approach of solving the qubit mapping problem. The name SABRE stands for **SWAP**-based **BidiRE**ctional heuristic search algorithm. It was developed to improve upon previous solutions that either map the qubit mapping problem into a similar mathematical problem to then apply solvers or follow a less advanced heuristic approach. While the former approaches lack the necessary limited runtime to scale efficiently, the latter were mostly developed for ideal lattice models that are not transferable to modern NISQ devices with restricted coupling. The bidirectionality suggested in the name is referring to the initial mapping technique that has been proposed by Li et al. [LDX19]. While this thesis applies said mapping technique to benefit from optimized initial mappings, this thesis focuses on improving the routing capabilities of SABRE [LDX19] by minimizing the number of SWAP gates inserted.

Like many other routing algorithms, SABRE can be divided into two phases: the preprocessing phase and the routing phase.

3.3.1 Preprocessing

Given a quantum circuit and a coupling map of the QPU, the SABRE algorithm precomputes a few key data structures. In a first step, the quantum circuit is transformed into a DAG. The coupling map is used to create a distance matrix (we access the values using the distance function $d(q_a, q_b)$). Furthermore, we initialize the so-called front-layer which we will introduce in the following.

DAG Generation. A circuit can be represented as a DAG. For the qubit mapping problem, it suffices to only consider two-qubit gates in the DAG since single-qubit gates can be executed trivially. The DAG can be constructed in $\mathcal{O}(|G|)$ with G representing the gates in the DAG. Semantically, the DAG represents the precedence constraints of two-qubit gates involved in the quantum circuit. As mentioned previously, a two-qubit gate $g = (q_a, q_b) \in G$ can be executed if and only if all predecessor gates of the DAG that operate on the physical qubit the logical qubits q_a and q_b are mapped to were executed before. After the execution, the DAG of the original circuit can be augmented using the SWAP gates that were computed during the routing stage.

Distance Matrix Computation. Given the topology (coupling map) of the QPU, we precompute the All-Pair Shortest Path (APSP) distances between any two physical qubits of the coupling map. To achieve this, SABRE utilizes the Floyd Warshall Algorithm [Joh77]. Since the SABRE algorithm uses a nearest neighbor cost function, the heuristic regularly needs the distance between the two physical qubits a gate operates on. As mentioned before, the distance between two physical qubits is indicated by $d(q_a, q_b)$. Since a SWAP gate only influences the mapping of local qubits to physical qubits, the distances between physical qubits are static throughout the lifecycle of the algorithm.

Front Layer Initialization. An important part of the SABRE algorithm is the so-called front-layer. It represents the gates of the DAG that do not have any unexecuted predecessors. That is, the gates in the front-layer represent the gates that have to be executed before any subsequent gates can be executed. Recognizing that the gates in the front-layer are the primary factor in making routing decisions resembles the main premise behind the SABRE algorithm. The front-layer is initialized by picking all gates (vertices in the DAG) that have 0 incoming edges.

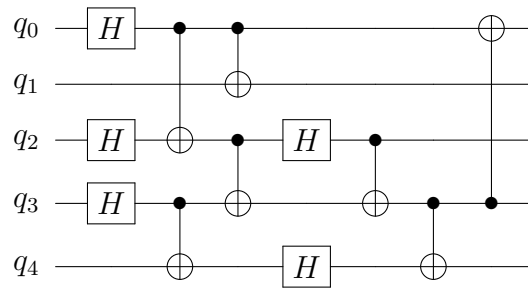


Figure 3.2: Example of a 5-qubit quantum circuit composed out of 5 two-qubit gates.

Figure 3.2 shows an example 5 qubit circuit with 5 single-qubit gates and 7 two-qubit gates. The DAG corresponding to that circuit can be seen in Figure 3.3. It represents the precedence constraints between two-qubit gates. Hence, the 5 single-qubit gates

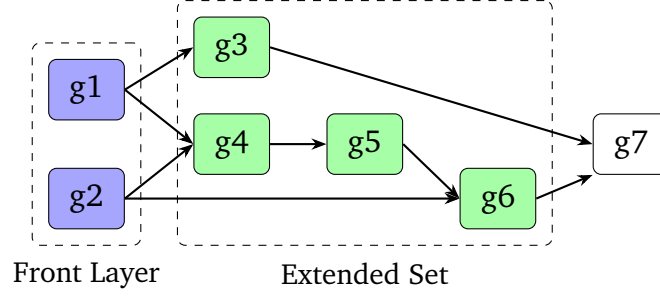


Figure 3.3: Example DAG corresponding to the quantum circuit depicted in Figure 3.2. Furthermore, the front-layer and extended-set are highlighted.

can be omitted. Furthermore, the DAG in Figure 3.3 highlights the initial front-layer and the so-called extended-set. Gate g_1 and g_2 are the only gates that do not have any incoming edges and therefore resemble the initial front-layer. Suppose a trivial linear topology as already assumed in previous examples. The extended-set contains the gates that can be executed without inserting any additional SWAP gates once the front-layer gates are executed. This set will be used to equip the heuristic with a limited lookahead ability. The number of items in the extended-set is usually bound to some constant. In the Qiskit implementation, this constant is set to 20 to limit the computation overhead while maximizing the information gain.

3.3.2 Routing

After having precomputed the DAG, the distance matrix and initialized the front-layer, we can begin the routing stage. This stage essentially traverses through the DAG once. During that traverse, we satisfy all the two-qubit constraints in the quantum circuit by inserting SWAP gates. Algorithm 3.1 shows the core logic of the algorithm designed by Li et al. [LDX19].

Given an input DAG, a front-layer F , the distance function $d(q_a, q_b)$ and an initial layout π , SABRE iterates until there are no remaining items in the front-layer, that is, all two-qubits in the front-layer have been successfully routed (executed).

Each iteration begins by checking whether any gates in the front-layer can be executed without inserting any SWAP gates. This step essentially just checks whether the logical qubits of the gate currently in question are mapped to two adjacent physical qubits. If that is the case, we append that gate to the `execute_gate_list` that tracks the gates ready to execute. In a next step, we then check whether there are any gates in this `execute_gate_list`. The gates in this list can be executed, which we handle in that we remove those gates from the front-layer and obtain the successors of the gates that do

Algorithm 3.1 SABRE Routing: Run

```

1: function RUN( )
2:   while  $F \neq \emptyset$  do
3:     executeGateList =  $\emptyset$ ;
4:     for gate  $\in F$  do
5:       if gate can be executed on device then
6:         executeGateList.APPEND(gate);
7:       end if
8:     end for
9:     if executeGateList  $\neq \emptyset$  then
10:      for gate  $\in$  executeGateList do
11:        F.REMOVE(gate);
12:        obtain successor gates from DAG;
13:        if successor gates' dependencies are resolved then
14:          F.APPEND(gate);
15:        end if
16:      end for
17:      Continue;
18:    else
19:      scores =  $\emptyset$ ;
20:      candidates = COMPUTESWAPCANDIDATES(F, G);
21:      for SWAP  $\in$  candidates do
22:         $\pi_{temp} = \pi.UPDATE(SWAP)$ ;
23:        scores[SWAP] = H(F, DAG,  $\pi_{temp}$ , D, SWAP);
24:      end for
25:      Find the SWAP with minimal score;
26:       $\pi = \pi.UPDATE(SWAP)$ ;
27:    end if
28:  end while
29: end function

```

not have any unmet precedence constraints. We then continue into the next iteration of the loop to check again whether those gates can be executed and we can advance the front-layer further.

If the `execute_gate_list` is empty, this means that no gates can be routed as is and we need to insert at least one SWAP gate to continue routing the circuit. To achieve this, we will first determine the so-called swap candidates.

The swap candidates are computed by determining all edges of the coupling graph that are adjacent to a physical qubit involved in the current front-layer. The number of swap

candidates is proportional to the number of qubits in the front-layer F and depends on the coupling map of the QPU. As an example, assume a ring topology. Each qubit is connected to two other qubits, that is, each vertex has a degree of 2. This would then lead to 2 candidate SWAPs for every qubit involved in the front-layer.

After this, each swap candidate is temporarily applied to the layout and the layout is then scored using a heuristic function h . After computing the scores for all swap candidates, the minimum score determines the best SWAP. We complete the iteration by applying the swap candidates with the minimum score to the layout.

The subsequent iterations will then begin to check whether there are any gates that are executable. Sometimes, more than one SWAP gate is required to make the next gate executable. In that case, the algorithm will iterate through the front-layer, see that no gates are executable and continue to determine the next best SWAP gate that can be inserted into the circuit.

Eventually, all gates are executed. So far, we have assumed that we have a heuristic function to determine the *best* swap candidate. The following section will cover the heuristic function h , since the quality of results largely depends on the heuristic function design.

3.3.3 Heuristic Function

As mentioned before, SABRE is built upon the observation that the gates in the front-layer need to be executed before any other gates can be executed [LDX19]. Hence, it should be the primary factor when designing a heuristic function.

SABRE utilizes a nearest neighbor cost function utilizing the distance function $d(q_a, q_b)$ that we introduced in Section 3.3.1. This function returns the distance between two physical qubits. Since the qubits need to be adjacent, a distance $d(q_a, q_b) = 1$ renders a gate executable. Based on this, Li et al. define a basic heuristic function summing over the distances between physical qubits of gates in the front-layer as shown in Equation 3.1 [LDX19].

$$h_{basic} = \sum_{(q_a, q_b) \in F} d(q_a, q_b) \quad (3.1)$$

Intuitively, this equation prefers swap candidates that bring the qubits in the front-layer closer together. That is, the *best* swap candidate is the SWAP gate that brings the qubits of the front-layer together the most. Moving the qubits closer together will eventually render the front-layer gates executable.

While h_{basic} provides a valid heuristic function to guide the routing stage, it is not able to determine whether a SWAP will be beneficial for the subsequent iterations of the algorithm. Only the gates in the current front-layer are considered. We are therefore unable to consider whether a swap candidate will be beneficial for successive iterations of the algorithm.

To mitigate that risk, Li et al. introduced the notion of the so-called extended-set E [LDX19]. The extended-set is defined as the set of successor gates that become executable – without inserting any additional SWAP gates – once the front-layer gates have been executed.

The extended-set is populated by performing a Depth First Search (DFS) [Knu13]. As mentioned previously, the number of gates in the extended-set is usually bound by some constant to reduce computational complexity. Since the front-layer is still the primary factor for heuristic decisions, the impact of the extended-set is usually limited by some weight λ , e.g. 0.5. To avoid a large extended-set to outweigh a small front-layer, F and E are normalized by $|F|$ and $|E|$, respectively. Although the original paper by Li et al. does not mention this heuristic explicitly, the introduction of the extended-set implicitly leads to a second heuristic function $h_{lookahead}$ as defined in Equation 3.2.

$$h_{lookahead} = \underbrace{\frac{1}{|F|} \sum_{(q_a, q_b) \in F} d(q_a, q_b)}_{\text{basic component}} + \lambda \cdot \underbrace{\frac{1}{|E|} \sum_{(q_a, q_b) \in E} d(q_a, q_b)}_{\text{lookahead component}} \quad (3.2)$$

This lookahead heuristic $h_{lookahead}$ can be computed in $\Theta(|F| + |E|)$. In the original paper, $h_{lookahead}$ is then extended by introducing a decay term to prioritize executing gates in parallel and thereby reducing depth. This decay term leads to the heuristic picking solutions with more SWAPs to achieve a decreased overall depth. Since we settled on the assumption to minimize the number of SWAP gates in section 3.1, this term will not be discussed in more detail.

