TECHNISCHE
UNIVERSITÄT
WIEN

Bachelor's Thesis

Quantum circuit compression using higher dimensional systems

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Technical Physics

by

Simon Babovic
Registration Number: 12002084

to the Faculty of Physics at the TU Wien

Supervisor: Dr. Paul Erker

Vienna, October 7, 2025

“If quantum mechanics hasn’t
profoundly shocked you,
you haven’t understood it yet.”

- *Niels Bohr (Nobel prize 1922)*

Abstract

Quantum computing has the potential to solve complex problems beyond the reach of classical computers, but the need for implementing quantum circuits with increasing depth and size pose challenges in terms of noise, decoherence, and hardware limitations. One approach to mitigate these issues is quantum circuit compression, which aims to reduce the number of required quantum gates, particularly entangling gates, that contribute to error propagation. This thesis explores the use of higher-dimensional quantum systems, known as qudits, to achieve more compact and efficient quantum circuit representations. By encoding quantum information in multi-level systems, qudits allow for a reduction in circuit depth and the number of entangling operations.

This work investigates the mathematical foundations of quantum circuit transformation from qubit-based to qudit-based architectures and presents an algorithm for circuit compression based on graph-based clustering techniques. A software implementation of the algorithm is developed to automate the conversion process, demonstrating the feasibility of qudit-based optimization. The results indicate that qudit-based circuits can lead to improved efficiency and lower error rates, offering a promising path toward enhancing quantum computation on Noisy Intermediate-Scale Quantum (NISQ) devices. Future work includes refining optimization techniques, expanding support for diverse quantum gate sets, and integrating the developed framework with existing quantum computing platforms.

Contents

1. Introduction

Quantum computing has emerged as a transformative paradigm in computation, offering the potential to efficiently solve problems that are very hard to solve for classical computers [1]. At the heart of quantum computation lies the concept of quantum circuits, which leverages quantum gates to manipulate qubits and perform computations. However, as quantum algorithms grow in complexity, so do their circuit representations. The increasing need for depth and size of quantum circuits introduce significant challenges, particularly in the presence of noise and decoherence in Noisy Intermediate-Scale Quantum (NISQ) devices [2]. As a result, optimizing and compressing quantum circuits is a crucial step toward improving computational efficiency and the feasibility of real-world quantum applications.

One promising approach to circuit optimization is the utilization of higher-dimensional quantum systems, or qudits. Unlike conventional qubits, which operate with a two-level quantum state, qudits exploit high dimensional degrees of freedom, enabling more compact representations of quantum operations. By encoding information in multi-level systems, qudit-based architectures can reduce the number of required gates, particularly entangling gates, which are a major source of error in quantum computations. This approach has the potential to enhance quantum circuit efficiency and mitigate error propagation, contributing to the overall stability and performance of quantum algorithms [3].

This thesis explores the compression of quantum circuits using higher-dimensional systems. It investigates the mathematical foundations of quantum computation and the transformation of quantum circuits from qubit to qudit architectures. A program for quantum circuit compression is developed, leveraging graph-based clustering techniques to identify optimal qudit encodings. This program provides a flexible framework that can be extended to incorporate new optimization techniques and support further advancements in quantum circuit compression.

The structure of this thesis is as follows: Chapter 2 provides a theoretical background on quantum mechanics, qubits, and the role of entanglement. Chapter 3 discusses the construction and properties of quantum circuits, including universal gate sets. Chapter 4 introduces the concept of quantum circuit compression using qudits, details the proposed algorithm, and describes its implementation. Finally, Chapter 5 presents the conclusions and discusses possible extensions and improvements to the developed program, including support for additional gate sets, circuit optimization techniques, and packaging for seamless integration into other software.

2. Background

2.1. Schrödinger's equation

The most fundamental equation for closed nonrelativistic quantum system is the Schrödinger equation [4]

$$-\frac{\hbar^2}{2m}\Delta\Psi(\vec{r},t) + V(\vec{r})\Psi(\vec{r},t) = i\hbar\frac{\partial}{\partial t}\Psi(\vec{r},t), \tag{2.1}$$

where Δ denotes the Laplace-operator and $\hbar$ the reduced Planck constant. It assigns wave functions Ψ to a particle with mass m in a real potential $V(\vec{r})$ and describes it's time evolution. By making the product ansatz $\Psi(\vec{r},t) = \psi(\vec{r})\chi(t)$ and after separating variables,

$$-\frac{\hbar^2}{2m}\frac{1}{\psi(\vec{r})}\Delta\psi(\vec{r}) + V(\vec{r}) = E = i\hbar\frac{1}{\chi(t)}\frac{\partial}{\partial t}\chi(t), \tag{2.2}$$

the stationary Schrödinger equation is obtained:

$$-\frac{\hbar^2}{2m}\Delta\psi(\vec{r}) + V(\vec{r})\psi(\vec{r}) = E\psi(\vec{r}) \tag{2.3}$$

For a particle in one dimension, the stationary Schrödinger equation can be written as

$$-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\psi(x) + V(x)\psi(x) = E\psi(x). \tag{2.4}$$

By checking the units of the equation, it becomes clear, that regardless of the units of ψ, E is a scalar and has the dimension *Energy*.

$$\left[-\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\psi(x)\right] + [V(x)\psi(x)] = [E\psi(x)] \tag{2.5}$$

$$\frac{J^2s^2}{kg}\left[\frac{\partial^2}{\partial x^2}\psi(x)\right] + [V(x)]\,[\psi(x)] = [E]\,[\psi(x)] \tag{2.6}$$

$$Jm^2\frac{1}{m^2}\,[\psi(x)] + J\,[\psi(x)] = [E]\,[\psi(x)] \tag{2.7}$$

$$J = [E] \tag{2.8}$$

To further simplify the equation, an operator H can be defined:

$$H = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2} + V(x) \tag{2.9}$$

It is called Hamilton-operator or Hamiltonian and it depends on $V(x)$, therefore it is different for each physical system. The Schrödinger equation now simplifies to

$$H\psi(x) = E\psi(x). \tag{2.10}$$

2.1.1. Particle in a box

One simple one-dimensional problem to give an intuition for the Schrödinger equation is a one-dimensional particle trapped between two infinitely large potential barriers.

$$V(x) = \begin{cases} 0 & 0 < x < L \\ \infty & \text{otherwise} \end{cases} \tag{2.11}$$

The resulting differential equation can easily be solved by making an exponential Ansatz

$$\psi(x) = \begin{cases} Ae^{ikx} + Be^{-ikx} & 0 < x < L \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$

and demanding continuity of the wave function

$$\psi(0) = \psi(L) = 0. \tag{2.13}$$

By applying the boundary conditions and Euler's formula [5], k can be found and inserted into the wave function:

$$\psi_n(x) = N sin\left(\frac{n\pi}{L}x\right), \quad n \in \mathbb{Z}, \quad N \in \mathbb{C} \tag{2.14}$$

Because the wave function itself can be complex-valued and does not directly correspond to measurable quantities, squaring its magnitude ensures a real, non-negative value that can be normalized to represent the probability distribution of finding a particle in a given state or position. In this case, normalizing the wave function

$$\int_{-\infty}^{\infty} |\psi(x)|^2 dx = 1, \tag{2.15}$$

yields N which is then inserted into the wave function

$$\psi_n(x) = \sqrt{\frac{2}{L}} sin\left(\frac{n\pi}{L}x\right). \tag{2.16}$$

Technically not N was found but $|N|$. N can have a complex phase, but as this global phase does not affect the probability distribution $|\psi(x)|^2$

$$\left|e^{i\theta}\psi(x)\right|^2 = \left(e^{i\theta}\psi(x)\right)\left(e^{-i\theta}\psi^*(x)\right) = |\psi(x)|^2, \tag{2.17}$$

this global phase is ignored. Plugging the wave function back into the Schrödinger equation (2.4) yields the energies of each wave function:

$$E_n = \frac{\hbar^2 n^2 \pi^2}{2mL^2} \tag{2.18}$$

These are not the only solutions to the Schrödinger equation. In fact, every linear combination of functions $\psi_n(x)$ is also a solution. When constructing a simple example $\Phi(x) = a\psi_1(x)+b\psi_2(x)$, it becomes clear, that this solution cannot be assigned a constant energy. A particle with this wave function is in superposition. Only when measuring the energy of the particle the wave function collapses into either $\psi_1(x)$ or $\psi_2(x)$ with a probability based on the value of the coefficients a and b [6]. The functions $\psi_n(x)$ are special in the sense that they have well defined energies associated with them. These functions are called eigenstates or eigenfunctions of the system. The energies E_n are called eigenenergies.

2.2. The Postulates of Quantum Mechanics

While the previous section aims to provide an initial intuition for quantum mechanics, this section aims to provide a more rigorous framework to describe quantum systems. The postulates of quantum mechanics establish the connection between the physical world and the mathematical formalism of the theory [4].

> **Postulate 1.** *Associated to any isolated physical system is a complex vector space with inner product (that is, a Hilbertspace) known as the state space of the system. The system is completely described by its state vector, which is a unit vector in the system's state space.*

The wavefunction $\psi_n(\vec{r})$ is now represented as an abstract state vectore $|\psi\rangle$ that lives in an N-dimensional Hilbertspace $\mathcal{H}$. The inner product of that Hilbertspace [7] is defined as

$$\langle\phi|\psi\rangle = \int_{\mathbb{R}^3} \phi(\vec{r})^*\psi(\vec{r})d\vec{r}. \tag{2.19}$$

It follows, that if and only if the wavefunctions $\phi(\vec{r})$ and $\psi(\vec{r})$ are orthogonal to each other, the inner product $\langle\phi|\psi\rangle = 0$. Also, for the state vector $|\psi\rangle$ to be a unit vector, it has to be normalized such that

$$\langle\psi|\psi\rangle = 1. \tag{2.20}$$

The number of dimensions N of the Hilbertspace $\mathcal{H}$ equals the number of eigenstates of the system. In the example of a particle in a box, discussed in Section 2.1.1, N would be countable infinite.