In case of a tie, SABRE chooses randomly which swap candidate to pick among all candidates with the minimum heuristic. This prevents getting stuck at a local minimum across multiple executions of the algorithm.

3.3.4 Relative Scoring

Applying a single SWAP can at most make two gates of the front-layer executable. This is due to how the front-layer is defined. The front-layer only contains gates that have met their precedence constraints. This implies that all gates in the front-layer have to operate on distinct qubits. Having two gates operating on the same qubits would contradict the

definition to only contain gates with satisfied precedence constraints. Hence, at most two gates in the front-layer can operate on the qubits involved in a single SWAP gate.

At the same time, a single SWAP can potentially lead to all gates in the extended-set to become executable. This is due to the beforementioned way the extended-set is populated i.e. only containing gates that can be instantly executed when a front-layer gate was routed successfully.

Based on this, Zou et al. argue that a single SWAP can affect at most a total of $2 + |E|$ terms in Equation 3.2. The basic term will render either one or two gates of the front-layer executable while all gates in the lookahead term could become executable after applying a single SWAP gate. Hence, $\mathcal{O}(1)$ terms of Equation 3.2 are affected by a single SWAP gate [ZTH+24].

Suppose $h_{q_a \leftrightarrow q_b}$, with (q_a, q_b) being a swap candidate and $h_{q_a \leftrightarrow q_b}$ being the heuristic of Equation 3.2 after applying the SWAP (q_a, q_b) . Let h_0 be the heuristic without applying a SWAP gate to the layout. Zou et al. argue that when $h_{q_a \leftrightarrow q_b}$ is minimized for a SWAP gate, $h_{q_a \leftrightarrow q_b} - h_0$ is also minimized (for any constant h_0). Hence, the relative change of each SWAP, which affects only $\mathcal{O}(1)$ terms since h_0 does not have to be computed explicitly, can be taken as a base for scoring swap candidates [ZTH+24].

As a result, they managed to reduce the complexity from $\mathcal{O}(n)$ with n defined as the number of gates in the front layer to $\mathcal{O}(1)$.

3.3.5 Release Valve

In addition to the relative scoring, Zou et al. show how the SABRE $h_{lookahead}$ heuristic is prone to getting stuck at a local minimum if only considering the minimum score to make SWAP decisions. This can happen in a scenario where the number of front-layer gates heavily outweighs the number of gates in the extended-set. In that case, bringing two front-layer qubits closer together to make progress while pulling a gate from the extended-set apart can be more costly than to SWAP two qubits of the same gate and not make any forward progress. Consequently, this leads to a scenario in which the SABRE algorithm continuously applies the same SWAP gates without making any forward progress. To solve this, Zou et al. introduce a release valve to ensure the SABRE heuristic makes progress. By keeping track of the number of SWAP gates since the last gate was successfully routed, a loop caused by the heuristic getting stuck can be resolved by applying Dijkstra's algorithm on the front-layer gate that operates on the qubits having the minimal distance to each other [Dij22]. The path determined is then applied in a bidirectional fashion [ZTH+24]. This release valve can then be triggered once a certain threshold of SWAP gates has been applied without making any progress. The Qiskit implementation sets this threshold ρ to $\rho = 10 \times \text{num_qubits}$ where num_qubits represents the number of qubits in the circuit.

4 Exhaustive K-SWAP-Sequence Search

The central operation of the SABRE algorithm is to repeatedly make gates in the front-layer physically adjacent (executable), by inserting individual SWAP gates [LDX19]. While this scales very well, it relies on a local heuristic. Considering individual SWAP gates and evaluating them on the local front-layer and extended-set can lead to decisions that are locally optimal, but suboptimal in a global scope.

This behavior can be motivated using the following example. For simplicity, assume we are using h_{basic} and a trivial initial mapping from logical to physical qubits as depicted on the right in Figure 4.1. Suppose we want to execute the circuit depicted on the left in Figure 4.1. According to SABRE, the initial front-layer contains only the first gate from the left operating on q_0 and q_2 . Let that gate be G_1 . The swap candidates on the coupling map are $\{Q_0, Q_1\}$, $\{Q_1, Q_2\}$ and $\{Q_2, Q_3\}$. h_{basic} yields a summed distance of 1 for the swap candidates $\{Q_0, Q_1\}$ and $\{Q_1, Q_2\}$. Applying $\{Q_2, Q_3\}$ results in a score of 3 since we increase the distance of the single front-layer gate by 1. Due to identical heuristic scores, we randomly choose $\{Q_1, Q_2\}$ as shown in the right of Figure 4.1. While this leads to a valid solution that allows us to execute G_1 , choosing $\{Q_0, Q_1\}$ would have resulted in the same, while also reducing the distance in favor of the second gate from the left (G_2).

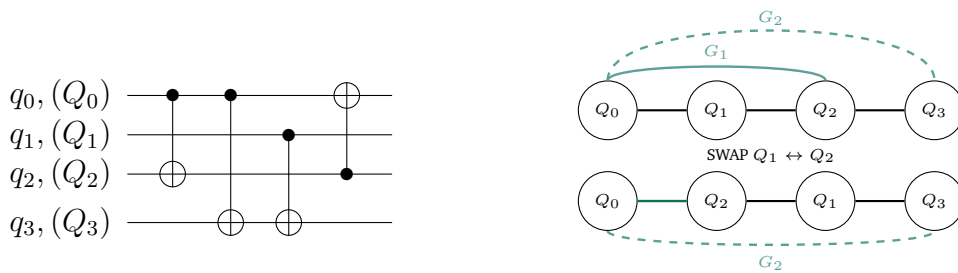


Figure 4.1: Example circuit to illustrate the local heuristic of the classical SABRE algorithm. The left subfigure depicts a quantum circuit, while the right subfigure depicts the topology over time (top to bottom).

K-SWAP SABRE extends the classical SABRE algorithm by determining the best sequence $\mathcal{S} = (\sigma_1, \dots, \sigma_k)$ with $|\mathcal{S}| \leq k$ SWAPs instead of scoring individual SWAPs. As demonstrated in Figure 4.1, considering individual SWAPs can lead to bad decisions where a

larger search space could have led to more informed decisions. Furthermore, it allows for choosing worse SWAP gates at the beginning of a sequence $\mathcal{S}$ that temporarily move the gates in the front-layer apart if these worse SWAPs lead to better SWAPs that we could not have reached without picking a seemingly bad SWAP at first. This is achieved by performing a DFS [Knu13] bounded by the sequence length k over SABRE's swap candidates to find the best sequence under a heuristic function h . Applying this methodology increases the considered search space and can thereby lead to improved global solutions. Furthermore, introducing a variable sequence length k allows to adjust the desired amount of computational resources one is willing to spend in the routing stage.

We will begin this chapter by proposing the *K-SWAP SABRE* algorithm. Then, we will continue to propose an updated heuristic function optimized for scoring candidate sequences. Finally, this chapter will be concluded with some considerations regarding algorithmic complexity.

4.1 Algorithm Formulation

This section explains the core algorithm of *K-SWAP SABRE*. On an abstract level, the algorithm works similar to the classical SABRE algorithm designed by Li et al. introduced in Chapter 3 [LDX19]. It executes gates and inserts SWAP gates until there are no gates left in the front-layer.

However, instead of applying individual SWAP gates, we are applying a sequence of SWAP gates $\mathcal{S} = (\sigma_1, \dots, \sigma_k)$ with $|\mathcal{S}| \leq k$. The algorithm is separated into three functions: Run contains the abstract main loop, ChooseBestSwaps constructs the sequences and scores them to determine the *best* sequence and ApplyPrefix replays a sequence prefix (an incomplete sequence) to explore the solution space and construct the candidate sequences.

Algorithm 4.1 defines the Run function. Note that the release valve was omitted from the following algorithmic description for the sake of brevity. We begin by initializing the output structures swapMap and gateOrder, as well as the insertionQueue. This queue is later used to insert a sequence of SWAP gates. Next, we initialize the front-layer structure and advance the front-layer as much as possible. That is, we execute all gates that do not require the insertion of any SWAP gates. Now, the function enters the main loop of the algorithm.

Analogous to SABRE, we terminate once the front-layer becomes empty. Next, we initialize the executeGateList which tracks the executable gates and currentSwaps that tracks the SWAPs that have been inserted without making progress (executing a gate).

Due to having advanced the front-layer at the initialization of the data structures, we are in a state in which no gates from the front-layer can be executed without the insertion of SWAPs (assuming that the circuit requires the insertion of at least one SWAP). Hence, we enter the inner loop that iterates as long as no gate from the front-layer can be executed ($\text{executeGateList} = \emptyset$). We then check whether there are items in the insertionQueue . In the first iteration, it will be empty. In that case, the algorithm will call the ChooseBestSwaps function. For now, assume that the function returns the best sequence $\mathcal{S}$ of $|\mathcal{S}| \leq k$ SWAPs. All SWAPs σ_i of sequence $\mathcal{S}$ are then pushed into

Algorithm 4.1 K-SWAP SABRE Routing: Run

```

1: function RUN( $k$ )
2:   swapMap  $\leftarrow \emptyset$ ; gateOrder  $\leftarrow \emptyset$ ; insertionQueue  $\leftarrow \emptyset$ 
3:   INITIALIZEFRONT( )
4:   ADVANCEFRONTLAYER( $\emptyset$ )
5:   while  $F \neq \emptyset$  do
6:     executeGateList  $\leftarrow \emptyset$ ; currentSwaps  $\leftarrow \emptyset$ 
7:     while executeGateList =  $\emptyset$  do
8:       if insertionQueue =  $\emptyset$  then
9:         for all  $\sigma \in \text{CHOOSEBESTSWAPS}(k)$  do
10:          insertionQueue.PUSH( $\sigma$ )
11:        end for
12:      end if
13:      if insertionQueue  $\neq \emptyset$  then
14:         $\sigma = \text{insertionQueue.POP}()$ 
15:        currentSwaps = currentSwaps  $\cup \sigma$ 
16:        APPLYSWAP( $\sigma$ )
17:        for all  $Q \in \sigma = (Q_0, Q_1)$  do
18:          node = EXECUTABLENODEONQUBIT( $Q$ )
19:          if node  $\neq \emptyset$  then
20:            executeGateList = executeList  $\cup$  node
21:            F.REMOVE(node)
22:          end if
23:        end for
24:      end if
25:    end while
26:    swapMap[executeGateList[0]] = currentSwaps
27:    ADVANCEFRONTLAYER(executeGateList)
28:  end while
29:  return swapMap, gateOrder
30: end function

```

the insertionQueue. The algorithm continues by checking whether the insertionQueue contains any items. If not, we continue into the next iteration of the inner loop which then calls the ChooseBestSwaps function again to determine the next candidate sequence. Now, assuming the InsertionQueue contains at least one SWAP gate, we pop the first SWAP from the queue, append it to the currentSwaps list and apply it. We then check whether the gates that were involved in the SWAP are executable now.

As mentioned previously, at most two gates from the front-layer can become executable when a SWAP gate is applied. Hence, it is sufficient to only check for those two. If the SWAP gate did not make any gate executable, the next iteration of the inner loop will insert another SWAP from the queue. If eventually, a gate from the front-layer becomes executable, it is added to the executeGateList and removed from the current front-layer. An executable gate will result in the algorithm to break out of the inner loop (since $\text{executeGateList} \neq \emptyset$).

Before going into the next iteration of the main loop, we insert the currentSwaps (all SWAPs that are required until the next front-layer gate is able to execute) into the swapMap. The swapMap resembles a hashmap. The value is indexed (keyed) by the first gate that was made executable by inserting the SWAPs in currentSwaps. Finally, we advance the front-layer based on the new SWAP gates we inserted and continue into the next iteration of the loop.

Up to now, we assumed that ChooseBestSwaps will return the best sequence of SWAPs. We will now discuss the implementation details on how a sequence is constructed and scored. This is shown in Algorithm 4.2. As mentioned before, the algorithm performs a stack-based DFS bounded by k over the set of swap candidates. While *K-SWAP SABRE* could also be implemented using recursion, we will not further discuss this implementation due to inefficient scaling and limited stack size.

The function begins by creating a snapshot of the initial state S_0 . This state includes the state of the front-layer, the required predecessors that need to be executed before a gate can execute, the current layout, the gate order, and the gates that have been executed so far. This snapshot allows for reliable backtracking after a sequence changed the application state. After that, the stack is initialized and an empty sequence is pushed onto the stack. In addition to the sequence itself, we also track the remaining sequence length. As a result, a stack item has the following structure: $\{\text{swap_sequence} = \emptyset, \text{remaining} = k\}$. The remaining depth indicates how far the construction of the sequence has progressed so far. With $k = 3$ and a sequence prefix that contains one SWAP, the remaining sequence length will be set to 2.

The main portion of this function iterates as long as there are items on the stack. Intuitively, this means that we run as long as not all sequences have been determined and scored. We start an iteration by popping the top-most item from the stack. After that, we load

Algorithm 4.2 K-SWAP SABRE Routing: ChooseBestSwaps

```

1: function CHOOSEBESTSWAPS( $k$ )
2:    $S_0 \leftarrow \text{CREATE\_SNAPSHOT}()$ 
3:    $\mathcal{S}^* \leftarrow \text{INIT\_BEST\_RECORD}()$ 
4:    $\text{stack} \leftarrow [(\text{swap\_sequence} = \emptyset, \text{remaining} = k)]$ 
5:   while  $\text{stack} \neq \emptyset$  do
6:      $(\mathcal{S}, r) \leftarrow \text{stack.POP}()$ 
7:      $\text{LOAD\_SNAPSHOT}(S_0)$ 
8:      $(\text{executedGates}, U_{\text{front}}, U_{\text{ext}}) \leftarrow \text{APPLY\_PREFIX}(\mathcal{S})$ 
9:      $\text{isLeaf} \leftarrow (r = 0) \vee (F = \emptyset)$ 
10:    if  $\text{isLeaf}$  then
11:       $h \leftarrow H(S_0.\pi, U_{\text{front}}, U_{\text{ext}}, \lambda)$ 
12:       $\text{UPDATE\_BEST\_RECORD}((\mathcal{S}, |\text{executedGates}|, h))$ 
13:      continue
14:    end if
15:     $\text{candidates} \leftarrow \text{COMPUTE\_SWAP\_CANDIDATES}()$ 
16:    for all  $\sigma \in \text{candidates}$  do
17:       $\text{stack.PUSH}((\mathcal{S} \cup \sigma, r - 1))$ 
18:    end for
19:  end while
20:   $\text{LOAD\_SNAPSHOT}(S_0)$ 
21:  return  $\mathcal{S}^*$ 
22: end function

```

the initial state S_0 to allow for convenient backtracking. Then, we start replaying the prefix sequence that we just popped from the stack. That is, we apply the SWAPs of the sequence one by one to the current application state. This replaying is handled by `ApplyPrefix` which we will discuss next. For now, assume that `ApplyPrefix` returns the number of executed gates, and a union of all front-layer and extended-set gates that were traversed while applying the sequence. We will further discuss the computation of these metrics in Section 4.2.

If the sequence that was just applied has resulted in a sequence of length k ($\text{remaining} = 0$) or an empty front-layer, we score the sequence ($\text{isLeaf} = \text{true}$). Scoring involves computing the heuristic h , as well as updating the best record seen so far. If the sequence is not ready for scoring yet, we compute the next set of swap candidates. This process is similar to how it was done in SABRE. That is, we determine all edges from the coupling map connected to a physical qubit that is currently mapped to a logical qubit involved in the front-layer. Now, to exhaustively explore all branches up to depth k , we extend the current sequence prefix with the different swap candidates and push each resulting

candidate sequence onto the stack and continue processing the next candidate sequence on the stack.

Once all sequences have been explored ($\text{stack} = \emptyset$), the initial snapshot is loaded to reset any changes the function made to the global state of the data structures and the *best* sequence is returned.

The last missing piece other than the heuristic itself is how a sequence prefix is replayed once it is popped from the stack. This is achieved using `ApplyPrefix` as defined in Algorithm 4.3. First, we populate the unions with the current gates in the front-layer and the extended-set before applying any SWAP gates from the sequence. Then we iteratively apply all the SWAP gates of the sequence. After applying a SWAP gate, we check whether it rendered a front-layer gate executable. We then advance the front-layer to prepare for the next iteration. All gates that have been executed due to the SWAP will be removed from the front-layer and extended-set unions. Furthermore, we will track the executed

Algorithm 4.3 K-SWAP SABRE Routing: `ApplyPrefix`

```

1: function APPLYPREFIX( $\mathcal{S}$ )
2:   executeGateList  $\leftarrow \emptyset$ ; advanced  $\leftarrow \emptyset$ 
3:   ( $U_{\text{front}}, U_{\text{ext}}$ )  $\leftarrow$  POPULATEUNION( )
4:   for all  $\sigma \in \mathcal{S}$  do
5:     APPLYSWAP( $\sigma$ )
6:     for all  $Q \in \sigma = (Q_0, Q_1)$  do
7:       node = EXECUTABLENODEONQUBIT( $Q$ )
8:       if node  $\neq \emptyset$  then
9:         executeGateList = executeGateList  $\cup$  node
10:        F.REMOVE(node)
11:      end if
12:    end for
13:    executed  $\leftarrow$  ADVANCEFRONTLAYER(executeGateList)
14:    for all node  $\in$  executed do
15:       $U_{\text{front}}$ .REMOVE(node)
16:       $U_{\text{ext}}$ .REMOVE(node)
17:      advanced.PUSH(node)
18:    end for
19:    executeGateList  $\leftarrow \emptyset$ 
20:     $U_{\text{front}} \leftarrow U_{\text{front}} \cup F$ 
21:     $U_{\text{ext}} \leftarrow U_{\text{ext}} \cup \text{GETEXTENDEDSET}( )$ 
22:  end for
23:  return (advanced,  $U_{\text{front}}, U_{\text{ext}}$ )
24: end function

```

gates in the advanced list. We clear the `executeGateList` and append the front-layer and the extended-set after the SWAP to the respective union. Then we proceed to insert the next SWAP of the sequence.

Through repeated application of this procedure, the algorithm incrementally routes the entire circuit. The next step is to design a heuristic function that evaluates candidate sequences in order to determine the *best* one.

4.2 Heuristic Function Design

After introducing the *K-SWAP SABRE* algorithm, we still need a heuristic function h to identify the *best* sequence $\mathcal{S}^*$ with $|\mathcal{S}^*| \leq k$ among a set of candidate sequences. Recall that the front-layer and extended-set are constructed based on a local state of the algorithm. The heuristic is then computed based on this local subset of gates that represent the measure of improvement. Hence, determining a measure that tracks the improvement across multiple SWAP gates can be difficult. Comparing the summed distances of front-layer gates and extended-set gates before and after inserting a SWAP gate lacks comparability.

Recall that a heuristic function needs to be admissible to guarantee optimality [NI02]. That is, the heuristic function never overestimates the solution. However, note that the nearest neighbor cost function used in SABRE and *K-SWAP SABRE* is not admissible. One reason for this is that multiple gates in the front-layer can be affected by a single SWAP. Hence, summing up the distances over all gates in the front-layer overestimates the true cost of SWAP gates required. Furthermore, optimizing for the minimization of distances in the front-layer does not necessarily imply that we are optimizing for the global optimization goal at all times.

Definition 4.2.1 (Heuristic Improvement) *Given the sequence length k , layout π with π_0 and $\pi_{\mathcal{S}}$ denoting the layouts before and after applying sequence $\mathcal{S}$, respectively, and a heuristic function h , determine the best sequence of swap candidates $\mathcal{S} := (\sigma_1, \dots, \sigma_k)$ with $|\mathcal{S}| \leq k$ s.t. $\Delta h := h(\pi_0) - h(\pi_{\mathcal{S}})$ is maximized.*

When designing a heuristic function based on the above definition of heuristic improvement, there are two central problems when comparing sequences of SWAPs instead of individual SWAP gates:

1. **Divergence in Branch Exploration.** As k increases, the gates in the front-layer and the extended-set diverge. Due to the local characteristic of the front-layer, computing the heuristic improvement of a candidate sequence can be difficult since

the set of gates that are accounted for in the computation of the heuristic might be distinct.

2. **Comparing different Sequences.** While the previous problem was concerned with inter-sequence comparability, the core problem persists when considering comparisons across sequences. Different sequences that explore different branches of the quantum circuit might compute the heuristic on distinct sets.

As introduced in Section 3.3, Li et al. implicitly proposed three heuristic functions with increasing complexity and different optimization goals. While h_{basic} and $h_{lookahead}$ minimize the number of SWAP gates, h_{decay} , which was only briefly mentioned as part of this thesis, tries to minimize the depth of the transpiled circuit. They mention three design goals they tried to optimize for when designing the original set of heuristic functions [LDX19]:

1. A heuristic function h should encourage the execution of gates by inserting SWAP gates that reduce the distances of gates in the front-layer F .
2. The heuristic should be equipped with a lookahead ability to account for gates beyond the current front-layer to improve the global mapping capabilities of the function h .
3. h should prioritize decisions that lead to an increased number of gates in the front-layer. This increases parallelism and can potentially lead to decreased depth of the transpiled circuit.

In the following, we propose a heuristic function that tries to address the beforementioned problems while implementing the above design objectives. However, to align with the overall objective to minimize SWAPs, we will not trade solutions with more SWAPs for less depth.

4.2.1 Heuristic Function

Recall that SABRE relies on the observation that the front-layer gates are the primary factor to ensure progress of the algorithm [LDX19]. Consequently, we do not want to dismiss this integral part of the algorithm design when it comes to adapting the heuristic to support a candidate sequence S with $|S| \leq k$. However, we need to solve the two central challenges to support exhaustive sequence search for the SABRE algorithm. The *Divergence in Branch Exploration* leads to computing less meaningful heuristic improvement across the sequence as the sequence length k increases. For similar reasons, we need to ensure that the heuristic improvement of two different sequences is comparable, even in cases

when different branches of the DAG are being explored. To address these issues, we introduce the heuristic function $h(\mathcal{S})$ shown in Equation 4.1.

$$h(\mathcal{S}) = (-|X_{\mathcal{S}}|, \eta(\mathcal{S}), |\mathcal{S}|), \quad h : \mathcal{S} \rightarrow \mathbb{Z}^- \times \mathbb{R} \times \mathbb{N} \quad (4.1)$$

The heuristic function $h(\mathcal{S})$ is defined as a lexicographic vector with three components. Let $X_{\mathcal{S}}$ denote the set of gates executed by applying the candidate sequence $\mathcal{S}$. To ensure progress on a global level, the primary component of the heuristic $|X_{\mathcal{S}}|$ measures the total number of gates that can be executed when $\mathcal{S}$ is applied. This includes not only the (at most two) gates in the front-layer, but also any two-qubit gates in the extended-set that become executable once a front-layer gate is applied, without requiring additional SWAPs. In this way, the primary component implicitly captures how many gates remain to be executed until the circuit is fully routed. Since executing more gates corresponds to greater progress, we multiply the count by -1 , allowing the lexicographic vector to be minimized while keeping smaller values preferable in the subsequent components.