Postulate 2. *The evolution of a closed quantum system is described by a Unitary transformation. That is, the state $|\psi\rangle$ of the system at time t_1 is related to the state $|\psi'\rangle$ of the system at time t_2 by a Unitary operator $\hat{U}$ which depends only on the times t_1 and t_2,*

$$\left|\psi'\right\rangle = \hat{U}\left|\psi\right\rangle. \tag{2.21}$$

This postulate describes the time evolution of closed physical systems. An equivalent description of that postulate would be: *The time evolution of the state of a closed quantum system is described by the Schrödinger equation,*

$$\hat{H}\left|\psi\right\rangle = i\hbar\frac{d\left|\psi\right\rangle}{dt}. \tag{2.22}$$

Equation (2.22) was alredy introduced in equation (2.1) with the difference being the description using a state vector $|\psi\rangle$ instead of a wave function in position space and $\hat{H}$ being the Hamilton-operator describing the physical system. In this notation it is independent of the basis. In this thesis, operators written with a hat (e.g., $\hat{A}$) denote basis-independent representations, while those without a hat (e.g., A) denote a representation in a specific basis. In the position-basis, H has the form

$$H = -\frac{\hbar^2}{2m}\Delta + V(\vec{r}), \tag{2.23}$$

which is similar to equation (2.9), with the only difference being that it uses three dimensions instead of one. This is not the only representation of the Hamilton-operator. It can also for example be represented in its eigenbasis $\{|\psi_1\rangle, |\psi_2\rangle, |\psi_3\rangle, ..., |\psi_N\rangle\}$:

$$H = \begin{pmatrix} E_1 & 0 & 0 & \cdots & 0 \\ 0 & E_2 & 0 & \cdots & 0 \\ 0 & 0 & E_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & E_N \end{pmatrix} \tag{2.24}$$

where E_n are the eigenenergies of the system, giving the stationary Schrödinger equation

$$H\left|\psi_n\right\rangle = E_n\left|\psi_n\right\rangle. \tag{2.25}$$

Postulate 3. *Quantum measurements are described by a collection $\hat{M}_m$ of measurement operators. These are operators acting on the state space of the system being measured. The index m refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediately before the measurement then the probability that result m occurs is given by*

$$p(m) = \langle\psi| \hat{M}_m^\dagger \hat{M}_m |\psi\rangle , \tag{2.26}$$

and the state of the system after the measurement is

$$|\psi_m\rangle = \frac{\hat{M}_m |\psi\rangle}{\sqrt{\langle\psi| \hat{M}_m^\dagger \hat{M}_m |\psi\rangle}} \tag{2.27}$$

The measurement operators satisfy the completenes equation,

$$\sum_m \hat{M}_m^\dagger \hat{M}_m = \hat{\mathbb{1}}. \tag{2.28}$$

From equation (2.27) follows, that a measurement alters the state. The statevector $|\psi\rangle$ collapses into one of the possible results $|\psi_m\rangle$ with the probability $p(m)$, while giving the measurement result m. For example, if a state $|\psi\rangle$ in a two dimensional Hilbertspace is prepared as

$$|\psi\rangle = \frac{1}{\sqrt{2}} \left(|\psi_1\rangle + |\psi_2\rangle\right), \tag{2.29}$$

where $|\psi_1\rangle$ and $|\psi_2\rangle$ are eigenstates of the system described by an operator $\hat{A}$ with the distinct eigenvalues a_1 and a_2, the measurement operators are given by

$$\hat{M}_{a_1} = |\psi_1\rangle\langle\psi_1| , \tag{2.30}$$

$$\hat{M}_{a_2} = |\psi_2\rangle\langle\psi_2| . \tag{2.31}$$

This definition satisfies the completeness equation (2.28), as $|\psi_1\rangle$ and $|\psi_2\rangle$ form a complete orthonormal system

$$\sum_i |\psi_i\rangle\langle\psi_i| = \hat{\mathbb{1}}. \tag{2.32}$$

Using equation (2.26), a general expression for the probability $p(a_i)$ can be found:

$$p(a_i) = |\langle\psi_i|\psi\rangle|^2 \tag{2.33}$$

In the given example, the probability of measuring a_1 results in $\frac{1}{2}$ and the state immediately after the measurement - if a_1 is the outcome of the measurement - becomes $|\psi_1\rangle$.

Equation (2.28) also ensures, that the probabilities $p(m)$ of all possible measurement outcomes add to 1:

$$\sum_m p(m) = \sum_m \langle\psi| \hat{M}_m^\dagger \hat{M}_m |\psi\rangle = \langle\psi| \sum_m \hat{M}_m^\dagger \hat{M}_m |\psi\rangle = \langle\psi|\psi\rangle \overset{(2.20)}{=} 1 \tag{2.34}$$

> **Postulate 4.** *The state space of a composite physical system is the tensor product of the state spaces of the component physical systems. Moreover, iff there are independent systems numbered* 1 *through* n*, and system number* i *is prepared in the state* $|\psi_i\rangle$*, then the joint state of the total system is* $|\psi_1\rangle \otimes |\psi_2\rangle \otimes ... \otimes |\psi_n\rangle$*.*

This postulate provides the mathematical framework to describe composite systems, which play a crucial role in quantum entanglement and in the description of quantum circuits discussed later in this thesis.

2.3. Qubits

A quantum bit (qubit) is the basic building block of quantum information processing [8]. It describes a physical system that has two distinctly measurable eigenstates, for example Spin-½ particles [9]. Using the notation

$$|\psi_n\rangle \equiv |n\rangle\,, \tag{2.35}$$

these eigenstates are called $|0\rangle$ and $|1\rangle$. That means the qubit lives in a two dimensional Hilbertspace with the basis $\{|0\rangle, |1\rangle\}$. This basis is also called computational basis. The state of the qubit can then be represented by a linear combination of the basis vectors:

$$|\psi\rangle = a|0\rangle + b|1\rangle \quad a, b \in \mathbb{C} \tag{2.36}$$

From the normalization, a relationship for a and b follows.

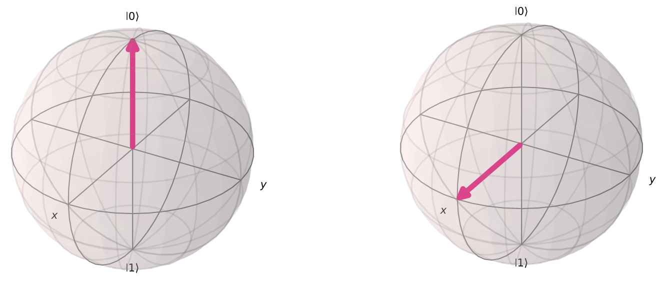
$$\langle\psi|\psi\rangle \overset{!}{=} 1 \quad \Rightarrow \quad |a|^2 + |b|^2 = 1 \tag{2.37}$$

2.3.1. Bloch representation

The amplitudes a and b of the qubit can be parameterized by angles θ and ϕ [10]:

$$|\psi\rangle = cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi} sin\left(\frac{\theta}{2}\right)|1\rangle\,. \tag{2.38}$$

Now the state can be visualized as a unit vector with angles θ and ϕ. Two different states are visualized in Figure 2.1.



(a) $|\psi\rangle = |0\rangle$ (b) $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$

Figure 2.1.: Two different qubit states visualized as bloch vectors. (a) has a probability of 100% to be in the state $|0\rangle$, (b) has a 50% chance to be either in the state $|0\rangle$ or $|1\rangle$. These visuals were generated using Qiskit [11]. The source code for this figure can be found in Listing A.1

2.4. Entanglement

With the help of postulate 4, as discussed in section 2.2, two qubits

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle \tag{2.39}$$

$$|\phi\rangle = c\,|0\rangle + b\,|1\rangle \tag{2.40}$$

can be described together in a composite system by joining their respective Hilbertspaces:

$$\mathcal{H}_{\psi\phi} = \mathcal{H}_\psi \otimes \mathcal{H}_\phi. \tag{2.41}$$

Using the notation

$$|nm\rangle \equiv |n\rangle \otimes |m\rangle\,, \tag{2.42}$$

the computational product basis for this new Hilbertspace is $\{|00\rangle\,, |01\rangle\,, |10\rangle\,, |11\rangle\}$, allowing a state in this Hilbertspace to be in the form of

$$\alpha\,|00\rangle + \beta\,|01\rangle + \gamma\,|10\rangle + \delta\,|11\rangle\,. \tag{2.43}$$

Postulate 4 also states, that the joint state of the two qubits equals to $|\psi\rangle \otimes |\phi\rangle$, which, when considering that both qubits must be normalized, only has two degrees of freedom, while a state prepared in the new Hilbertspace has three degrees of freedom. That means there are states in a composite system, that cannot be realized by only manipulating the qubits on its own. Such a state cannot be seperated into two qubits anymore and is called an entangled state [4]. In this state, the qubits are correlated to each other. The measurement of one qubit determines the measurement outcome of the other qubit. One example of such a preparation would be

$$|\Psi\rangle = \frac{1}{\sqrt{2}} \left(|01\rangle + |10\rangle \right). \tag{2.44}$$

For this state, there do not exist two qubit states $|\psi\rangle$ and $|\phi\rangle$ such that $|\psi\rangle \otimes |\phi\rangle = |\Psi\rangle$. Another way of thinking about the state $|\Psi\rangle$ is, that if qubit $|\psi\rangle$ is measured and yields the result $|0\rangle$ the only possible outcome when measuring qubit $|\phi\rangle$ in the same basis is $|1\rangle$. The qubits are dependent on each other.