The secondary term $\eta(\mathcal{S})$ is defined as the normalized improvement of applying a sequence $\mathcal{S}$ and builds upon $h_{lookahead}$ to compare multiple candidate sequences that execute the same number of gates $|X_{\mathcal{S}}|$. A smaller value for $\eta(\mathcal{S})$ implies a greater reduction of SWAP gates in the front-layer and extended-set. In practice, we introduce a small threshold $\varepsilon > 0$ when comparing heuristic values. This ensures that sequences whose $\eta(\mathcal{S})$ differ only by an insignificant amount are treated as equivalent, avoiding over-sensitivity to tiny numerical differences and promoting more stable and consistent selection of candidate sequences. We will further discuss the normalized improvement in Section 4.2.3.

The tertiary term of the lexicographic vector represents the sequence length $|\mathcal{S}|$. That is, in case multiple candidate sequences have the same primary heuristic $|X_{\mathcal{S}}|$ and secondary heuristic $\eta(\mathcal{S})$, the shorter sequence is considered to be better. Intuitively, this implies that for two sequences $\mathcal{S}_a$ and $\mathcal{S}_b$ with $|\mathcal{S}_a| < |\mathcal{S}_b|$, the shorter sequence is considered better if this sequence $\mathcal{S}_a$ is able to execute the same number of gates as a second sequence $\mathcal{S}_b$ ($|X_{\mathcal{S}_a}| = |X_{\mathcal{S}_b}|$) and is able to move the gates in the front-layer together in the same way the other sequence does ($\eta(\mathcal{S}_a) = \eta(\mathcal{S}_b)$). This ensures we are optimizing for the least number of SWAP gates inserted during the algorithm.

Example 4.2.1 (Comparing Sequence Scores) We compare lexicographic sequences by their score triple $(-|X_{\mathcal{S}}|, \eta(\mathcal{S}), |\mathcal{S}|)$. Recall that a better solution is characterized by minimizing the lexicographic vector. Thus, for sequences $\mathcal{S}_A$ and $\mathcal{S}_B$,

$$(-|X_{\mathcal{S}_A}|, \eta(\mathcal{S}_A), |\mathcal{S}_A|) \succ (-|X_{\mathcal{S}_B}|, \eta(\mathcal{S}_B), |\mathcal{S}_B|)$$

if $|X_{\mathcal{S}_A}| > |X_{\mathcal{S}_B}|$, or if $|X_{\mathcal{S}_A}| = |X_{\mathcal{S}_B}|$ and $\eta(\mathcal{S}_A) < \eta(\mathcal{S}_B)$, or if $|X_{\mathcal{S}_A}| = |X_{\mathcal{S}_B}|$, $\eta(\mathcal{S}_A) = \eta(\mathcal{S}_B)$, and $|\mathcal{S}_A| < |\mathcal{S}_B|$.

If for two different sequences S_a and S_b , all terms of the lexicographic vector are equal, a random decision determines the better sequence. As mentioned in Section 3.3, this randomness is introduced to prevent getting stuck at local minima across multiple executions of the algorithm.

Remark 4.2.1 (Considerations on Optimality for $|S^*| \leq k$) *The combination of the minimization of the primary term $-|X_S|$ and the tertiary term $|S|$ ensures that, within the explored candidate space, the algorithm prefers sequences that execute all gates with as few SWAPs as possible. Specifically, assume that a sequence S^* exists in the explored space that executes all gates with minimal SWAP count $|S^*| \leq k$. In terms of the heuristic function $h(S)$, all sequences that execute all gates have the same primary score $|X_S|$, and the secondary heuristic $\eta(S)$, which accounts for gates not yet executed excludes the entire set of gates. Consequently, the tertiary term $|S|$ ensures that the shortest sequence among the explored candidates is preferred. We emphasize that this guarantee holds only for sequences present in the explored candidate space, which depends on the swap candidate generation strategy.*

Remark 4.2.2 (Alternative Formulation) *The heuristic function of Equation 4.1 can alternatively be expressed as the polynomial shown in Equation 4.2.*

$$h(S) = -\omega^3|X_S| + \omega^2\eta(S) + \omega|S|. \quad (4.2)$$

Due to being able to tweak the weights, this equation allows for a more nuanced approach to making routing decisions. Since fine-tuning the weights was out of scope for this thesis, we continue to use the lexicographic formulation introduced before. To mimic the behavior of Equation 4.1, ω can be determined such that the primary term $-|X_S|$ outweighs any possible change in the secondary term $\eta(S)$ and tertiary term $|S|$. Analogously, the secondary term $\eta(S)$ has to outweigh any possible change in the tertiary term $|S|$. A trivial choice for ω could e.g. be $\omega = 100$. These weights can be fine-tuned by putting tight bounds on the terms to estimate their impact.

The heuristic proposed above represents a progress-oriented function. It incorporates a lookahead component while taking sequence complexity into account. Next, we will discuss how heuristic improvement can be modeled and applied to candidate sequences.

4.2.2 Addressing Divergence in Branch Exploration

Now that we introduced the heuristic function $h(S)$, we will continue to discuss how it solves the two major problems mentioned before. We begin with the question on

how heuristic improvement can be tracked in scenarios when the front-layer and the extended-set diverge with increasing sequence length k .

Let $\mathcal{G}$ be the set of two-qubits gates in the DAG, π denote a layout (logical $\mapsto$ physical mapping). For a gate $g = (a, b) \in \mathcal{G}$, recall function $d(q_a, q_b)$ that denotes the physical graph distance between the two logical qubits of gate g under layout π with $d \geq 0$ as introduced in Chapter 3. Recall that the two qubits a gate operates on have to be adjacent to be able to execute it ($d(q_a, q_b) = 1$). The number of SWAP gates required to make two physical qubits adjacent can be defined as shown in Equation 4.3.

$$\phi(d) := \max\{0, d(q_a, q_b) - 1\} \quad (4.3)$$

For any finite subset $U \subset \mathcal{G}$ and a given layout π , Equation 4.4 defines the aggregate number of SWAP gates required to make all gates in U adjacent. Note that this definition represents a generalized version of h_{basic} .

$$L_U(\pi) := \sum_{g=(a,b) \in U} \phi(d(q_a, q_b)) \quad (4.4)$$

For a fixed SWAP sequence $\mathcal{S}$ with $|\mathcal{S}| \leq k$, let π_0 be the starting layout before applying the sequence and let $\pi_{\mathcal{S}}$ be the layout after replaying the whole sequence of SWAPs. The aggregate lower bounds for U_0 and $U_{\mathcal{S}}$ can then be defined as $L_{U_0}(\pi_0)$ and $L_{U_{\mathcal{S}}}(\pi_{\mathcal{S}})$, respectively. $L_{U_0}(\pi_0)$ represents the initial aggregate while $L_{U_{\mathcal{S}}}(\pi_{\mathcal{S}})$ represents the aggregate after applying the sequence. The absolute improvement can then be expressed as $\Delta L(\mathcal{S})$ as it is stated in Equation 4.5. It represents the absolute improvement when applying sequence $\mathcal{S}$.

$$\Delta L(\mathcal{S}) := L_{U_0}(\pi_0) - L_{U_{\mathcal{S}}}(\pi_{\mathcal{S}}) \quad (4.5)$$

Naively applying a sequence of k consecutive SWAPs and computing the absolute improvement using methods introduced in SABRE will result in the subsets of gates U to diverge. When computing the absolute improvement $\Delta L(\mathcal{S})$, this membership drift leads to two different sets U_0 and $U_{\mathcal{S}}$ being used for π_0 and $\pi_{\mathcal{S}}$ when computing the aggregate improvement, respectively. The larger k , the bigger the divergence of branches and hence the bigger the difference of the subset U which is utilized as a base to compute the absolute improvement of a sequence.

So far, we assumed a finite subset $U \subset \mathcal{G}$. In the case of SABRE, let the subsets considered be the front-layer $F \subset \mathcal{G}$ and the extended-set $E \subset \mathcal{G}$. Consequently, a longer sequence length $\mathcal{S}$ results in a bigger divergence of F and E . As alluded to before, a single SWAP

gate can cause the execution of $\mathcal{O}(2 + |E|)$ gates. Hence, the insertion of a single SWAP can lead to two distinct sets F_0 and F_S (in the case that $|F| = 2$) and E_0 and E_S .

Now, to address this divergence of subsets when computing the absolute improvement $\Delta L(\mathcal{S})$ across a sequence of k SWAPs, we instead consider the union of all front-layers and extended-sets as shown in Equation 4.6.

$$U_F(\mathcal{S}) := \bigcup_{t=0}^k F_t, \quad U_E(\mathcal{S}) := \bigcup_{t=0}^k E_t \quad (4.6)$$

For a subset $X \subset \mathcal{G}$ of gates executed while applying sequence $\mathcal{S}$, define the residual sets

$$R_F(\mathcal{S}, X) := U_F(\mathcal{S}) \setminus X, \quad R_E(\mathcal{S}, X) := U_E(\mathcal{S}) \setminus X \quad (4.7)$$

These sets quantify the front-layer and extended-set gates that remain unexecuted after applying $\mathcal{S}$, thereby measuring the extent to which the candidate sequence $\mathcal{S}$ reduces the distance cost for subsequent iterations. Intuitively, this indicates how much sequence $\mathcal{S}$ improved the front-layer and extended-set distances.

Suppose we are now using this $R_F(\mathcal{S}, X)$ and $R_E(\mathcal{S}, X)$ to compute the aggregates $L_{R_F}(\pi)$ and $L_{R_E}(\pi)$, such that

$$\Delta L_{R_F}(\mathcal{S}) := L_{R_F}(\pi_0) - L_{R_F}(\pi_S), \quad \Delta L_{R_E}(\mathcal{S}) := L_{R_E}(\pi_0) - L_{R_E}(\pi_S) \quad (4.8)$$

This leads to the absolute improvement $\Delta L(\mathcal{S})$ being computed on the same subset of gates before and after applying the sequence $\mathcal{S}$. This allows for an accurate computation of absolute improvements in a sequence. At the same time, note that this prevents us from applying relative scoring as proposed by Zou et al. to the heuristic function [ZTH+24]. This is due to the dependency on the union of front-layer and extended-set gates across the entirety of the sequence before any heuristic can be computed.