3. Quantum Circuits

Multiple qubits in combination with quantum gates can be arranged to a quantum circuit. Initially, each qubit is set to the state $|0\rangle$ and subsequently transformed through the application of quantum gates. Apart from qubits and quantum gates, a quantum circuit also consists of classical bits and measurements. Qubits can be measured at any point in the circuit. Upon measurement, the state of a qubit collapses to either $|1\rangle$ or $|0\rangle$, and the result of the measurement is stored in one of the classical bits. Figure 3.1 shows an example for a simple quantum circuit. The aim of this chapter is to fully understand and describe this circuit.

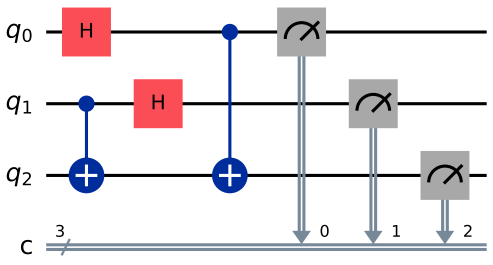


Figure 3.1.: This is an example of a simple quantum circuit. There are three qubits $\{q_0, q_1, q_2\}$, whose states are transformed by two single-qubit gates—Hadamard gates (H) in this case—and two multi-qubit gates, specifically controlled-X (CX) gates. At the end, each qubit is measured, and the result is stored in three classical bits. This visualization of a quantum circuit was generated using Qiskit [11]. The source code for this figure can be found in Listing A.2

3.1. Quantum gates

A quantum gate in a quantum circuit can be understood as an Unitary operator that acts either on the state of an individual qubit or on the composite state of multiple qubits.

3.1.1. Single qubit gates

As the name implies, a single qubit gate acts on single qubits. Its purpose is to map a state of a qubit into another state, therefore it can be described as an Unitary operator

$$\hat{A} = \alpha \, |0\rangle\langle 0| + \beta \, |1\rangle\langle 0| + \gamma \, |0\rangle\langle 1| + \delta \, |1\rangle\langle 1| \, , \tag{3.1}$$

acting on the state $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$ of a qubit,

$$\hat{A}\,|\psi\rangle = (a\alpha + b\gamma)\,|0\rangle + (a\beta + b\delta)\,|1\rangle\,. \tag{3.2}$$

The operator $\hat{A}$ can be represented as a 2×2 matrix using the computational basis $\{|0\rangle\,, |1\rangle\}$:

$$A^{\{|0\rangle,|1\rangle\}} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \tag{3.3}$$

With equation (3.2), gates can be constructed at will for different systems. An example is the Hadamard gate H, which transforms a qubit in the $|0\rangle$ or $|1\rangle$ state into a state with equal probabilities for $|0\rangle$ and $|1\rangle$ [12]:

$$H\,|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{3.4}$$

$$H\,|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{3.5}$$

With equations (3.4) and (3.5) all components $\{\alpha, \beta, \gamma, \delta\}$ of H can be calculated.

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.6}$$

A more general gate is the U gate. In this gate, three Euler angles [13] are used to rotate the state vector of the qubit.

$$U(\theta, \phi, \lambda) = \begin{pmatrix} cos\left(\frac{\theta}{2}\right) & -e^{i\lambda} sin\left(\frac{\theta}{2}\right) \\ e^{i\phi} sin\left(\frac{\theta}{2}\right) & e^{i(\phi+\lambda)} cos\left(\frac{\theta}{2}\right). \end{pmatrix} \tag{3.7}$$

3.1.2. Multi qubit gates

The real power of quantum circuits lies in manipulating states of qubits based on the state of other qubits. This can create entanglement. One example of an entangling gate is the Controlled-NOT ($CNOT$ or CX) gate. It acts on two qubits where the first qubit acts as a control. If the control qubit is in the state $|1\rangle$ it performs the NOT operation on the second qubit. Otherwise the second qubit remains unchanged:

$$CX\,|00\rangle = |00\rangle \tag{3.8}$$

$$CX\,|01\rangle = |01\rangle \tag{3.9}$$

$$CX\,|10\rangle = |11\rangle \tag{3.10}$$

$$CX\,|11\rangle = |10\rangle \tag{3.11}$$

With these conditions, the matrix representation of the CX gate in the computational product basis can be found:

$$CX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{3.12}$$

Similar to an entangled state vector, as discussed in Section 2.4, the controlled-X gate cannot be constructed by the tensorproduct of two single-qubit gates. This can easily be proven by assuming it can be separated into two 2×2 matrices,

$$\begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \otimes \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.13}$$

leading to a contradiction:

$$\alpha a = \delta c = 1 \Rightarrow \alpha \neq 0 \land c \neq 0 \Rightarrow\!\Leftarrow \alpha c = 0. \tag{3.14}$$

3.2. Circuit calculation

To calculate the final state of all three qubits in the circuit of Figure 3.1, the initial state and all gates will be transformed into the three-qubit product basis $\mathcal{B} = \{|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle\}$. The composite initial state $|\psi_{initial}\rangle$ of the circuit in the basis $\mathcal{B}$ is $|000\rangle$.

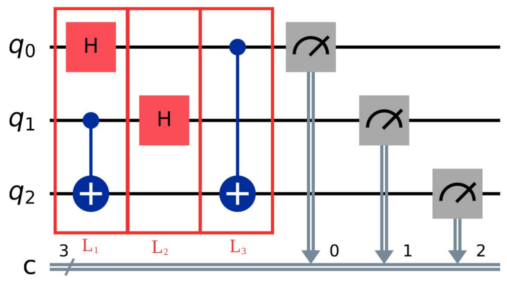


Figure 3.2.: The circuit from Figure 3.1 can be split into layers. In every layer, a maximum of one gate act on each qubit.

As shown in Figure 3.2, the circuit can be divided into three layers L_1, L_2 and L_3. Each layer can be imagined as an Unitary operator acting on the composite state of all qubits. The final state $|\psi_{final}\rangle$ can then be determined by applying all operators corresponding to the layers of the circuit to the initial state $|\psi_{initial}\rangle$,

$$|\psi_{final}\rangle = \hat{L}_3\hat{L}_2\hat{L}_1 |\psi_{initial}\rangle \,. \tag{3.15}$$

To determine the matrix representation of the layers, the matrix representation in the basis $\mathcal{B}$ for each single-qubit gate $U_{(i)}$ and two-qubit gate $U_{(i,j)}$ contained in the layer has to be found. In this notation i and j denote the number of the qubit on which the gate acts, while demanding $i < j$. As single-qubit gates leave other qubits untouched, their matrix representation in a basis $\mathcal{B}_N$ for N qubits can be constructed by tensor-products with the identity matrix:

$$U_{(i)}^{\{\mathcal{B}_N\}} = \mathbb{1}^{\otimes i} \otimes U_{(i)} \otimes \mathbb{1}^{\otimes N-1-i} \tag{3.16}$$

Entangling gates cannot be seperated into single-qubit gates as discussed in Section 3.1.2, so to find the matrix representation of entangling gates $U_{(i,j)}$ in the basis $\mathcal{B}_N$ while only knowing the matrix representation of $U_{(i,i+1)}$, a permutation matrix P, more specifically a transposition matrix $P_{n\mapsto m}$, has to be applied for entangling gates of non-adjacent qubits that swaps two qubits in a way that qubit q_i and q_j are adjacent,

$$q_j \leftrightarrow q_{i+1} \tag{3.17}$$

leading to

$$U_{(i,j)}^{\{\mathcal{B}_N\}} = P_{i+1\mapsto j}^{-1} \left(\mathbb{1}^{\otimes i} \otimes U_{(i,i+1)} \otimes \mathbb{1}^{\otimes N-2-i}\right) P_{i+1\mapsto j}. \tag{3.18}$$

Coming back to the example from figure 3.2 the first layer L_1, a Hadamard gate acts on the qubit q_0 and a controlled-X gate acts on q_1 and q_2 where q_1 is the control qubit and q_2 is the target qubit. Using equations (3.16) and (3.18), L_1 can be calculated,

$$L_1 = (H \otimes \mathbb{1} \otimes \mathbb{1})\,(\mathbb{1} \otimes CX) = H \otimes CX, \tag{3.19}$$

while the permutation matrix $P = \mathbb{1}$ for L_1 as q_1 and q_2 are already adjacent. L_2 can be found in a similar fashion as in this layer only a single-qubit Hadamard gate is present:

$$L_2 = \mathbb{1} \otimes H \otimes \mathbb{1}. \tag{3.20}$$

In L_3 only a controlled-X gate is present. But the qubits it acts on are not adjacent. To make them adjacent a transposition matrix $P_{1\mapsto 2}$ is used. This transposition matrix swaps qubit q_1 and q_2, so qubit q_0 is adjacent to q_2. When these qubits are swapped, the basis state order changes. This yields the conditions to determine the permutation matrix:

$$\begin{aligned} P_{1\mapsto 2}\,|000\rangle &= |000\rangle \\ P_{1\mapsto 2}\,|001\rangle &= |010\rangle \\ P_{1\mapsto 2}\,|010\rangle &= |001\rangle \\ P_{1\mapsto 2}\,|011\rangle &= |011\rangle \\ P_{1\mapsto 2}\,|100\rangle &= |100\rangle \\ P_{1\mapsto 2}\,|101\rangle &= |110\rangle \\ P_{1\mapsto 2}\,|110\rangle &= |101\rangle \\ P_{1\mapsto 2}\,|111\rangle &= |111\rangle \end{aligned} \Rightarrow P_{1\mapsto 2} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}. \tag{3.21}$$

L_3 can then be determined by using equation (3.18):

$$L_3 = P_{1\mapsto 2}^{-1}\,(CX \otimes \mathbb{1})\,P_{1\mapsto 2}. \tag{3.22}$$