4.2.3 Normalized Improvement

Now that we addressed the divergence of the front-layer and extended-set gates in a single candidate sequence, we continue to address the second problem of comparing two different sequences that can potentially explore different regions of the DAG. Comparing the raw improvements $\Delta L(\mathcal{S})$ of different candidate sequences directly is not meaningful, since sequences may affect different sets of gates. To address this, we apply

per gate normalization, which maps each sequence’s raw improvement to a uniform unit representing the average SWAP reduction per gate. Building on the definitions introduced above – the distance function $\phi(d)$, the aggregate distance $L_U(\pi)$, and the absolute improvement $\Delta L(\mathcal{S})$ over residual sets $R_F(\mathcal{S}, X)$ and $R_E(\mathcal{S}, X)$ – we derive the normalized improvement heuristic $\eta(\mathcal{S})$ for the unions of front-layer and extended-set gates, as shown in Equation 4.9. Intuitively, this corresponds to the arithmetic mean of improvement per gate, allowing fair comparison between sequences of different lengths and gate coverage. Note that a smaller value of $\eta(\mathcal{S})$ corresponds to a bigger reduction in SWAPs and thus to a better candidate sequence.

$$\begin{aligned}
\eta(\mathcal{S}) &= \frac{1}{|R_F(\mathcal{S}, X)|} \sum_{g \in R_F(\mathcal{S}, X)} \left(\phi(d_{\pi_0}(g)) - \phi(d_{\pi_{\mathcal{S}}}(g)) \right) \\
&\quad + \lambda \cdot \frac{1}{|R_E(\mathcal{S}, X)|} \sum_{g \in R_E(\mathcal{S}, X)} \left(\phi(d_{\pi_0}(g)) - \phi(d_{\pi_{\mathcal{S}}}(g)) \right) \\
&= \underbrace{\frac{\Delta L_{R_F}(\mathcal{S})}{|R_F(\mathcal{S}, X)|}}_{\text{basic component}} + \lambda \cdot \underbrace{\frac{\Delta L_{R_E}(\mathcal{S})}{|R_E(\mathcal{S}, X)|}}_{\text{lookahead component}}
\end{aligned} \tag{4.9}$$

Now, two sequences can be compared in that e.g. sequence $\mathcal{S}_1$ yields a larger per gate improvement than $\mathcal{S}_2$. That statement can be made even though $R_{F_1} \neq R_{F_2}$ and $R_{E_1} \neq R_{E_2}$. In other words, normalization converts an absolute, context dependent quantity into an arithmetic mean per gate which are comparable even when their underlying sets differ. This normalization removes bias toward sequences with larger union cardinality and renders secondary scores $\eta(\mathcal{S})$ comparable across sequences. A smaller value for $\eta(\mathcal{S})$ unambiguously implies a greater mean SWAP reduction per gate.

However, due to applying a nearest neighbor cost function as done in SABRE, it is crucial to state that $\Delta L(\mathcal{S})$ and thus $\eta(\mathcal{S})$ is computed based on per gate lower bounds $\phi(d)$. This implies that $\eta(\mathcal{S})$ does potentially overcount the effective number of distinct SWAPs required to route the circuit. This is due to several gates potentially being able to execute using the same SWAP gate. So a larger $\eta(\mathcal{S})$ does not strictly mean fewer SWAPs in the final routing – it means a larger expected per gate decrease in the per gate lower bound. In other words, $\eta(\mathcal{S})$ is a conservative per gate proxy for SWAP improvement, not an admissible SWAP improvement metric.

Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be two candidate sequences, and let $\eta(\mathcal{S}_1)$, $\eta(\mathcal{S}_2)$ be their normalized improvements as defined above. Then $\eta(\mathcal{S}_i)$ is the average improvement of the SWAP lower bound per gate. If $\eta(\mathcal{S}_1) < \eta(\mathcal{S}_2)$, then, on average, $\mathcal{S}_1$ improved the per gate SWAP lowerbound more than $\mathcal{S}_2$.

4.3 Algorithmic Complexity

Let N be the number of physical qubits in the device, G the number of two-qubit gates in the circuit (DAG nodes), and n the size of the front layer. Since every front-layer gate involves exactly two qubits, the maximum size of the front layer is bounded by

$$n \leq \left\lfloor \frac{N}{2} \right\rfloor.$$

The *K-SWAP SABRE* algorithm generalizes SABRE by considering sequences of SWAPs of length at most k , rather than inserting single SWAPs. Let b denote the maximum number of swap candidates per step. In the worst case, $b = \Theta(n)$ (each front-layer qubit generates candidates to its neighbors in the coupling graph), while for low-degree hardware topologies $b = \mathcal{O}(1)$. The algorithm explores all candidate sequences of length up to k , which gives a total of

$$\sum_{i=0}^k b^i = \frac{b^{k+1} - 1}{b - 1} = \Theta(b^k), \quad b > 1$$

candidate sequences per decision. Each candidate sequence corresponds to applying a prefix of SWAPs and evaluating the heuristic, which requires layout updates and front/extended-set computations. These operations are dominated by the front-layer size and take $\Theta(n)$ time per gate. Hence, the cost of a single decision is

$$\mathcal{O}(b^k \cdot n).$$

A single sequence does not necessarily enable execution of a front-layer gate. In practice, several sequences may be required before two qubits of a gate are brought together. Following the reasoning of SABRE [LDX19], the number of SWAPs required to satisfy one two-qubit gate is bounded by the diameter D of the coupling graph. For a two-dimensional grid of N qubits, this diameter is $D = \mathcal{O}(\sqrt{N})$. Thus, the cost to satisfy one gate is bounded by

$$\mathcal{O}(D \cdot b^k \cdot n),$$

and summing over all G gates in the circuit gives the total worst-case runtime

$$T_{\text{total}} = \mathcal{O}(G \cdot D \cdot b^k \cdot n).$$

If we pessimistically set $b = \mathcal{O}(n)$ and $D = \mathcal{O}(\sqrt{N})$, this simplifies to

$$T_{\text{total}} = \mathcal{O}(G \cdot n^{k+1} \cdot \sqrt{N}).$$

Since in the worst case the front layer may involve all qubits, i.e. $n = \Theta(N)$, this bound can be relaxed further to

$$T_{\text{total}} \leq \mathcal{O}(G \cdot N^{k+1.5}).$$

Two important consequences follow from this analysis. First, N^k – and therefore the running time – grows exponentially in k , which means that k must remain small in practice. Second, if k is treated as a fixed constant (e.g., $k = 2$ or 3), then N^k is a polynomial in N of degree k , and the algorithm runs in polynomial time in the problem size (with a degree that grows with k). In contrast, if k is allowed to grow with N , then the number of sequences explored becomes exponential, approaching the complexity of exact mapping methods. Exact algorithms explore the entire permutation space of size $\mathcal{O}(N!)$ [ZPW18], which is infeasible for realistic circuits. The K-SWAP approach therefore provides a tunable compromise: by fixing k to a small constant, one obtains a tractable polynomial-time algorithm that explores a richer search space than SABRE, thereby improving mapping quality while avoiding the infeasibility of exact methods.

5 Experimental Evaluation

In this chapter, we will first briefly discuss the implementation of the proposed algorithm. We then want to evaluate the effects of the *K-SWAP SABRE* algorithm compared to Qiskit’s SABRE implementation. To achieve that, we run a benchmark suite (*Benchpress*) against both implementations.

5.1 Implementation

The algorithm is implemented in the Rust¹ programming language and the code is available on GitHub². Analogously to how Qiskit³ is architected, the preprocessing is implemented in Python⁴. The preprocessing includes loading and parsing the OpenQASM file and applying several transpiler stages to prepare the circuits. Rust is then called from Python using the PyO3⁵ library that allows for creating native Python extensions written in Rust to construct the DAG, as well as to execute the core algorithm outlined in Chapter 4. For the benchmarks, Qiskit’s *Benchpress*⁶ is utilized and extended by a custom *transpiler gym* to support the implementation of this thesis. This fork of *Benchpress* that implements this extension can be found on GitHub as well⁷.

The transpilation of a quantum circuit includes more than just the routing stage that was tackled as part of this thesis. Furthermore, we made some assumptions at the beginning of Chapter 3, e.g. no three- or more qubit gates and no barriers. The following transpiler stages will help to transform the circuit into a valid input for our algorithm while adhering to the self-imposed constraints.

¹<https://www.rust-lang.org/>

²<https://github.com/JakWai01/microscope>

³<https://github.com/Qiskit/qiskit>

⁴<https://www.python.org/>

⁵<https://github.com/PyO3/pyo3>

⁶<https://github.com/Qiskit/benchpress/>

⁷<https://github.com/JakWai01/benchpress>

1. **Unroll3QOrMore.** This transpilation stage expands three- or more qubit gates recursively until only one or two-qubit gates remain in the circuit. That is, the circuit gets converted into a form that only contains one and two-qubit gates⁸.
2. **SabreLayout.** As mentioned in Section 3.3, SABRE also proposed a solution for applying an efficient initial mapping by running SABRE once and then taking the final layout as an initial layout for the actual pass⁹. Since this thesis was primarily concerned with the routing stage, the initial mapping was not modified. This stage applies the initial bidirectional mapping strategy of the SABRE algorithm. Doing so primarily helps in removing a funnel of SWAPs at the beginning of the routing that moves the front-layer gates to the middle of the circuit. Since the gates in the funnel can account for substantial amounts of gates, a potential improvement in the routing strategy itself is harder to observe if that funnel is present. Hence, this stage removes it.
3. **FullAncillaAllocation.** This stage is a prerequisite for the next stage. It ensures that all qubits that are not utilized in the initial mapping of the circuit will still get allocated as so-called ancilla qubits. This ensures that the entire topology is initialized correctly¹⁰.
4. **ApplyLayout.** This transpilation stage is responsible for transforming the circuit by applying the previously configured layout to all the physical qubits of the topology¹¹. For this, we assign the logical qubits of the circuit to the physical qubits of the topology.
5. **RemoveBarriers.** This transpilation stage removes all barriers defined on the initial quantum circuit¹². As mentioned previously in Chapter 3, barriers are a way to segment the circuit into multiple partitions that will be treated independently. This transformation will hence change the semantics of the input circuit but provide us with a set of commonly defined circuits to work with.

5.2 Benchmark Setup

In the following, we will evaluate the *K-SWAP SABRE* algorithm on a publicly available data-set of Hamiltonian circuits provided by HamLib, short for Hamiltonian Library

⁸<https://quantum.cloud.ibm.com/docs/en/api/qiskit/qiskit.transpiler.passes.Unroll3qOrMore>

⁹<https://quantum.cloud.ibm.com/docs/en/api/qiskit/qiskit.transpiler.passes.SabreLayout>

¹⁰<https://quantum.cloud.ibm.com/docs/en/api/qiskit/qiskit.transpiler.passes.FullAncillaAllocation>

¹¹<https://quantum.cloud.ibm.com/docs/en/api/qiskit/qiskit.transpiler.passes.ApplyLayout>

¹²<https://quantum.cloud.ibm.com/docs/en/api/qiskit/qiskit.transpiler.passes.RemoveBarriers>

[SMH+24]. In quantum computing, a Hamiltonian is an operator that indicates the energy and interactions of a system. Consequently, a computational problem is mapped to the physical qubits of the system. The optimal solution to the problem lies in the ground state of the Hamiltonian [Llo96]. In gate based quantum computers, the Hamiltonian dynamics are simulated utilizing quantum gates.

The data encompasses various quantum circuits ranging from 2 to 930 qubits and up to 10^6 two-qubit gates. It covers various problems from different fields such as condensed matter physics, chemistry, binary optimization and discrete optimization [SMH+24]. A total of one hundred different circuits are included in this benchmark. For the following benchmark, each circuit will be executed on three topologies: linear, square and heavy-hex. Since this thesis is concerned with optimizing for the number of SWAP gates, including the all-to-all topology does not provide any information since no SWAPs need to be inserted in a fully connected graph. Hence, a total of 300 tests are considered for the following benchmarks.

All benchmarks were conducted on an AMD Ryzen 9 3950X 16-Core processor with 128 Gb of memory running Ubuntu 24.04.3 LTS. The *Benchpress* library that is build on top of *pytest* is utilized to execute the benchmarks [NSB+25]. The tests are executed in parallel on all 16 cores. Since this thesis is not concerned with benchmarking the exact runtime performance and a single instance of the *K-SWAP SABRE* algorithm does not require multi-core processing, running the benchmark suite in parallel does not influence the results. However, we will consider the relative runtime implications of increasing the sequence length k in Section 5.3.2. In the following experimental evaluation, we compare the implementation of *K-SWAP SABRE* that was implemented as part of this thesis with the SABRE implementation as it is found in Qiskit v2.1.1 [JTK+24]. As alluded to before, the Qiskit implementation contains the improvements by Zou et al. [ZTH+24].

To ensure comparability, both tests execute a single trial on the circuit and layout prepared by applying the beforementioned transpiler stages (see Section 5.1). After that, the Qiskit implementation is tested by executing the *SabreSwap* transpiler stage. The stage is configured to use $h_{lookahead}$ and a single execution is enforced by passing `trials = 1` to the *SabreSwap* transpiler stage. When testing *K-SWAP-SABRE*, we invoke the algorithm manually on the prepared quantum circuit instead of relying on a pre-built transpiler stage. Both implementations use a fixed extended-set size of 20 and a weight $\lambda = 0.5$ for weighing the extended-set portion of the heuristic function. After every test, the *CheckMap* transpiler stage is executed. This stage verifies that all two-qubit gates are executed on adjacent physical qubits on the final mapping. With this, we verify the validity of a solution.

Figure 5.1 illustrates the distribution of test circuits that are part of the benchmark suite. By plotting the number of qubits and the resulting SWAP gates when executing the circuits using Qiskit’s *SABRE*, Figure 5.1 visually conveys the range of the tests included. As can

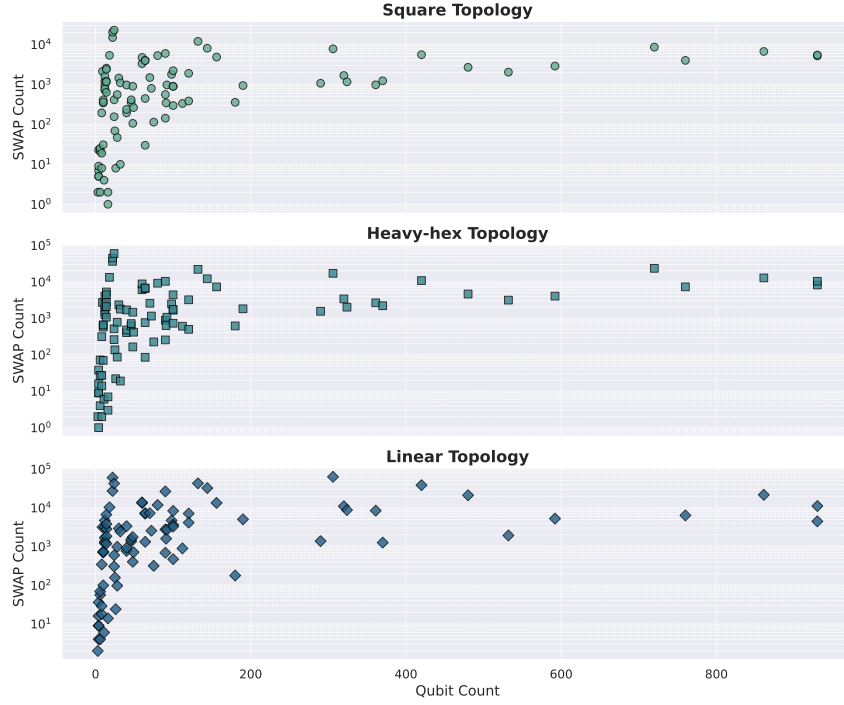


Figure 5.1: Distribution of test data in the HamLib *AbstractTranspile* suite [SMH+24]. Each point represents a single test circuit. The x-axis shows the number of qubits in the circuit, while the y-axis indicates the number of SWAP gates inserted by Qiskit’s SABRE during routing.

be observed, most tests are circuits with less than 150 qubits. This reflects the current state of the quantum hardware. For larger circuits and problem sizes to become more common, advanced error correction techniques are required.

5.3 Results and Discussion

This section evaluates *K-SWAP SABRE* against Qiskit’s SABRE across various hardware topologies and sequence lengths k . We first present aggregate benchmark results, examining how SWAP insertion varies with the connectivity of the topology and sequence length k . Next, we conduct a focused case study on three circuits, each with different behavior, to analyze the effect of increasing k on SWAP count, circuit depth and runtime. We also demonstrate the practical trade-offs between solution quality and computational cost.

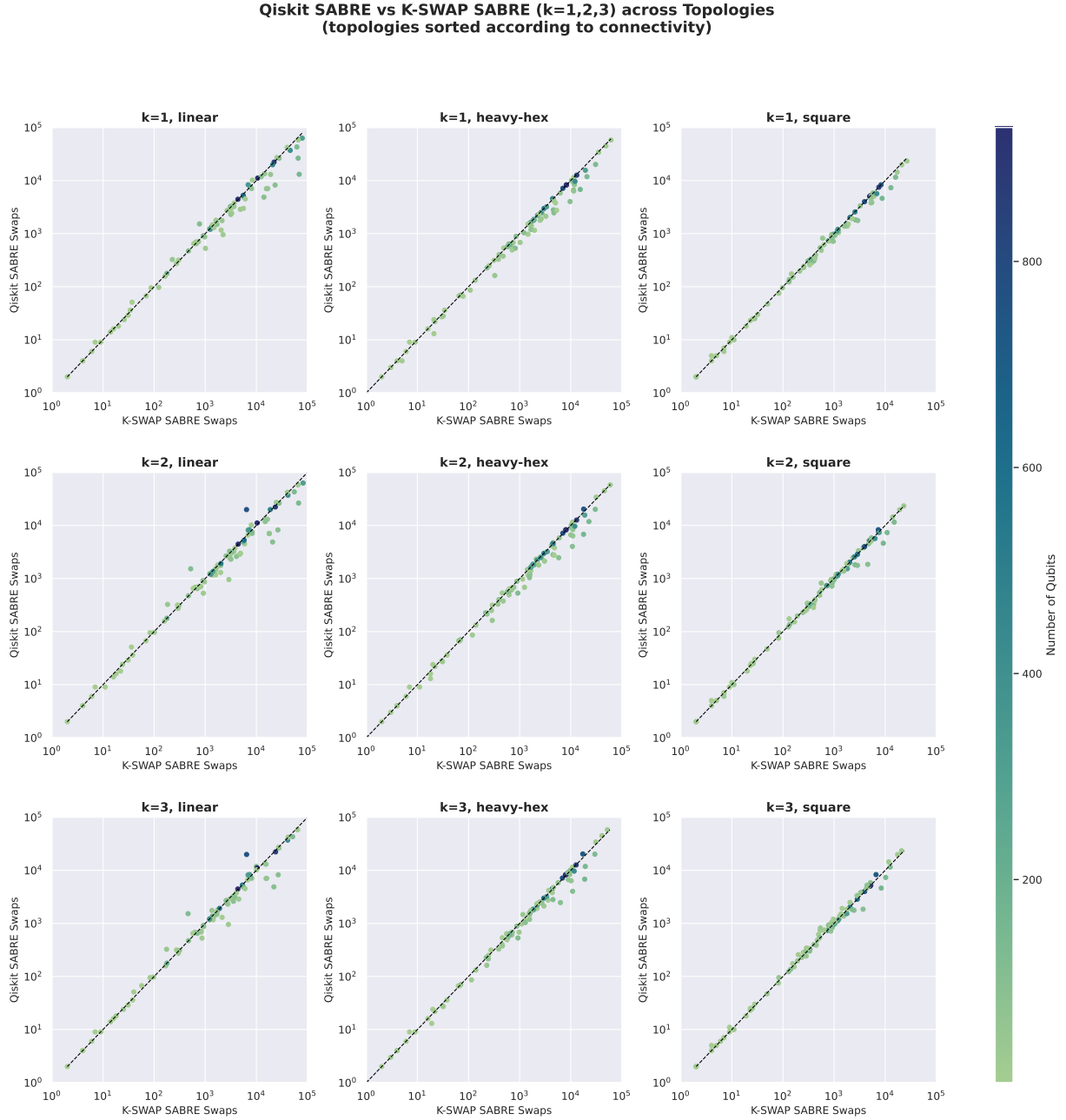


Figure 5.2: Mean SWAP counts for *K-SWAP SABRE* (x-axis) vs *Qiskit SABRE* (y-axis) across topologies and sequence length k . Each point is the mean over $n=10$ independent runs. Color encodes number of qubits in the circuit. Both axes are log-scaled.

5.3.1 Swap Count Comparison: K-SWAP vs SABRE

Figure 5.2 visualizes the benchmark results across topologies ordered by increasing connectivity (left to right) and increasing sequence length k (top to bottom). Due to the exponential growth in runtime with increasing k , a maximum of $k = 3$ was feasible for all 300 test cases (ranging from 2 to 930 qubits). Additionally, a test is terminated after 1 hour. Two circuits were excluded due to the timeout: one MaxKCut and one Fermi-Hubbard instance. Each data point represents the average result after 10 test executions ($n = 10$). While the x-component visualizes the number of SWAPs *K-SWAP SABRE* inserted, the y-axis visualizes the Qiskit SABRE result. Tests for which the SABRE result is equal to the *K-SWAP SABRE* result lie on the diagonal line. A point above the line indicates less SWAPs being inserted by *K-SWAP SABRE* than by Qiskit’s SABRE. Analogously, a point below the line represents a test for which Qiskit’s SABRE produces less SWAPs in the solution. The darker the color of a point, the more qubits are part of the test circuit. The data is plotted on a log-scale.

Overall, *K-SWAP SABRE* and Qiskit SABRE produce similar SWAP counts, but *K-SWAP* yields tangible improvements on higher-connectivity topologies at larger k . On low connectivity topologies (linear), Qiskit SABRE often outperforms *K-SWAP SABRE*, particularly at $k = 1$. In contrast, for heavy-hex and square topologies, the distribution tightens around or above the diagonal as k increases, with *K-SWAP SABRE* inserting fewer SWAPs than Qiskit in a substantial number of instances. The effect becomes more pronounced as k increases, especially at $k = 3$. The best results can be observed on the square topology at $k = 3$ in which the majority of points move above the line. Also note that while most points are located close to the diagonal, the log-scaling means that the solutions can still differ by a notable margin. The results suggest that *K-SWAP SABRE*’s relative advantage tends to increase with both topology connectivity and sequence length k , particularly for $k \geq 3$.

In summary, Figure 5.2 shows that *K-SWAP SABRE* and Qiskit SABRE are broadly aligned in SWAP insertion behavior, but that *K-SWAP SABRE* can outperform SABRE for larger and more connected topologies, particularly at higher sequence lengths k . This trend suggests that while *K-SWAP SABRE* is not consistently superior, it offers benefits as the connectivity increases. Additionally, the data suggests that increasing the sequence length k leads to an increased quality of results.