Combining all layers yields the circuit Unitary C

$$C = L_3 L_2 L_1 = \frac{1}{2}\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & -1 & -1 \\ 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & -1 & 1 \\ 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 \end{pmatrix}. \tag{3.23}$$

By applying C to the initial circuit state $|\psi_{intial}\rangle = |000\rangle$ the final state $|\psi_{final}\rangle$,

$$|\psi_{final}\rangle = C\,|000\rangle = \frac{1}{2}\left(|000\rangle + |010\rangle + |101\rangle + |111\rangle\right) \tag{3.24}$$

is computed and the probabilities of measuring each eigenstate can be calculated using equation (2.33):

$$|\langle 000|\psi_{final}\rangle|^2 = \tfrac{1}{4}, \tag{3.25}$$

$$|\langle 001|\psi_{final}\rangle|^2 = 0, \tag{3.26}$$

$$|\langle 010|\psi_{final}\rangle|^2 = \tfrac{1}{4}, \tag{3.27}$$

$$|\langle 011|\psi_{final}\rangle|^2 = 0, \tag{3.28}$$

$$|\langle 100|\psi_{final}\rangle|^2 = 0, \tag{3.29}$$

$$|\langle 101|\psi_{final}\rangle|^2 = \tfrac{1}{4}, \tag{3.30}$$

$$|\langle 110|\psi_{final}\rangle|^2 = 0, \tag{3.31}$$

$$|\langle 111|\psi_{final}\rangle|^2 = \tfrac{1}{4}. \tag{3.32}$$

And indeed, when simulating the circuit with qiskits Aer-Simulator [11] with 100000 runs, the same results are achieved, as seen in Figure 3.3.

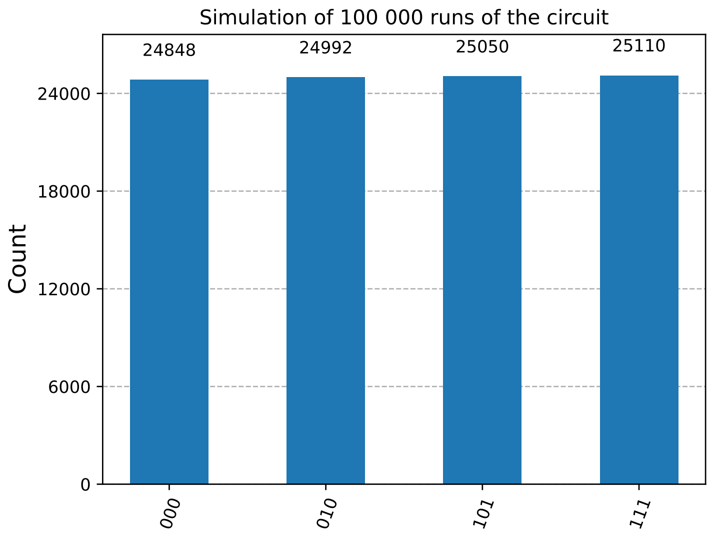


Figure 3.3.: This plot shows the outcome of a simulation of the circuit from Figure 3.1. While the states $|000\rangle$, $|010\rangle$, $|101\rangle$ and $|111\rangle$ all have a probability close to $\frac{1}{4}$, the states $|001\rangle$, $|011\rangle$, $|100\rangle$ and $|110\rangle$ are not realized. This coincides with the exact result from equation (3.24). This plot was generated using Qiskit [11]. The source code for this figure can be found in Listing A.3

3.3. Universal gate sets

The term *Universal Gate Set* is usually defined as follows:

> *A universal gate set is a finite set of gates that can be combined to approximate any Unitary operation on any number of qubits to arbitrary precision.* [4]

It is possible to check if a set of gates is universal by using group-theory methods [14] or by examining its relation to Unitary t-designs [15]. One example for a universal gate set that is commonly used is the Clifford + T set [16]. It consists of the Hadamard gate H

(3.6), the phase gate S (3.33), the T-gate (3.34) and the CX gate (3.12).

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \tag{3.33}$$

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{pmatrix} \tag{3.34}$$

By removing the restriction that the set of gates must be finite, the definition becomes simpler:

Definition 1. *A universal gate set is a set of gates that can be combined to produce any Unitary operation on any number of qubits.*

This definition excludes finite sets of gates, as there are infinetly many Unitary operations. While this definition may not be applicable to real-world scenarios, it is sufficient for this thesis and is therefore used instead of the usual definition. One example for a universal gate set that complies with this definition is a set of single-qubit rotations (3.7) in conjunction with the CZ-gate,

$$CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} \tag{3.35}$$

Any set of single-qubit gates that is universal for $SU\left(2\right)$, combined with the CX-gate (3.12) forms a universal gate set [4]. As single-qubit rotations $U\left(\theta, \phi, \lambda\right)$ are universal for $SU\left(2\right)$ by definition and the CZ-gate can be constructed using a CX gate and two Hadamard gates,

$$CZ = \left(\mathbb{1} \otimes H\right) CX \left(\mathbb{1} \otimes H\right), \tag{3.36}$$

the gate set $\{U\left(\theta, \phi, \lambda\right), CZ\}$ is also universal. This gate set is later used in this thesis to construct circuits to test circuit compression on.

4. Circuit Compression

As discussed in Section 3.1.2, the workhorse resource for quantum computing is entanglement. Unfortunately, entanglement also contributes to the fragility of qubits. Entangled qubits in current noisy intermediate-scale quantum (NISQ) devices [2] are extremely sensitive to environmental interactions, causing rapid quantum decoherence [17], which leads to the degradation of the entangled state [18]. Another problem of NISQ devices is error propagation. Even with little noise, single-qubit errors spreading throughout the quantum circuit can negate any quantum advantage [19]. One way to combat these errors are quantum error correction (QEC) algorithms, which can determine whether an error has occured by a suitable measurement and then applying Unitary corrections [20]. This chapter however investigates another way of mitigating errors – quantum circuit compression – by utilizing higher dimensional systems known as *qudits*. By reducing the number of quantum information carriers, the performance of experimental setups can be improved, while simultaneously lowering the resource requirements for quantum error correction through a reduction in entangling-gate count [3].

4.1. Qudits

As discussed in section 2.1.1, physical systems are not only limited to two states, in fact, most degrees of freedom of physical systems naturally posess more than two eigenstates. As qubits are modeled as two-state systems, they lack the capability to utilize higher dimensions, creating the need for a generalization – the quantum dit (qudit). It represents quantum information in a d-dimensional Hilbertspace $\mathcal{H}_d$, where $d > 2$. For example, a 4-dimensional qudit (ququart) can be described using a 4-dimensional hilberstspace $\mathcal{H}_4$ with the computational basis states $\{|0\rangle, |1\rangle, |2\rangle, |3\rangle\}$. Such a quqart can be modeled by a composite system of two qubits by encoding the basis states of the quqart onto the composite systems product states $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ living also on a 4-dimensional Hilbertspace $\mathcal{H}_4 = \mathcal{H}_2 \otimes \mathcal{H}_2$. The key difference is, that gates that are capable to entangle qubits, like CX or CZ gates, act locally rather than entangling when applied to a single ququart state:

$$\begin{aligned} CX\,|0\rangle &= |0\rangle \\ CX\,|1\rangle &= |1\rangle \\ CX\,|2\rangle &= |3\rangle \\ CX\,|3\rangle &= |2\rangle\,. \end{aligned} \tag{4.1}$$

So by combining qubits to qudits, the amount of entangling operations in a circuit can be reduced, with the limitation being the existence of a real physical system that can accurately represent the qudit.

4.1.1. Qudit Gates

When combining N qubits to one qudit it is also critical to understand how arbitrary local qubit gates U_i (4.2) and entangling qubit gates $U_{i,j}$ (4.3) transform to local qudit gates U_d, where q_i and q_j denote qubits being combined, while $i, j \in \{0, 1, ..., N-1\}$ and $i < j$ and $\mathbb{1}$ represents the two dimensional identity matrix.

$$U_d = \mathbb{1}^{\otimes i} \otimes U_i \otimes \mathbb{1}^{\otimes N-1-i} \tag{4.2}$$

If the entangling qubit gate $U_{i,j}$ acts on qubits which are not adjacent, a transposition matrix P is needed to rearrange the qubits into adjacent positions, as discussed in section 3.2, i.e.

$$U_d = P^{-1}_{i+1\mapsto j} \left(\mathbb{1}^{\otimes i} \otimes U_{i,i+1} \otimes \mathbb{1}^{\otimes N-2-i}\right) P_{i+1\mapsto j} \tag{4.3}$$

To give a better intuition on how qubit-gates transform when merging qubits, it will be demonstrated in the following example.

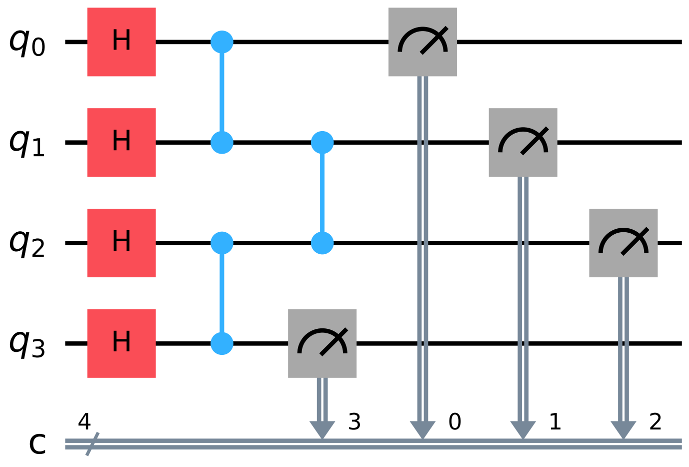


Figure 4.1.: Qubit circuit with four H-gates and three CZ-gates. This graphic was created using qiskit [11]. The source code can be found in Listing A.4.

In this example, using the circuit from Figure 4.1, the qubits q_0 and q_1 will be merged into a ququart ${q^4}_0$ and the qubits q_2 and q_3 will be merged into one ququart ${q^4}_1$. There are two ways how to transform local gates. Either they are transformed each into two local 4-dimensional ququart gates, that act only on a specific level of the ququart, or all gates of one column are combined to one 4-dimensional ququart gate. In this example, the latter is used. Two H-gates transform into a H_4-gate:

$$H_4 = H \otimes H = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \tag{4.4}$$

Two of the CZ-gates, that are entangling in the qubit circuit become local in the ququart circuit. Their matrix representation does not change. The last CZ-gate will still be entangling in the ququart circuit, so it will transform into a 16-dimensional ququart gate called CZ_{16}, its matrix form must be calculated using equation (4.3):

$$CZ_{16} = \mathbb{1} \otimes CZ \otimes \mathbb{1}, \tag{4.5}$$

where $\mathbb{1}$ denote 2-dimensional identity matrices. The resulting ququarts have four eigenstates, so at least two classical bits are needed to store the measurement results. The resulting ququart circuit is depicted in Figure 4.2.

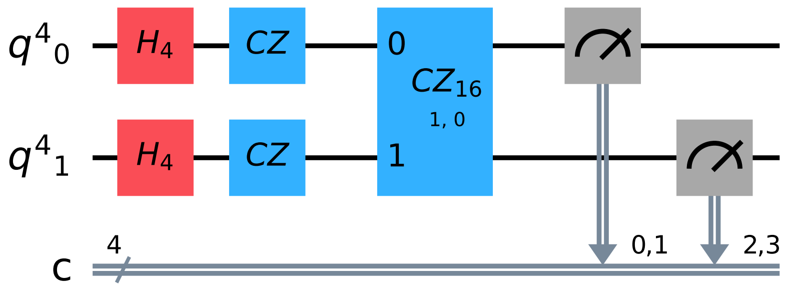


Figure 4.2.: Ququart circuit with two 4-dimensional H-gates denoted by H_4, two local CZ-gates and one 16-dimensional entangling CZ-gate denoted by CZ_{16}. The vertically arranged numbers in the CZ_{16}-gate show the numbers of the ququarts it acts on. The horizontally arranged numbers show the internal level of the ququart it acts on. It acts on level 1 of ququart $q^4{}_0$ (which originated from qubit q_1) and on level 0 of ququart $q^4{}_1$ (from original qubit q_2). This graphic was created using qiskit [11]. The source code can be found in Listing A.5.

4.2. Algorithm

One way to decide what qubits to merge is to create a weighted graph from the circuit and then to cluster nodes of that graph based on their edge-weight with their neighbours [3]. Using the universal gate set $\{U(\phi, \theta, \lambda), CZ\}$ discussed at the end of section 3.3, the graph is constructed as follows:

1. From every qubit in the circuit create two graph nodes. The first node represents the state $|0\rangle$ while the second node represents the state $|1\rangle$.

2. For every local gate U add a graph edge with weight w_l connecting the two nodes that originated from the qubit the gate acts on.

3. For every entangling gate CZ add a graph edge with weight w_e connecting the second nodes of both qubits it acts on.

The CZ gate only affects the $|11\rangle$ state, hence edges are only added to the nodes representing a $|1\rangle$ state. This is also the reason why this algorithm only works for this specific gate set. This simplifies the algorithm, however, it makes it important that $w_l > w_e$ is chosen, otherwise the clustering algorithm may attempt to generate qubits with local CZ gates, which is not possible. A visualization of a graph created from the circuit from Figure 4.1 is shown in Figure 4.3.

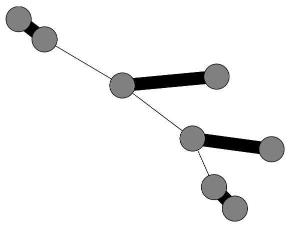


Figure 4.3.: A graph generated from the example circuit from Figure 4.1 using the algorithm described above with $w_l = 4w_e$. The thickness of the edges represent the weights. Nodes connected with a thick line originated from the same qubit, the H-gate binds them together. Nodes connected with a thin line are connected through a CZ gate. This graph was visualized using scikit-network [21].

In the next step a graph clustering algorithm groups highly connected nodes together by giving each node a label. In this implementation, any clustering method can be used. The algorithms *Louvain*, *Leiden* and *K-Centers* from scikit-network [21] are alredy implemented, but other more specialized algorithms, such as for example *gclu* [22] can easily be added - more on this in section 4.3.

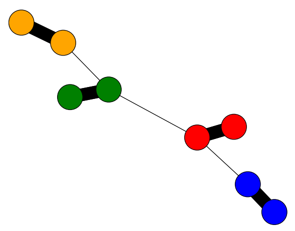


Figure 4.4.: The graph from Figure 4.3 got labeled by the *Louvain* clustering algorithm. Each color represents one label. This graph was visualized using scikit-network [21].

In Figure 4.4 the result from labeling the graph from Figure 4.3 is visualized. After labeling, all nodes with the same labels get merged into one new node while preserving edge data in a seperate datastructure, leading to a graph where every node now represents a qubit. Such a graph is visualized in Figure 4.5.

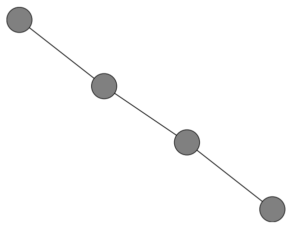


Figure 4.5.: This graph is the result of merging every node with the same label from the graph in Figure 4.4. Every node now represents a qubit. This graph was visualized using scikit-network [21].

This step did not produce a graph representing a qudit circuit but a graph representing a qubit circuit. Therefore, the clustering algorithm is applied again on the new graph, resulting in a ququart circuit graph, shown in Figure 4.6.

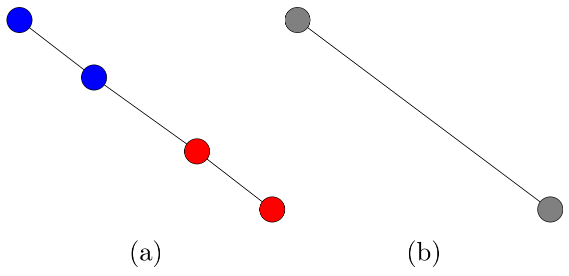


Figure 4.6.: The *Louvain* clustering algorithm assigns two labels to the nodes of the graph (a), effectevely merging four qubits into two ququarts. The resulting graph (b) represents exactly the circuit from Figure 4.2. These graphs were visualized using scikit-network [21].

In summary, the algorithm can be described as follows:

1. Create a weighted graph from a circuit
2. Run a clustering algorithm c that assigns labels to nodes of the graph
3. Save all gate information from each cluster in a seperate datastructure
4. Merge all nodes with the same label
5. Repeat steps 2. to 4. n times

where c and n can be chosen depending on the wanted dimension d of the qudit and width of the qubit circuit.

4.3. Implementation

The entire source code of the project is publicly available at `https://github.com/SimShady/qudit-compression` licensed under the GNU General Public License version 3 [23]. A copy of the source code can also be found in Appendix A.2. The project is structured in the following folders and files:

- `circuits/`
  - In this folder, all input circuits are stored in OpenQASM-format [24]. Except for the file `thesis_circuit_qiskit_4.qasm`, every file in this folder is taken from the mqt-qudit-compression project [25].
- `solutions/`
  - In this folder, all output circuits and visualizations are stored.
  - Circuit data is stored in `solutions/<circuitname>/<algorithmname>/data`
  - Visuals are stored in `solutions/<circuitname>/<algorithmname>/visuals`
- `graph.py`
- `circuit.py`
- `clustering.py`
- `processing.py`
- `main.py`
- `circuitinfo.py`
- `requirements.txt`
- `flake.nix`
- `flake.lock`

In the following, the source files are described in more detail.

`graph.py` (Listing A.6)

This file provides a simple datastructure to model a graph using only a list of edges to do so. That means, that if during the clustering process a node loses all edges it is destroyed. That has to be kept in mind when analysing results. Additionaly, an array `node_weights` is maintained that keeps track of the size of each node, with the size meaning the number of original nodes it has been merged with. The datastructure has basic methods to create and copy it as well as two special methods `get_reduced_graph_by_label` and `reduce_graph`.

- `get_reduced_graph_by_label` creates a new graph based on provided labels. It is a helper function for `reduce_graph`.
- `reduce_graph` calls the provided clustering algorithm one or more times and creates new graphs while keeping track of the history of graphs and labels used. These history arrays are then returned.

`circuit.py` (Listing A.7)

This file contains every datastructure needed to describe a qudit circuit.

- `Instruction`
- `Qudit`
- `QuditGate`
- `Circuit`

The `Instruction` class is just a helper structure that is used to convert qiskit circuits into an instance of `Circuit`. The classes `Qudit` and `QuditGate` are mainly structures to store data about qudits and gates. They have a function to serialize to JSON and some error handling to prevent invalid states. The most important datastructure is the `Circuit`. It stores a list of qudits and a ordered list of qudit gates, and can be created by passing such two lists to the constructor or from a QASM file. It also has a method to serialize to JSON which calls the serializers of its qudits and gates. The two important methods in this class are:

- `generate_weighted_graph`, which converts the circuit to a weighted graph. The weights w_l and w_e can be chosen. The defaults are $w_l = 400$ and $w_e = 100$. The implementation of converting a qiskit circuit into a graph was based on the mqt-qudit-compression project [25].
- `get_updated_circuit_by_labels`, which creates a new circuit based on the current circuit and a set of labels from a graph clustering algorithm.

`clustering.py` (Listing A.8)

This file is used as a library for clustering algorithms and also serves as example on how the interface of a clustering function needs to look like to be compatible with this project. A clustering function needs to take a `Graph` for an argument and return a list of integers representing the labels. The index i of this list denotes the number of the node, while the value at i denotes the label. If additional parameters need to be passed to the underlying clustering function, this can be done using a higher order function demonstrated by the three implemented functions `louvain`, `leiden` and `kcenters`.

`processing.py` (Listing A.9) and `main.py` (Listing A.10)

The file `processing.py` provides a function that prepares and applies the compression of one circuit using one clustering algorithm. It writes the resulting circuit data and visuals including the history to the `solutions` folder. The file `main.py` serves as entrypoint of the program. It looks up all files in the `circuits` folder and processes each circuit with multiple clustering algorithms using the function from `processing.py`.

`circuitinfo.py` (Listing A.12)

This program serves as utility to inspect generated circuits. For the example circuit from Figure 4.2, it prints:

```
Number of qudits: 2
Qudit dimensions: [4, 4]
Number of local gates: 6
Number of entangling gates: 1
```

Or for a circuit a little bit more complex (from `qpeexact_indep_qiskit_31.qasm`):

```
Number of qudits: 7
Qudit dimensions: [38, 4, 4, 4, 4, 4, 4]
Number of local gates: 1408
Number of entangling gates: 571
```

Utility files

The files `requirements.txt` (Listing A.11), `flake.nix` (Listing A.13), `flake.lock` (Listing A.14) handle dependency management and support the reproducibility even years after this writing. `requirements.txt` is used by the python package manager *pip* to install the correct versions of python packages that the project depends on. The other two files are used by Nix [26] to create a development shell that replicates the exact environment as it exists at the time of writing. This shell can be activated using the command `nix-shell`.

5. Conclusion

This thesis explored the concept of quantum circuit compression using higher-dimensional quantum systems (qudits) as an approach to optimizing quantum computations. By leveraging qudits, it is possible to reduce circuit width and the number of entangling gates, which are known to be a major source of noise and decoherence in Noisy Intermediate-Scale Quantum (NISQ) devices. The research focused on the transformation of qubit-based circuits into qudit-based representations and implemented a framework to compress qubit circuits to qudit circuits using clustering algorithms to identify optimal qudit encodings.

The results demonstrate that qudit-based architectures can effectively reduce the number of entangling operations while maintaining computational integrity. The proposed algorithm provides a flexible framework for circuit compression and can be extended to support additional optimization techniques. The developed program successfully converts quantum circuits into qudit equivalents, theoretically enabling more efficient execution on quantum hardware that supports higher-dimensional states.

Future enhancements to the program could expand support for a broader range of universal gate sets, enabling applications on real quantum hardware. Additionally, refining existing clustering algorithms to incorporate machine learning techniques may further enhance the optimization process. To simplify its adoption in practical use cases, the program could be packaged as a modular software library with well-defined APIs, allowing seamless integration into custom circuit compression applications.

By providing a framework for quantum circuit compression, this work contributes to the ongoing effort to improve quantum computing efficiency, paving the way for more scalable and noise-resilient quantum algorithms.

List of Figures

Bibliography

[1] Gill, S. S. *et al.* Quantum computing: Vision and challenges. *arXiv preprint arXiv:2403.02240* (2024).

[2] Preskill, J. Quantum computing in the nisq era and beyond. *Quantum* **2**, 79 (2018). URL `http://dx.doi.org/10.22331/q-2018-08-06-79`.

[3] Gao, X., Appel, P., Friis, N., Ringbauer, M. & Huber, M. On the role of entanglement in qudit-based circuit compression. *Quantum* **7**, 1141 (2023).

[4] Nielsen, M. A. & Chuang, I. L. *Quantum Computation and Quantum Information* (Cambridge University Press, Cambridge, England, 2010).

[5] Riley, K. F., Hobson, M. P. & Bence, S. J. *Mathematical methods for physics and engineering* (Cambridge University Press, Cambridge, England, 2006), 3 edn.

[6] Sakurai, J. J. & Napolitano, J. *Modern quantum mechanics* (Cambridge University Press, Cambridge, England, 2020), 3 edn.

[7] Jackson, J. D. *Mathematics for quantum mechanics: An introductory survey of operators, eigenvalues, and linear vector spaces* (Dover Publications, Incorporated, 2013).

[8] Aeschbacher, U., Hansen, A. & Wolf, S. *Invitation to quantum informatics* (vdf Hochschulverlag AG, an der ETH Zurich, Zürich, Switzerland, 2020).

[9] Levy, J. Universal quantum computation with spin-1/2 pairs and heisenberg exchange. *Physical Review Letters* **89**, 147902 (2002).

[10] Desurvire, E. *Classical and quantum information theory* (Cambridge University Press, Cambridge, England, 2009).

[11] Javadi-Abhari, A. *et al.* Quantum computing with Qiskit (2024). `2405.08810`.

[12] Rieffel, E. G. & Polak, W. H. *Quantum Computing*. Scientific and Engineering Computation (MIT Press, London, England, 2011).

[13] Morin, D. *Introduction to classical mechanics* (Cambridge University Press, Cambridge, England, 2008).

[14] Sawicki, A. & Karnas, K. Universality of single-qudit gates. *Annales Henri Poincaré* **18**, 3515–3552 (2017). URL `http://dx.doi.org/10.1007/s00023-017-0604-z`.

[15] Sawicki, A., Mattioli, L. & Zimborás, Z. Universality verification for a set of quantum gates. *Physical Review A* **105** (2022). URL `http://dx.doi.org/10.1103/PhysRevA.105.052602`.

[16] Vandaele, V., Martiel, S., Perdrix, S. & Vuillot, C. Optimal hadamard gate count for clifford+ t synthesis of pauli rotations sequences. *ACM Transactions on Quantum Computing* **5**, 1–29 (2024).

[17] Schlosshauer, M. Quantum decoherence. *Physics Reports* **831**, 1–57 (2019). URL `https://www.sciencedirect.com/science/article/pii/S0370157319303084`. Quantum decoherence.

[18] Yu, T. & Eberly, J. H. Finite-time disentanglement via spontaneous emission. *Phys. Rev. Lett.* **93**, 140404 (2004). URL `https://link.aps.org/doi/10.1103/PhysRevLett.93.140404`.

[19] González-García, G., Trivedi, R. & Cirac, J. I. Error propagation in nisq devices for solving classical optimization problems. *PRX Quantum* **3**, 040326 (2022). URL `https://link.aps.org/doi/10.1103/PRXQuantum.3.040326`.

[20] Brun, T. A. Quantum error correction (2020). URL `https://oxfordre.com/physics/view/10.1093/acrefore/9780190871994.001.0001/acrefore-9780190871994-e-35`.

[21] Bonald, T., de Lara, N., Lutz, Q. & Charpentier, B. Scikit-network: Graph analysis in python. *Journal of Machine Learning Research* **21**, 1–6 (2020). URL `http://jmlr.org/papers/v21/20-412.html`.

[22] Sieranoja, S. & Fränti, P. Adapting k-means for graph clustering. *Knowl. Inf. Syst.* (2021).

[23] Gnu general public license. URL `http://www.gnu.org/licenses/gpl.html`.

[24] Cross, A. W., Bishop, L. S., Smolin, J. A. & Gambetta, J. M. Open quantum assembly language (2017). URL `https://arxiv.org/abs/1707.03429`. `1707.03429`.

[25] Mato, K., Hillmich, S. & Wille, R. Compression of qubit circuits: Mapping to mixed-dimensional quantum systems. In *2023 IEEE International Conference on Quantum Software (QSW)*, 155–161 (IEEE, 2023).

[26] Dolstra, E. *The Purely Functional Software Deployment Model*. Phd thesis, Utrecht University, Utrecht, NL (2006). Available at `https://edolstra.github.io/pubs/phd-thesis.pdf`.

A. Appendix

A.1. Figure source code

```python
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector
from qiskit.visualization import plot_bloch_multivector

qc1 = QuantumCircuit(1)
qc2 = QuantumCircuit(1)
qc2.h(0)

state1 = Statevector(qc1)*1.03
state2 = Statevector(qc2)*1.03
fig1 = plot_bloch_multivector(state1)
fig1.get_axes()[0].set_title("")
fig2 = plot_bloch_multivector(state2)
fig2.get_axes()[0].set_title("")

fig1.savefig("bloch_0.pdf")
fig2.savefig("bloch_superposition.pdf")

```

Listing A.1: Source code for Figure 2.1

```python
from qiskit import QuantumCircuit
from qiskit.circuit import QuantumRegister, ClassicalRegister

qr = QuantumRegister(3, name="q")
cr = ClassicalRegister(3, name="c")
qc = QuantumCircuit(qr, cr)

q0,q1,q2 = qr
c0,c1,c2 = cr

qc.h(q0)
qc.cx(q1,q2)
qc.h(q1)
qc.cx(q0, q2)
qc.measure(q0, c0)
qc.measure(q1, c1)
qc.measure(q2, c2)

qc.draw(output="mpl", filename="example_quantum_circuit.pdf")

```

Listing A.2: Source code for Figure 3.1

```python
from qiskit import QuantumCircuit, transpile
from qiskit.circuit import QuantumRegister, ClassicalRegister
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram

qr = QuantumRegister(3, name="q")
cr = ClassicalRegister(3, name="c")
qc = QuantumCircuit(qr, cr)

q0,q1,q2 = qr
c0,c1,c2 = cr

qc.h(q0)
qc.cx(q1,q2)
qc.h(q1)
qc.cx(q0, q2)
qc.measure(q0, c0)
qc.measure(q1, c1)
qc.measure(q2, c2)

simulator = AerSimulator()
circ = transpile(qc, simulator)

result = simulator.run(qc, shots=100000).result()
counts = result.get_counts(qc)
plot_histogram(
  counts,
  title="Simulation of 100 000 runs of the circuit",
  filename="qc_results.pdf"
)

```

Listing A.3: Source code for Figure 3.3

```python
from qiskit import QuantumCircuit

qc = QuantumCircuit(4,4)
qc.h(0)
qc.h(1)
qc.h(2)
qc.h(3)
qc.cz(0,1)
qc.cz(2,3)
qc.cz(1,2)
qc.measure(0,0)
qc.measure(1,1)
qc.measure(2,2)
qc.measure(3,3)

qc.draw(output="mpl", filename="qubit_circuit.pdf")

```

Listing A.4: Source code fore Figure 4.1

```python
from matplotlib.figure import Figure
from qiskit import ClassicalRegister, QuantumCircuit, QuantumRegister
from qiskit.circuit import Gate

qr = QuantumRegister(2, name='q^4')
cr = ClassicalRegister(4, name='c')
qc = QuantumCircuit(qr,cr)
qc.append(Gate(name=r'$H_4$', num_qubits=1, params=[]),[0])
qc.append(Gate(name=r'$H_4$', num_qubits=1, params=[]),[1])
qc.append(Gate(name=r'$CZ$', num_qubits=1, params=[]),[0])
qc.append(Gate(name=r'$CZ$', num_qubits=1, params=[]),[1])
qc.append(Gate(name=r'$CZ_{16}$', num_qubits=2, params=[1,0]),[0,1])
qc.measure(0,0)
qc.measure(1,2)

fig: Figure = qc.draw(output="mpl", style={
  'displaycolor': {
    '$H_4$': ('#fA4D56', '#000000'),
    '$CZ$': ('#33B1FF', '#000000'),
    '$CZ_{16}$': ('#33B1FF', '#000000')
  }
}) # type: ignore
ax = fig.get_axes()[0]
ax.text(
    2.77608 + 1.6,
    -2 + 0.1,
    ",1",
    ha="left",
    va="bottom",
    fontsize=0.8 * 13,
    color="#000000",
    clip_on=True,
    zorder=13,
)
ax.text(
    3.77608 + 1.6,
    -2 + 0.1,
    ",3",
    ha="left",
    va="bottom",
    fontsize=0.8 * 13,
    color="#000000",
    clip_on=True,
    zorder=13,
)
fig.savefig("compressed.pdf")

```

Listing A.5: Source code fore Figure 4.2

A.2. Project source code

```
from typing import Callable

class Graph:
  def __init__(
    self,
    edge_list: list[tuple[int, int, int]],
    node_weights: list[int]
  ):
    self.edge_list = edge_list.copy()
    self.node_weights = node_weights.copy()

  def get_reduced_graph_by_label(
    self,
    labels: list[int]
  ) -> "Graph":
    new_node_weights = [0]*len(set(labels))
    for i in range(len(labels)):
      label = labels[i]
      new_node_weights[label] += self.node_weights[i]

    new_edge_list = []
    for edge in self.edge_list:
      node1 = edge[0]
      node2 = edge[1]
      weight = edge[2]
      if labels[node1] != labels[node2]:
        new_edge_list.append((labels[node1], labels[node2], weight))
    return Graph(new_edge_list, new_node_weights)

  def reduce_graph(
    self,
    clustering_function: Callable[["Graph"],list[int]],
    max_runs=1
  ) -> tuple[list["Graph"], list[list[int]]]:
    graph_history: list["Graph"] = []
    labels_history = []
    graph = self.copy()

    graph_history.append(graph)
    for runs in range(max_runs):
      labels = clustering_function(graph)
      graph = graph.get_reduced_graph_by_label(labels)
      if not graph.edge_list:
        break # stop if reduced graph does not have any edges left
      labels_history.append(labels)
      graph_history.append(graph.copy())
    return graph_history, labels_history

  def copy(self) -> "Graph":
    return Graph(self.edge_list.copy(), self.node_weights.copy())

```

Listing A.6: graph.py

```
import math
import json
from qiskit import QuantumCircuit

from graph import Graph

class Instruction:
  def __init__(
    self, name: str,
    qubits: list[int],
    params: list[float]
  ):
    self.name = name
    self.qubits = qubits.copy()
    self.params = params.copy()

class Qudit:
  def __init__(
    self, number: int,
    dimensions: int
  ):
    self.number = number
    self.dimensions = dimensions

  def toJSON(self) -> str:
    return json.dumps({
      "number": self.number,
      "dimensions": self.dimensions
    })

  def has_level(self, level: int) -> bool:
    return level >=0 and level < self.dimensions

class QuditGate:
  def __init__(
    self, name: str,
    qudits: list[Qudit],
    levels: list[int],
    euler_angles: tuple[float, float, float]=(0, 0, 0)
  ):
    self.name = name
    self.qudits = qudits
    self.euler_angles = euler_angles
    for i in range(len(self.qudits)):
      qudit = self.qudits[i]
      level = levels[i]
      if (not qudit.has_level(levels[i])):
        raise ValueError(
          f"Qudit {qudit.number} has dimension {qudit.dimensions}, it has
    no level {level}"
        )
    self.levels = levels

  def toJSON(self) -> str:
    return json.dumps({
      "name": self.name,
      "qudits": [qudit.number for qudit in self.qudits],
      "levels": self.levels,
      "euler_angles": self.euler_angles
    })

class Circuit:
  def __init__(
    self, qudits: list[Qudit],
    gate_list: list[QuditGate]
  ):
    self.qudits = qudits
    self.gate_list = gate_list

  def toJSON(self) -> str:
    return json.dumps({
      "qudits": [json.loads(qudit.toJSON()) for qudit in self.qudits],
      "gates": [json.loads(gate.toJSON()) for gate in self.gate_list]
    })

  @staticmethod
  def from_qasm(file: str) -> "Circuit":
    qc = QuantumCircuit.from_qasm_file(file)
    instructions, num_qubits = Circuit.__collect_instructions(qc)
    qudits: list[Qudit] = [Qudit(i, 1) for i in range(num_qubits*2)]
    gate_list: list[QuditGate] = []
    for instruction in instructions:
      if instruction.name != "measure" and instruction.name != "barrier":
        if instruction.name == "cz":
          control_qubit = instruction.qubits[0]
          target_qubit = instruction.qubits[1]
          control_qudit = qudits[control_qubit*2+1] # qubit level 1
          target_qudit = qudits[target_qubit*2+1] # qubit level 1
          gate = QuditGate("cz", [control_qudit, target_qudit], [0, 0])
          gate_list.append(gate)
        else:
          qubit = instruction.qubits[0]
          gate_qudits = [qudits[qubit*2], qudits[qubit*2+1]]
          euler_angles: tuple[float, float, float] = (0, 0, 0)
          match instruction.name:
            case "u1":
              euler_angles = (0, 0, instruction.params[0])
            case "u2":
              euler_angles = (math.pi/2, instruction.params[1],
  instruction.params[0])
            case "u3":
              euler_angles = (instruction.params[2], instruction.params
  [1], instruction.params[0])
          case _:
            raise ValueError("Unsupported gate " + instruction.name)
        gate = QuditGate("u", gate_qudits, [0, 0], euler_angles)
        gate_list.append(gate)
    return Circuit(qudits, gate_list)

  def generate_weighted_graph(self, local_weight=400, entangling_weight
  =100) -> Graph:
    edge_list: list[tuple[int, int, int]] = []
    for gate in self.gate_list:
      edge_list.append((gate.qudits[0].number, gate.qudits[1].number,
  entangling_weight if gate.name == "cz" else local_weight))
    qudit_dimensions = [qudit.dimensions for qudit in self.qudits]
    return Graph(edge_list, node_weights=qudit_dimensions)

  def get_updated_circuit_by_labels(self, labels: list[int]) -> "Circuit"
  :
    if (len(labels) != len(self.qudits)):
      raise ValueError("Every qudit needs a label")

    qudits = [Qudit(i, 0) for i in range(len(set(labels)))]
    for i, label in enumerate(labels):
      qudits[label].dimensions += self.qudits[i].dimensions

    new_gate_list: list[QuditGate] = []

    for gate in self.gate_list:
      gate_qudits: list[Qudit] = []
      gate_qudit_levels: list[int] = []
      first_level = (gate.qudits[0].number, gate.levels[0])
      second_level = (gate.qudits[-1].number, gate.levels[-1])
      first_compressed_level = self.__get_compressed_qudit_level(labels,
  first_level)
      second_compressed_level = self.__get_compressed_qudit_level(labels,
   second_level)
      gate_qudits.append(qudits[first_compressed_level[0]])
      if first_compressed_level[0] != second_compressed_level[0]:
        gate_qudits.append(qudits[second_compressed_level[0]])
      gate_qudit_levels.append(first_compressed_level[1])
      gate_qudit_levels.append(second_compressed_level[1])
      new_gate = QuditGate(gate.name, gate_qudits, gate_qudit_levels,
  gate.euler_angles)
      new_gate_list.append(new_gate)
```

```python
    return Circuit(qudits, new_gate_list)

  def __get_compressed_qudit_level(self, labels: list[int], qudit_level: tuple[int, int]) -> tuple[int, int]:
    grouped_labels: list[list[int]] = [[] for i in range(len(set(labels)))]
    for i in range(len(labels)):
      grouped_labels[labels[i]].append(i)

    original_qudits = [(qudit.number, qudit.dimensions) for qudit in self.qudits]

    original_qudit_idx, level_within_qudit = qudit_level

    for compressed_idx, label in enumerate(grouped_labels):
      if original_qudit_idx in label:
        offset = sum(original_qudits[i][1] for i in label if i < original_qudit_idx)
        new_level_within_compressed = offset + level_within_qudit
        return (compressed_idx, new_level_within_compressed)

    raise ValueError("No possible qudit level found")


  @staticmethod
  def __collect_instructions(qc: QuantumCircuit) -> tuple[list[Instruction], int]:
    num_qubits = len(qc.qubits)
    register_offset_mapper = {}
    last_offset = 0

    for register in qc.qregs:
      register_offset_mapper[register.name] = last_offset
      last_offset = last_offset + register.size

    instructions: list[Instruction] = []
    for _i, gate in enumerate(qc.data):
      name = ""
      qubits: list[int] = []
      params: list[float] = []

      for field in gate:
        if "library" in str(field) or "circuit" in str(field) or "Instruction" in str(field):
          name = field.name
          params = field.params

        if "QuantumRegister" in str(field):
          qubits += [(register_offset_mapper[f._register.name] + f._index) for f in field]

      instructions.append(Instruction(name, qubits, params))

    return instructions, num_qubits

```

Listing A.7: `circuit.py`

```python
from sknetwork.clustering import Louvain, Leiden, KCenters
from sknetwork.data.parse import from_edge_list

from graph import Graph
from typing import Callable

def louvain(modularity="dugue")-> Callable[[Graph], list[int]]:
  def clustering_function(graph: Graph) -> list[int]:
    louvain = Louvain(modularity=modularity)
    adjacency = from_edge_list(graph.edge_list, weighted=True)
    return louvain.fit_predict(adjacency) # type: ignore
  return clustering_function

def leiden(modularity="dugue") -> Callable[[Graph], list[int]]:
  def clustering_function(graph: Graph) -> list[int]:
    leiden = Leiden(modularity=modularity)
    adjacency = from_edge_list(graph.edge_list, weighted=True)
    return leiden.fit_predict(adjacency) # type: ignore
  return clustering_function

def kcenters(mean_cluster_size: int) -> Callable[[Graph], list[int]]:
  def clustering_function(graph: Graph) -> list[int]:
    kcenters = KCenters(n_clusters = len(graph.node_weights) // mean_cluster_size)
    adjacency = from_edge_list(graph.edge_list, weighted=True)
    return kcenters.fit_predict(adjacency) # type: ignore
  return clustering_function

```

Listing A.8: `clustering.py`

```python
from os import makedirs
from os.path import join, exists
from typing import Callable
from sknetwork.visualization import visualize_graph
from sknetwork.data.parse import from_edge_list
from circuit import Circuit
from graph import Graph

def process_circuit(circuit_file: str, out_dir: str, clustering_function: Callable[[Graph], list[int]], max_runs=1) -> int:
  circuit = Circuit.from_qasm(circuit_file)
  graph = circuit.generate_weighted_graph()
  graph_history, labels_history = graph.reduce_graph(clustering_function, max_runs=max_runs)

  for evolution in range(len(graph_history)):
    evolution_dirname = "evolution-" + str(evolution)
    visual_dir = join(out_dir, "visuals", evolution_dirname)
    data_dir = join(out_dir, "data", evolution_dirname)
    if not exists(visual_dir):
      makedirs(visual_dir)
    if not exists(data_dir):
      makedirs(data_dir)

    # save data
    with open(join(data_dir, "circuit.json"), "w+") as data_file:
      data_file.write(circuit.toJSON())

    # generate svg
    sknetwork_graph = from_edge_list(graph_history[evolution].edge_list, weighted=True)
    visualize_graph(sknetwork_graph, filename=join(visual_dir, "graph"), node_size=20) # type: ignore
    visualize_graph(sknetwork_graph, filename=join(visual_dir, "graph_w"), node_size=20, display_edge_weight=True) # type: ignore

    if (evolution < len(labels_history)): # Last graph has no labels
      circuit = circuit.get_updated_circuit_by_labels(labels_history[evolution])
      visualize_graph(sknetwork_graph, filename=join(visual_dir, "graph_l"), node_size=20, labels=labels_history[evolution]) # type: ignore
      visualize_graph(sknetwork_graph, filename=join(visual_dir, "graph_wl"), node_size=20, labels=labels_history[evolution], display_edge_weight=True) # type: ignore
  return len(labels_history) # return actual runs

```

Listing A.9: `processing.py`

```
from os import listdir
from os.path import isfile, join
from processing import process_circuit

import clustering

circuits_dir = "circuits"
solutions_dir = "solutions"

circuit_files = [f for f in listdir(circuits_dir) if isfile(join(
    circuits_dir, f))]

for circuit_file in circuit_files:
  out_dir = join(solutions_dir, circuit_file[:-5])
  runs = process_circuit(join(circuits_dir, circuit_file), join(out_dir,
    "louvain"), clustering.louvain(modularity="dugue"), max_runs=2)
  print("processed file '%s' with algorithm louvain (dugue) in %d runs" %
     (circuit_file, runs))
  runs = process_circuit(join(circuits_dir, circuit_file), join(out_dir,
    "leiden"), clustering.leiden(modularity="dugue"), max_runs=2)
  print("processed file '%s' with algorithm leiden (dugue) in %d runs" %
    (circuit_file, runs))
  runs = process_circuit(join(circuits_dir, circuit_file), join(out_dir,
    "kcenters"), clustering.kcenters(mean_cluster_size=4))
  print("processed file '%s' with algorithm kcenters (mean_cluster_size
    4) in %d runs" % (circuit_file, runs))

```

Listing A.10: `main.py`

```
contourpy==1.3.0
cycler==0.12.1
dill==0.3.8
fonttools==4.53.1
kiwisolver==1.4.5
matplotlib==3.9.2
mpmath==1.3.0
numpy==1.26.4
packaging==24.1
pbr==6.1.0
```

```
pillow==10.4.0
pylatexenc==2.10
pyparsing==3.1.4
python-dateutil==2.9.0.post0
qiskit==1.2.0
rustworkx==0.15.1
scikit-network==0.33.0
scipy==1.14.1
six==1.16.0
stevedore==5.3.0
symengine==0.11.0
sympy==1.13.2
typing_extensions==4.12.2

```

Listing A.11: `requirements.txt`

```
import sys
import json

def help():
  print("Usage: python circuitinfo.py path/to/circuit.json")

file = sys.argv[1]

if (len(sys.argv) < 2):
  help()
else:
  filename = sys.argv[1]
  try:
    with open(filename, 'r', encoding='utf-8') as file:
      data = json.load(file)
      qudits = data["qudits"]
      dimensions = [qudit["dimensions"] for qudit in qudits]
      local_gates = 0
      entangling_gates = 0
      for gate in data["gates"]:
        if len(gate["qudits"]) > 1:
          entangling_gates += 1
        else:
          local_gates += 1
      print("Number of qudits:", len(qudits))
      print("Qudit dimensions:", dimensions)
      print("Number of local gates:", local_gates)
      print("Number of entangling gates:", entangling_gates)
  except FileNotFoundError:
    print(f"Error: The file '{filename}' was not found.")

```

Listing A.12: circuitinfo.py

```
{
  description = "qudit-compression";
  inputs.nixpkgs.url = "github:NixOS/nixpkgs/nixos-unstable";
  inputs.flake-utils.url = "github:numtide/flake-utils";

  outputs = { self, nixpkgs, flake-utils }:
    flake-utils.lib.eachDefaultSystem (system: let
      pkgs = nixpkgs.legacyPackages.${system};
      lib = nixpkgs.lib;
      pythonldlibpath = lib.makeLibraryPath (with pkgs; [
        zlib
        zstd
        stdenv.cc.cc
        curl
        openssl
        attr
        libssh
        bzip2
        libxml2
        acl
        libsodium
        util-linux
        xz
        systemd
      ]);
    in {
      devShell = pkgs.mkShell {
        nativeBuildInputs = [ pkgs.bashInteractive ];
        buildInputs = with pkgs; [
          python311
        ];
        shellHook = with pkgs; ''
          ${python311}/bin/python -m venv ./.virtualenv
          source ./.virtualenv/bin/activate
          export LD_LIBRARY_PATH="${pythonldlibpath}"
          pip install -r requirements.txt
        '';
      };
    });
}

```

Listing A.13: `flake.nix`

```
{
  "nodes": {
    "flake-utils": {
      "inputs": {
        "systems": "systems"
      },
      "locked": {
        "lastModified": 1731533236,
        "narHash": "sha256-l0KFg5HjrsfsO/JpG+r7fRrqm12kzFHyUHqHCVpMMbI=",
        "owner": "numtide",
        "repo": "flake-utils",
        "rev": "11707dc2f618dd54ca8739b309ec4fc024de578b",
        "type": "github"
      },
      "original": {
        "owner": "numtide",
        "repo": "flake-utils",
        "type": "github"
      }
    },
    "nixpkgs": {
      "locked": {
        "lastModified": 1734119587,
        "narHash": "sha256-AKU6qqskl0yf2+JdRdD0cfxX4b9x3KKV5RqA6wijmPM=",
        "owner": "NixOS",
        "repo": "nixpkgs",
        "rev": "3566ab7246670a43abd2ffa913cc62dad9cdf7d5",
        "type": "github"
      },
      "original": {
        "owner": "NixOS",
        "ref": "nixos-unstable",
        "repo": "nixpkgs",
        "type": "github"
      }
    },
    "root": {
      "inputs": {
        "flake-utils": "flake-utils",
        "nixpkgs": "nixpkgs"
      }
    },
    "systems": {
      "locked": {
        "lastModified": 1681028828,
        "narHash": "sha256-Vy1rq5AaRuLzOxct8nz4T6wlgyUR7zLU309k9mBC768=",
        "owner": "nix-systems",
        "repo": "default",
        "rev": "da67096a3b9bf56a91d16901293e51ba5b49a27e",
        "type": "github"
      },
      "original": {
        "owner": "nix-systems",
        "repo": "default",
        "type": "github"
      }
    }
  },
  "root": "root",
  "version": 7
}

```

Listing A.14: `flake.lock`