Figure 5.3 shows the distribution of relative SWAP improvements across the three topologies. Distributions shown are post-filtering (IQR rule). The three panels reveal topology-dependent behavior: on the square topology the median improvement shifts from -5.8% at ($k = 1$) to approximately -0.2% at ($k = 2$) and 0.0% at ($k = 3$), indicating a clear upward trend as the sequence length increases. Heavy-hex exhibits a similar but less pronounced trend (median $\approx$ -3.0%) at $k = 1$ to $\approx$ -0.2% at $k = 3$, whereas the linear

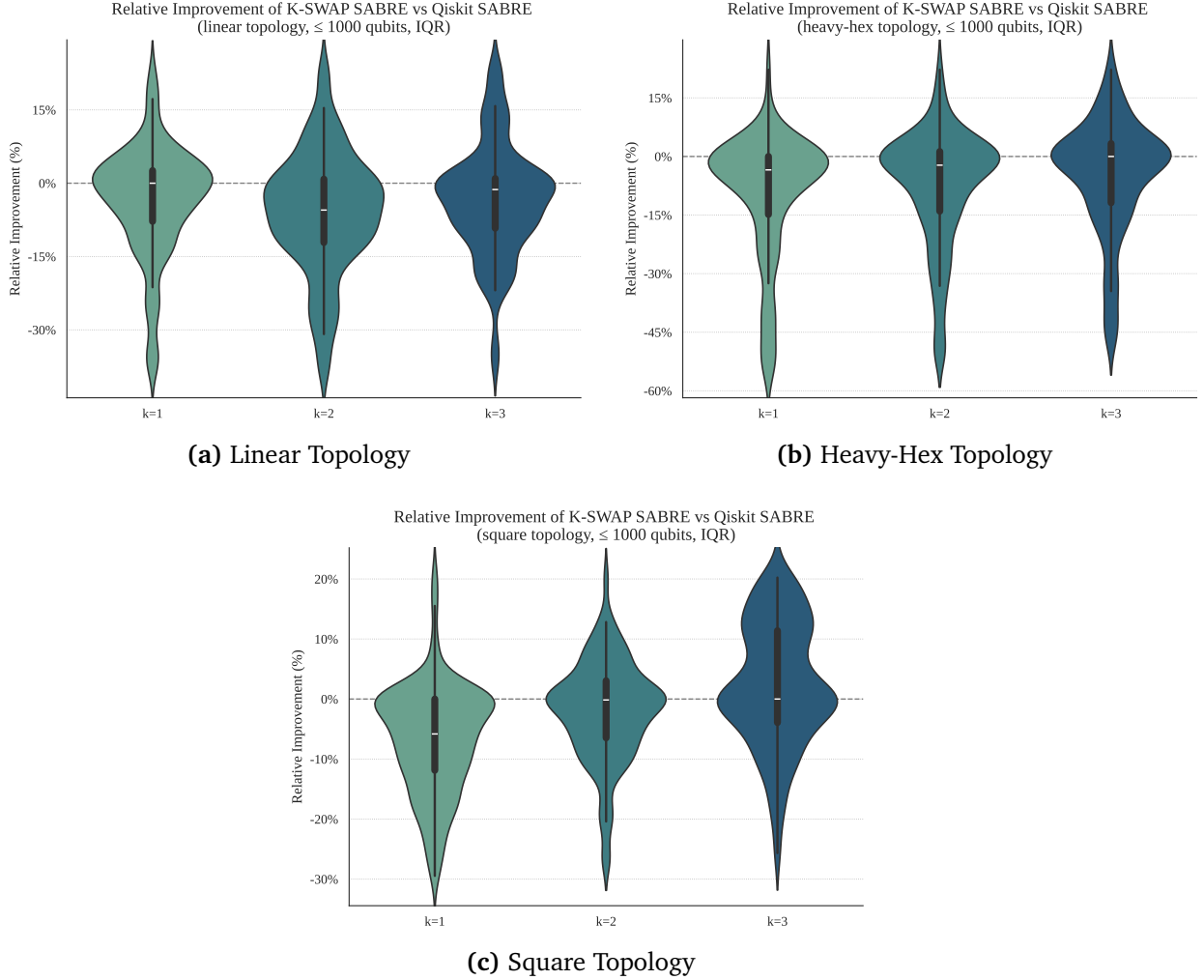


Figure 5.3: Distribution of relative SWAP improvement on linear, heavy-hex and square topology for $k \in \{1, 2, 3\}$. Violins show filtered distributions (IQR rule).

topology does not display a systematic improvement with larger k . These trends highlight that both increasing the connectivity and increasing k contributes to better results.

The overall median across all topologies remains near 0%, indicating that *K-SWAP SABRE* does not always outperform Qiskit SABRE. However, the distributions are wide, with improvements up to 20% and degradations down to -30%. Increasing k reduces the negative tail and shifts the distribution upward. Notably, $k = 3$ in combination with the square topology captures a large fraction of positive improvements, suggesting that increasing sequence lengths k improves the quality of solutions.

To explore the effect of higher k , Figure 5.4 isolates circuits with fewer than 40 qubits on the square topology, allowing evaluation of up to $k = 4$. At $k = 1$, median performance is

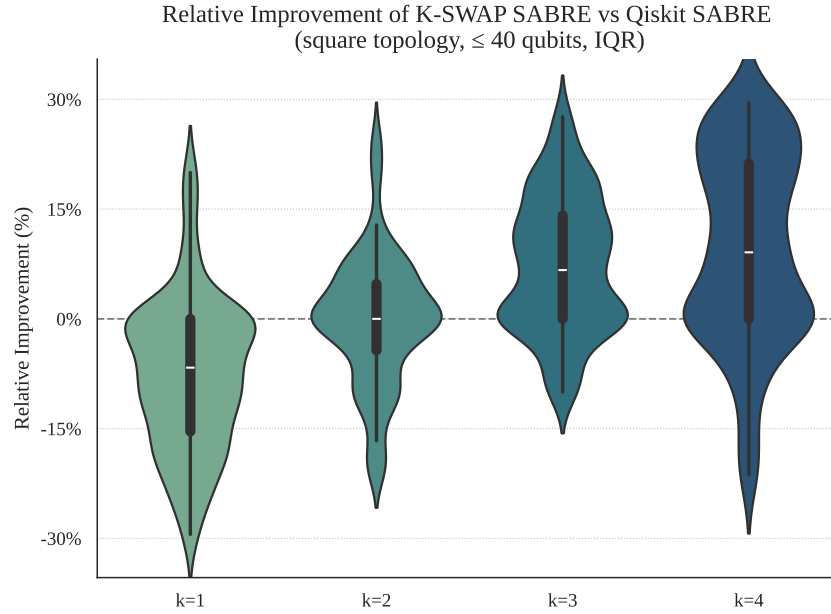


Figure 5.4: Distribution of relative SWAP improvement of *K-SWAP SABRE* compared to Qiskit SABRE for circuits with fewer than 40 qubits on a square topology. Results shown for $k \in \{1, 2, 3, 4\}$.

-6.5% with occasional larger degradations. Increasing k progressively shifts the median upward: $k = 2$ centers near 0.0%, $k = 3$ reaches 6.6% and $k = 4$ reaches 9.1%. The distribution narrows as k increases, reducing the likelihood of severe regressions and demonstrating that larger k enhances robustness and solution quality.

5.3.2 Per-Circuit Trade-offs Across k

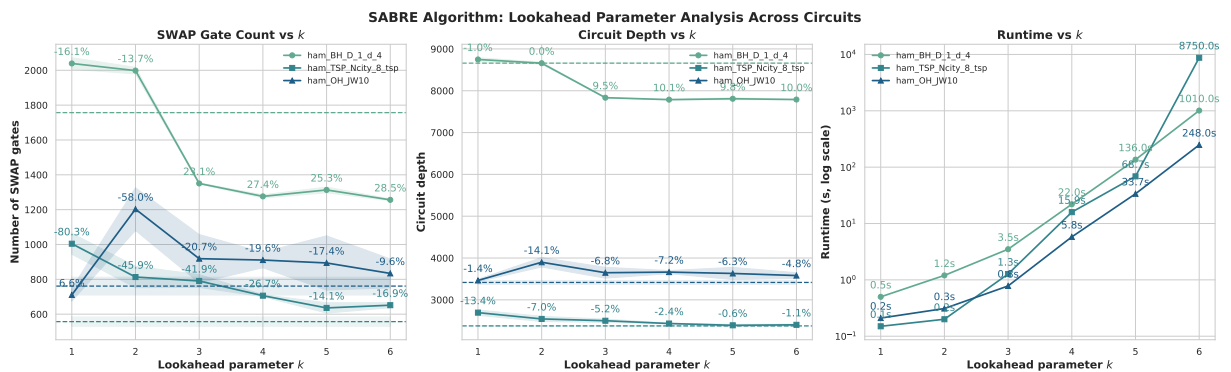


Figure 5.5: Analysis of individual circuit performance. We analyze the number of SWAPs, circuit depth and runtime across varying sequence lengths k .

To understand how the sequence length k affects the behavior of *K-SWAP SABRE*, we continue to analyze three representative benchmarks deliberately chosen for their contrasting performance profiles at $k = 3$. For this, all circuits are mapped to a *linear* QPU topology. This choice ensures that (i) each circuit remains feasible to execute 10 independent times even for $k = 6$, and (ii) the set covers both favorable and unfavorable cases for *K-SWAP SABRE*, enabling an assessment of the algorithm’s strengths and limitations.

The first circuit, `ham_BH_D_1_d_4_bh_graph_1D_grid` with 48 qubits and 11904 DAG nodes, highlights the potential of sequence-based lookahead. For small k , performance is clearly inferior to Qiskit; at $k = 1$ the algorithm produces 2039.2 ± 33.0 SWAPs compared to Qiskit’s 1757, and 1998.2 ± 19.1 at $k = 2$. Once k is increased to 3, however, the algorithm achieves 1350.3 ± 3.4 SWAPs, corresponding to a 23.1% reduction relative to the baseline. The best result is reached at $k = 6$ with 1256.3 ± 5.1 SWAPs, a 28.5% improvement. Circuit depth follows a similar trajectory, with a reduction of roughly 10% at $k = 4$. The runtime grows from 0.5s at $k = 1$ to more than 1000s at $k = 6$. Taken together, these observations suggest the existence of a sweet spot at $k = 3$ or $k = 4$, where quality improvements are substantial while the runtime remains manageable. Beyond this point the marginal gains are negligible compared to the exponential growth in runtime.

The second benchmark, `ham_TSP_Ncity_8_tsp` with 24 qubits and 2240 DAG nodes, shows that Qiskit’s *SABRE* consistently outperforms *K-SWAP SABRE* across all tested values of k . At $k = 1$, the algorithm produces 1004.5 ± 60.2 SWAPs, almost double Qiskit’s baseline of 557.2 ± 26.8 . Although the gap narrows as k increases, even at $k = 6$ the algorithm requires 651.5 ± 14.5 SWAPs, which is still 16.9% worse than the baseline. Circuit depth mirrors this trend, remaining consistently above Qiskit’s values. This circuit therefore constitutes a *bad case*, where the increased sequence length not only fails to help but actually degrades performance and increases runtime instability. The likely reason is that the circuit’s structure already aligns well with Qiskit’s heuristic, while longer SWAP sequences proposed by *K-SWAP SABRE* introduce overhead without yielding meaningful global benefits.

The third circuit, `ham_0H_JW10` with 10 qubits and 3851 DAG nodes, illustrates a more mixed picture with unstable behavior. At $k = 1$, the algorithm slightly outperforms Qiskit, with 710.6 ± 36.0 SWAPs compared to 761.2 ± 51.1 , a +6.6% improvement. At $k = 2$, however, performance deteriorates sharply, yielding 1203.0 ± 122.5 SWAPs (-58.0%). From $k = 3$ onward, the results gradually recover, reaching 834.0 ± 83.0 SWAPs at $k = 6$, only 9.6% worse than Qiskit. Depth shows a similar instability, and variance across runs is much larger than for the other circuits. This sensitivity to randomness indicates that the algorithm’s stochastic choices – such as the initial mapping or tie-breaking in the heuristic – can push the search into very different regions of the solution space, leading to unstable outcomes. Runtime growth is more moderate than for the large `ham_BH` circuit but still rises substantially, with runs at $k = 6$ taking around 248s.

From these examples, we conclude that no single k is universally optimal. Moderate sequence length ($k = 3$ or $k = 4$) often balances solution quality and runtime, yielding substantial SWAP reductions for favorable circuits while remaining computationally feasible. Runtime increases rapidly with k , highlighting the need for adaptive or selective use of larger sequence lengths.

Across all benchmarks, we observe:

1. **Advantage grows with connectivity.** On average, higher-connectivity topologies (square, heavy-hex) benefit more from increased k , while low-connectivity topologies (linear) show minimal gains.
2. **Improved solution quality with larger k .** Increasing the sequence length k improves robustness and median SWAP reductions, particularly on smaller circuits and connected topologies.

In summary, *K-SWAP SABRE* offers meaningful advantages for connected topologies and moderate-to-large sequence lengths, with a practical trade-off between solution quality and computational cost.

6 Conclusion and Outlook

This thesis addresses the qubit mapping problem, a key bottleneck in compiling quantum circuits for near-term devices, by extending heuristic approaches to more effectively balance scalability and solution quality. Motivated by the limitations of existing heuristic methods, in particular the SABRE algorithm, our goal was to bridge the gap between a scalable heuristics and exact approaches. The central idea was to extend the SABRE algorithm with a bounded exhaustive search over sequences of SWAPs, thereby improving the algorithm’s ability to consider routing decisions beyond the immediate front-layer of the circuit.

We proposed *K-SWAP SABRE*, an algorithm that integrates SWAP sequences of length k into the routing stage of the transpilation process. This required adapting SABRE’s heuristic to operate on sequences rather than single SWAPs, ensuring comparability between alternative routing choices while avoiding divergence.

The resulting algorithm was implemented and evaluated on a large and diverse set of benchmark circuits. Our experiments confirmed that increasing the sequence length k tends to reduce the number of inserted SWAPs, thus improving the fidelity of the resulting circuits. At the same time, we observed a trade-off: larger values of k increase the runtime of the algorithm. We also found that the algorithm benefits significantly from increased device connectivity: as the coupling graph becomes denser, the advantage of our sequence-based approach becomes more pronounced.

Taken together, these results show that *K-SWAP SABRE* is a promising way to complement scalable heuristics in the process of routing a quantum circuit.

The results of this thesis open several avenues for further investigation. One promising direction is to more explicitly consider the occupancy of the front-layer in routing decisions. By doing so, the algorithm could be designed either to favor moves that maintain a fuller front-layer, thereby preserving more routing options, or to dynamically adjust the sequence length k when the front-layer is sparse, enabling more informed exploration of candidate sequences. Another important line of research is the exploration of alternative hardware topologies. Our experiments indicate that denser connectivity enhances the effectiveness of *K-SWAP SABRE*, suggesting that evaluating the algorithm on 3D topologies with higher connectivity could yield further improvements. In addition, understanding

the structural properties of quantum circuits that correlate with strong performance under *K-SWAP SABRE* could inform circuit-specific heuristics or pre-processing strategies, potentially improving results across a broader class of circuits. Finally, working with the alternative polynomial formulation of the heuristic offers another avenue for optimization. By systematically exploring and fine-tuning this formulation, it may be possible to further enhance routing efficiency and solution quality.

In conclusion, this thesis demonstrates that extending the heuristic qubit mapping of SABRE with bounded swap-sequence search can improve the quality of results while maintaining scalability.

Bibliography

- [BE21] N. P. Breuckmann, J. N. Eberhardt. “Quantum Low-Density Parity-Check Codes.” In: *PRX Quantum* 2.4 (Oct. 2021). ISSN: 2691-3399. DOI: [10.1103/prxquantum.2.040101](https://doi.org/10.1103/prxquantum.2.040101). URL: <http://dx.doi.org/10.1103/PRXQuantum.2.040101> (cit. on p. 15).
- [BPB21] S. Brandhofer, I. Polian, H. P. Büchler. “Optimal mapping for near-term quantum architectures based on Rydberg atoms.” In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE. 2021, pp. 1–7 (cit. on p. 17).
- [BPK23] S. Brandhofer, I. Polian, K. Krsulich. “Optimal qubit reuse for near-term quantum computers.” In: *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. Vol. 1. IEEE. 2023, pp. 859–869 (cit. on p. 17).
- [CBSG17] A. W. Cross, L. S. Bishop, J. A. Smolin, J. M. Gambetta. “Open quantum assembly language.” In: *arXiv preprint arXiv:1707.03429* (2017) (cit. on pp. 13, 18).
- [CJA+22] A. Cross, A. Javadi-Abhari, T. Alexander, N. De Beaudrap, L. S. Bishop, S. Heide, C. A. Ryan, P. Sivarajah, J. Smolin, J. M. Gambetta, et al. “OpenQASM 3: A broader and deeper quantum assembly language.” In: *ACM Transactions on Quantum Computing* 3.3 (2022), pp. 1–50 (cit. on p. 13).
- [Com09] W. Commons. *Bloch sphere, a geometrical representation of a two-level quantum system*. 2009. URL: https://commons.wikimedia.org/wiki/File:Bloch_sphere.svg (cit. on p. 12).
- [Coo23] S. A. Cook. “The complexity of theorem-proving procedures.” In: *Logic, automata, and computational complexity: The works of Stephen A. Cook*. 2023, pp. 143–152 (cit. on p. 21).
- [Dij22] E. W. Dijkstra. “A note on two problems in connexion with graphs.” In: *Edsger Wybe Dijkstra: his life, work, and legacy*. 2022, pp. 287–290 (cit. on p. 28).

-
- [DJ92] D. Deutsch, R. Jozsa. “Rapid solution of problems by quantum computation.” In: *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439.1907 (1992), pp. 553–558 (cit. on p. 9).
 - [Fey18] R. P. Feynman. “Simulating physics with computers.” In: *Feynman and computation*. cRc Press, 2018, pp. 133–153 (cit. on p. 9).
 - [Gro96] L. K. Grover. “A fast quantum mechanical algorithm for database search.” In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 212–219 (cit. on p. 9).
 - [Joh77] D. B. Johnson. “Efficient algorithms for shortest paths in sparse networks.” In: *Journal of the ACM (JACM)* 24.1 (1977), pp. 1–13 (cit. on p. 23).
 - [JTK+24] A. Javadi-Abhari, M. Treinish, K. Krsulich, C. J. Wood, J. Lishman, J. Gacon, S. Martiel, P. D. Nation, L. S. Bishop, A. W. Cross, et al. “Quantum computing with Qiskit.” In: *arXiv preprint arXiv:2405.08810* (2024) (cit. on pp. 10, 18, 46).
 - [Knu13] D. E. Knuth. *Art of Computer Programming, Volume 4, Fascicle 4, The: Generating All Trees—History of Combinatorial Generation*. Addison-Wesley Professional, 2013 (cit. on pp. 27, 30).
 - [KYG+07] J. Koch, T. M. Yu, J. Gambetta, A. A. Houck, D. I. Schuster, J. Majer, A. Blais, M. H. Devoret, S. M. Girvin, R. J. Schoelkopf. “Charge-insensitive qubit design derived from the Cooper pair box.” In: *Physical Review A—Atomic, Molecular, and Optical Physics* 76.4 (2007), p. 042319 (cit. on p. 14).
 - [LDX19] G. Li, Y. Ding, Y. Xie. “Tackling the qubit mapping problem for NISQ-era quantum devices.” In: *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*. 2019, pp. 1001–1014 (cit. on pp. 17, 22, 24, 26, 27, 29, 30, 36, 42).
 - [Llo96] S. Lloyd. “Universal quantum simulators.” In: *Science* 273.5278 (1996), pp. 1073–1078 (cit. on p. 46).
 - [NC10] M. A. Nielsen, I. L. Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010 (cit. on pp. 11–13, 15).
 - [NI02] P. R. Norvig, S. A. Intelligence. “A modern approach.” In: *Prentice Hall Upper Saddle River, NJ, USA: Rani, M., Nayak, R., & Vyas, OP (2015). An ontology-based adaptive personalized e-learning system, assisted by software agents on cloud storage. Knowledge-Based Systems* 90 (2002), pp. 33–48 (cit. on p. 35).
 - [NSB+25] P. D. Nation, A. A. Saki, S. Brandhofer, L. Bello, S. Garion, M. Treinish, A. Javadi-Abhari. “Benchmarking the performance of quantum computing software for quantum circuit creation, manipulation and compilation.” In: *Nature Computational Science* (2025), pp. 1–9 (cit. on pp. 18, 46).

- [SBS+24] M. Steinberg, M. Bandić, S. Szkuclarek, C. G. Almudever, A. Sarkar, S. Feld. “Lightcone bounds for quantum circuit mapping via uncomplexity.” In: *npj Quantum Information* 10.1 (2024), p. 113 (cit. on p. 17).
- [Sho99] P. W. Shor. “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer.” In: *SIAM review* 41.2 (1999), pp. 303–332 (cit. on p. 9).
- [SMH+24] N. P. Sawaya, D. Marti-Dafcik, Y. Ho, D. P. Tabor, D. E. B. Neira, A. B. Magann, S. Premaratne, P. Dubey, A. Matsuura, N. Bishop, et al. “HamLib: A library of Hamiltonians for benchmarking quantum algorithms and hardware.” In: *Quantum* 8 (2024), p. 1559 (cit. on pp. 46, 47).
- [SSCP18] M. Y. Siraichi, V. F. d. Santos, C. Collange, F. M. Q. Pereira. “Qubit allocation.” In: *Proceedings of the 2018 international symposium on code generation and optimization*. 2018, pp. 113–125 (cit. on pp. 16, 21).
- [TC20] B. Tan, J. Cong. “Optimal layout synthesis for quantum computing.” In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020, pp. 1–9 (cit. on p. 16).
- [TGO+22] T. Tomesh, P. Gokhale, V. Omole, G. S. Ravi, K. N. Smith, J. Visslai, X.-C. Wu, N. Hardavellas, M. R. Martonosi, F. T. Chong. “Supermarq: A scalable quantum benchmark suite.” In: *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2022, pp. 587–603 (cit. on p. 18).
- [YDI+15] K. Yamanaka, E. D. Demaine, T. Ito, J. Kawahara, M. Kiyomi, Y. Okamoto, T. Saitoh, A. Suzuki, K. Uchizawa, T. Uno. “Swapping labeled tokens on graphs.” In: *Theoretical Computer Science* 586 (2015), pp. 81–94 (cit. on p. 21).
- [ZPW18] A. Zulehner, A. Paller, R. Wille. “An efficient methodology for mapping quantum circuits to the IBM QX architectures.” In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.7 (2018), pp. 1226–1236 (cit. on pp. 17, 43).
- [ZTH+24] H. Zou, M. Treinish, K. Hartman, A. Ivrii, J. Lishman. “LightSABRE: A Lightweight and Enhanced SABRE Algorithm (2024).” In: *arXiv preprint arXiv:2409.08368* (2024) (cit. on pp. 18, 22, 28, 40, 46).

All links were last followed on September 30, 2025.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature