Zurich University
of Applied Sciences

School of
Engineering

InIT Institute of Applied
Information Technology

# Project work Computer Science

## Programming a Quantum Computer – the Future of Programming?

| | |
|---|---|
| **Author** | Tobias Fankhauser<br>Marc Elias Solèr |
| **Main supervisor** | Prof. Dr. Kurt Stockinger |
| **Sub supervisor** | Prof. Dr. Rudolf Marcel Füchslin |
| **Date** | 18.12.2020 |

Zurich University
of Applied Sciences

School of Engineering

# DECLARATION OF ORIGINALITY

## Project Work at the School of Engineering

By submitting this project work, the undersigned student confirm that this work is his/her own work and was written without the help of a third party. (Group works: the performance of the other group members are not considered as third party).

The student declares that all sources in the text (including Internet pages) and appendices have been correctly disclosed. This means that there has been no plagiarism, i.e. no sections of the project work have been partially or wholly taken from other texts and represented as the student's own work or included without being correctly referenced.

Any misconduct will be dealt with according to paragraphs 39 and 40 of the General Academic Regulations for Bachelor's and Master's Degree courses at the Zurich University of Applied Sciences (Rahmenprüfungsordnung ZHAW (RPO)) and subject to the provisions for disciplinary action stipulated in the University regulations.

City, Date: Signature:

..... .....

.....

.....

The original signed and dated document (no copies) must be included after the title sheet in the ZHAW version of all project works submitted.

**Abstract**

Practical approaches to quantum algorithms are found seldom in the literature. With public access to quantum computers, exploration of such algorithms has however become feasible for small problem instances. In this work, we apply quantum algorithms on two kinds of problems: (1) Boolean satisfaction problems, where variable assignments that satisfy a logical (Boolean) expression are sought, and (2) combinatorial optimisation problems, where variable assignments with possibly low cost are desired.

For (1), we employ Grover's algorithm and discover that, while it performs well when simulated by a classical computer, it often fails on quantum computers due to error. We examine a high-level error-correction technique for quantum computers and find that this is insufficient to mitigate error and conclude that error-correction techniques would best be implemented on a lower level.

For (2) we examine the variational quantum eigensolver and the quantum approximate optimisation algorithm, two hybrid algorithms that operate between a quantum and a classical computer. We find that both algorithms perform well both when simulated by a classical computer as well as when executed on a quantum computer. With a more practical inclination, we solve a query optimisation problem, a classically hard database-related problem with the hybrid algorithms and obtain correct solutions.

We conclude that quantum algorithms are currently inferior to classical algorithms, mainly due to insufficient power and high error of quantum computers. However, many quantum algorithms are promising because of their - at least theoretically - superior performance in terms of time and space complexity against classical algorithms, practically enabled by sufficiently powerful and error-free quantum computers. We claim that hybrid algorithms may even be suitable for application in near-term quantum computers, as they tolerate more error and require less qubits.

***Keywords:*** quantum computing, satisfiability, query optimisation

**Zusammenfassung**

Quantenalgorithmen werden in der Literatur selten praktisch betrachtet. Die Untersuchung solcher Algorithmen für einfache Probleminstanzen wurde neuerdings mit öffentlichem Zugriff auf Quantencomputern einfacher. In dieser Arbeit wenden wir Quantenalgorithmen für zwei Problemklassen an: (1) Boolesche Erfüllbarkeitsprobleme, bei welchen eine Variablenzuweisung gesucht wird, die einen logischen (Booleschen) Ausdruck erfüllt, und (2) kombinatorische Optimierungsprobleme, wo Variablenzuweisungen mit möglichst geringen Kosten gesucht sind.

Für (1) wenden wir Grover's Algorithmus an, wobei wir feststellen, dass dieser befriedigende Resultate liefert wenn er mit einem herkömmlichen Rechner simuliert wird. Er scheitert jedoch, wenn er auf einem Quantenrechner ausgeführt wird aufgrund von hardware-induzierten Fehlern. In der Folge untersuchen wir eine Methode zur Fehlerkorrektur auf Niveau der Software, wobei wir feststellen, dass diese Fehler nur ungenügend mildern kann. Wir schliessen darauf, dass eine effektive Fehlerkorrektur auf einer tieferen Ebene, das heisst in der Nähe der Hardware, besser geeignet ist.

Für (2) betrachten wir den *Variational Quantum Eigensolver* und den *Quantum Approximate Optimisation Algorithm* - zwei hybride Algorithmen, die zu einem Teil auf Quanten- wie zu einem anderen Teil auf einem herkömmlichen Rechner ausgeführt werden. Wir konstatieren, dass beide dieser Algorithmen sowohl in der klassischen Simulation wie auch auf der Quantenhardware befriedigende Ergebnisse erzielen. Für eine praktischere Betrachtung lösen wir mithilfe der hybriden Algorithmen ein Query-Optimierungsproblem, ein Datenbankproblem, welches klassisch schwierig zu lösen ist, und wo beide Algorithmen nützliche Lösungen liefern.

Wir folgern, dass Quantenalgorithmen zum aktuellen Zeitpunkt herkömmlichen Algorithmen unterlegen sind, was auf fehlende Leistungsfähigkeit und hohe Fehleranfälligkeit von aktuellen Quantenrechnern zurückzuführen ist. Jedoch sind viele Quantenalgorithmen vielversprechend, da sie - zumindest theoretisch - in puncto Zeit- und Speicherkomplexität ihren klassischen Varianten überlegen sind. Diese Überlegenheit könnte künftig durch genügend leistungsfähige und fehlerfreie Quantenrechner umgesetzt werden. Wir behaupten, dass hybride Algorithmen bereits in naher Zukunft eingesetzt werden könnten, da sie Fehler gut tolerieren und Qubits effizient einsetzen.

***Schlüsselwörter:*** Quantenrechner, SAT, Queryoptimierung

# Preface

> (...) all the calculations that would ever be needed in this country could be done on the three digital computers (...)
>
> *B. V. Bowden, Baron Bowden quoting Douglas Hartree*

For more than 60 years, the computer industry is dominated by the integrated circuit. During this time, it has been enhanced with an impressive pace that has since become a law - *Moore's law* [1]. This success has only left little desire to seek alternative architectures for computers - but not none. Richard Feynman is famously accredited with considering a quantum computer that could surpass digital computers in certain physical problems in 1982 [2]. In 1994, Peter Shor showed that quantum computers were, at least in theory, able to factorise integers faster than previously suspected [3], simultaneously showing that quantum computers are applicable for tasks outside physics.

Now, quantum computers exist in different kinds, though generally not surpassing digital computers (yet). Providers of quantum hardware allow public access on some of their devices, allowing a broad audience to program these novel devices. Here, the ellipsis to the introductory quote, which is often attributed to Thomas Watson, closes: as many, including the authors of this work, suppose quantum computers not to attain a widespread adoption as digital computers, the question remains open whether, or which calculations may be done on quantum computer in the future. With this, another ellipsis closes, as the very author of the quoted thought, by partially providing the *Hartree-Fock*-method, contributed to an ingredient used in a quantum algorithm, which we examine.

We close the preface with one of the founding fathers of computer science, Alan Turing, who, at his time, was mostly agnostic about the architecture a computer is built upon. Now, after more than 60 years, with the evaluation of architectures based on quantum physical effects, or biological principles [4], we may see how computing architectures beyond integrated circuits might look like.

Finally, we thank our supervisors, Prof. Dr. Kurt Stockinger and Prof. Dr. Rudolf M. Füchslin to wake our interest in the exciting field of quantum computing. A field which atmosphere is not unsimilar with the beginnings of digital computing a few decades ago.

# Contents

# Chapter 1

# Introduction

This project work would not have been possible just five years ago. Since then, a number of quantum computers have become publicly available for researchers or students. During the last ten years, the pace of quantum technology in research and industry has accelerated, measured on the number of publications and commissioning of quantum devices [5]. In computer science, the digital computer has been the working horse since the sixties and has influenced the reasoning and design of algorithms and data structures. One may arguably ask: *What can a quantum computer do, what a digital computer cannot?*

To a computer scientist, we would answer: *no more than a Turing Machine. But, in certain cases, a quantum computer could do it more efficiently.*

To a physicist, we would answer: *a quantum computer could simulate quantum mechanical systems that are intractable on a classical computer.*

Although current quantum computers are generally inferior at solving tasks than digital computers, they obtain their potential power by principles such as as superposition and entanglement, which we introduce more formally in the next chapter. Still, we want to demonstrate the quantum advantage of superposition and entanglement in the following example.

**Example.** An ideal quantum computer with one qubit can represented two basis states $\alpha_0 |0\rangle + \alpha_1 |1\rangle$ (the $|...\rangle$ notation can be neglected for now), where the $\alpha$ s describe a complex coefficient, this is a number representable by three parameters. Assume that each $\alpha$ requires 8 bytes to store on a digital computer. Therefore, this quantum system requires 16 bytes to store digitally. Each additional qubit doubles the amount of complex coefficients, such that the potential of a quantum computer scales with $2^N$ with $N$ qubits.

An ideal quantum computer with 32 qubits would therefore require 256 gigabytes of digital storage, and a 64 qubit system were able to comprehend all information on the internet [6].

Today's quantum computers are, however, restricted as they are error prone. This work explores the capabilities of these imperfect quantum computers and error mitigation techniques theoretically and practically.

## 1.1 Organisation of this document

The remainder of this document is organised as follows.

Chapter 2 gives a theoretical introduction to quantum computing. Components and operation of quantum computers are explained mathematically and using program code. It also covers mathematical foundations and common quantum circuits.

Chapter 3 examines applications of Grover's algorithm, a quantum algorithm for unstructured search suggested for databases. It also covers quantum error correction briefly.

Chapter 4 analyses quantum optimisation algorithms. Two recent algorithms are introduced and discussed in more detail and applied on a database-related problem.

Chapter 5 reviews the results from the examination of the previous chapters.

Chapter 6 concludes this document with a discussion of the attained results and compares the quantum algorithms with classical counterparts.

## 1.2 Guide for running code

Code used in this project work is available on the **PA-Companion** repository on ZHAW GitHub, by the following URL: `https://github.zhaw.ch/QC-PA-2020/PA-Companion`. In this document, the files used are referred to as follows:

**Code reference.** Here, the usage of the code is quickly described, followed by the path relative to the repository above. For example: `Optimisation/ClassicSolver.py`.

Note that the repository may contain files not referred from the document. They are intended as supplementary material. We provide a quick installation guide below. Alternatively, we provide a list of the required software here:

To execute the code, the user requires the following software:

- `Qiskit`
- `numpy`
- `matplotlib`
- `networkx` *(for the MaxCut example only)*

To execute code on the Amazon Braket code (for the Rigetti quantum computer), the following software is required:

- `boto3`
- `amazon-braket-sdk`

### 1.2.1 Installation

1. If not present, install Python 3.8 (Python 3.9 has not been tested, but should work as well)
2. Install the `conda` software to better handle Python libraries: `https://docs.conda.io/projects/conda/en/latest/user-guide/install/`
   This automatically installs *Jupyter Notebook*, an environment in which most our files are written.

3. Install `Qiskit` by typing following command: `pip install qiskit[visualization]` (Qiskit should automatically install `matplotlib` and `numpy`. The user can however install them manually by: `pip install matplotlib` and `pip install numpy`)
4. For Amazon Braket, install `boto3` and `amazon-braket-sdk` by typing `pip install boto3` and `pip install amazon-braket-sdk`.
5. Enable access to the quantum hardware providers installing the API keys found on `https://github.zhaw.ch/QC-PA-2020/PA-Companion/blob/master/README.md`.

### 1.2.2 API keys for quantum hardware providers

To access quantum devices used in this work, we provide API keys and a quick documentation how to enable them on the **PA-Companion** repository: `https://github.zhaw.ch/QC-PA-2020/PA-Companion/blob/master/README.md`.

### 1.2.3 Running code

To run the code, we suggest to clone the **PA-Companion** repository and then accessing the `.ipynb` files by *Jupyter Notebook* with the following commands:

1. Clone the repository by
   `git clone https://github.zhaw.ch/QC-PA-2020/PA-Companion`
2. Open *Jupyter Notebook* by typing
   `jupyter notebook`
   into a console

For further reference, we list a few web resources related to the above software and libraries:

- Qiskit installation guide: `https://qiskit.org/documentation/install.html`
- Qiskit Website: `https://qiskit.org`
- Amazon Braket SDK README: `https://github.com/aws/amazon-braket-sdk-python`

# Chapter 2

# Introduction to Quantum Computing

In this chapter the basics of quantum computing are presented with the corresponding implementations in Qiskit.

## 2.1 Quantum Circuits

A circuit typically fulfils three tasks:

1. Encoding the input
2. Do actual computations
3. Extract an output

Qubits posses properties only describable by quantum mechanics. To make use of this characteristics, new gates have been created and new algorithms invented to manipulate the information contained in qubits.

To determine a function on a classical computer the function needs to be tested on a variety of different inputs to get sufficient information about the wanted property of that function. On a quantum computer on the other hand, due to the feature of creating superposition states, the function can be applied on many possible inputs simultaneously. Since only one possible output can be accessed through measurement, a quantum interference effect needs to be induced to reveal the global property require. For example a global maximum.

A different strategy towards developing quantum algorithms is to use quantum computers to solve quantum problems. To describe a quantum system with a classical computer the amount of information scales exponentially with the number of states the system can be in. On a quantum computer the number of qubits does not increase exponentially [7]. Therefore quantum systems such as molecules or fundamental particles can be successfully projected onto quantum computers while intractable on classical computers.

Especially promising are problems which do not require large classical data sets to be loaded and for which classical algorithms face inherent scaling limits. To have quantum advantage on a given problem the answer needs to be describable through entangled states and a inner structure such that quantum mechanics can dissolve the solution without exploring all possible paths [8].

The precise property of problems that are easy for quantum computers to solve is still an open question [8].

Listing 2.1: Creating a quantum circuit with Qiskit

```
# Creating a quantum-circuit with three qubits and two
   classical bits
from qiskit import QuantumRegister, ClassicalRegister,
   QuantumCircuit
q = QuantumRegister(3)
c = ClassicalRegister(2)
qc = QuantumCircuit(q, c)
```

Listing 2.1 describes the necessary imports and the code to create a quantum circuit with three qubits and 2 classical bits. In Qiskit qubits are always initialised in the state $|0\rangle$. Different gates can be applied to perform the manipulation intended. After the gates for measurement are placeed the circuit can be run on a simulator without or with artificial error in place or on a real quantum computer to test the implementation.

## 2.2 Qubits

Qubits hold the information gained from quantum calculations. A qubit is in the form

$$\alpha \cdot |0\rangle + \beta \cdot |1\rangle$$
$$\alpha, \beta \in \mathbb{C}$$

where $\alpha$ and $\beta$ are called the amplitudes [9]. The amplitudes are complex numbers and it is important to note that at all time

$$|\alpha|^2 + |\beta|^2 = 1$$

that is because $|\alpha|^2$ corresponds to the probability of measuring the qubit in the state $|0\rangle$ and $|\beta|^2$ for measuring the state $|1\rangle$. The states $|0\rangle$ and $|1\rangle$ can also be described by vectors.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$
$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

After measurement the qubit collapses his state to one of the basis states. This is also called **the observer effect**. In the Z-Basis this would be either $|0\rangle$ or $|1\rangle$. Further measurement has a $100\%$ chance of finding the qubit in the collapsed state. At all other times before measurement a qubits' state can be something more complex [9, 8].

### 2.2.1 Measuring Qubits

To find the probability of measuring the state $|x\rangle$ on a qubit in the state $|\psi\rangle$ the calculation

$$p(|x\rangle) = |\langle x|\psi\rangle|^2$$

is performed. The notation $\langle v|w\rangle$ is called the Dirac- or Bra-Ket-Notation [10].

$bra = \langle x| =$ **row vector with complex conjugate entries** of the state $|x\rangle$

$ket = |y\rangle =$ **column vector**

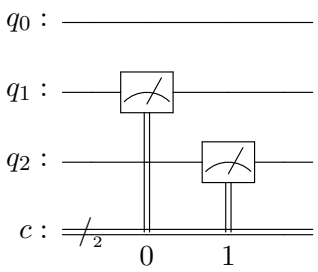
To measure the state $|1\rangle$ when the qubit is in the state $|\psi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$:

$$\begin{aligned}
\langle 1|\psi\rangle &= \frac{1}{\sqrt{2}}\langle 1|0\rangle + \frac{1}{\sqrt{2}}\langle 1|1\rangle \\
&= \frac{1}{\sqrt{2}}[0,1]\cdot\begin{bmatrix}1\\0\end{bmatrix} + \frac{1}{\sqrt{2}}[0,1]\cdot\begin{bmatrix}0\\1\end{bmatrix} \\
&= \frac{1}{\sqrt{2}}\cdot 0 + \frac{1}{\sqrt{2}}\cdot 1 \\
&= \frac{1}{\sqrt{2}} \\
\Rightarrow |\langle 1|\psi\rangle|^2 &= \frac{1}{2}
\end{aligned}$$

Qubits can be measured in infinitely many bases. The standard basis to measure is the Z-Basis. Measuring in the Z-Basis forces the qubit to collapse either into the state $|0\rangle$ or $|1\rangle$ [8]. Listing 2.2 shows the Qiskit code to measure selected qubits onto specified classical bits.

Listing 2.2: Measuring qubits

```
1 # Measuring Qubit q1 and q2 on classical bit c0, c1
2 qc.measure([1,2], [0,1])
```




### 2.2.2 Normalisation

As discussed before the amplitudes are related to probabilities. To gurantee the probabilities add up to 1 we need to normalize the statevector. The magnitude of the state vector has to be 1 [8].

$$\begin{aligned}
|\psi\rangle &= \alpha|0\rangle + \beta|1\rangle \\
\Rightarrow \sqrt{|\alpha|^2 + |\beta|^2} &= 1
\end{aligned}$$

### 2.2.3 Alternative measurement

There are infinite numbers of possible ways to measure a qubit. A measurement can be defined for any orthogonal pair of states to force a qubit to choose between the two [8]. The most common basis pairs are listed in 2.1.

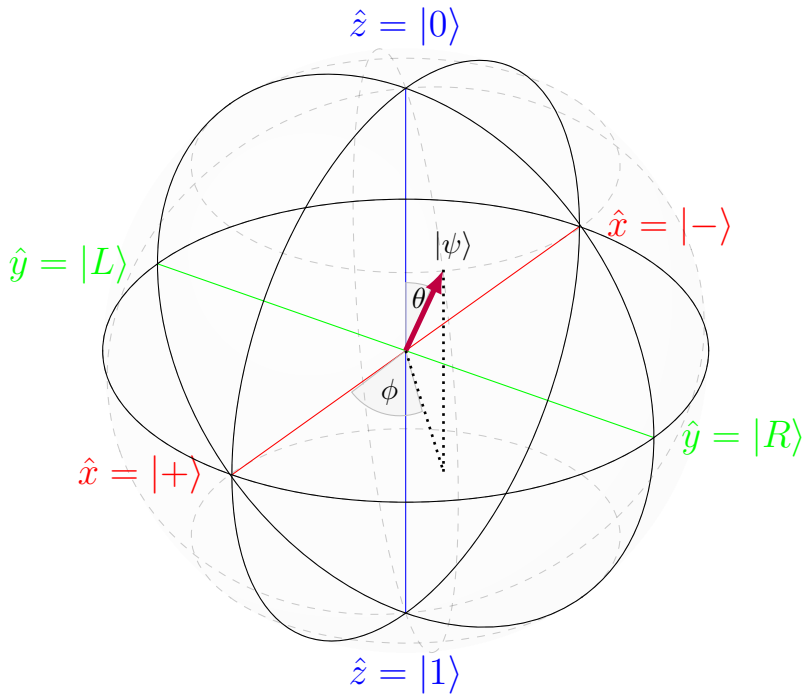
$$\begin{aligned}
Z - Basis &: |0\rangle, |1\rangle \\
X - Basis &: |+\rangle, |-\rangle \\
Y - Basis &: |R\rangle, |L\rangle
\end{aligned} \tag{2.1}$$

### 2.2.4 Global Phase

The probability to measure the state $|1\rangle$ is identical to $i|1\rangle$. Since measuring is the only way to extract information from a qubit, these two states are physically indistinguishable. Any factor $\gamma$ on a state for which $|\gamma| = 1$ is called a **global phase** [8].

$$|\langle x|(\gamma \cdot |a\rangle)\rangle|^2 = |\gamma \cdot \langle x|a\rangle|^2 = |\langle x|a\rangle|^2$$

### 2.2.5 The Bloch Sphere




If $\alpha$ and $\beta$ are confined to real numbers and a term is added to tell us the relative phase between them the state of a qubit can be written as

$$|\psi\rangle = \alpha|0\rangle + e^{i\phi}\beta|1\rangle$$
$$\alpha, \beta, \phi \in \mathbb{R}$$

because $\sqrt{\alpha^2 + \beta^2} = 1$, we can use the trigonometric identity $\sqrt{sin^2x + cos^2x} = 1$ to describe the real $\alpha$ and $\beta$ in terms of one variable $\theta$:

$$\alpha = cos\frac{\theta}{2}, \beta = sin\frac{\theta}{2}$$

hence we can describe the state of any qubit using $\phi$ and $\theta$:

$$\begin{aligned}|\psi\rangle &= cos\frac{\theta}{2}|0\rangle + e^{i\phi}sin\frac{\theta}{2}|1\rangle, \theta, \phi \in \mathbb{R} \\ &= \alpha|0\rangle + e^{i\phi}\beta|1\rangle, \alpha, \beta, \phi \in \mathbb{R} \\ &= \alpha|0\rangle + \beta|1\rangle, \alpha, \beta \in \mathbb{C}\end{aligned}$$

$\phi$ and $\theta$ can be interpreted as spherical coordinates on a sphere with radius $r = 1$. Consequently we can visualise any qubit on the surface of this sphere known as the Bloch sphere [8]. Listing 2.3 shows the Qiskit code to display the measured results from a quantum circuit on Bloch spheres.

Listing 2.3: Display Qubits on the Bloch sphere

```
from qiskit import execute, Aer
from qiskit.visualization import plot_histogram,
    plot_bloch_multivector

backend = Aer.get_backend("statevector_simulator")
statevector = execute(qc, backend=backend).result().
    get_statevector()
plot_bloch_multivector(statevector)
```

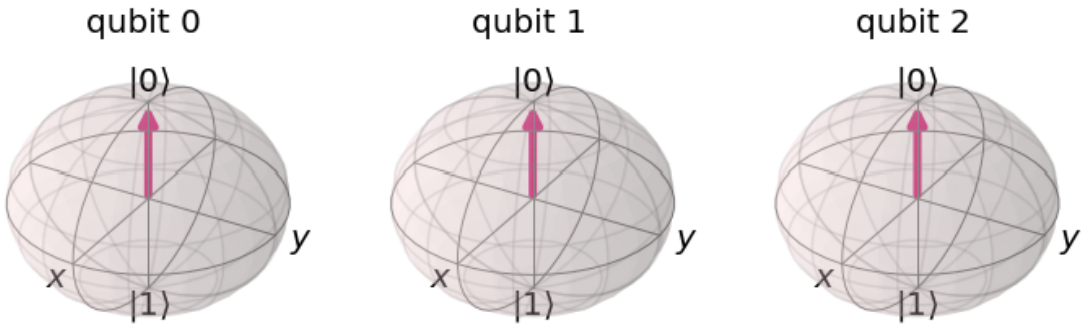



Figure 2.1: Qubits as state vectors in the Bloch sphere

## 2.3 Single Qubit Gates

Computation on a quantum computer is performed by gates which manipulate the state of qubits. In this section we introduce the most common transformations and the corresponding gates.

### 2.3.1 The X-gate, The Y-gate, The Z-gate

$$\text{Pauli-}X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0|$$

Listing 2.4: X-gate on qubit0

```
qc.x(0)
```

$$\text{Pauli-}Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = -i\,|0\rangle\langle 1| + i\,|1\rangle\langle 0|$$

Listing 2.5: Y-gate on qubit0

```
qc.y(0)
```

$$\text{Pauli-}Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = |0\rangle\langle 0| - |1\rangle\langle 1|$$

Listing 2.6: Z-gate on qubit0

```
1 qc.z(0)
```

The Pauli-gates have the effect of switching the amplitudes in the corresponding basis. In case of the X-gate $\alpha|0\rangle + \beta|1\rangle$ becomes $\beta|0\rangle + \alpha|1\rangle$. This is the same as a rotation by $\pi$ around the x-Axis of the Bloch sphere as shown in 2.2. Listings 2.4, 2.5 and 2.6 show the Qiskit code to add the corresponding gates to the qubits in a quantum circuit.

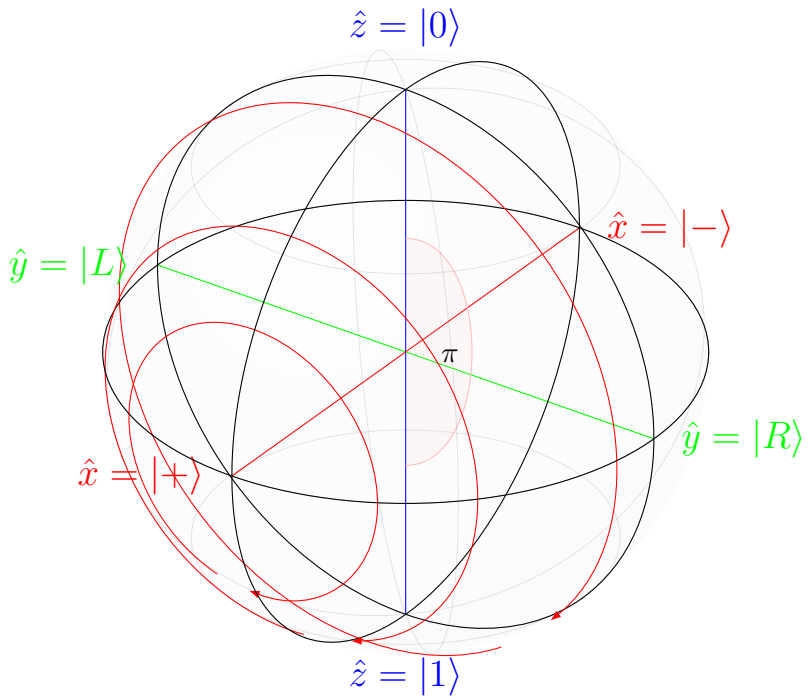



Figure 2.2: Rotation around the x-axis induced by a X-gate

### 2.3.2 The Hadamard Gate

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The Hadamard gate corresponds to a rotation by $\pi$ around the Bloch vector $[1, 0, 1]$, the line between the x & z-Axis.

$$\begin{aligned} H|0\rangle &= \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ &= \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ &= |+\rangle \end{aligned}$$

$$\begin{aligned} H|1\rangle &= \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ &= \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &= |-\rangle \end{aligned}$$

By applying the Hadamard gate to the state $|0\rangle$ or $|1\rangle$ the qubit is put into a **superposition** where measuring the states $|0\rangle$ or $|1\rangle$ have the same probability [8]. Listing 2.7 shows how to add the Hadamard gate to a qubit in Qiskit.

Listing 2.7: Hadamard gate on qubit0

```
1 qc.h(0)
```

### 2.3.3 General U-Gates

The $U_3$-gate is the most general of the single-qubit-gates. It allows to define three parameters $\theta$, $\phi$ and $\lambda$ and can perform any rotation around the Bloch sphere.

$$U_3(\theta, \phi, \lambda) = \begin{bmatrix} cos(\frac{\theta}{2}) & -e^{i\lambda}sin(\frac{\theta}{2}) \\ e^{i\phi}sin(\frac{\theta}{2}) & e^{i\lambda+i\phi}cos(\frac{\theta}{2}) \end{bmatrix}$$

Every of the above described gates could be specified as a $U_3(\theta, \phi, \lambda)$. For example the X-gate can be described as a $U_3(\pi, 2\pi, \pi)$.

$$\begin{aligned} X &= U_3(\pi, 2\pi, \pi) \\ &= \begin{bmatrix} cos(\frac{\pi}{2}) & -e^{i\pi}sin(\frac{\pi}{2}) \\ e^{i2\pi}sin(\frac{\pi}{2}) & e^{i\pi+i2\pi}cos(\frac{\pi}{2}) \end{bmatrix} \\ &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{aligned}$$

Listing 2.8 shows how to implement the $U_3$-gate in Qiskit. The `numpy` library is imported to use its value for $\pi$.

Listing 2.8: U-gate on qubit0

```
1 import numpy as np
2 pi = np.pi
3 qc.u(pi, 2*pi, pi, 0)
```

Every single-qubit-operation run on a IBM quantum computer is compiled down to a $U$-gate and implemented as such [8].

## 2.4 Multiple Qubit-Gates

To perform complex operations and make use of the potential of quantum computers, superposition and entanglement operating on more than a single qubit is mandatory. There exist various multiple qubit gates to perform those transformations. In this section we introduce some of those gates and concepts.

### 2.4.1 Multi-Qubit States

To describe the state of two qubits, four complex amplitudes are required. Those are stored in a 4D-vector, the **state vector**.

$$|a\rangle = a_{00}\,|00\rangle + a_{01}\,|01\rangle + a_{10}\,|10\rangle + a_{11}\,|11\rangle = \begin{bmatrix} a_{00} \\ a_{01} \\ a_{10} \\ a_{11} \end{bmatrix}$$

The rules of measurement remain the same for multi-qubit states as for single states.

$$p(|00\rangle) = |\langle 00|a\rangle|^2 = |a_{00}|^2$$

apparently also the normalisation has to hold.

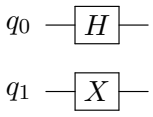
$$|a_{00}|^2 + |a_{01}|^2 + |a_{10}|^2 + |a_{11}|^2 = 1$$

The collective state of multiple qubits is described by the **tensor product** $\otimes$ of the individual qubits [8].

$$|a\rangle = \begin{bmatrix} a_0 \\ a_1 \end{bmatrix}, \quad |b\rangle = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}, \quad |c\rangle = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$$

$$\begin{aligned} |cba\rangle &= |c\rangle \otimes |b\rangle \otimes |a\rangle \\ &= |c\rangle \otimes \begin{bmatrix} b_0 \times \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \\ b_1 \times \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} \end{bmatrix} \\ &= |c\rangle \otimes \begin{bmatrix} b_0a_0 \\ b_0a_1 \\ b_1a_0 \\ b_1a_1 \end{bmatrix} \\ &= \begin{bmatrix} c_0 \times \begin{bmatrix} b_0a_0 \\ b_0a_1 \\ b_1a_0 \\ b_1a_1 \end{bmatrix} \\ c_1 \times \begin{bmatrix} b_0a_0 \\ b_0a_1 \\ b_1a_0 \\ b_1a_1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} c_0b_0a_0 \\ c_0b_0a_1 \\ c_0b_1a_0 \\ c_0b_1a_1 \\ c_1b_0a_0 \\ c_1b_0a_1 \\ c_1b_1a_0 \\ c_1b_1a_1 \end{bmatrix} \end{aligned}$$

In the same manner gates can be applied on multiple qubits using the tensor product of the gate's matrix representations [8].



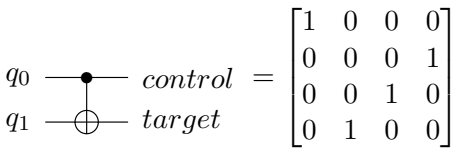

$$= X|q_0\rangle \otimes H|q_1\rangle = (X \otimes H)|q_1q_0\rangle$$

$$X \otimes H = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & H \\ H & 0 \end{bmatrix}$$

$$= \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \times \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & 1 \times \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \\ 1 \times \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & 0 \times \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \\ 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \end{bmatrix}$$

### 2.4.2 The CNOT-Gate

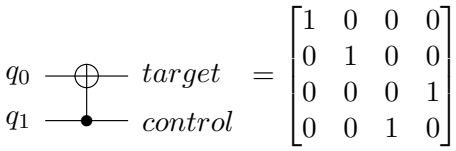
The CNOT-gate or controlled-NOT performs an X-gate on the target qubit if the control-qubit is $|1\rangle$. In the circuit below the qubit with the dot is the control- and the one with the XOR the target-qubit [8].




Listing 2.9 and 2.10 show how to implement the CNOT-gate in Qiskit with different target and control qubits.

Listing 2.9: CNOT-gate with q0 as control and q1 as target

```
1 qc.cx(0,1)
```



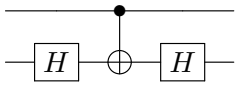

Listing 2.10: CNOT-gate with q1 as control and q0 as target

```
1 qc.cx(1,0)
```

As we can see from the matrices above the CNOT-gate swaps the amplitudes of the states $|01\rangle$ and $|11\rangle$ if $q_0$ is the control-qubit respectively the states $|10\rangle$ and $|11\rangle$ if $q_1$ is the control-qubit.

### 2.4.3 The Controlled-Z

The controlled-Z applies a Z-gate to the target qubit if the control is in state $|1\rangle$. We can build a controlled-Z out of a CNOT-gate and two Hadamard gates [8].




Listing 2.11 shows the two different ways to implement the controlled-Z-Gate in Qiskit.

Listing 2.11: Controlled-Z build with Hadamard and CNOT

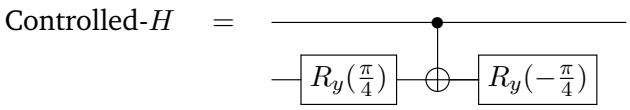
```
1 qc.h(1)
2 qc.cx(0,1)
3 qc.h(1)
```

or just

```
1 qc.cz(0,1)
```

### 2.4.4 The Controlled-Gates

We can build any controlled gate using a CNOT preceded and followed by the right rotation [8].




The above controlled operation is implemented in Qiskit in listing 2.12.

Listing 2.12: Controlled-H build with RY and CNOT

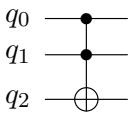
```
1 qc.ry(pi/4, 1)
2 qc.cx(0,1)
3 qc.ry(pi/4, 1)
```

or just

```
1 qc.ch(0,1)
```

### 2.4.5 The Toffoli Gate

The Toffoli gate is a three qubit gate with two control qubits and one target qubit. Both control qubits need to be in state $|1\rangle$ to perform the X-gate on the third qubit.




Listing 2.13 shows how to use the Toffoli gate in Qiskit.

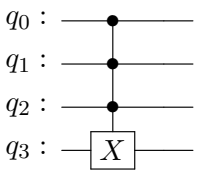
Listing 2.13: Toffoli

```
1 qc.ccx(0,1,2)
```

In Qiskit already exists an implementation to use a arbitrary number of control qubits, the MCT-gate. This gate is shown in Listing 2.14.

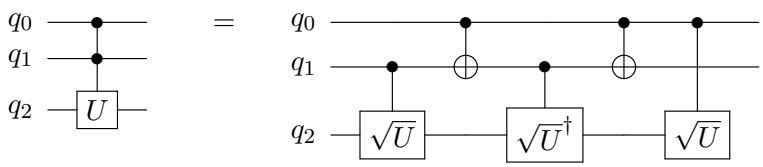
Listing 2.14: Multi-controlled X-gate

```
1 qc.mct([0, 1, 2], 3)
```



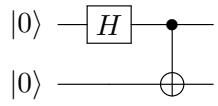

### 2.4.6 General rotation gate with two control qubits

to build a general rotation gate **U** with two control qubits we need to define a controlled version of $\sqrt{U}$ and $\sqrt{U}^\dagger$.



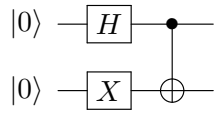
## 2.5 Entanglement

If two qubits are in an entangled state, depending on the measurement of one of the qubits the state of the other can be determined with $100\%$ certainty without measuring it.




$$CNOT\,|0+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

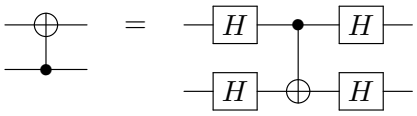
or




$$CNOT\,|1+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$

These are entangled states. The state of the control qubit is totally random. Depending on the measurment of one of the qubits the other has to be in the corresponding state. It does not matter which qubit is measured first. It is important to note that the state of one qubit is not affected by any operation applied to the other qubit after the entanglement. So there is no way to use shared quantum states to communicate. This is known as the no-communication theorem [8].

## 2.6 Phase Kickback




$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This identity is an example of **phase kickback**. When a controlled operation applies the eigenvalue added by a gate to a qubit into a different qubit this process is called phase kickback [8].

Performing an X-gate on a qubit in the state $|-\rangle$ gives it the phase **-1**.

$$\begin{aligned} X\,|-\rangle &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \\ &= \begin{bmatrix} -\frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \\ &= -\,|-\rangle \end{aligned}$$

The CNOT affects the state only if the control qubit is in state $|1\rangle$. Because this is a global phase it has no observable effect.

$$\begin{aligned} CNOT\,|-0\rangle &= X\,|-\rangle \otimes |0\rangle \\ &= |-0\rangle \end{aligned}$$

$$\begin{aligned} CNOT\,|-1\rangle &= X\,|-\rangle \otimes |1\rangle \\ &= -\,|-\rangle \otimes |1\rangle \\ &= -\,|-1\rangle \end{aligned}$$

Although when the control qubit is in superposition, this phase factor adds a relative phase to our control qubit [8].

$$\begin{aligned} CNOT\,|-+\rangle &= \frac{1}{\sqrt{2}}(CNOT\,|-0\rangle + CNOT\,|-1\rangle) \\ &= \frac{1}{\sqrt{2}}(|-0\rangle + X\,|-1\rangle) \\ &= \frac{1}{\sqrt{2}}(|-0\rangle - |-1\rangle) \\ &= |-\rangle \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &= |--\rangle \end{aligned}$$

The state of the control qubit is changed while the target qubit remains unchanged. This phenomenon is later applied in the oracle of Grover's algorithm to mark the searched item in our list with a negative phase.

## 2.7 Outer Product

$$|0\rangle\langle 0| = \begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1 & 0\end{bmatrix} = \begin{bmatrix}1 & 0\\0 & 0\end{bmatrix}$$

A matrix can be written purely in terms of outer products [8].

$$M = \begin{bmatrix}m_{00} & m_{01}\\m_{10} & m_{11}\end{bmatrix} = m_{00}\,|0\rangle\langle 0| + m_{01}\,|0\rangle\langle 1| + m_{10}\,|1\rangle\langle 0| + m_{11}\,|1\rangle\langle 1|$$

## 2.8 Unitary matrices

A unitary matrice is a matrix witch fulfills specific properties:

1. Unitary transformations are linear - can be described as matrices
2. The Hermitian conjugate of a unitary is its inverse
$$UU^\dagger = I$$
3. Unitary transformations preserve the length of the vectors applied on
$$||\,|\psi\rangle\,|| = ||U\,|\psi\rangle\,||$$
4. Unitary transformations preserve the angle of the vectors applied on
$$\langle U\phi|U\psi\rangle = \langle\phi|\psi\rangle$$

All operations in a quantum circuit are unitary and therefore reversible [8].

# Chapter 3

# Using Grover's Algorithm for SAT-problems

Grover's Algorithm was introduced by Lov Grover in 1996 [11]. He showed that quantum computers are able to explore a search space much faster than classical computers by the use of superposition, an oracle and amplitude amplification. A classical computer needs to check $\frac{N+1}{2}$ elements on average to find a desired element in a search space of $N$ elements. A quantum computer with the use of Grover's Algorithm can reduced this to $\sqrt{N}$ tries [9]. How this algorithm works in detail is discussed in this chapter.

## 3.1 Motivation

Although SAT are a intensively studied topic of theoretical computer science. In fact, the *NP*-hardness of a problem is often proved by reducing the problem to a SAT problem. Nevertheless, SAT problems are not only relevant in theory, but also in practice. In the following section, a few practical problems are presented that can be reformulated into SAT problems[12].

### 3.1.1 Semiconductors

The semiconductor industry has different applications of SAT problems. A common example is the validation of correctness of a hardware design (but not the hardware itself, which is subject to physical effects). For this, the circuit's logic is represented by a transition relation. A SAT solver would then validate whether a given property holds true for all, or a range of input values.

### 3.1.2 Artificial intelligence

In the field of artificial intelligence, SAT occur in different forms. Here, we only introduce a few. *Constraint satiscaction problems*, or CSP, which are common in AI research, can often easily be reformulated into SAT problems. For instance, Sudoku can be represented as CSP, and can thus be solved by a SAT solver. Furthermore, many logic-based AI approaches i.e. inference machines, expert systems and planning agents can be formulated as Boolean satisfiability problems by using resolution calculus. [13].

### 3.1.3 Bioinformatics

In Bioinformatics, the influence of DNA mutations that may cause genetic diseases are described by haplotype data. This data can be gathered using a procedure that can be represented as SAT problem. *Lynce and Marques-Silva* [14] demonstrate that this method is faster than existing approaches.

### 3.1.4 Others

In manufacturing, the planning of assembly lines and robots may be modelled as SAT problems, by which impossible configurations can be determined. Other examples are the planning of meetings, the loading of vehicles, route planning and so forth. As SAT problems are common in practical applications, competitive SAT solvers exist, able to solve thousands of clauses. A variety of such solvers are presented by *Gomes et al.* [15]. There are even hardware-accelerated approaches.

## 3.2 Grover's algorithm in brief

Grover's Algorithm uses the concept of phase kickback to mark a searched state with the global phase -1. Afterwards it uses the the diffusion operator to rotate the value of the amplitudes for all the states around the mean. By that the negative amplitude is increased while the others are decrease. This algorithm has to be applied $\sqrt{N}$ times, when $N = 2^n$ with $n$ is the number of bits to represent the states. If there is more than one solution, the number of repetitions of the algorithm is equal to $\left\lfloor \frac{\pi}{4}\sqrt{\frac{N}{s}} \right\rfloor$ with $s$ as the number of states of interest [16]. Because the number of solutions is typically not known beforehand, there exist different algorithms to evaluate the optimal number of repetitions [17].

### 3.2.1 Oracle

The function f($|x\rangle$) maps from the set of all possible states for a given amount of qubits, to 1 or 0:

$$\begin{aligned} f(x) : |\{0,1\}^n\rangle &\to 0,1 \\ x &\mapsto f(x) \end{aligned}$$

The oracle function $U_f$ performs the phase kickback on all states, for which $f(|x\rangle) = 1$

$$U_f \left|x\right\rangle \left|y\right\rangle = \left|x\right\rangle \left|y \oplus f(x)\right\rangle$$

### 3.2.2 Phase Kickback

Using a CNOT gate on a qubit in state $|-\rangle$ leaves the target qubit unchanged while applying a negative phase to the control qubit:

$$\begin{aligned} CNOT \left|-1\right\rangle &= X \left|-\right\rangle \otimes \left|1\right\rangle \\ &= - \left|-\right\rangle \otimes \left|1\right\rangle \\ &= \left|-\right\rangle \otimes - \left|1\right\rangle \end{aligned}$$

therefore we need the target qubit $|y\rangle$ in the state $|-\rangle$ to apply a negative relative phase to all states for which the oracle returns 1.

### 3.2.3 Diffusion Operator

The diffusion operator rotates the amplitudes of all the possible states around the mean of all the amplitudes. This process is called amplitude amplification because it amplifies the amplitude of the marked state [9].

if $|s\rangle$ is a superposition over all possible states with n bits

$$\left|s\right\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} \left|x\right\rangle$$

then

$$U_s = 2 \left|s\right\rangle \left\langle s\right| - I$$

## 3.3 SAT-Solver

SAT problems are defined as a formula consisting of variables (also referred to as literals) $v_1, v_2, ..., v_n$ that can represent $true$ or $false$, conjunctions $\wedge$, disjunctions $\vee$ and negations $\neg$. The goal is to find for each variable an assignment $true$ or $false$, such that the formula evaluates to $true$. Variables are often grouped into clauses, containing disjunct variables $x$, while the clauses $c \in C$ where $|C| = m$ are conjuncted and the variables may be negated. Of course, variables are allowed to be in multiple clauses. Note that the number of variables in each clause must not be identical.

$$(v_{1,1} \vee ... \vee v_{1,n_1}) \wedge ... \wedge (v_{m,1} \vee ... \vee v_{m,n_m}) \tag{3.1}$$

For the formula to be $true$, each clause must evaluate to $true$. The form in equation 3.1 is called conjunctive normal form, or CNF and is a standard representation for SAT problems. Note that each Boolean formula can be transformed into a CNF using logical transformations. This procedure is however omitted here, as many problems are already formulated in this form.

Boolean satisfiability problems essentially are search problems, where solutions are represented as bit strings, that are sequences of binary variables. The Boolean formula is expressed by an oracle which consists of logical operators readily available on classical and quantum computers. Grover's algorithm exploits the qubit's superposition and entanglement principles to prompt the oracle to find satisfying solutions. Each variable and each clause are assigned a qubit. A single qubit is used to represent the result, that is 1 when the Boolean formula is satisfied, 0 otherwise. The structure of this high-level overview is shown in figure 3.1.

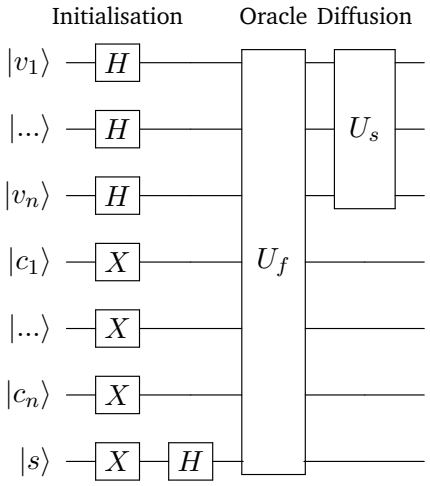



Figure 3.1: Schematic overview of Grover's algorithm

For the following explanations, we introduce a simple example SAT, given by 3.2.

$$B = (\neg A \vee \neg B) \wedge (A \vee B) \wedge A \tag{3.2}$$

First, we introduce a technique to implement the oracle with logic gates. Boolean conjunctions can be implemented conveniently using controlled gates, i.e. CNOT or Toffoli

gates. Disjunctions, however, are more difficult to implement because quantum computers lack an OR gate equivalent, that must be constructed from conjunctions and negations. This procedure can be thought of replacing disjunctions inside clauses with conjunctions, such that the operations in the Boolean formula can be realised by equivalent quantum gates. For this, we preprocess the Boolean formula by applying De Morgan's law on each clause, as demonstrated in equation 3.3. One can see that the clauses on the right-hand side are preceded by a negation, which is realised in the quantum circuit by applying a X-gate on the clause qubits. Additionally, each variable in the clause is negated as well. Although syntactically equivalent, we call the new clause $B'$[1].

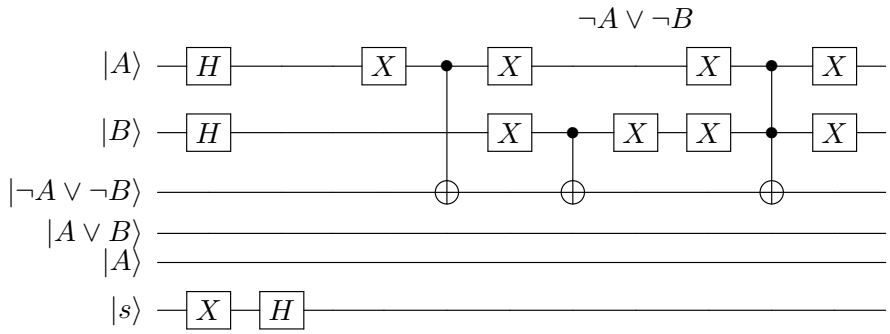
$$\underbrace{(\neg A \vee \neg B) \wedge (A \vee B) \wedge A}_{B} = \underbrace{\neg(A \wedge B) \wedge \neg(\neg A \wedge \neg B) \wedge \neg(\neg A)}_{B'} \tag{3.3}$$

The procedure described above can be comprehended as follows. If a state, i.e. an assignment of the variables $|x\rangle$ fulfils a given clause, we perform a bit flip on the qubit for this clause $|c\rangle$ from $|1\rangle$ to $|0\rangle$.

**Example.** Consider the Boolean formula

$$(\neg A \vee \neg B) \wedge (A \vee B) \wedge A.$$

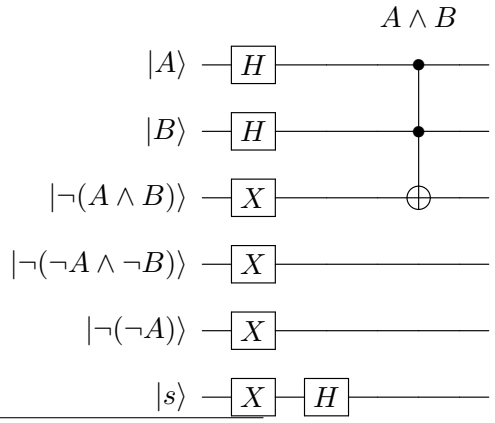
Originally the clauses have to be true, leading to the following circuit for the first clause:




but could be written, using De Morgan's laws, as:

$$B' = \neg(A \wedge B) \wedge \neg(\neg A \wedge \neg B) \wedge \neg(\neg A).$$

So instead of switching the state $|(\neg A \vee \neg B)\rangle$ on, when $(\neg A \vee \neg B)$ is true, we switch the state $|\neg(A \wedge B)\rangle$ off if $(A \wedge B)$ is true. This circuit would look like this for the first clause:




[1]The disjunctions could equivalently be replaced when translating the Boolean formula into a quantum circuit, the preprocessing approach is however more intuitive to understand.

Figure 3.2 shows the oracle's workings in more detail. The construction of the clauses is abstracted with the $U_c$ operation. The principle for the SAT-solver is applying the gates to switch the state of the constraints to $|1\rangle$ if $|v_1\rangle$ to $|v_n\rangle$ have satisfying values assigned. As the clauses are all conjunct, we can easily apply a CNOT-gate on all the clause qubits $|c\rangle$ on the result qubit $|s\rangle$ as target, that is initialised in the state with the negative phase $|-\rangle$.

This negative phase causes, together with the CNOT-gate, a phase kickback to the states that have activated the CNOT-gate, which also are the sought states that satisfy the Boolean formula. To propagate this negative phase $|-\rangle$ to the variable qubits, the clause operation $U_c$ is applied again. This procedure is known as *uncomputation*.

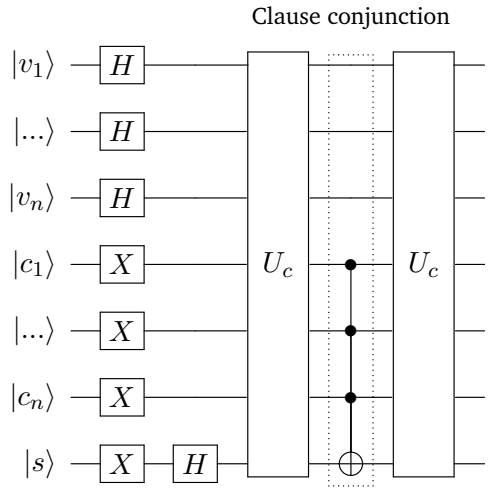



Figure 3.2: Detailed view of oracle. $U_C$ constructs the clauses, the MCT-operation in the centre constructs the conjunction of the clauses and the final $U_C$ propagates the negative phase back to the individual variable qubits.

## 3.4 CSP-Solver

CSP are a prominent problem class in AI research. The *graph colouring problem* is a mostly pedagogical instance of a CSP, but it is chosen as example in the following chapter because of its pureness and short abstraction from a CSP. CSPs itself can easily be formulated as Boolean formula for which a satisfying assignment is sought. Therefore the goal is to reduce the CSP to a SAT problem. Figure 3.3 shows the example of the graph colouring problem this section is about. The goal is to assign every section of the Australian continent a colour but no adjacent tiles are allowed to have the same colour.

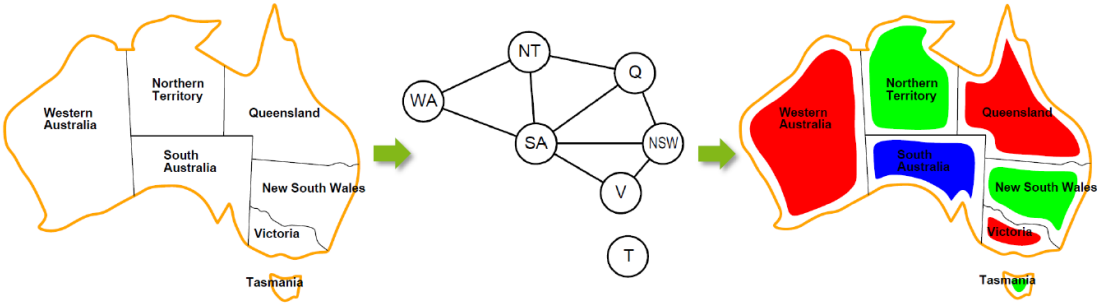



Figure 3.3: A graph colouring problem as a CSP problem instance [13]

When formulating a CSP first the variables are determined. The variables correspond to the different entities affected by the constraints.

$$\text{Variables: } WA, NT, Q, NSW, V, SA, T$$

The next step is to set the domains. The domains correspond to the states a variable can be in. In this example the selection of colours are red, green and blue.

$$\text{Domains:} D_i = \{red, green, blue\}$$

The constraints define the rules under which the domains have to be assigned to the variables. In this case two adjacent regions must have different colours.

$$WA \neq NT \Rightarrow (WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), ...\}$$

to formulate this problem as a SAT we first have to formulate the constraints as a logical formula. Therefore the domains are encoded in binary. Because there are 3 different domains we have to use $log_2(3) \Rightarrow 2$ bits to encode them.

$$\text{Domains:} D_i = \{00, 01, 10\}$$

To fulfil the constraint of two adjacent tiles having different colours the constraint is formulated as follows, explained on the example of $WA \neq NT$.

| $WA_0 WA_1$ | $NT_0 NT_1$ | $WA \neq NT$ |
|---|---|---|
| 0 0 | 0 0 | F |
| 0 0 | 0 1 | T |
| 0 0 | 1 0 | T |
| 0 0 | 1 1 | F |
| 0 1 | 0 0 | T |
| 0 1 | 0 1 | F |
| 0 1 | 1 0 | T |
| 0 1 | 1 1 | F |
| 1 0 | 0 0 | T |
| 1 0 | 0 1 | T |
| 1 0 | 1 0 | F |
| 1 0 | 1 1 | F |
| 1 1 | 0 0 | F |
| 1 1 | 0 1 | F |
| 1 1 | 1 0 | F |
| 1 1 | 1 1 | F |

This are all the possible states the two variables are allowed to be in. This truth table formulated as a logical formula in CNF is the starting point of the SAT-Problem.

$$\begin{aligned}(WA \neq NT) =& (\neg WA_0 \wedge \neg WA_1 \wedge \neg NT_0 \wedge NT_1) \\ & \vee (\neg WA_0 \wedge \neg WA_1 \wedge NT_0 \wedge \neg NT_1) \\ & \vee (\neg WA_0 \wedge WA_1 \wedge \neg NT_0 \wedge \neg NT_1) \\ & \vee (\neg WA_0 \wedge WA_1 \wedge NT_0 \wedge \neg NT_1) \\ & \vee (WA_0 \wedge \neg WA_1 \wedge \neg NT_0 \wedge \neg NT_1) \\ & \vee (WA_0 \wedge \neg WA_1 \wedge \neg NT_0 \wedge NT_1)\end{aligned}$$

this can be formulated shorter:

$$\begin{aligned}(WA \neq NT) =& (WA_0 \wedge \neg WA_1 \wedge \neg NT_0) \\ & \vee (\neg WA_0 \wedge ((\neg WA_1 \wedge ((\neg NT_0 \wedge NT_1) \\ & \vee (NT_0 \wedge \neg NT_1))) \vee (WA_1 \wedge \neg NT_1)))\end{aligned}$$

and as CNF:

$$\begin{aligned}(WA \neq NT) =& (WA_0 \vee WA_1 \vee NT_0 \vee NT_1) \\ & \wedge (\neg WA_0 \vee \neg WA_1) \\ & \wedge (\neg WA_1 \vee \neg NT_1) \\ & \wedge (\neg WA_0 \vee NT_0) \\ & \wedge (\neg NT_0 \vee \neg NT_1)\end{aligned}$$

As last step, De Morgan's law is used to formulate the clauses as negations as already explained for the SAT problem.

$$\begin{aligned}(WA \neq NT) =& \neg(\neg WA_0 \wedge \neg WA_1 \wedge \neg NT_0 \wedge \neg NT_1) \\ & \wedge \neg(WA_0 \wedge WA_1) \\ & \wedge \neg(WA_1 \wedge NT_1) \\ & \wedge \neg(WA_0 \wedge NT_0) \\ & \wedge \neg(NT_0 \wedge NT_1)\end{aligned}$$

now this formula has to be implemented for all the constraints - all the paths between our nodes. The transformations were made by the help of *Gottschall* [18].

## 3.5 Complexity

because every variable can be either True or False, the complexity of the SAT-Problem scales with the variables. In case of n variables there would be $2^n$ possible assignments for all the variables. Therefore the complexity of a SAT-Problem with a classical computer is

$$\mathcal{O}(2^n)$$

because quantum computers can reduce the complexity of a unstructured search problem, using Grover's algorithm, from $\mathcal{O}(N)$, with $N$ being the number of elements in the search space, to $\mathcal{O}(\sqrt{N})$. Therefore we can reduce the complexity of the SAT-Problem using a quantum computer:

$$\mathcal{O}(2^n) \Rightarrow \mathcal{O}(\sqrt{2^n})$$

## 3.6 The Experiment using Qiskit and the IBM qasm-simulator

Because the variable $T$ can be assigned any possible value, and is not limited by any constraints, we leave it out of the experiment. For all the other variables, two qubits were used each, to differentiate between the colours assigned to each variable.

In python several imports must be done to work with the Qiskit library. The necessary imports are listed in Listing 3.1.

Listing 3.1: import necessary libraries

```
# Initialization
import matplotlib.pyplot as plt
import numpy as np
import math
%matplotlib inline
%config InlineBackend.figure_format = 'svg' # Makes the
    images look nice

# importing Qiskit
from qiskit import IBMQ, Aer
from qiskit.providers.ibmq import least_busy
from qiskit import QuantumCircuit, ClassicalRegister,
    QuantumRegister, AncillaRegister, execute

# import basic plot tools
from qiskit.visualization import plot_histogram

# import Operator
from qiskit.quantum_info.operators import Operator

#SAT
import numpy as np
from qiskit import BasicAer
from qiskit.visualization import plot_histogram
from qiskit.aqua import QuantumInstance
from qiskit.aqua.algorithms import Grover
```

```
25 from qiskit.aqua.components.oracles import
      LogicalExpressionOracle, TruthTableOracle
26
27 # Real Device
28 from qiskit.tools.monitor import job_monitor
29 from qiskit.compiler import transpile
30
31 # State Vector
32 from qiskit_textbook.tools import vector2latex
33
34 import collections
35 from collections import OrderedDict
36 import operator
```

The above declared variables are initialised to create our circuit. As explained above, we need two qubits for each variable to encode them into the different colours. The corresponding code is described in listing 3.2.

Listing 3.2: initializing variables

```
1 # assigning each variable specific qubits
2 WA, NT, Q, NSW, V, SA = [0,1], [2,3], [4,5], [6,7], [8,9],
     [10,11]
3 # combining the qubits of the variables to one array
4 variables = WA + NT + Q + NSW + V + SA
5
6 # assigning each color a two bit code
7 red, green, blue, fail = '00', '01', '10', '11'
8 colors = {red: 'red', green: 'green', blue: 'blue', fail: '11
     ' }
```

Subsequently, `QuantumRegisters` are created for each variable, constraint, ancillary (`ancbit`) and the qubit representing the solution (`solutionbit`). They form the `QuantumCircuit`. Listing 3.3 shows the code to accomplish this task.

Listing 3.3: Creating the QuantumCircuit

```
1 # creating QuantumReigister for each variable
2 wa = QuantumRegister(2, 'wa')
3 nt = QuantumRegister(2, 'nt')
4 q = QuantumRegister(2, 'q')
5 nsw = QuantumRegister(2, 'nsw')
6 v = QuantumRegister(2, 'v')
7 sa = QuantumRegister(2, 'sa')
8
9 # creating QuatumRegister for each constraint
10 want = QuantumRegister(1, 'want')
11 wasa = QuantumRegister(1, 'wasa')
12 ntq = QuantumRegister(1, 'ntq')
13 ntsa = QuantumRegister(1, 'ntsa')
14 saq = QuantumRegister(1, 'saq')
15 sansw = QuantumRegister(1, 'sansw')
16 sav = QuantumRegister(1, 'sav')
17 qnsw = QuantumRegister(1, 'qnsw')
```

```
18 nswv = QuantumRegister(1, 'nswv')
19
20 # creating QuantumRegister for the ancbits and the
      solutionbit
21 anc = QuantumRegister(7, 'anc')
22 constraintAncs = QuantumRegister(5, 'constraintAnc')
23 sol = QuantumRegister(1, 's')
24
25 # creating a ClassicalRegister to measure the qubits of the
      variables
26 measure = ClassicalRegister(12, 'measure')
27
28 # building the QuantumCircuits from the QuantumRegister
29 qc = QuantumCircuit(wa, nt, q, nsw, v, sa, want, wasa, ntq,
      ntsa, saq, sansw, sav, qnsw, nswv, anc, constraintAncs,
      sol, measure)
30
31 # creating an array which contains the QuantumRegisters of
      the constrains
32 constraints = [want, wasa, ntq, ntsa, saq, sansw, sav, qnsw,
      nswv]
```

### The Qubits in the Circuit

The code from listing 3.3 leads to the following circuit.

$$
\begin{array}{r}
|WA_0\rangle \ — \\
|WA_1\rangle \ — \\
|NT_0\rangle \ — \\
|NT_1\rangle \ — \\
|Q_0\rangle \ — \\
|Q_1\rangle \ — \\
|NSW_0\rangle \ — \\
|NSW_1\rangle \ — \\
|V_0\rangle \ — \\
|V_1\rangle \ — \\
|SA_0\rangle \ — \\
|SA_1\rangle \ — \\
|WA \neq NT\rangle \ — \\
|WA \neq SA\rangle \ — \\
|NT \neq Q\rangle \ — \\
|NT \neq SA\rangle \ — \\
|SA \neq Q\rangle \ — \\
|SA \neq NSW\rangle \ — \\
|SA \neq V\rangle \ — \\
|Q \neq NSW\rangle \ — \\
|NSW \neq V\rangle \ — \\
|anc_0\rangle \ — \\
|anc_1\rangle \ — \\
|anc_2\rangle \ — \\
|anc_3\rangle \ — \\
|anc_4\rangle \ — \\
|anc_5\rangle \ — \\
|anc_6\rangle \ — \\
|conAnc_0\rangle \ — \\
|conAnc_1\rangle \ — \\
|conAnc_2\rangle \ — \\
|conAnc_3\rangle \ — \\
|conAnc_4\rangle \ — \\
|sol\rangle \ — \\
12 classicalBits \ —
\end{array}
$$

The `anc` qubits are used to store the combined state of the variable qubits, to evaluate later on whether all requirements of a constraint are met.

The `conAnc` qubits are used so store the combined state of the constraint qubits, to evaluate later on whether all requirements for the solution are met.

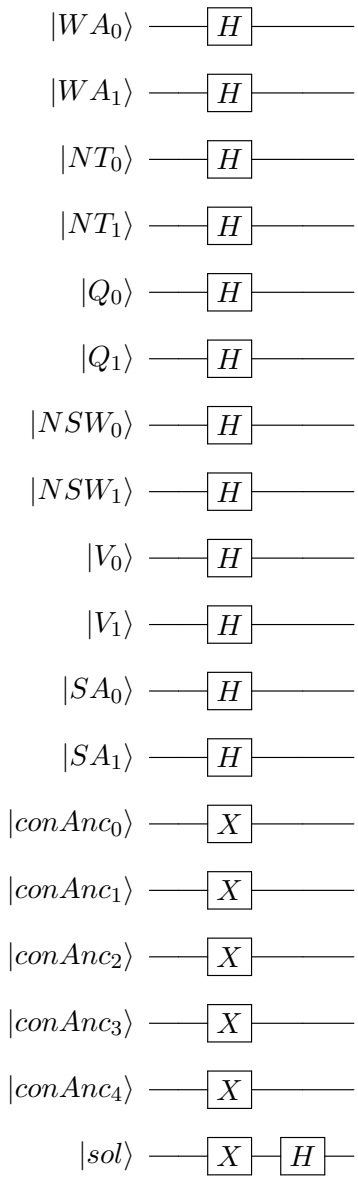
Because of comprehensibility, future circuit-diagrams will not show the whole circuit, but only the relevant qubits.

### Instantiation

First the qubits for the variables are brought into superposition $|+\rangle$, all the constrain qubits in state $|1\rangle$ and the solution qubit in state $|-\rangle$. The method defined to carry out this task is described in listing 3.4.

Listing 3.4: Instantiating the `QuantumCircuit`.

```
1 def instantiate(qc, constraints, variables):
2   qc.h(variables)
3   qc.x(sol)
4   qc.h(sol)
5   for constraint in constraints:
6     qc.x(constraint)
```

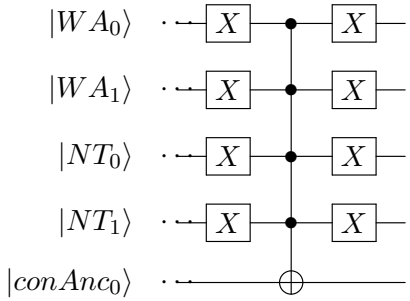



**First Clause**

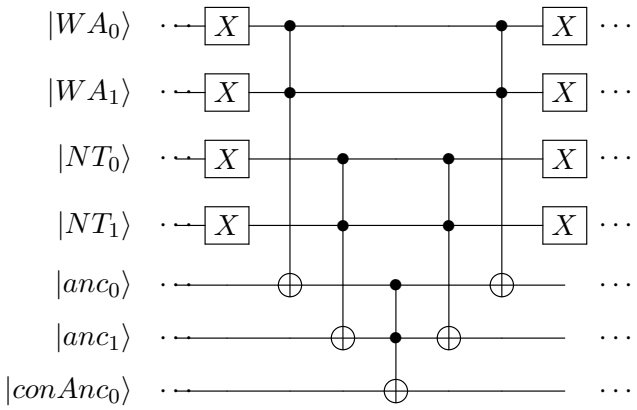
The first clause of the constraint $WA \neq NT$ is

$$\neg(\neg WA_0 \wedge \neg WA_1 \wedge \neg NT_0 \wedge \neg NT_1)$$

Because the formula evaluates to true if $(\neg WA_0 \wedge \neg WA_1 \wedge \neg NT_0 \wedge \neg NT_1)$ is false, the constraint qubit is flipped from $|1\rangle$ to $|0\rangle$ if it is true.



because Qiskit internally breaks down MCT to CNOT or Toffoli gates for execution on a quantum computer, the circuit looks as follows [8, 19].




The code in listing 3.5 defines a method to implement the first clause for any two variables submitted to the method.

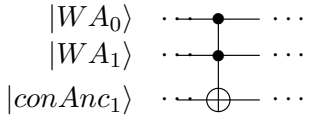
Listing 3.5: implementing the first clause

```
1  def firstClause(qc, variable1, variable2, constraintAncs,
       ancBits):
2    qc.x(variable1)
3    qc.x(variable2)
4    qc.ccx(variable1[0], variable1[1], ancBits[0])
5    qc.ccx(variable2[0], variable2[1], ancBits[1])
6
7    qc.ccx(ancBits[0], ancBits[1], constraintAncs[0])
8
9    qc.ccx(variable2[0], variable2[1], ancBits[1])
10   qc.ccx(variable1[0], variable1[1], ancBits[0])
11   qc.x(variable1)
12   qc.x(variable2)
```

**Second Clause**

$$\neg(WA_0 \wedge WA_1)$$

Listing 3.6 shows the implementation of the second clause. The code contains a single Toffoli gate making sure no variable is assigned the invalid colour code 11.



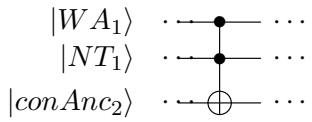

Listing 3.6: implementing the second clause

```
1  def secondClause(qc, variable, constraintAncs):
2    qc.ccx(variable[0], variable[1], constraintAncs[1])
```

### Third Clause

The third clause is implemented as shown in Listing 3.7.

$$\neg(WA_1 \wedge NT_1)$$



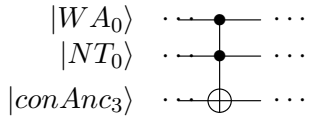

Listing 3.7: implementing the third clause

```
1 def thirdClause(qc, variable1, variable2, constraintAncs):
2   qc.ccx(variable1[1], variable2[1], constraintAncs[2])
```

### Fourth Clause

Although the implementation of the third clause matches the code for the fourth clause they are implemented separately to avoid confusion and make the implementation more clear. The code is shown in Listing 3.8.

$$\neg(WA_0 \wedge NT_0)$$



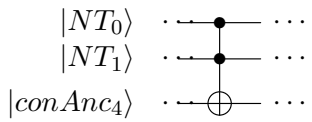

Listing 3.8: implementing the fourth clause

```
1 def fourthClause(qc, node1, node2, constraintAncs):
2     qc.ccx(node1[0], node2[0], constraintAncs[3])
```

### Fifth Clause

The implementation of the fifth clause is the same as the code for the second clause. Again, for clarity the two clauses are implemented separately. The code for the fifth clause is shown in listing 3.9.

$$\neg(NT_0 \wedge NT_1)$$

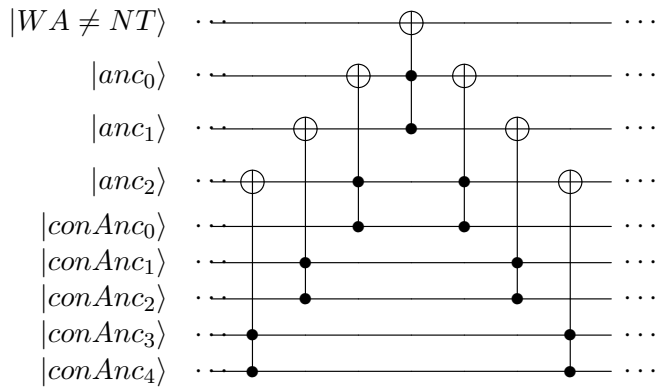



Listing 3.9: implementing the fifth clause

```
1 def fifthClause(qc, node2, constraintAncs):
2     qc.ccx(node2[0], node2[1], constraintAncs[4])
```

### Check Constraint

The next step is to control the state of the constraint $WA \neq NT$ itself. So for all states, for which the first constraint - $WA \neq NT$ - is true, this qubit is set to true or $|1\rangle$. Listing 3.10 shows the implementation in Qiskit.




Listing 3.10: Check single constraint

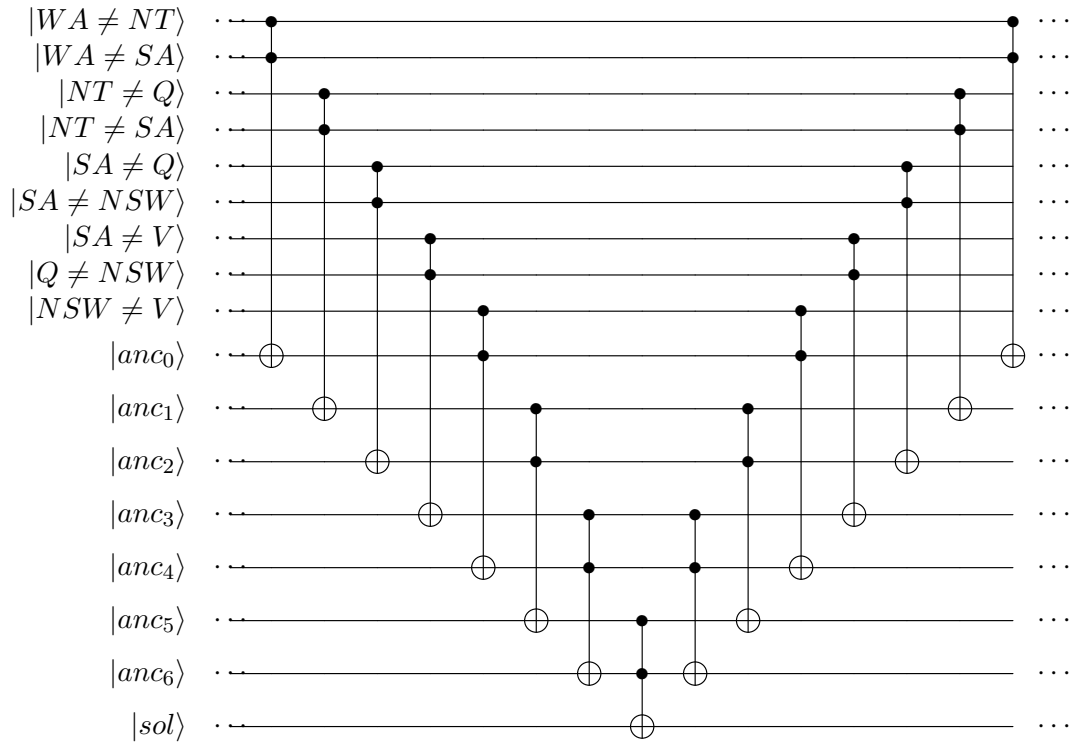
```
1  def checkConstraint(qc, constraintBit, constraintAncs,
       ancBits):
2    qc.ccx(constraintAncs[0], constraintAncs[1], ancBits[0])
3    qc.ccx(constraintAncs[2], constraintAncs[3], ancBits[1])
4    qc.ccx(constraintAncs[4], ancBits[0], ancBits[2])
5
6    qc.ccx(ancBits[1], ancBits[2], constraintBit)
7
8    qc.ccx(constraintAncs[4], ancBits[0], ancBits[2])
9    qc.ccx(constraintAncs[2], constraintAncs[3], ancBits[1])
10   qc.ccx(constraintAncs[0], constraintAncs[1], ancBits[0])
```

### Revert Gates

After all the qubits are set for the current constraint, the `conAnc-bits` are left in a messy state. All the operations on them need to be reverted to use them again for the next constraint. In quantum computing reverting a operation is as easy as applying all the operations, all the clauses again. Here, first to fifth clause has to be applied again. After that the `conAnc-bits` are again in the initial state $|1\rangle$ and the circuit is ready for the next constraint. The implementation of the first to the fifth clause and the reversion has to be repeated for all further constraints.

### Check all Constraints, Phase Kickback

After performing the above described algorithm on all constrains, the phase kickback has to be applied on all states, for which the constraints are met i.e. all adjacent tiles have different colours.




After performing the phase kickback, every gate needs to be executed again in order to apply the global phase to the right states. The code to do the task is described in listing 3.11.

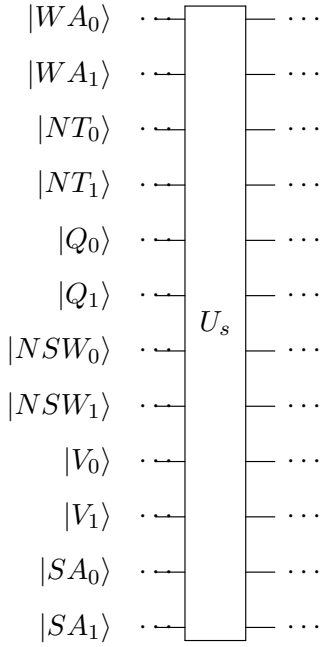
Listing 3.11: check all constraints

```
def checkConstraints(qc, constraints, ancBits, sol):
  qc.ccx(constraints[0], constraints[1], ancBits[0])
  qc.ccx(constraints[2], constraints[3], ancBits[1])
  qc.ccx(constraints[4], constraints[5], ancBits[2])
  qc.ccx(constraints[6], constraints[7], ancBits[3])
  qc.ccx(constraints[8], ancBits[0], ancBits[4])
  qc.ccx(ancBits[1], ancBits[2], ancBits[5])
  qc.ccx(ancBits[3], ancBits[4], ancBits[6])

  qc.ccx(ancBits[5], ancBits[6], sol)

  qc.ccx(ancBits[3], ancBits[4], ancBits[6])
  qc.ccx(ancBits[1], ancBits[2], ancBits[5])
  qc.ccx(constraints[8], ancBits[0], ancBits[4])
  qc.ccx(constraints[6], constraints[7], ancBits[3])
  qc.ccx(constraints[4], constraints[5], ancBits[2])
  qc.ccx(constraints[2], constraints[3], ancBits[1])
  qc.ccx(constraints[0], constraints[1], ancBits[0])
```

### Diffusion, Amplitude Amplification

The next step is to rotate the amplitudes around the mean. More precisely, because the searched states have now a negative phase, this amplitudes will be increased by rotation

around the mean of all amplitudes, while the positive amplitudes will be decreased [9].




To implement the diffusion operator as a matrix in Qiskit, the code from Listing 3.12 is used.

Listing 3.12: Diffusion operator $U_s$

```
n = len(variables)
N = 2 ** n
Us = []

for i in range(N):
  us = []
  for j in range(N):
    if (i == j):
      us.append(2/N-1)
    else:
      us.append(2/N)
  Us.append(us)

Us = Operator([*((Us[i]) for i in range(N))])
```

### Repetitions

The whole above described algorithm, except the instantiation, generally needs to be repeated $\sqrt{N}$-times to increase the amplitude of the searched state sufficiently to recognise it as a solution.

The final code looks as follows. Although the algorithm should be repeated $\sqrt{N}$ times for one solution, because of practical limitations, the number of iterations $N = 4$ is chosen, for which the experiment also returns the right result. The code to implement the algorithm is described in listing 3.13.

Listing 3.13: Constriction of the complete circuit

```
sqrtN = 4 #math.floor(math.sqrt(N))
```

```
2
3  #the instantiation is performed
4  instantiate(qc, constraintAncs, variables)
5
6  for i in range(sqrtN):
7
8  # all constraints are implemented
9    implementConstraint(qc, wa, nt, want, constraintAncs, anc)
10   implementConstraint(qc, wa, sa, wasa, constraintAncs, anc)
11   implementConstraint(qc, nt, q, ntq, constraintAncs, anc)
12   implementConstraint(qc, nt, sa, ntsa, constraintAncs, anc)
13   implementConstraint(qc, sa, q, saq, constraintAncs, anc)
14   implementConstraint(qc, sa, nsw, sansw, constraintAncs, anc
       )
15   implementConstraint(qc, sa, v, sav, constraintAncs, anc)
16   implementConstraint(qc, q, nsw, qnsw, constraintAncs, anc)
17   implementConstraint(qc, nsw, v, nswv, constraintAncs, anc)
18
19 # the constraints are checked and if all are fulfiled the
     phase kickback is performed on the corresponding states
20   checkConstraints(qc, constraints, anc, sol)
21
22 # the constraints are implemented again to assign the
     negative phase to the right states
23   implementConstraint(qc, wa, nt, want, constraintAncs, anc)
24   implementConstraint(qc, wa, sa, wasa, constraintAncs, anc)
25   implementConstraint(qc, nt, q, ntq, constraintAncs, anc)
26   implementConstraint(qc, nt, sa, ntsa, constraintAncs, anc)
27   implementConstraint(qc, sa, q, saq, constraintAncs, anc)
28   implementConstraint(qc, sa, nsw, sansw, constraintAncs, anc
       )
29   implementConstraint(qc, sa, v, sav, constraintAncs, anc)
30   implementConstraint(qc, q, nsw, qnsw, constraintAncs, anc)
31   implementConstraint(qc, nsw, v, nswv, constraintAncs, anc)
32
33 # the diffusion operator is implemented to amplify the
     negative states
34   qc.unitary(Us, variables, label='Us')
```

### Measurement

At the end, the result has to be measured. Because the states for which all constraints are true were amplified, the probabilities to measure those states was enlarged. The measurement process in Qiskit is described in listing 3.14.

Listing 3.14: implement measurement

```
1 # implements the measurement in the circuit
2 qc.measure([i for i in range(12)], measure)
```

The circuit is now completeley implemented and can be executed. Because there are currently no quantum computers with a sufficient amount of qubits to perform the task,

the classical simulator *qasm_simulator* from the Qiskit library is selected. To run the code on the selected device, the code from listing 3.15 is used.

Listing 3.15: run experiment on device

```
simulator = Aer.get_backend('qasm_simulator')
backend_opts_mps = {"method":"matrix_product_state"}

shots = 4096

# The result of the circuit is safed in the variable result
result = execute(qc, simulator, backend_options=
    backend_opts_mps, shots=shots).result()

# the counts of the different states is saved into the
    variable counts
counts = result.get_counts(qc)
```

The resulting bit strings are counted and the corresponding colour combination is assigned. The code to perform the task is shown in listing 3.16.

Listing 3.16: assign corresponding colors to results

```
names = {'WA': WA, 'NT': NT, 'Q': Q, 'NSW': NSW, 'V': V, 'SA'
    : SA}
solution_colors = []
for solution in result:
  for name in names:
    color = solution[names[name][0]] + solution[names[name
       ][1]]
    solution_colors.append(colors[color])
```

Finally, the solution of our circuit, which appeared most frequently is printed as shown in listing 3.17.

Listing 3.17: create histogram of the result

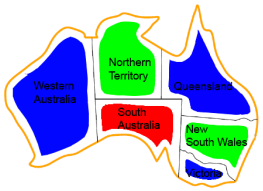
```
maxValue = 0

for key in counts:
if counts[key] > maxValue:
maxValue = counts[key]

possible_solutions = {}
for key in counts:
if counts[key] > maxValue / 2:
possible_solutions[key] = counts[key]

names = {'WA': WA, 'NT': NT, 'Q': Q, 'NSW': NSW, 'V': V, 'SA'
    : SA}
# because the LSB is the rightmost bit WA, [0, 1] = 11, 10
for solution in possible_solutions:
for name in names:
color = solution[(n-1)-names[name][0]] + solution[(n-1)-names
    [name][1]]
```

```
17 print(name + " = " + colors[color])
18
19 # The bitstring and the amount it appeard
20 print(solution + ", " + str(possible_solutions[solution]) + "
       \n")
```

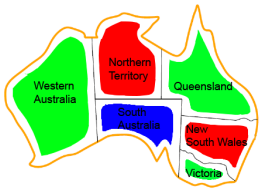
### 3.6.1 Solution

The bit strings which appeared most frequently in the simulation results are shown below. Out of $2^{12}$ possible assignments for the variables and 4096 results from our circuit, almost all of our solutions appeared more than 80 times. Hence, they appeared significantly more often than what would be probable in a random experiment and can therefore by assumed to be correct solutions.

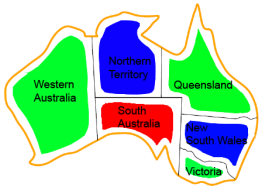
$$WA = blue$$
$$NT = green$$
$$Q = blue$$
$$NSW = green$$
$$V = blue$$
$$SA = red$$
$$000110011001, 87$$



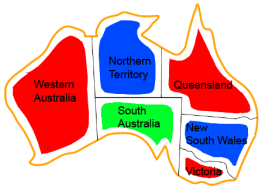

$$WA = green$$
$$NT = red$$
$$Q = green$$
$$NSW = red$$
$$V = green$$
$$SA = blue$$
$$011000100010, 58$$



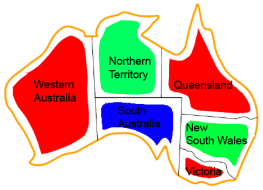

$$WA = green$$
$$NT = blue$$
$$Q = green$$
$$NSW = blue$$
$$V = green$$
$$SA = red$$
$$001001100110, 85$$



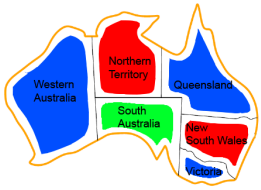

$$WA = red$$
$$NT = blue$$
$$Q = red$$
$$NSW = blue$$
$$V = red$$
$$SA = green$$
$$100001000100, 80$$




$$WA = red$$
$$NT = green$$
$$Q = red$$
$$NSW = green$$
$$V = red$$
$$SA = blue$$
$$010010001000, 91$$




$$WA = blue$$
$$NT = red$$
$$Q = blue$$
$$NSW = red$$
$$V = blue$$
$$SA = green$$
$$100100010001, 85$$



## 3.7 Conversion algorithm

In this section, we present a generic procedure to solve a Boolean satisfiability problem, in future referred to as SAT, using Grover's algorithm. We call this procedure *conversion algorithm* to better distinguish it from Grover's algorithm. In contrast to classical SAT solvers, we pursue a brute-force approach enabled by quantum parallelism. Finally, we implement the conversion algorithm with Python and for two hardware providers.

**Code reference.** The conversion algorithm is available as `SAT/Qiskit/boolean_oracle.py` as Qiskit variant and `SAT/Braket/boolean_oracle_braket.py` as Amazon Braket variant.

SAT problems are further categorised. For instance in the 3-SAT problem, there are exactly three variables in each clause. The number of variables in each clause is however not relevant for this algorithm, the only precondition necessary is that the formula is in CNF.

For the algorithm, we consider a generic Boolean formula in CNF in the form

$$(v_{1,1} \vee ... \vee v_{1,n_1}) \wedge ... \wedge (v_{m,1} \vee ... \vee v_{m,n_m}) \tag{3.4}$$

0. *Preparatory:* Parse the SAT expression, determine the variables and clauses and create utility data structures

1. For each variable $v_n$, create a qubit $|v_n\rangle$, initialise it in state $|0\rangle$ and apply a Hadamard gate to it

2. For each clause, create a new qubit $|c_m\rangle$, and initialise it in state $|1\rangle$, calculate the negated clause $\neg c_m$ and store it for later

3. Initialise a final qubit $|s\rangle$, initialise it in state $|1\rangle$ and apply a $H$ gate to it. Apply a barrier on all qubits

4. For each clause $c_m$, create a multi-controlled Toffoli gate MCT which takes the qubits of the contained variables $|v_n\rangle$ as control bits and the clause's qubit $c_m$ as target bit. If the variable $v_n$ is negated in the clause, surround the MCT gate on the variable's qubit $|v_n\rangle$ with two X-gates. Finish this step by applying a barrier on all qubits

    (a) *Optional:* For each variable's gate $|v_n\rangle$ check whether there are two consecutive X gates i.e. without another gate in between. They can be removed as they cancel each other out

5. Create a multi-controlled Toffoli-gate MCT with each clause qubit $|c_m\rangle$ as input bit and $|s\rangle$ as output bit and apply a barrier on all qubits

6. Apply all the gates created in step 4 one more time and apply a barrier on all qubits

Note that steps 1 to 3 are only done upon creation of the circuit. When the Boolean expression has to be appended to a quantum circuit, only steps 4 to 6 are repeated. Also note that this procedure only implements the quantum oracle and that the additional steps for Grover's algorithm are omitted here and introduced later. A schematic overview of the resulting quantum circuit after a single iteration is given in figure 3.4.

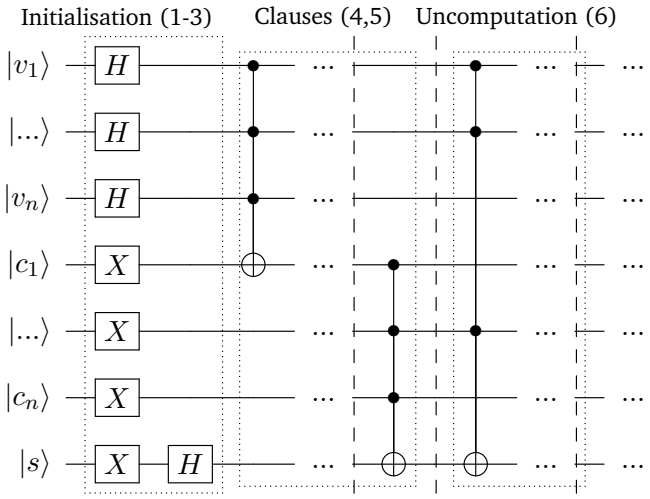



Figure 3.4: Schematic overview of quantum oracle. The initialisation is shown left, the application of clauses is shortened. In the centre, the MCT-gate is visible and followed by the uncomputation.

### 3.7.1 Analysis

The algorithm can roughly be subdivided into three sections. In the first part which consists of steps 1-3, qubits are initialised. In steps 4 and 5, gates are applied and in step 6, the quantum circuit is reversed, which is known as uncomputation. This subdivision is seen in figure 3.4, where they are marked by the dotted boxes. Now, the individual steps are analysed. We use the example formula that we already have used above:

$$(\neg A \vee \neg B) \wedge (A \vee B) \wedge A$$

In the preparatory step 0, the input is stratified and data structures are created for convenience. These tasks are done in the main routine `build_circuit` shown in listing 3.18. In line 2, the string-typed expression is converted into a symbolic expression used by the `sympy`-library. With this symbolic expression, variables and clauses can easily be extracted in lines 4 and 5. In line 6, the `setup_circuit` subroutine is called, which is explained in a following paragraph. Finally, the associative arrays `var_mapping` and `clause_mapping` are created in lines 8 and 9, enabling convenient access to qubit objects given symbolic variables from the expression.

Listing 3.18: Preparatory tasks in the main routine

```
1 def build_circuit(expression, n_solutions=1):
2     symbolic_expression = parse_expr(expression)
3
4     # Extract variables and clauses from symbolic expression
5     variables = sorted(symbolic_expression.binary_symbols,
          key=str)
6     clauses = symbolic_expression.args
7     qc, var_reg, clause_reg, res_reg, cr = setup_circuit(
          variables, clauses)
8
9     # Associate Variables and Clauses to their Quantum
          Register counterparts
```

```
10      var_mapping = dict(zip(variables, var_reg))
11      clause_mapping = dict(zip(clauses, clause_reg))
12      # ...
```

In step 1, variables are initialised in the $|0\rangle$ state such that all following operations are executed on the same basis. For the example, two variable qubits, each for variables $A$ and $B$ are created. Then, the qubits are put into a superposition $|+\rangle$ of $|0\rangle$ and $|1\rangle$ using a H gate. This way, quantum parallelism is possible. In step 2, the clause qubits are created. As there are three clauses in the example, three qubits are initialised in state $|1\rangle$ by applying a X-gate on each qubit. This flip is conducted because of the De Morgan trick explained in the next paragraph. Due to this trick, the clauses left to each qubit are also negated.

Finally, the qubit $|s\rangle$ is used to represent the SAT's result. It is different from the other qubits by being initialised in the $|1\rangle$ state using a X-gate and then put into a superposition $|-\rangle$ with a H-gate. This is important, as this negative phase is applied to the superposition for which the SAT evaluates $true$. The state of the quantum circuit at this point is shown in figure 3.5.

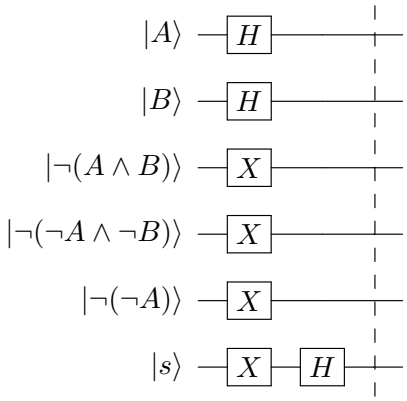



Figure 3.5: The quantum circuit's state after steps 1-3

For the creation of the quantum circuit, the subroutine in listing 3.19 is invoked. It determines the number of qubits needed for variables and clauses in lines 2 and 3 and creates quantum registers in lines 5 to 7. Quantum registers are groups of qubits and are easier to process compared to single qubits. In line 9 classical bits are created to store the measuring. Finally, the quantum circuit is created in line 11.

Listing 3.19: Subroutine for the initial creation of the quantum circuit

```
1  def setup_circuit(variables, clauses):
2      n_vars = len(variables)
3      n_clauses = len(clauses)
4      # Create the registers
5      qr_vars = QuantumRegister(n_vars, name="vars")
6      qr_clauses = QuantumRegister(n_clauses, name="clauses")
7      qr_result = QuantumRegister(1, name="res")
8      # Classical register to store measurement
9      cr = ClassicalRegister(n_vars)
10     # Create the Quantum Circuit
11     qc = QuantumCircuit(qr_vars, qr_clauses, qr_result, cr)
12     return qc, qr_vars, qr_clauses, qr_result, cr
```

To initialise the qubits, the subroutine in listing 3.20 is used. In line 2, the qubits are put into superposition, in line 3 the clauses are negated and in lines 4 and 5, the negative phase is added to the result qubit. Finally, a barrier is applied to separate the implemented gates visually.

Listing 3.20: Subroutine for initialisation of qubits

```
1 def initialise_circuit(qc, var_reg, clause_reg, res_reg):
2     qc.h(var_reg)
3     qc.x(clause_reg)
4     qc.x(res_reg)
5     qc.h(res_reg)
6     qc.barrier()
7     return qc
```

The rationale of negating all the clauses $c \in C$, previously introduced as De Morgan trick, follows using De Morgan's laws to transform the disjunctions in each clause into conjunctions, as described in 3.3.

Step 4 mechanically constructs the clauses from the variable qubits $|v_n\rangle$ on the clause qubits $|c_m\rangle$. For the example, we take the first clause $c_1 = (\neg A \vee \neg B)$. Applying De Morgan's law yields $c_1' = \neg(A \wedge B)$. The leading negation can be neglected, as described above. Here, the variables $A$ and $B$ are not negated and the clause can be built directly. For this, we use a Toffoli gate with qubits $|A\rangle$ and $|B\rangle$ as control bits and the clause's qubit $|\neg(A \wedge B)\rangle$ as target bit. Logically, this process can be understood by that the clause is flipped in the case the clause's condition is not satisfied and only then the clause's qubit is flipped into $|0\rangle$.

Next, we consider a situation in which a negated variable occurs, as in the last clause $c_3' = \neg(\neg A)$. To construct the clause, $A$ must be negated first. This is done by applying a X-gate to $|A\rangle$, then using it with a CNOT-gate as control bit and $|\neg(\neg A)\rangle$ as target bit. To restore the variable qubit's prior state, another X-gate is applied. In figure 3.6, the complete formula has been applied.

As two consecutive X-gates cancel each other out ($XX = I$), they can be omitted. As it is desired to create possibly compact quantum circuits[2], sparing of gates using simplifications is generally advisable. In figure 3.6, there would be two $X$-gates on qubit $|A\rangle$ between the second and third controlled-gate without this optimisation (one for restoring $|A\rangle$, another for negating it for the last clause).

To create the the formula's logic, i.e. step 4, the main program calls a subroutine displayed in listing 3.21.

Listing 3.21: Python code for steps 1-3. It generally consists of a loop iterating through all clauses on line 2. In lines 5 to 11, clauses with only one variable are processed, in lines 13 to 23, clauses with multiple variables are created. The De Morgan trick is applied by flipping qubit variables of variables originally not negated and not flipping variables which are already negated. The conjunction inside the clause $c'$ can now easily be represented using an MCT-gate.

```
1 def create_logic(qc, var_mapping, clause_mapping):
2     for clause, c_qubit in clause_mapping.items():
3         clause_vars = []
4     # if the clause only contains one variable
```

[2]As current quantum computers are noisy i.e. they are *NISQ* devices, small circuits are less prone to error than large circuits

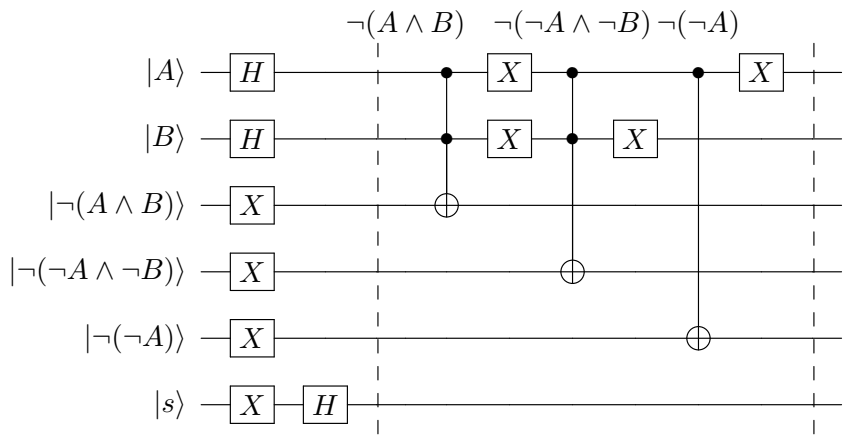



Figure 3.6: The quantum circuit's state after step 4 (including (a))

```
5         if len(clause.args) <= 1: # i.e. the clause contains
              only one variable
6             abs_var = Not(clause)
7             if not clause.is_Not: # if clause is not already
                  negated
8                 abs_var = clause
9                 qc.x(var_mapping[abs_var])
10            qc.cx(var_mapping[abs_var], c_qubit)
11            if not clause.is_Not:
12                qc.x(var_mapping[abs_var])
13    # for multi-variable clauses
14        else:
15            for var in clause.args:
16                abs_var = Not(var)
17                if not var.is_Not:
18                    abs_var = var
19                    qc.x(var_mapping[abs_var])
20                clause_vars.append(var_mapping[abs_var])
21            qc.mct(clause_vars, c_qubit)
22            for var in clause.args:
23                if not var.is_Not:
24                    qc.x(var_mapping[var])
25    qc.barrier()
26    return
```

In the case of only one variable, a simple CNOT-gate is applied with the variable qubit as control-, and the clause qubit as target qubit. Similarly, clauses with multiple variables are stored in `clause_vars` to be used in the MCT-gate as control bits.

*Pro memoria:* `var_mapping` and `clause_mapping` are associative arrays returning a qubit object given a symbolic variable or clause. In this program's version, the optimisation by removing consecutive X-gates is not implemented, therefore each negated variable is surrounded by two X-gates.

The multi-controlled Toffoli gate MCT in step 5 represents the conjunction of clauses of the SAT problem. It flips the $|s\rangle$ qubit only if all the clauses are satisfied, i.e. all

their qubits $|c_m\rangle$ are in state $|1\rangle$. In the example, the MCT-gate uses the clauses' qubits $|\neg(A \wedge B)\rangle$, $|\neg(\neg A \wedge \neg B)\rangle$ and $|\neg(\neg A)\rangle$ as control bits and $|s\rangle$ as target bit. After it is implemented, another barrier is applied.

This procedure is realised by the subroutine in listing 3.22. In line 2, the MCT-gate is created with the clause qubits as control-, and the result qubit as target qubit. Then, a barrier is applied.

Listing 3.22: Python code for steps 1-3

```
1 def evaluate(qc, clause_reg, res_reg):
2     qc.mct(clause_reg, res_reg)
3     qc.barrier()
4     return qc
```

In addition to the flip, another effect occurs: informally, the negative phase of $|s\rangle$ marks the satisfying superposition (or superpositions) of the SAT by phase kickback. This negative phase is then propagated to the variable qubits $|A\rangle$ and $|B\rangle$ in the example by the uncomputation in step 6, denoted by $U_C$. The uncomputation repeats the exact operations from step 4, by which the original state of all the qubits is restored, except that the negative phase is carried to the variable qubits. The state of the circuit after this operation is rendered in figure 3.7.

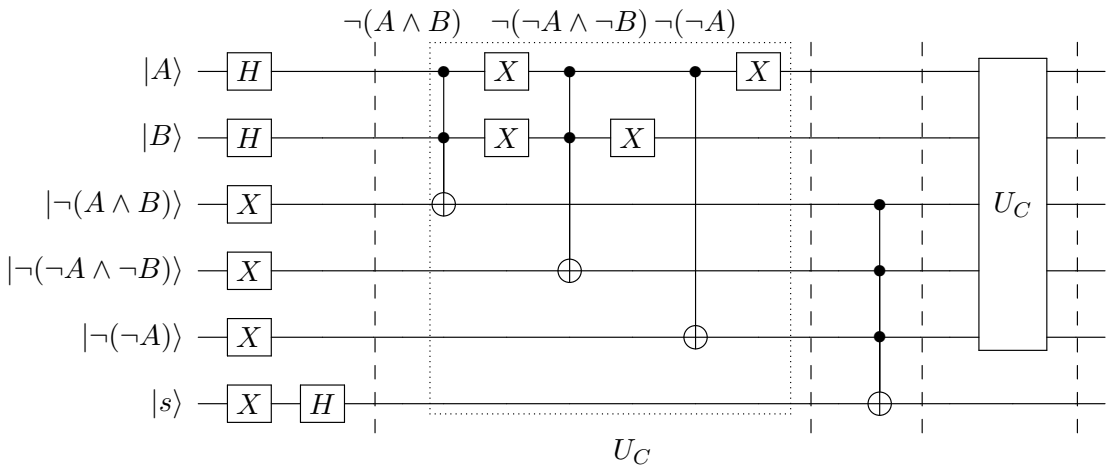



Figure 3.7: The finished quantum circuit

The steps 5 and 6 are again executed in the main program, of which second part is shown in listing 3.23. Now, the SAT-problem is entirely represented as quantum circuit and may be used for example as oracle.

Listing 3.23: Python code for steps 1-3. In line 1, the MCT-gate creates the conjunction of the clauses by simply using the clause register `qr_clauses` as control bits and the result register `qr_result` as target bit. This operation is concluded by placing a barrier on all qubits. The uncomputation is easily realised by calling the `create_logic` subroutine again, which is also finalised by creating a barrier on all qubits.

```
1 # create conjunction with all clauses
2 qc.mct(qr_clauses, qr_result)
3 qc.barrier()
4
5 # Uncompute logic
```

```
6 create_logic(qc, var_mapping, clause_mapping)
7 qc.barrier()
8 return qc
```

### 3.7.2 Complexity

We find that the conversion algorithm has a complexity of $\mathcal{O}(n)$ with $n$ clauses and $m$ variables, assuming that multi-controlled Toffoli gates are available. The algorithm then requires $n+m+1$ qubits, that is, a qubit for each variable, each clause and for the result.

## 3.8 Qiskit implementation details

The mathematical formulation used for SAT problems is not well suited for algorithmic processing. With the *DIMACS CNF* [20] format, a standardised format is provided. The CNF Boolean expression

$$(\neg A \vee \neg B) \wedge (A \vee B) \wedge A$$

can be represented by the *DIMACS CNF* format as in listing 3.24.

Listing 3.24: Example SAT in *DIMACS CNF*

```
1    c example DIMACS SAT
2    p cnf 2 3
3    -1 -2 0
4    1   2 0
5    1     0
```

In *DIMACS CNF*, lines starting with a `c` are regarded as comments. The first non-comment line starts with `p cnf n_variables n_clauses` with `n_variables` being the number of variables and `n_clauses` being the number of clauses. Here, we have two variables in three clauses. Every line following represents a clause, with each number representing a variable. Variable $A$ is translated into `1`, variable $B$ into `2`. A negated variable is preceded by a minus $-$ sign. Each line containing a clause is terminated by `0`. A clause cannot contain a variable and its negation.

The usage of Boolean formulae in quantum circuits is alternatively facilitated by Qiskit's *LogicalExpressionOracle*, that also enables optimisation. However, the Qiskit evaluates the logical expression classically and does not produce a quantum circuit. The program code provided here is also intended to be more general and to be more easily ported to other quantum assembly languages.

The quantum circuit created by our algorithm is slightly optimised by Qiskit, therefore the clauses appear in reverse order and one X-gate is applied earlier, as shown in figure 3.8. The logic is however not altered. At the moment, we do not know the reasons of the modifications. We suspect that Qiskit attempts to modify the circuit in order to reduce noise when executing it on quantum hardware, for instance by simultaneously applying gates on different qubits. The figure shows the circuit containing initialisation and a single oracle application.

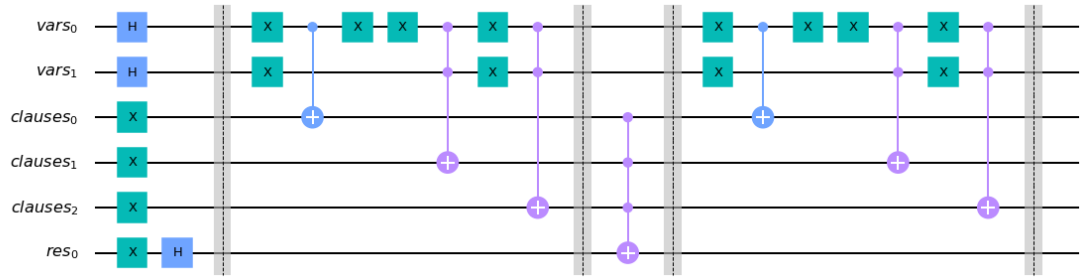



Figure 3.8: The oracle part of the quantum circuit in Qiskit

## 3.9 Solving SAT instances with the conversion algorithm

In this section, we solve three SAT problem instances using Grover's algorithm while we use the conversion algorithm to represent the SAT on a quantum circuit. For this purpose, we extend the program introduced above. The extension applies the oracle $U_f$, the diffusion operator $U_s$ for the required number of iterations. Finally, it measures the variable qubits.

The extension is incorporated in the program's main routine `build_circuit`, with the relevant part shown in listing 3.25. The number of iterations for Grover's algorithm is commonly indicated as $\sqrt{\frac{n}{m}}$ with $n$ possible entries and $m$ known solutions. We however use an optimised number of iterations by *Boyer et al.* [21], which is calculated in line 4. In line 5 the loop creates the oracle by creating the logic (l. 7), marking the solution (l. 9) and uncomputing the logic (l. 11). It then applies the diffusion operator on line 13. The variable qubits are finally measured and the measurement stored into the classical register in line 15. The quantum circuit `qc`, the variable- and clause, result and classical registers are returned for operation outside of the main routine.

Listing 3.25: Grover iteration in the main routine

```
1  def build_circuit(expression, n_solutions=1):
2      # ...
3      # Conduct the Grover iteration (Oracle and diffusion)
4      n_iterations = floor(pi * sqrt(2 ** len(variables) /
           n_solutions) / 4) # n of iterations dependend on
           number of solutions
5      for i in range(n_iterations):
6          # Create oracle
7          create_logic(qc, var_mapping, clause_mapping)
8          # Mark the solution
9          evaluate(qc, clause_reg, res_reg)
10         # Uncompute
11         create_logic(qc, var_mapping, clause_mapping)
12         # Apply diffusion operator
13         apply_diffuser(qc, var_reg)
14     # Finally, measure the variable qubits
15     qc.measure(var_reg, cr)
16     return qc, var_reg, clause_reg, res_reg, cr
```

### 3.9.1 Simple instance

To acquire an intuition which size of problems is tractable by a current quantum device, we first use a simplified expression of the above example clause with only two clauses and two variables

$$(A \vee B) \wedge \neg B$$

The routine for converting the *DIMACS CNF* format into a boolean expression uses $v_1, v_2, ..., v_n$ for the variables, so the expression is internally represented as

$$(v_1 \vee v_2) \wedge \neg v_2$$

with the solution

| $v_2$ | $v_1$ |
|---|---|
| 0 | 1 |

Note that the solutions are listed in little-endian order, such that they are more conveniently comparable with the result.

**Code reference.** The code used for the "simple" SAT instance with the Qiskit platform is available as `SAT/Qiskit/Simple_instance.ipynb`.

It is however not necessary to indicate SAT problems in *DIMACS CNF* form. The problem instance can also be written as a string. In listing 3.26. The quantum circuit can be considered in a sense similar to machine instructions, as the quantum circuit can be simulated or executed on real quantum hardware.

Listing 3.26: Top-level creation of the quantum circuit. The Boolean expression is indicated as a string in line 1 and the quantum circuit containing the Grover algorithm for the SAT problem is constructed in line 2. In the last line, the quantum circuit is drawn.

```
1 simple_boolean = '(v1 | v2) & ~v2'
2 qc, var_reg, clause_reg, res_reg, cr = build_circuit(
     simple_boolean)
3 qc.draw(output='mpl')
```

This expression can be implemented with 5 qubits, 2 for variables and clauses each, and one qubit for the result qubit. The quantum circuit in figure 3.9 looks conceptually similar as the circuit in figure 3.8, but additionally contains a diffusion operator (highlighted red) and the final measuring steps (highlighted yellow). The initialisation part is highlighted green and the oracle $U_f$ is highlighted blue. It represents a functional Grover algorithm implementation.

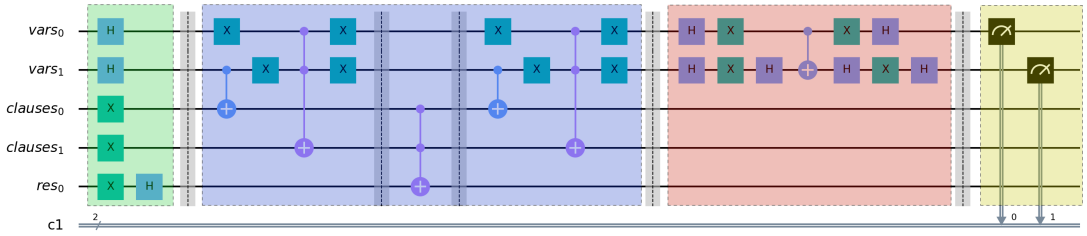



Figure 3.9: The complete quantum circuit in Qiskit. The initialisation part is highlighted green, the oracle $U_f$ is highlighted blue, the diffusion operator is in the red area, and the state is measured in the yellow area.

First, we simulate the circuit on classical hardware. The simulation provides helpful insight for debugging, as mixed states can be inspected without the destructive measuring process. Also, the simulation's output is ideal, that is, it is not disturbed by noise inherent to current day quantum hardware. The relevant code is shown in listing 3.27.

Listing 3.27: Simulation of the quantum circuit. In the first line, a simulator instance is created, then the circuit is simulated in the second line. The number of shots indicates how many times the circuit is executed. We explicitly use $n = 1024$ shots, as this is the default value. In this line, a job instance is returned and the result is directly accessed. In the third line, the states of the measured qubits, i.e. the classical register is plotted as a histogram.

```
1 backend = Aer.get_backend('qasm_simulator')
2 final_state = execute(qc, backend, shots=1024).result()
3 plot_histogram(final_state.get_counts())
```

For a more verbose output, we provide the `interpret_result` routine, which is displayed in listing 3.28. Given the counts and the original expression required for the variable assignment, the routine renders each state in the result in a more human-readable fashion.

Listing 3.28: Result interpretation routine

```
1  def interpret_result(counts, expression):
2      symbolic_expression = parse_expr(expression)
3      # Reverse order because result is little endian
4      variables = sorted(symbolic_expression.binary_symbols,
           key=str, reverse=True)
5      for result, count in counts.items():
6          print("Count: ", count)
7          expression = ""
8          for i in range(len(result)):
9              if result[i] == '0':
10                 expression += "~"
11             expression += str(variables[i])
12             expression += " & "
13         expression = expression[:-2] # Remove surplus &
14         print(expression)
15     return
```

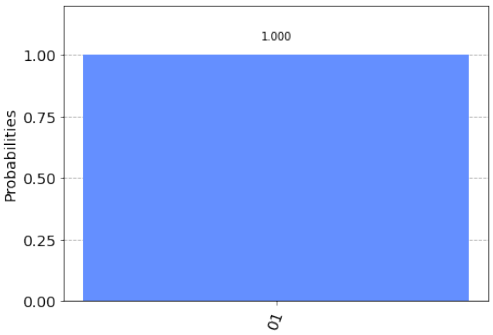
Now, we execute the circuit on a real IBM quantum device. For this, we execute the code shown in listing 3.29.

Listing 3.29: Execution of the quantum circuit on quantum hardware. In lines 2 and 3 the connection to IBM as provider is established, then the least busy device with a sufficient number of qubits is selected. Currently, the only publicly available IBM with more than five qubits is the *ibmq_melbourne* computer. On line 7, the circuit is executed with $n = 1024$ shots. On the following line, the job is monitored as it usually takes a while from job submission to the finished job due to a large number of jobs submitted to the devices. In lines 11 and 12, the result is extracted and displayed in the same manner as with the simulation.

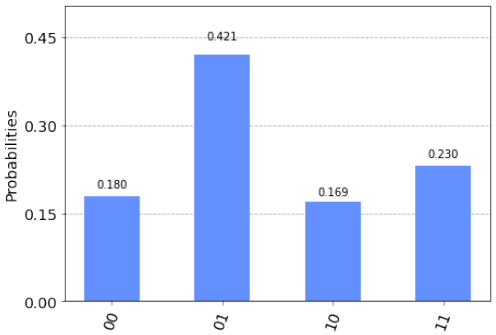
```
1 # Select quantum device
2 provider = IBMQ.load_account()
```

```
3  device = least_busy(provider.backends(filters=lambda x: x.
       configuration().n_qubits >=5 and not x.configuration().
       simulator and x.status().operational==True))
4  print("Chosen device: ", device)
5
6  # Run the job
7  job = execute(qc, backend=device, shots=1024)
8  job_monitor(job, interval = 2)
9
10 # Extract and display the result
11 results = job.result()
12 plot_histogram(results.get_counts(qc))
```

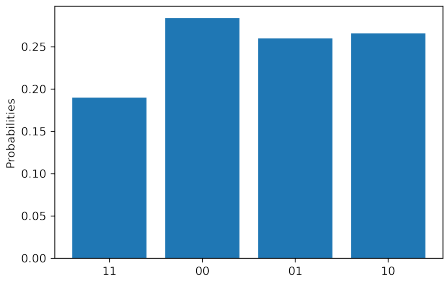
The results of both the simulation (left) and the execution of the circuit on a quantum computer (right) are shown in figure 3.10. Recall that state is shown as little-endian, therefore the least significant bit is the lowest bit in the register and the first variable. The resulting bit string is therefore in the form $v_n \ldots, v_2, v_1$. Also recall that the correct assignment is $v_2 v_1 = 01$. We interpret from the probability histogram that $v_2$ is assigned $0$ and $v_1$ is assigned 1. This is the result we expected. Also the quantum device produces a correct result. To better visualise the output, the `interpret_result` function can be invoked, yielding the variable assignment `~v2 & v1`.




(a) Result from simulation




(b) Result from IBM quantum device




(c) Result from Rigetti quantum device

Figure 3.10: The simulation (a) produced the same state for all shots, hence the probability for the state $01$ is 1. The result of quantum hardware (b) also produced the correct solution as the satisfying assignment is clearly more probable compared to the other states $00$, $10$ and $11$.
In (c), the correct assignment $v_1 v2 = 01$ (here in big-endian notation) has a similar amplitude as two other states, but should be considerably larger to be considered as correct result.

**Code reference.** The code used with the Amazon Braket platform is available as `SAT/Braket/Simple_instance.ipynb`.

For a manufacturer comparison, we rewrote the conversion algorithm to solve the above SAT instance on a *Rigetti* quantum device available with the Amazon *Braket* platform. Because the platform lacks MCT gates, the conversion algorithm is restricted. We expanded the conversion algorithm to emulate MCT gates with up to three controlling gates that suffices for this SAT instance. The resulting quantum circuits are however almost identical and the quantum circuit used on the Rigetti quantum device is therefore not shown.

The Rigetti device however fails at delivering the correct result. Note that results are in big-endian notation on the Braket platform, and thus the most significant bit represents the lowest qubit and the first variable, i.e. the resulting bit string is in the form $v_1 v_2 \dots v_n$. Recall that the satisfying assignment for this instance is therefore $v1v2 = 10$, but we receive $v1v2 = 01$ as most probable solution from the plot containing the results in figure 3.10. The amplitude is not small, but is not sufficiently significant or unique as $00$ and $01$ exhibit very similar amplitudes.

### 3.9.2 Medium SAT

Now, we solve the SAT used as example in the above section

$$(\neg v_1 \vee \neg v_2) \wedge (v_1 \vee v_2) \wedge v_1$$

that now contains one clause more than in the first instance. The SAT has the unique satisfying assignment

| $v_2$ | $v_1$ |
|---|---|
| 0 | 1 |

**Code reference.** The code used for the "medium" SAT instance is available as `SAT/Qiskit/Simple_instance.ipynb` for the Qiskit and as `SAT/Braket/Simple_instance.ipynb` for the Braket platform.

The resulting quantum circuit becomes slightly larger, the additional clause is represented by another controlled-not operation in the oracle and an additional qubit for the clause, as seen in figure 3.11. The quantum circuit is also first simulated and then executed on real quantum hardware.

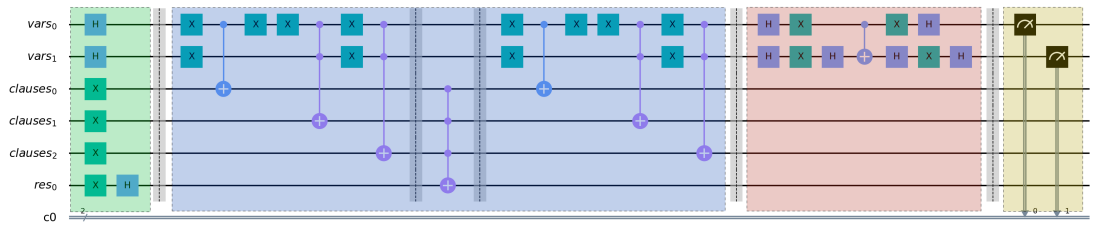
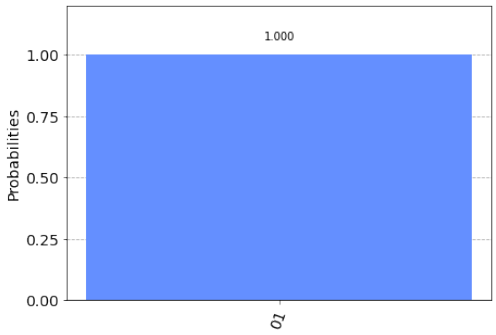


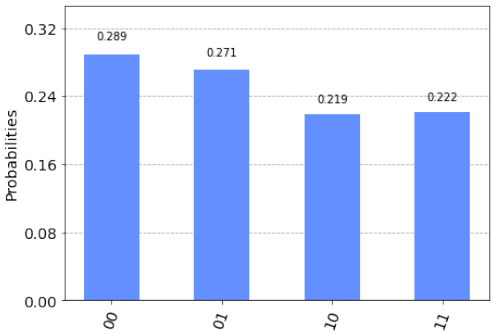

Figure 3.11: The complete quantum circuit in Qiskit. We use the same color scheme as above: the initialisation part is highlighted green, the oracle $U_f$ is highlighted blue, the diffusion operator is in the red area, and the state is measured in the yellow area.

The results from simulation and from the quantum hardware are shown in figure 3.12. The correct assignment is $v_2 v_1 = 01$, which is produced by the simulation, as expected.

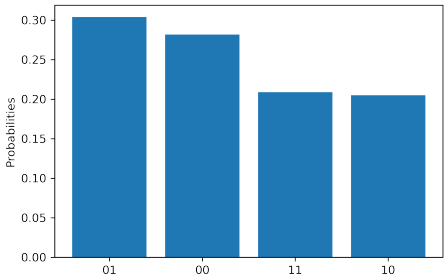
The result from the IBM quantum computer does not fit the expected result, as the assignment with the highest amplitude is $00$, which is clearly incorrect. The correct assignment $01$ has the second highest probability, which should however be considerably higher. The reason for the incorrect result is most likely that for quantum circuits in this depth i.e. with many gates in sequence, the current *NISQ* quantum devices are too noisy. It must be noted that *Qiskit's* tutorials only use the simulator for circuits with more than four qubits [22]. The quantum circuit is also executed on the Rigetti quantum device, which also fails as well to produce the correct assignment.




(a) Result from simulation




(b) Result from IBM quantum device




(c) Result from Rigetti quantum device

Figure 3.12: The IBM quantum device's result (b) does not exhibit the correct assignment $01$ (in little-endian), which, however, has the second-highest amplitude. The result from the Rigetti device (c) incorrectly features the correct assignment $10$ (as it is big-endian) only with the lowest amplitude.

### 3.9.3 Complex instance

We now employ the program to solve a more complex SAT instance[3]:

$$(v_1 \vee v_2 \vee v_3 \vee v_4) \wedge (\neg v_1 \vee \neg v_2) \wedge (\neg v_2 \vee \neg v_4) \wedge (\neg v_1 \vee \neg v_3) \wedge (\neg v_3 \vee \neg v_4) \tag{3.5}$$

with six satisfying assignments, where $1$ represents *True* and $0$ *False*

| $v_4$ | $v_3$ | $v_2$ | $v_1$ |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 |
| 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 |

[3]This expression is identical to the expression introduced in the CSP example 3.4

**Code reference.** The code used for the "complex" SAT instance is available as `SAT/Qiskit/Complex_instance.ipynb`.

This yields a larger circuit, as depicted in figure 3.13. Note that in spite the additional number of solutions, the Grover iteration only requires one repetition, although the search space consists of four (qu-)bits.

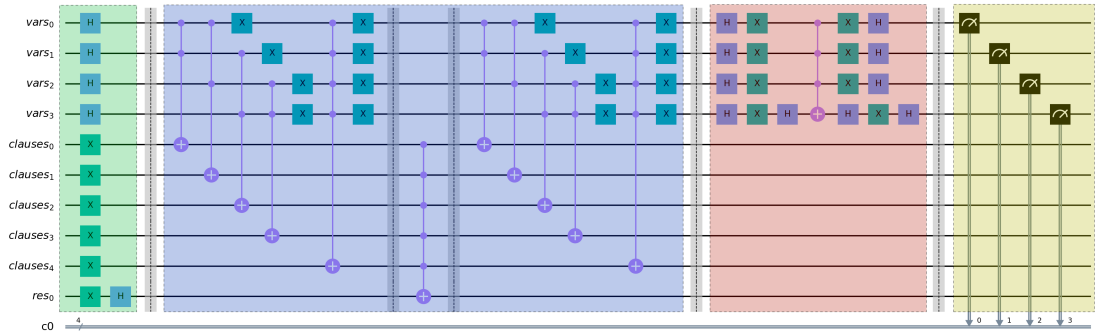
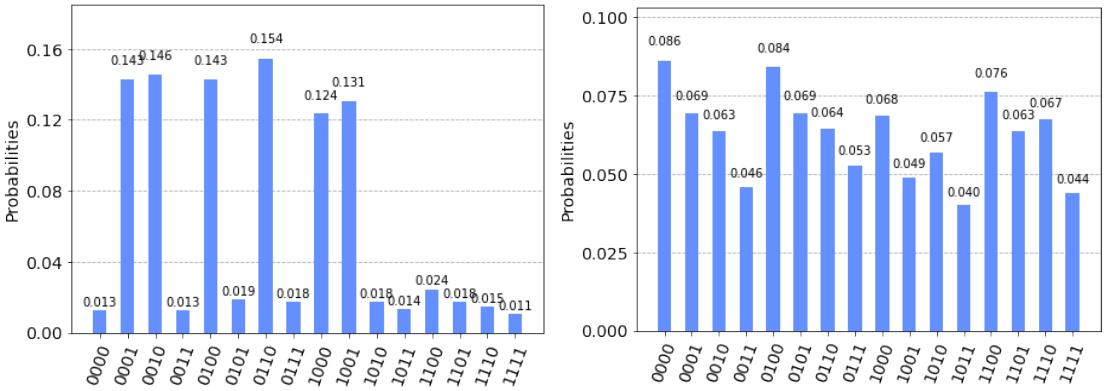



Figure 3.13: The complete quantum circuit for the "complex" instance in Qiskit. The highlighting semantics are the same as in figure 3.11

Again, we simulate the quantum circuit and run it on a quantum device analogous to the simple SAT instance. The result of the simulation is shown in figure 3.14. The result of the simulation is correct, considering that the states representing solutions are significantly more frequent, and the non-satisfying states are unlikely. However, the result from the IBM quantum device is false, as the probabilities are randomised by the noise. The solutions are not significantly more probable and non-satisfying states are too probable. The execution on the Rigetti quantum computer is omitted, as the device failed to produce correct assignments with the previous SAT instances.




(a) Result from simulation

(b) Result from IBM quantum device

Figure 3.14: The simulation's result (a) exhibits a significantly amplitude on the correct solutions $\{1000, 0100, 0010, 0110, 0001, 1001\}$. The results from the quantum device (b) are randomised by noise.

It becomes clear that quantum computers are probabilistic by nature when observing the results, and by the fact that a quantum circuit is executed with usually several thousand shots. The shots are required to obtain a statistically significant result, as the result of single shots may vary. As demonstrated, Grover's algorithm is very effective in an ideal quantum system without noise. Current *NISQ* devices are however prone to error due

to the noise and could not deliver satisfactory results for this application. As a remedy to the noise-introduced behaviour, error-correction techniques are employed [19]. We examine these techniques in the section 3.10.

### 3.9.4 Device comparison and error sources

Concluding from this single experiment above, the Rigetti quantum device appears inferior to the IBM quantum device. To compare the devices' performance reliably, more exhaustive experiments are required. In this paragraph we briefly compare technical properties of the IBM *ibmq-athens* used for the "simple" and "medium" SAT instances and the Rigetti *Aspen-8* device used in this subsection. We further provide a possible explanation of the IBM device's superior performance in this task.

Both devices operate on the superconducting architecture which is currently predominant with gate-based and annealing quantum computers. Although the Rigetti device features 32 physical qubits compared to the IBM's 5 qubits, their error rate differs significantly.

#### Gate error rates

The error rates of IBM's device is almost an order of magnitude smaller than the rate of Rigetti's device, as can be seen in table 3.1. As Grover's algorithm relies heavily on multi-gate operations and its circuits usually have great depth, that is, the amount of instructions that are executed sequentially, also small error rates accumulate. In the "medium" SAT instance, the least significant qubit $vars_0$ has 12 single-qubit gates and seven multi-qubit gates. The resulting error and accuracy for this single qubit on the IBM device could be calculated as

$$\begin{aligned} \text{Error} &= 12 \cdot 3.19e\text{-}04 + 7 \cdot 8.74e\text{-}03 = 6.500e\text{-}2 \\ \text{Accuracy} &= 100 \cdot (1 - 6.500e\text{-}2) = 93.5\% \end{aligned}$$

and for the Rigetti device

$$\begin{aligned} \text{Error} &= 12 \cdot 2.1e\text{-}03 + 7 \cdot 3.43e\text{-}02 = 2.653e\text{-}1 \\ \text{Accuracy} &= 100 \cdot (1 - 2.653e\text{-}1) = 73.47\%. \end{aligned}$$

The accumulated error of both of the devices is significant, in particular of the Rigetti device which loses more than a quarter of accuracy solely due to the gate errors.

This calculation is not generally applicable, as it ignores the circuit-specific error structure and other device-inherent effects as crosstalk. Crosstalk is the effect of one qubit influencing the state of other qubits and is poorly understood in quantum computers yet, as *Proctor et al.* show in [23]. As crosstalk generally increases with the amount of qubits, this could explain the difference in error rates between IBM's and Rigetti's device. *Proctor et al.* also give an introduction in the benchmarking of quantum devices as a whole, which has only been examined recently.

Because of a very different architecture and maturity of quantum computers it is difficult to compare the quantum error rates with classical semiconductors' rates. As illustrative figure it is however reported that $5e\text{-}5$ errors occur per hour per Mbit on DRAM, mainly due to radioactive decay and cosmic radiation [26]. Compared to the per-operation error of Quantum devices, which is in the range of $10^{-4}$ to $10^{-2}$, semiconductors have a considerably smaller error rate.

| **Metric** | **IBM *ibmq-athens*** [24] | **Rigetti *Aspen-8*** [25] |
|---|---|---|
| # of qubits | 5 | 32 |
| Median single-qubit gate error rate | 3.19e-04 | 2.1e-03 |
| Resulting accuracy [%] | 99.96 | 99.79 |
| Median multi-qubit gate error rate | 8.74e-03 | 3.43e-02 |
| Resulting accuracy [%] | 99.13 | 95.66 |

Table 3.1: Comparison of two of IBM's and Rigetti's quantum devices. The Rigetti quantum device has generally a higher error rate for its gates than the IBM quantum computer.

#### Architectural differences

Additionally, Rigetti's *Aspen-8* device is advertised as "hybrid-optimised", i.e. that it is designed for hybrid algorithms, which consist of a part executed on quantum computer and another part executed on a classical computer. These algorithms usually have quantum circuits with little depth and are executed repeatedly. Thus, they require less accuracy but profit from a larger number of qubits. In chapter 4, two examples of hybrid algorithms are examined.

Another source for the different performance could be the topological structure of the qubits, that is, their arrangement and connection. We however omit this aspect due to two reasons. First, the analysis of this aspect is complex and requires knowledge about how the logical quantum circuit is effectively mapped onto the quantum hardware. Secondly, providers of quantum devices employ optimisation techniques when mapping logical quantum circuits which are aware of the quantum hardware's properties. Those properties are usually not publicly available. We therefore assume that a close-to-ideal mapping is chosen for the respective hardware.

## 3.10 Error correction

As our experiments have shown the quantum computers in the NISQ-Era are very susceptible to errors. Although the construction of quantum gates improves and aims to reduce error constantly there will always be a certain inaccuracy due to the nature of quantum operations. For example $\pi$ is a transcendent number and we will never be able to represent it completely nor perform a accurate rotation by $\pi$ around any axis. This imperfection always leads to errors over time. Therefore it is mandatory to perform error correction on the qubits, to make our circuits more resistant and have a certain guarantee that our result is valid.

We conducted two kind of error correction algorithms for our experiment. The first and simplest is inspired by error correction on classical computer and corresponds to the repetition code. We encoded one qubit on three qubits.

**Code reference.** The code used for this section can be found as
`Error Correction/Error_correction.ipynb`.

### 3.10.1 Reducing errors due to measurement

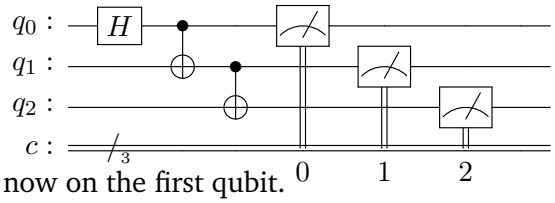
Our first attempt aimed to reduce errors induced by measurement. To prevent this, we entangled the qubits we would use for our circuit beforehand with two additional qubits as shown in listings 3.30 and 3.31. When measured in the end, the three entangled qubits should have the same output or the output can be reconstructed by the majority

of the qubit states after measuring. With this we would be able to detect a single qubit error induced by measurement.

Listing 3.30: Entanglement of qubits beforehand

```
# Creating a quantum-circuit and entangling the main qubit
    with two others
qc = QuantumCircuit(3, 3)
qc.h(0)

# The computation with qubit 0 would take place here

qc.cx(0, 1)
qc.cx(1, 2)

qc.measure([0, 1, 2], [0, 1, 2])
```

This leads to the simple circuit for entanglement of three qubits




The result depends now on the first qubit.

Listing 3.31: Measuring entangled state

```
backend = Aer.get_backend('qasm_simulator')

shots = 1024

job = execute(qc, backend, shots=shots)
job_monitor(job, interval = 2)
result = job.result()
counts = result.get_counts(qc)
plot_histogram(counts)
```

As expected and rendered in figure 3.15, the output is uniform.

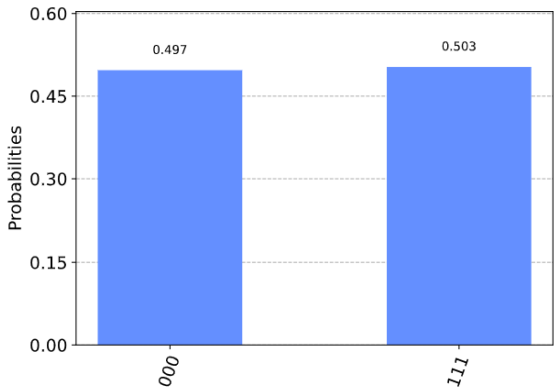



Figure 3.15: Measurement of the entangled state

We then try to apply this idea to our CSP solver with two variables as shown in listing 3.32.

Listing 3.32: Entangle variable-qubits with parity-qubits

```
# defining method to entangle the variable qubits with two
    parity qubits
def entanglement(qc, variables, parity_bits):
    qc.cx(variables[0], parity_bits[0])
    qc.cx(parity_bits[0], parity_bits[1])
    qc.cx(variables[1], parity_bits[2])
    qc.cx(parity_bits[2], parity_bits[3])
    qc.cx(variables[2], parity_bits[4])
    qc.cx(parity_bits[4], parity_bits[5])
    qc.cx(variables[3], parity_bits[6])
    qc.cx(parity_bits[6], parity_bits[7])
```

We then proceed to construct the CSP circuit for two variables, as shown in listing 3.33.

Listing 3.33: Creating the entangled CSP circuit for two variables

```
WA, NT = [0,1], [2,3]
variables = WA+NT
names = {'WA': WA, 'NT': NT}

red, green, blue, fail = '00', '01', '10', '11'
colors = {red: 'red', green: 'green', blue: 'blue', fail: '11
    ' }

wa = QuantumRegister(2, 'wa')
nt = QuantumRegister(2, 'nt')
parity = QuantumRegister(8, 'parity')

sol = QuantumRegister(1, 's')

want = QuantumRegister(1, 'want')

anc = QuantumRegister(3, 'anc')
constraintAncs = QuantumRegister(5, 'constraintAnc')

measure = ClassicalRegister(12, 'measure')

constraints = [want]

qc = QuantumCircuit(wa, nt, parity, want, anc, constraintAncs
    , sol, measure)
```

Finally, we run the circuit on the *qasm* simulator and find that the correct results have been produced, as shown in figure 3.16.

In our second attempt, we try to run the circuit multiple times on different qubits representing the same variable. We called the additional qubits parity qubits. We intend to deduce the final result by comparing the qubits, which should have the same result. The code we used is described in listing 3.34.

Listing 3.34: Creating the CSP-circuit for two variables

```
qubit = QuantumRegister(1, 'qubit')
```

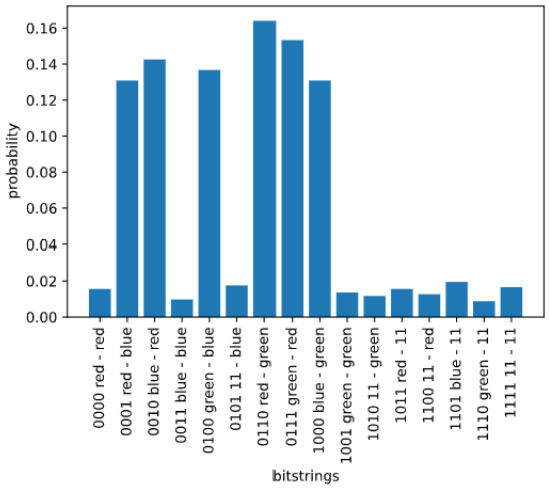
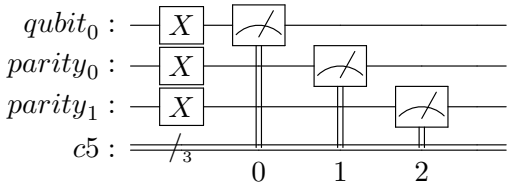



Figure 3.16: Simulator result from CSP Problem with two variables and entanglement

```
parity = QuantumRegister(2, 'parity')
measure = ClassicalRegister(3)

# create the circuit
qc = QuantumCircuit(qubit, parity, measure)

# define the operation
def some_operation(qc, bit):
    qc.x(bit)

# do the same opeartion on all qubits
some_operation(qc, 0)
some_operation(qc, 1)
some_operation(qc, 2)

qc.measure([0, 1, 2], measure)
```

Thus we create a circuit applying the same operation on different qubits:




Applying this method to our CSP problem leads to the code as in listing 3.35.

Listing 3.35: Creating the CSP-circuit for two variables with parity

```
WA, NT = [0,1], [2,3]
wa_parity0, nt_parity0, = [4,5], [6,7]
wa_parity1, nt_parity1,  = [8,9], [10,11]

variables = WA+NT + wa_parity0+nt_parity0 + wa_parity1+
   nt_parity1
```

```
6  names = {'WA': WA, 'NT': NT}
7
8  red, green, blue, fail = '00', '01', '10', '11'
9  colors = {red: 'red', green: 'green', blue: 'blue', fail: '11
       ' }
10
11 wa = QuantumRegister(2, 'wa')
12 nt = QuantumRegister(2, 'nt')
13
14 wa_parity0 = QuantumRegister(2, 'wa_parity0')
15 nt_parity0 = QuantumRegister(2, 'nt_parity0')
16
17 wa_parity1 = QuantumRegister(2, 'wa_parity1')
18 nt_parity1 = QuantumRegister(2, 'nt_parity1')
19
20 sol = QuantumRegister(1, 's')
21
22 want = QuantumRegister(1, 'want')
23 want_parity0 = QuantumRegister(1, 'want_parity0')
24 want_parity1 = QuantumRegister(1, 'want_parity1')
25
26 ancs = QuantumRegister(3, 'anc')
27 constraint_ancs = QuantumRegister(5, 'constraintAnc')
28
29 measure = ClassicalRegister(12, 'measure')
30
31 constraints = [want, want_parity0, want_parity1]
32
33 n = len(names) * 2
34 N = 2 ** n
35
36 qc = QuantumCircuit(wa, nt, wa_parity0, nt_parity0,
       wa_parity1, nt_parity1, want, want_parity0, want_parity1,
       anc, constraintAncs, sol, measure)
37
38 instantiate(qc, constraint_ancs, variables)
39
40 implementConstraint(qc, wa, nt, want, constraint_ancs)
41 implementConstraint(qc, wa_parity0, nt_parity0, want_parity0,
       constraint_ancs)
42 implementConstraint(qc, wa_parity1, nt_parity1, want_parity1,
       constraint_ancs)
43 checkConstraints(qc, constraints, sol)
44 implementConstraint(qc, wa, nt, want, constraint_ancs)
45 implementConstraint(qc, wa_parity0, nt_parity0, want_parity0,
       constraint_ancs)
46 implementConstraint(qc, wa_parity1, nt_parity1, want_parity1,
       constraint_ancs)
47
48 use_US_gates(qc, variables)
```

This works on the *qasm* simulator, as shown in figure 3.17, but fails on the real devices.

On an IBM quantum device it fails due to the circuit runtime surpassing the device's capacity. On the Rigetti quantum computer, the task fails due to an unknown reason we were not able to identify. We assume a similar problem as we experienced with the IBM quantum computer.

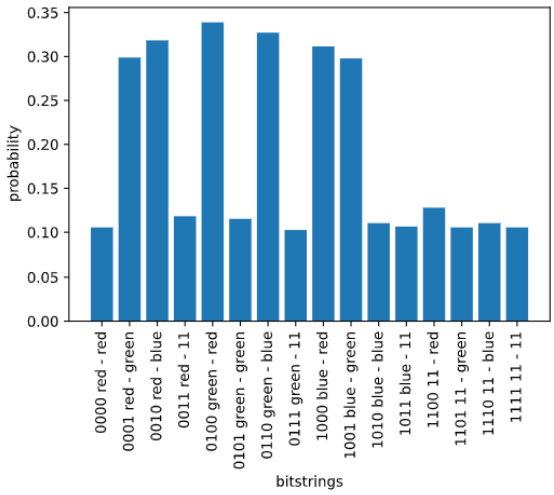



Figure 3.17: simulator result from CSP problem with two variables and parity

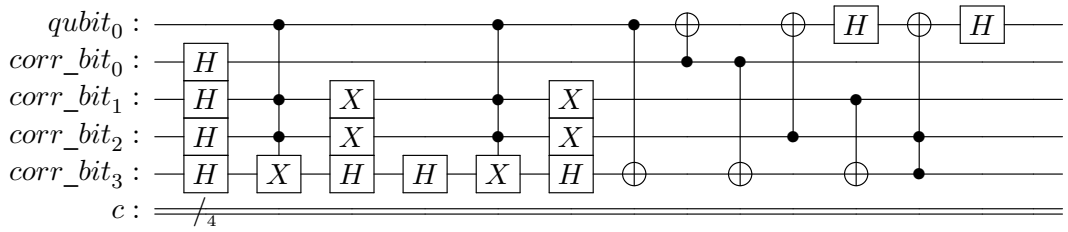
### 3.10.2 Syndrome error correction

Further research on the topic of quantum error correction codes leads us to encoding one qubit with four other qubits [27]. By measuring the four additional qubits, the error induced to the main qubit can be read from a table containing all the possible states of the control qubits. $S$ corresponds to a sign flip, $B$ to a bit flip and $BS$ to both. The complete lookup table is rendered in 3.2.

| Error | Syndrome $\|\text{corr\_bit}0,1,2,3\rangle$ | Resulting State $\|\text{qubit}_0\rangle$ |
|---|---|---|
| None | 0000 | $\alpha\|0\rangle + \beta\|1\rangle$ |
| BS3 | 1101 | -$\alpha\|1\rangle + \beta\|0\rangle$ |
| BS5 | 1111 | -$\alpha\|0\rangle + \beta\|1\rangle$ |
| B2 | 0001 | $\alpha\|0\rangle - \beta\|1\rangle$ |
| S3 | 1010 | |
| S5 | 1100 | |
| BS2 | 0101 | |
| B5 | 0011 | -$\alpha\|0\rangle - \beta\|1\rangle$ |
| S1 | 1000 | |
| S2 | 0100 | |
| S4 | 0010 | |
| B1 | 0110 | -$\alpha\|1\rangle - \beta\|0\rangle$ |
| B3 | 0111 | |
| B4 | 1011 | |
| BS1 | 1110 | |
| BS4 | 1001 | |

Table 3.2: Error with corresponding syndrom.

Before operations on the qubits are performed, they are encoded as follows:




With the corresponding code for the Qiskit implementation in listing 3.36.

Listing 3.36: encoding qubits for error correction

```
def five_bit_encoder(qc, qubit, corr_bits):
    qc.h([corr_bits[i] for i in range(len(corr_bits)-1)])

    qc.h(corr_bits[3])
    qc.mct([qubit, corr_bits[1], corr_bits[2]], corr_bits[3])
    qc.h(corr_bits[3])

    qc.x([corr_bits[i] for i in [1,2]])
    qc.h(corr_bits[3])
    qc.mct([qubit, corr_bits[1], corr_bits[2]], corr_bits[3])
    qc.h(corr_bits[3])
    qc.x([corr_bits[i] for i in [1,2]])

    qc.cx(qubit, corr_bits[3])

    qc.cx(corr_bits[0], qubit)
    qc.cx(corr_bits[0], corr_bits[3])

    qc.cx(corr_bits[2], qubit)

    qc.cx(corr_bits[1], corr_bits[3])

    qc.h(qubit)
    qc.mct([corr_bits[3], corr_bits[2]], qubit)
    qc.h(qubit)
```

After a operation is performed, the change can be extracted applying the same code backwards and measuring the correction qubits. This allows to find errors without changing the information encoded in the qubit. The code for the error finder is shown in listing 3.37.

Listing 3.37: decoding qubits for error extraction

```
def error_finder(qc, qubit, corr_bits):
    qc.h(qubit)
    qc.mct([corr_bits[3], corr_bits[2]], qubit)
    qc.h(qubit)

    qc.cx(corr_bits[1], corr_bits[3])

    qc.cx(corr_bits[2], qubit)
```

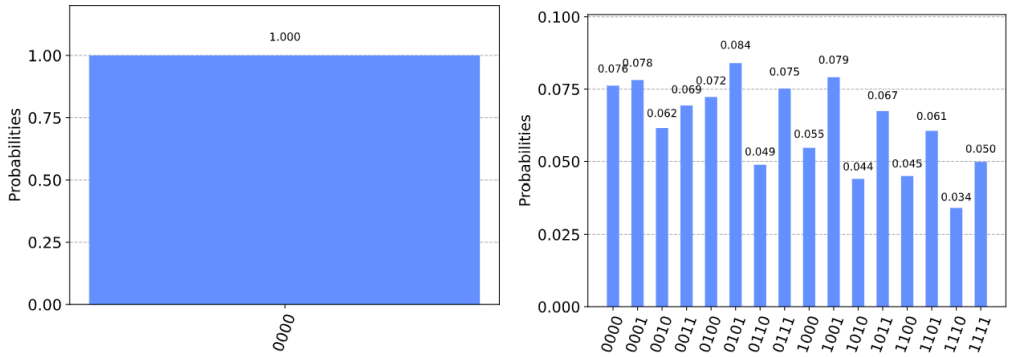
```
9
10      qc.cx(corr_bits[0], corr_bits[3])
11      qc.cx(corr_bits[0], qubit)
12
13      qc.cx(qubit, corr_bits[3])
14
15      qc.x([corr_bits[i] for i in [1,2]])
16      qc.h(corr_bits[3])
17      qc.mct([qubit, corr_bits[1], corr_bits[2]], corr_bits[3])
18      qc.h(corr_bits[3])
19      qc.x([corr_bits[i] for i in [1,2]])
20
21      qc.h(corr_bits[3])
22      qc.mct([qubit, corr_bits[1], corr_bits[2]], corr_bits[3])
23      qc.h(corr_bits[3])
24
25      qc.h([corr_bits[i] for i in range(len(corr_bits)-1)])
```

Those circuits would need to run before and after every single qubit operation to ensure no unintended change took place. In this example we ran the code without any gate in between, so there should be no error. The code in Listing 3.38 was used for this task.

Listing 3.38: Construction of test circuit for error correction

```
1 qubit = QuantumRegister(1, 'qbit')
2 corr_bits = QuantumRegister(4, 'corr_bit')
3 measure_bits = ClassicalRegister(4, 'c')
4 qc = QuantumCircuit(qubit, corr_bits, measure_bits)
5 five_bit_encoder(qc, qubit, corr_bits)
6 error_finder(qc, qubit, corr_bits)
7 qc.measure([i for i in range(1,5)], measure_bits)
```

For a single qubit the circuit ran successfully on both the simulator and the real device. But because of the use of five qubits and its inherent error the result on the real device is useless 3.18.




(a) Result from simulation (b) Result from quantum device

Figure 3.18: The result on the simulator shows there was no error, while the result on the real device represents a impossible combination of errors.

### 3.10.3 Results

The problem we encounter with error correction is the same problem we experience without correction. If the circuit on a real device is to deep i.e. contains to many gates or qubits, the error rate increases exponentially leading to errors in the main circuit and the correction code as well. We conclude from this that the best way to prevent errors in program code written for quantum computers is if error correction methods would already be implemented in the construction phase of the gates, that is, where physical qubits are translated to logical ones. In our experiment, error correction implemented in circuits leads to further errors because of the additional qubits used for error correction.

# Chapter 4

# Quantum Optimisation

In this chapter, we examine optimisation with quantum algorithms. Optimisation is, in addition to simulation of molecular systems and execution of machine learning models one of the often mentioned near-term applications of quantum computers [6, 8], although all of those applications make use of similar techniques. These techniques usually build upon the adiabatic principle, which is later introduced.

We focus on combinatorial optimisation problems, which are an extension of Boolean satisfaction problems, where each solution is assigned cost. It is therefore not sufficient to find a satisfying assignment, but rather a satisfying assignment with optimal cost. Conversely, Boolean satisfaction problems can be solved by combinatorial optimisation by assigning all satisfying assignments the same cost.

## 4.1 Motivation

In theoretical computer science, *Karp's* 21 NP-complete problems are combinatorial optimisation problems of relevance [28]. In practice, combinatorial optimisation can be found in the following areas.

### 4.1.1 Boosting in machine learning

In machine learning, different simple models are often combined to a larger model, where each simple model contributes to the larger model. This method is called *boosting*. Consider small models $m_1, m_2, \ldots, m_n \in M$, of which a subset is used for the large model $M_S \subset M$. The task of finding the subset of small models for which the large model produces the smallest error $\min_{M_S \subset M} Err(M_S)$ is a combinatorial optimisation problem.

### 4.1.2 Multiple query optimisation

This problem is later examined more thoroughly. It can be summarised as follows. A set of database queries $Q$ is given with a set of plans $P$ that generate the data asked by the queries. Assume that there are multiple plans for each query $q \in Q$ Similar to boosting, the task is to find the subset of plans in $P$ that generate the data for the set of queries $Q$ with minimal cost.

### 4.1.3 Portfolio optimisation

In a finance-related task, the selection of derivatives that minimise expected risk are sought. This selection, with cost evaluated using a expected risk function such as the *Black-Scholes* equation [29], represent a combinatorial optimisation problem.

## 4.2 Optimisation by Least Squares

We mention *least squares* as optimisation method which is different from the methods we introduce next. Here, the optimisation problem is represented as system of linear equations and subsequently solved by a quantum algorithm. With least squares for two dimensions, $n$ given data points $X$, $Y$ and a function $f(X)$, the coefficients for the functions are determined such that the squared error between $f(X)$ and $Y$ becomes minimal. The problem can be represented in the general case as a system of linear equations $AX = Y$, which is solved for A [30].

The system of linear equations can efficiently be solved in $\mathcal{O}(\log(n))$ with a quantum computer using the $HHL$-algorithm [31]. This algorithm is also often used for machine learning applications, for instance by *Weber and Mehmet* [32].

## 4.3 Variational Algorithms

In the remainder of this chapter, we present the *Quantum Approximate Optimisation Algorithm* (QAOA) and *Variational Quantum Eigensolver* (VQE). Variational algorithms can informally be thought of as methods that vary quantum mechanical parameters of a qubit in order to find a particular state. A problem is mapped to the qubits in a way that the sought state represents a solution to the problem. As the variation of the quantum mechanical parameters is usually done by a classical computer and the operation on the qubit on a quantum device, those algorithms are also known as *hybrid algorithms*. Those algorithms are particularly interesting because they require less coherency and fault-tolerance of quantum computer and are therefore more feasible in the near future[1] [33, 34].

This chapter is organised as follows. First, mathematical and physical prerequisites are introduced briefly. Then, the QAOA is introduced and used to solve the illustrative *Max-Cut* problem and explored for use on the *multiple query optimisation* problem (MQO), which is a typical problem in databases. Finally the VQE is introduced and applied for the same problems as the QAOA.

### 4.3.1 Mathematical and Physical Background

In this section we provide the mathematical background necessary to grasp the concept of the VQE.

#### Complex conjugate

The complex conjugate of a complex number $z$ is often denoted as $\bar{z}$

$$z = a + bi \Rightarrow \bar{z} = a - bi, a, b \in \mathbb{R} \tag{4.1}$$

[1] Although such problems could be solved quantum phase estimation, this requires full quantum coherency, which is currently not attainable.

### Transposed Matrix

The transposed matrix $\mathbf{M}^\top$ of a Matrix $\mathbf{M}$ can be seen as mirroring the entries of $\mathbf{M}$ on the main diagonal:

$$\mathbf{M} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \Rightarrow \mathbf{M}^\top = \begin{bmatrix} a_{00} & a_{10} & a_{20} \\ a_{01} & a_{11} & a_{21} \\ a_{02} & a_{12} & a_{22} \end{bmatrix} \tag{4.2}$$

### Eigenvector and Eigenvalue

The transformation of an eigenvector $|\psi_i\rangle$ of a matrix $A$ by the matrix $A$ is the same as the multiplication of $|\psi_i\rangle$ with a scalar $\lambda_i$:

$$A\,|\psi_i\rangle = \lambda_i\,|\psi_i\rangle \tag{4.3}$$

if this holds true for a $\lambda_i$ then $\lambda_i$ is called a eigenvalue of $A$ and $|\psi_i\rangle$ is its corresponding Eigenvector. Eigenvalues and the corresponding Eigenvectors of a Matrix $A$ can be seen as compressed data, providing a summary of a large matrix. They are useful in solving differential equations where the task is to find a rate of change or the relationships between two variables [35, 8].

### Hermitian matrix

A matrix H is called hermitian when it is equal to its conjugate transposed:

$$H = H^\dagger \tag{4.4}$$

If $H$ is Hermitian it may be expressed as the sum of the outer product of its normalised Eigenvectors multiplied by the corresponding Eigenvalues [8].

$$H = \sum_{i=1}^{N} \lambda_i\,|\psi_i\rangle\,\langle\psi_i| \tag{4.5}$$

### Hamiltonian and Time Evolution

The Hamiltonian of a physical system is a matrix describing the possible energies of the system. The sum of the potential and kinetic energy of all the particles associated with the system building the Hamiltonian of that system. The set of eigenvalues of a Hamiltonian corresponds to the set of possible outcomes of the system's total energy. By knowing the Hamiltonian of a system we can calculate its behaviour over time. A Hamiltonian can also be developed into a desired state by the concept of time evolution, which is explained below. As stated in [36] it is possible to formulate many optimisation problems or more precise cost functions of optimisation problems into Hamiltonians. Therefore we can use VQE to solve general optimisation problems. It is important to note that Hamiltonians are always hermitian matrices.

### Expectation values and Observables

The expectation value of the observable $H$ is defined as

$$E(\psi) = \langle\psi|\,H\,|\psi\rangle = \langle H\rangle_\psi$$

with $|\psi\rangle$ being a normalised state vector. If we substitute $H$ with its representation as a sum of eigenvectors

$$\begin{aligned}\langle H\rangle_\psi = \langle\psi| H |\psi\rangle &= \langle\psi| (\sum_{i=1}^{N} \lambda_i |\psi_i\rangle \langle\psi_i|) |\psi\rangle \\ &= \sum_{i=1}^{N} \lambda_i \langle\psi|\psi_i\rangle \langle\psi_i|\psi\rangle \\ &= \sum_{i=1}^{N} \lambda_i |\langle\psi_i|\psi\rangle|^2\end{aligned}$$

and because $|\langle\psi_i|\psi\rangle|^2$ has to be $\geq 0$, follows

$$\lambda_{min} \leq \langle H\rangle_\psi = \langle\psi| H |\psi\rangle = \sum_{i=1}^{N} \lambda_i |\langle\psi_i|\psi\rangle|^2$$

This equation is known as the variational method [8]. It implies that the minimum eigenvalue associated with $H$ will always be a lower bound to the expectation value.

$$\langle H\rangle_\psi \geq \lambda_{min}$$

hence

$$\langle H\rangle_{\psi_{min}} = \lambda_{min}$$

**Time Evolution**

Time evolution is a consequence of Schrödinger's equation, which states that a quantum mechanical state $|\Psi(0)\rangle$ evolves, by applying an operator in form of a time-dependent Hamiltonian $H$ during a time interval $t$ into a new state $|\Psi(t)\rangle$:

$$|\Psi(t)\rangle = e^{-iHt} |\Psi(0)\rangle$$

This allows evolving a ground state by a known Hamiltonian operator $H$ to obtain a new state containing a solution to $H$, if $H$ encodes a problem. This evolution is closely related to the *adiabatic theorem* and applied in *adiabatic quantum computing*, which are both described in the subsection below.

**Adiabatic Theorem**

Time evolution explains the general concept of evolving a quantum state into another. The adiabatic theorem states more precisely how this evolution occurs, namely that by slowly increasing $t$, the base state, itself described by a Hamiltonian $H_B$, evolves into another state $H_C$ [37]

$$H(\alpha) = (1-\alpha)H_B + \alpha H_C$$

where $\alpha = t/T$, $t$ is time and $T$ is the total time required for the evolution. Hence

$$\lim_{t\to T} H(t) = H_C.$$

Adiabatic quantum computation utilises this principle, realised for instance by the quantum annealing (QA) technique, or in another form by the QAOA.

### Suzuki-Trotter expansion

Due to limitations in current NISQ-era quantum computers, the simulation of quantum systems, such as time evolution are infeasible on current hardware. The Suzuki-Trotter expansion, also known as Trotterisation [38] provides a suitable approximation to simulate Hamiltonians. Consider a time evolution $|\Psi(t)\rangle = e^{-iHt}|\Psi(0)\rangle$ with the Hamiltonian operator $H$ over time $t$. The approximation provided by the Suzuki-Trotter expansion is achieved by dividing up $t$ in smaller time steps $\Delta t$

$$e^{-iHt} \approx e^{-ih_1\Delta t} \dots e^{-ih_n\Delta t} = (\prod_n e^{-ih_i\Delta t})^{\frac{t}{\Delta t}}$$

Note that the time steps $\Delta t$ do not have to be equidistant. Because the Hamiltonian $H$ in the time evolution must be applied by an adiabatic process, $H$ is represented as $H(\alpha) = (1-\alpha)H_B + \alpha H_C$ (with $\alpha = \frac{t}{T}$), which adds additional terms into the approximation product

$$e^{-i((1-\alpha)H_B+\alpha H_C)t} \approx e^{-i(1-\alpha)h_{B_1}\Delta t}e^{-i\alpha h_{C_1}\Delta t} \dots e^{-i(1-\alpha)h_{B_n}\Delta t}e^{-i\alpha h_{C_n}\Delta t}$$

with $\alpha = \frac{t}{T}$. In this context, a common choice for the base state $H_B$ is $\sum_{i=1}^{n}\sigma_X^i$, where $\sigma_X^i$ is a Pauli-X operator. As $H_C$ is described here in terms of Pauli-Z operators $\sigma_Z^i$, $H_C$ and $H_B$ do not commute. Therefore, the operators $H_C$ and $H_B$ must be applied sequentially with only small error.

To simplify the product, angles $\gamma$ and $\beta$ are introduced to substitute the time steps with $\beta_t = (1-\alpha)\Delta t$ and $\gamma_t = \alpha/\Delta t$. The final approximation term is

$$e^{-i\beta}h_{B_1\Delta t}e^{-i\gamma h_{C_1}\Delta t} \dots e^{-i\beta h_{B_n}\Delta t}e^{-i\gamma h_{C_n}\Delta t} = (\prod_n e^{-i\beta h_{B_i}\Delta t}e^{-i\gamma h_{C_i}\Delta t})^{\frac{t}{\Delta t}}$$

Although this procedure seems very theoretical, it is an essential step to simulate quantum systems on actual quantum hardware.

## 4.4 Quantum Approximate Optimisation Algorithm

The quantum approximate optimisation algorithm is a recent technique to conduct optimisation with gate-based quantum computers [39]. The algorithm is designed for solving combinatorial optimisation problems (COP), which can be represented as finding a bit string $z = z_1 z_2 \dots z_n$ for which the cost function $C$ becomes maximal

$$C(z) = \sum_{\alpha=1}^{m} C_\alpha(z) \tag{4.6}$$

where $C_\alpha(z)$ evaluates to $0$ if $z$ does not satisfy and $1$ if it satisfies the clause $\alpha$. The QAOA differs from the VQE as it (1) with combinatorial problems is intended for a different problem class, (2) seeks a state close to maximum, not the minimum of a problem and (3) represents the problem only in terms of Z-gates. Thus, the QAOA can be considered as a special case of the more general VQE. As the algorithm is approximate, it seeks to find a solution close to $C_{max}$.

The remainder of this section is organised as follows. First, we introduce the algorithm on a high level and introduce components used. Then, we show how to formulate a COP into a quantum circuit theoretically and by an example and execute it. We apply this algorithm to MQO (multiple query optimisation) problems and compare the QAOA with a classical approach. Finally, we compare the QAOA with similar techniques.

### 4.4.1 Algorithm overview

The QAOA is schematically shown in figure 4.1. It initialises the quantum circuit with the qubits $|x_1\rangle, |x_2\rangle \dots |x_n\rangle$ representing the bit string $z, |z| = n$ for the cost function $C(z)$ and creates an uniform superposition of the $n$ qubits with H gates. The superposition now has a non-zero overlap with the solution bit string because the qubits are in a mixed state

$$\frac{1}{\sqrt{2^n}}(|0\rangle + |1\rangle + \dots + |2^n - 1\rangle$$

Then, the state is prepared by applying the operators $H_C$ and $H_B$ with variational angles $\gamma$ for $H_C$ and $\beta$ for $H_B$. These two steps are executed $p$ times, with a better approximation to the solution to the optimisation problem for each iteration and adjustment of the angles $\gamma$ and $\beta$ after each state preparation.

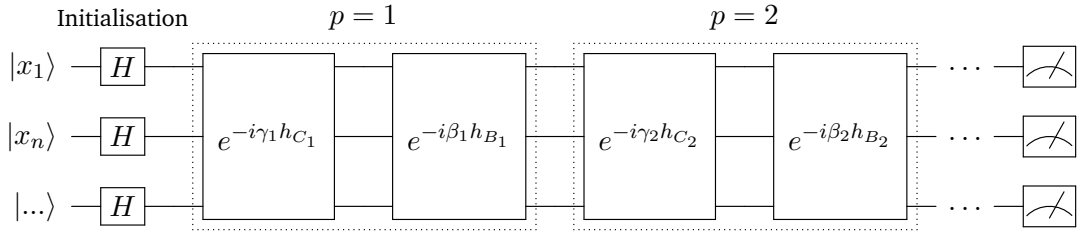



Figure 4.1: QAOA in in a general form. Two stages $p \in 1, 2$ are shown. In each stage $p$, the problem Hamiltonian $H_C$ with an angle $\gamma$ and the base state $H_B$ with an angle $\beta$ is partially evolved on the qubits.

Often, a depth of $p = 1$ is chosen out of two reasons: firstly, error and decoherence accumulate strongly with deep circuits, as shown in subsection 3.9.4. Secondly, the

algorithm already performs non-trivially with $p = 1$ [39]. $p$ is either chosen prior to the algorithm invocation, or for $p \to \infty$, in which case the QAOA simulates quantum annealing, that has no fixed steps. *Zhou et al.* examine the QAOA with $p \geq 1$ and suggest heuristic methods to adjust the angles [40]. The repeated application of $H_C$ and $H_B$ is known as trotterised annealing as it combines the methods of quantum annealing and the Suzuki-Trotter expansion. After $p$ repetitions, the qubits are measured and the resulting amplitudes represent the bit string $z$, for which $C(z)$ is close to the maximum $C_{max}$.

The following introduction is accompanied by the *MaxCut* problem. It is defined as a task in which a graph with weighted edges has to be cut into two subgraphs, such that the sum of the edges which are cut becomes maximal, as shown in figure 4.2. It is often used as introductory problem for the QAOA as it can be formulated well into a mathematical form used for QAOA.

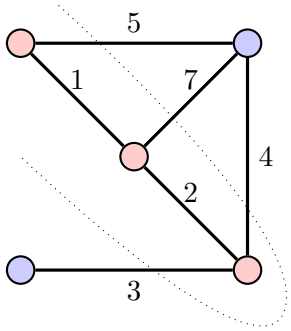



Figure 4.2: A cut through the edges with weights 5, 4 and 3 gives the maximum sum. The edges of one subgraph are highlighted red, the other edges blue.

**Code reference.** The code for the following example can be found as `Optimisation/QAOA_MaxCut.ipynb`

**Example.** For the purpose of the example, we choose a simpler graph, shown in figure 4.3. In our implementation, it only requires three qubits. Solutions are represented as bit strings in the form $x_0 x_1 \dots x_n$ with nodes $x_i$ that are in the domain $\{ 0, 1 \}$, depending in which subgraph they are contained. In this example, the maximal cut is obtained by cutting off $x_0$ from the graph, yielding the solutions

| $x_2$ | $x_1$ | $x_0$ |
|---|---|---|
| 1 | 1 | 0 |
| 0 | 0 | 1 |

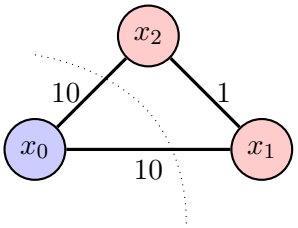



Figure 4.3: The maximum cut in this graph is through the edges with weight 10, between the node pairs 0, 1 and 0, 2.

For solving this task in Qiskit, the graph is first represented as data structure, as in code listing 4.1. The remainder of the code is placed at the end of this section, as the following subsections examine the problem formulation, for which no code is required.

Listing 4.1: The graph for the MaxCut problem is implemented using lists of vertices and edges (lines 2 and 3). On lines 6-8, the graph object is invoked and lines 8-18 are for graphical output of the graph.

```
1  # Create graph
2  V = [0, 1, 2]
3  E = [(0, 1, 10), (0, 2, 10), (1, 2, 1)] # In the form start,
       end, weight
4  num_vertices = len(V)
5
6  # Create graph data structure
7  G = nx.Graph()
8  G.add_nodes_from(V)
9  G.add_weighted_edges_from(E)
10
11 # Generate plot of the Graph
12 colors       = ['r' for node in G.nodes()]
13 default_axes = plt.axes(frameon=True)
14 pos          = nx.circular_layout(G)
15
16 nx.draw_networkx(G, node_color=colors, node_size=600, alpha
       =1, ax=default_axes, pos=pos)
```

### 4.4.2 Devising the problem Hamiltonian $H_C$

The cost function $C(z)$ can easily be reformulated into a quadratic unconstrained binary optimisation (QUBO) problem, which is very similar to the actual implementation into a quantum circuit. QUBO problems are in a form developed from the original combinatorial problem in equation 4.6:

$$\sum_{i \leq j} w_{ij} x_i x_j$$

with binary variables $x_i \in \{0, 1\}$ and weights $w_{ij}$. The sum range $i \leq j$ is chosen that combinations are only chosen once, as $x_i x_j = x_j x_i$.

This form is very similar to Ising problems

$$H(\sigma) = -\sum_{\langle i,j \rangle} J_{ij} \sigma_i \sigma_j$$

in which $\sigma_i$ represents spin properties of particles and $J_{ij}$ the strength of interaction of each other. $H$ is the Ising Hamiltonian, a special type of Hamiltonian. The spins $\sigma_i$ desire to have an identical spin to their neighbours, therefore, the above equation describes the minimum energy state. To express a system with different spin among particles, the sign of the Ising Hamiltonian is changed [41]. This property is later exploited. The similarity between QUBO and Ising problems also holds for the result, as extrema of $H(\sigma)$ are also extrema of the QUBO. This similarity now allows the mapping of the QUBO to a quantum system by replacing the QUBO's variables $x_i$ by Z-gates, in this context often written as $\sigma_Z^i$, on state $|i\rangle$. Hence, each variable requires one qubit. Finally, we can

obtain the Ising Hamiltonian $H_C$ from the original cost function

$$H_C = \sum_{i \leq j} w_{ij} \sigma_Z^i \sigma_Z^j$$

or in a more general form, where single variables $x_i$ occur

$$H_C = \sum_{i \leq j} w_{ij} \sigma_Z^i \sigma_Z^j + \sum_{i=0} v_i \sigma_Z^i$$

$H_C$ is often called problem Hamiltonian because it encodes a QUBO problem and is a main component of the QAOA.

**Example.** For our MaxCut example, we formulate the QUBO as follows. We encode each node as variable $x_i$ and edges as pair of nodes $x_i x_j$. The weight of a edge can directly be used as weight $w_{ij}$ for the QUBO. As any solution, i.e. any valid cut produces two subgraphs we assign the node variables the binary value 0 or 1, depending in which subgraph it is contained. To maximise the sum, we seek a state where heavy edges are in different subgraphs, that is, $x_i \neq x_j$. This is exactly the property that is described in Ising Hamiltonians. The QUBO sum contains all combinations of nodes, that is, a fully connected graph. If a graph is not fully connected, we can formally connect the remaining edges and assign the weight 0, thus removing these terms from the QUBO sum. In the given example, the QUBO is[2]:

$$\sum_{i \leq j} w_{ij}(x_i(1 - x_j) + x_j(1 - x_i))$$

where the weighted expression evaluates to 1 when $x_i$ and $x_j$ are different and 0 when they are both 0 or 1. This expression can be simplified in the Ising Hamiltonian formulation because in quadratic terms $\sigma_Z^i \sigma_Z^j$ the neighbouring spins property introduced with Ising Hamiltonians is exploited. For this, consider an $\sigma_Z^i \sigma_Z^j$ operation (later introduced as $R_{ZZ}$-gate) acting on states $|i\rangle$ and $|j\rangle$. It rotates $|i\rangle$ and $|j\rangle$ iff the states are different $|i\rangle \neq |j\rangle$. We replace variables $x_i$ by $\sigma_Z^i$ and obtain

$$H_C = \frac{1}{2}(10\sigma_Z^0\sigma_Z^1 + 10\sigma_Z^0\sigma_Z^2 + 1\sigma_Z^1\sigma_Z^2)$$

where we multiply the weights by 0.5 such that the weights are in the interval $[0, 2\pi]$, as this is the value range of the gate which is later used to implement this Hamiltonian and it has empirically proved to produce better results.

### 4.4.3 Representation of Hamiltonian operators with quantum gates

Thus far, the application of Hamiltonian operators $H_C$ and $H_B$ has been written as $e^{-i\gamma H_C}$ and $e^{-i\beta H_B}$, and the problem Hamiltonian $H_C$ has been formulated as

$$H_C = \sum_{i \leq j} w_{ij} \sigma_Z^i \sigma_Z^j + \sum_{i=0} v_i \sigma_Z^i.$$

Now, we show how to implement these Hamiltonians with quantum gates. In the QUBO, respectively the Ising Hamiltonian formulation, only linear terms $x_i$ or $\sigma_Z^i$ respectively

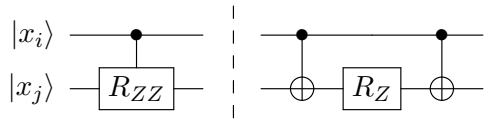
[2]In the literature, variables $x_i$ are often chosen with domain $\{-1, 1\}$ to write a simpler QUBO form. To formulate an equivalent problem Hamiltonian, this domain must however be adjusted to $\{0, 1\}$

$x_i x_j$ or $\sigma_Z^i \sigma_Z^j$ occur. Linear terms are simply represented as $R_Z$ gates with parameter $\lambda$ represented as matrix (left) and gate (right) [38]:

$$R_Z(\lambda) = \begin{pmatrix} e^{-i\frac{\lambda}{2}} & 0 \\ 0 & e^{i\frac{\lambda}{2}} \end{pmatrix} \qquad |x_i\rangle \; — \boxed{R_Z} —$$

Quadratic terms are represented as $R_Z$ gates surrounded by CNOT gates with the $R_Z$ gate on the target qubit. Alternatively, we can use $R_{ZZ}$ gates, which are the shorthand notation of the former. Due to the gates property to act when its control- and target qubits are different, it can be considered as analogue to a XOR gate. The control- and target qubits can be interchanged as the phase kickback acts in both directions. The $R_{ZZ}$ gate takes $\theta$ as argument. The matrix representation is given on the left and the gate representation on the right side [38]:

$$R_{ZZ}(\theta) = \begin{pmatrix} e^{-i\theta} & 0 & 0 & 0 \\ 0 & e^{i\theta} & 0 & 0 \\ 0 & 0 & e^{i\theta} & 0 \\ 0 & 0 & 0 & e^{-i\theta} \end{pmatrix}$$




$R_Z$ and $R_{ZZ}$ gates are a more general version of Z gates, where the rotation angle around the Z-axis is indicated by the parameter $\lambda$, respectively $\theta$. For example, the Z gate could be simulated by a $R_Z(2/\pi)$ gate.

As introduced above, the Hamiltonian $H_B$, which we call the driver Hamiltonian, is often chosen as $H_B = \sum_{i=1}^{n} \sigma_X^i$ with the Pauli-$X$ operator $\sigma_X^i$. As $H_B$ is applied with the angle $\beta$, the parametrised version of the $X$ gate, that is, the $R_X$ gate with parameter $\theta$ is chosen. It is defined as

$$RX(\theta) = \begin{pmatrix} \cos\theta & -i\sin\theta \\ -i\sin\theta & \cos\theta \end{pmatrix} \qquad |x_i\rangle \; — \boxed{R_X} —$$

as only linear $\sigma_X^i$ terms are contained in Hamiltonian $H_B$, no further gates are needed.

**Example.** With this, the necessary prerequisites for creating the quantum circuit are given. Recall that in our example, the problem Hamiltonian $H_C$ is given as

$$H_C = \frac{1}{2}(10\sigma_Z^0\sigma_Z^1 + 10\sigma_Z^0\sigma_Z^2 + 1\sigma_Z^1\sigma_Z^2) \tag{4.7}$$

and the driver Hamiltonian $H_B$ that acts on all qubits is

$$H_B = \sum_{i=1}^{n} \sigma_X^i = \sigma_X^0 + \sigma_X^1 + \sigma_X^2 + \sigma_X^3 \tag{4.8}$$

Additionally we choose the angles $\gamma = 1$ and $\beta = 0.3$ and conduct one repetition, i.e. set $p = 1$. We chose them because they were empirically effective to obtain correct results. In general, the choice of these angles is a non-trivial problem and is chosen for multiple repetitions $p > 1$ after each repetition by a classical algorithm [39, 40].

Now, we present a generalised algorithm to implement the quantum circuit for finding approximate solutions for the MaxCut problem on arbitrary graphs with QAOA. The code used for the initialisation is shown in listing 4.2, from which we refer to the colour highlighting scheme also used in figure 4.4.

Listing 4.2: In lines 2 and 3, we define the angles $\beta$ and $\gamma$ as lists, as they usually vary with multiple repetitions $p$. Then, the quantum circuit is created (lines 6-9). Finally, the qubits are brought into an uniform superposition in line 10, and a barrier is applied (green area).

```
1  # List of angles beta and gamma
2  betas = [0.3] # only one, as p = 1
3  gammas = [1]
4
5  # Set up Quantum circuit
6  vertex_register = QuantumRegister(num_vertices, name="
       vertices")
7  classical_register = ClassicalRegister(num_vertices, name="
       classical")
8
9  qc = QuantumCircuit(vertex_register, classical_register)
10 qc.h(vertex_register) # Create entangled state
11 qc.barrier()
```

Now, the problem and driver Hamiltonians $H_C$ and $H_B$ are applied by the code in listing 4.3.

Listing 4.3: In line 2, the number of repetitions is set, for which the outer loop on line 5 is executed. In this loop, the problem Hamiltonian $H_C$ is applied in the loop on line 8 by applying a $R_{ZZ}$-gate for each edge, which consists of a pair of vertices (blue area). This operation corresponds to the binary terms of $H_C$ as in equation 4.7. The gates' first argument equals to $w_{ij}\gamma$, the second and third are the control- and target qubits. On line 13, $H_B$ from as in equation 4.8 is implemented with single qubit $R_X$ gates that only take $\beta$ as argument (red area). The qubits are then finally measured with the instruction in line 16 (yellow area).

```
1  # Set the number of repetitions p
2  p_max = 1
3
4  # Create the Hamiltonians (with p = 1)
5  for p in range(1, p_max+1):
6      # If not predefined, calculate new values for beta and
           gamma
7      # Apply H_c
8      for e in E:
9          qc.rzz(0.5*e[2]*gammas[p-1], e[0], e[1]) # Apply
               weight
10     qc.barrier()
11
12     # Apply H_b
13     qc.rx(betas[p-1], vertex_register)
14     qc.barrier()
15
16 qc.measure(vertex_register, classical_register)
```

The complete quantum circuit for the MaxCut problem is shown in figure 4.4.

Now, we execute the algorithm first as simulation and with real quantum hardware, with the results in figure 4.5. Recall that the solutions to the example are $110$ and $001$, which is achieved by both simulation and execution on quantum hardware.

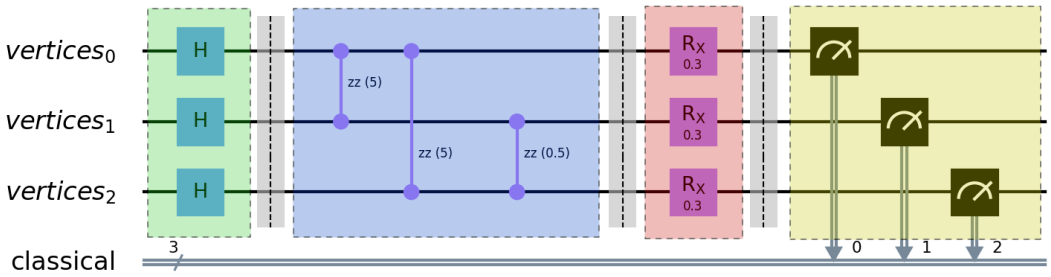
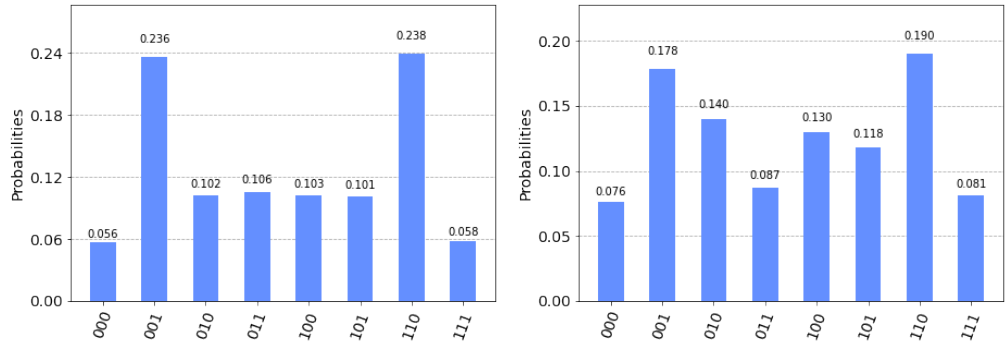



Figure 4.4: Before applying the Hamiltonians, the qubits are brought into superposition by applying a Hadamard $H$ gate on each qubit in the area highlighted green. Then, the problem Hamiltonian $H_C$ is applied using the $R_{ZZ}(\theta)$ gates with $\theta = \gamma w_{ij}$, highlighted blue. Then, the $R_X(\theta)$ gates with $\theta = \beta$ from the driver Hamiltonian $H_B$ are applied on all qubits in the red area. Finally the state is measured, highlighted yellow.




(a) Result from simulation

(b) Result from quantum device

Figure 4.5: The results of the simulation and of the quantum hardware are both correct and solutions show significantly higher amplitudes than non-solutions. This accuracy is likely because of a compact circuit that does not propagate much error.

## 4.5 Solving query optimisation problems with QAOA

In summary, query optimisation seeks to assign a plan (the procedure that describes how to acquire the data) with lowest cost to a query (an expression that describes the desired data). *Query optimisation* is an important database problem which has been examined since IBM's System R [42] and has been generalised for multiple queries as *multiple query optimisation* [43]. The problem is classically well analysed, as different approaches, such as integer linear programming, genetic algorithms and hill-climbing algorithms and others are suggested [44]. *Trummer and Koch* explore the use of a D-Wave quantum annealer for the MQO [45]. The problem is probably not solvable in polynomial time [45].

In more detail, and in case of an MQO, the set of plans $p \in P$ that generates the data required for a set of queries $q \in Q$ with minimal cost is desired. For our example, we assume that there are multiple plans for each query, that the exact cost of each plan is known and only one cost metric is considered. That is, we do not consider a cost vector to simplify this example. We call the set of plans considered for a query $P_{q_i} \subset P$ and the cost of a plan $c_{p_i} \in \mathbb{Z}$. In our model, we additionally consider cost savings by intermediate results that can be shared among two or more plans, which we denote

pairwise as $s_{p_i p_j}$.

**Example.** Consider a database with the following tables and their attributes, where primary and foreign keys are italic:

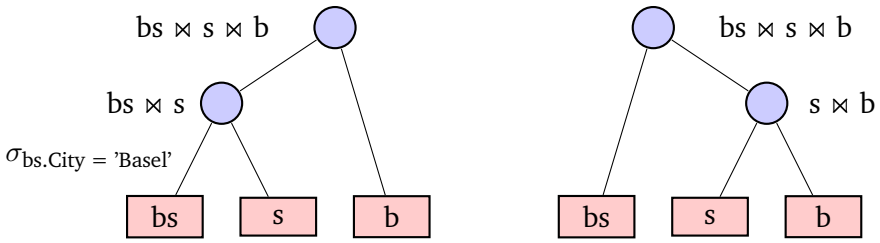
| **Table** | **Attributes** | **# Tuples** |
|---|---|---|
| Bookstore | *BSID*, BSName, City, Manager | 15 |
| Sale | *SID*, *BSID*, *ISBN*, Amount, YearMonth | 45 |
| Book | *ISBN*, *AID*, Title, Category | 19 |
| Author | *AID*, Name | 25 |

Now consider the following queries on this database:

$q_1$ **SELECT** * **FROM** Bookstore bs **NATURAL JOIN** Sale s **NATURAL JOIN** Book b **WHERE** bs.City = 'Basel'

$q_2$ **SELECT** * **FROM** Sale s **NATURAL JOIN** Book b **NATURAL JOIN** Author a **WHERE** a.name = 'D. Knuth'

For each of the two queries, we find two plans that produce the desired result, as shown in figures 4.6 and 4.7. They distinguish by the order of the join operations and by an early selection (i.e. on a low tree level) that greatly reduces the amount of tuples in intermediate results. As each join operation $m \bowtie n$ generally has to compare each entries of $m$ with each entry of $n$, $mn$ comparisons are required. It is therefore desired to maintain as few intermediate results as possible. Conducting selection operations $\sigma$ as early as possible is an effective way to achieve this goal. Note that $p_2$ (query $q_1$) and $p_3$ (query $q_2$) both create the intermediary result $s \bowtie b$ that can be shared.

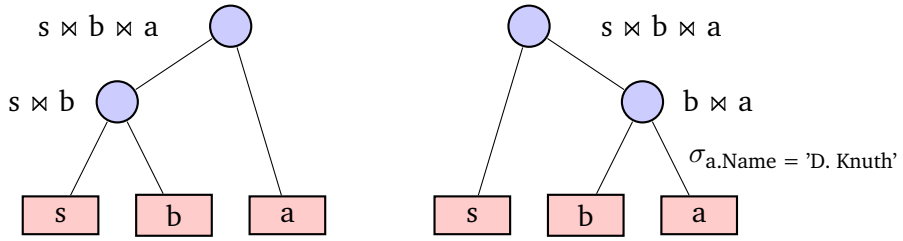



(a) First plan $p_1$ for $q_1$ (b) Second plan $p_2$ for $q_1$

Figure 4.6: Plans for Query $q_1$. In $p_2$ the intermediate result $s \bowtie b$ is created, which is also used in $q_2$. In $p_1$, the selection $\sigma$ bs.City = 'Basel' is done early in the plan, thus reducing the amount of tuples processed by subsequent joins. For the example, we expect the selection to reduce the amount of tuples from the table **Bookstore** $bs$ from 15 to 3.

The cost of plans depend largely, (although not only [44]), on the amount of disk or memory accesses caused by comparisons [43]. A plan with few comparisons also requires few disk or memory accesses. Here, we derive possible costs for plans $p_1 - p_4$ based on comparison operations

Additionally, we calculate the savings of the shared intermediate result $s \bowtie b$ among $p_2$ and $p_3$: $s_{p_2 p_3} = s \times b = 855$. In our model, we require that only one plan is selected per query, which is a straightforward restriction, as there are no negative cost. Also, for each query $q \in Q$, at least one plan $p \in P$ must be selected. Now, we have a complete model




(a) First plan $p_3$ for $q_2$ (b) Second plan $p_4$ for $q_2$

Figure 4.7: Plans for query $q_2$. In $p_3$ the intermediate result $s \bowtie b$ is created, which is also used in $q_1$. In $p_4$, the selection $\sigma$ a.Name = 'D. Knuth' is done early in the plan. For the example, we expect the selection to reduce the amount of tuples from the table **Author** $a$ from 19 to 1.

$$\begin{aligned} c_{p_1} &\quad \sigma_{\text{bs.City = 'Basel'}}(bs) \times s \times b = 2565 \\ c_{p_2} &\quad bs \times s \times b = 12825 \\ c_{p_3} &\quad s \times b \times a = 21375 \\ c_{p_4} &\quad s \times b \times \sigma_{\text{a.Name = 'D. Knuth'}}(a) = 855 \end{aligned}$$

for choosing the query with lowest cost. Solution candidates are encoded in a bit string in the form $p_1 p_2 \dots p_n$ where plans are denoted with $1$ if they are selected and $0$ if not.

$$\underbrace{p_1 p_2}_{q_1} \underbrace{p_3 p_4}_{q_2}$$

To simplify, we linearly scale the cost by factor $1/1000$ and round to the closest integer. Thus we obtain:

$$\begin{aligned} c_{p_1} &\quad 3 \\ c_{p_2} &\quad 13 \\ c_{p_3} &\quad 21 \\ c_{p_4} &\quad 1 \end{aligned}$$

and for the savings $s_{p_2 p_3} = 1$.

From now, the problem can be solved with an optimiser that seeks the bit string with the smallest cost assigned. First, we present a naïve classical solver, then we use QAOA to solve the MQO.

### 4.5.1 Classical solver

**Code reference.** The classical solver is available as `Optimisation/classical_query_solver.py` with the Jupyter Notebook `Optimisation/ClassicSolver.ipynb`.

To encode a MQO configuration, we use three data structures:

- a list containing the cost for each plan in the form $c_{p_1}, c_{p_2}, \dots c_{p_n}$, called `cost`
- a $k \times 3$ matrix containing $k$ pairwise savings as rows in the form $p_i p_j - s_{p_i p_j}$, called `savings`
- a $m \times n$ matrix with the rows containing the plans $P_{q_i}$ for each query $q_i \in Q || Q| = m$, called `ppq` (plans per query)

With the above restrictions, admissible bit strings are exactly the combinations of sets containing alternative plans for a query, that is in our encoding `ppq`.

**Example.** Consider the above configuration, where $p_1$ and $p_2$ yield $q_1$ and $p_3$ and $p_4$ yield $q_2$. We encode this assignment into the variables `cost = [3, 13, 21, 1]`, `savings = [[1, 2, -1]]` and `ppq = [[0, 1], [2, 3]]`. The combination of these yields the admissible solutions

| $p_1$ | $p_2$ | $p_3$ | $p_4$ |
|---|---|---|---|
| 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 |

In our classical solver, the function `get_admissible_solutions` in listing 4.4 creates bit strings of admissible solutions as combinations.

Listing 4.4: Python code for creating admissible solutions

```
1 def get_admissible_solutions(ppq, n_plans):
2     combinations = list(itertools.product(*ppq, repeat=1))
3     admissible_solutions = np.zeros([len(combinations),
          n_plans])
4     for i in range(len(combinations)):
5         admissible_solutions[i, combinations[i]] = 1
6     return admissible_solutions
```

Now, we only have to assign costs to each of the plans. This is a straight forward task, as we only add the cost, given by `cost` of each selected plan to the admissible solutions and subtract the cost savings given by `savings`.

**Example.** In our example, we obtain following cost for admissible solutions. Note that the second solution qualifies for savings, thus subtracting cost of 1 from the sum.

| $p_1$ | $p_2$ | $p_3$ | $p_4$ | Cost |
|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 4 |
| 0 | 1 | 1 | 0 | 33 |

The cost evaluation is done by the function `evaluate_cost` in listing 4.5.

Listing 4.5: Python code for assigning cost to each admissible solution

```
1 def evaluate_cost(solutions, cost, savings):
2     cost_matrix = np.multiply(solutions, cost)
3     savings_per_solution = np.zeros([len(solutions)])
4     for i in range(len(solutions)):
5         for v in savings:
6             if(solutions[i, v[0]]*solutions[i, v[1]] != 0): #
                  then both plans involving the saving are
                 selected
7                 savings_per_solution[i] = v[2]
8     return np.sum(cost_matrix, 1)+savings_per_solution
```

Finally, the solution with the lowest cost is selected. As multiple solutions may have the same cost, multiple solutions may be selected, of which one is finally selected for execution. As we represent cost as scalars, no weighting or preference is required.

**Example.** The lowest-cost solution is clearly encoded by $1001$, with total cost of 4. Therefore, the set of plans to acquire data asked by queries $q_1$ and $q_2$ is $\{ p_1, p_4 \}$.

Our classical solver selects the lowest-cost solutions with the function `get_minimal_plans`, that also invokes the above subroutines and is shown in listing 4.6.

Listing 4.6: Python code for selecting the lowest-cost solution and the invokation of subroutines.

```
1 def get_minimal_plans(cost, savings, ppq):
2     admissible_solutions = get_admissible_solutions(ppq, len(
          cost))
3     plan_cost = evaluate_cost(admissible_solutions, cost,
          savings)
4     minimum_cost = min(plan_cost)
5     minimum_cost_idx = np.where(plan_cost == minimum_cost)
6     minimum_solutions = admissible_solutions[minimum_cost_idx
          ]
7     return minimum_solutions, minimum_cost
```

### 4.5.2 MQO implementation

To solve any optimisation problem with QAOA, we have to formulate the problem in terms of a QUBO. In the MaxCut example, the problem is easily representable as QUBO, as desired states have only one goal, i.e. heavy edges in different subgraphs. With MQO different goals are pursued. (3) we have to guarantee that for each query, exactly one plan is selected, (2) that cost savings are regarded, and finally, (1) that low plan costs are preferred. Thus, we have to formulate a QUBO for each of the goals. We treat each plan as variable, which suits the bit string notation of solutions well.

**Code reference.** The QAOA implementation is available as `Optimisation/QAOA_MQO.ipynb`.

**Example.** We start devising the QUBO for the cost of each plan (1). As there is no dependency among the plans $P$, we simply express them as linear terms and weight them according to cost. Since the maximum is sought and high cost should be penalised, we use negative weights and prescale them by the factor $1/2$, because smaller values are desired in quantum circuits, as we explain later. We finally obtain the QUBO sum

$$\sum_{i \leq j} w_{ij} x_i x_j = -3x_1 - 13x_2 - 21x_3 - x_4$$

where we have replaced the plans $p_i$ by variables $x_i$ that can take values $\{ 0, 1 \}$. We now proceed to encode the goal of regarding cost savings $s_{p_i p_j}$ (2). Here, the pairwise notation of cost savings is beneficial, as we can only represent binary terms by default[3]. As savings are desired, we choose a positive weight of the saved cost and write the QUBO sum

$$\sum_{i \leq j} w_{ij} x_i x_j = x_2 x_3.$$

Because the cost savings are exactly 1, no explicit weighting factor is visible. Finally, we write the QUBO that enforces to choose exactly one plan per query (3). This goal is non-negotiable and must therefore be able to override cost goals, for which we use a weight

[3] *Whitfield et al.* [38] suggest an approach for general Ising coupling gates for $\bigotimes_n \sigma_Z^i$. In our trials, we failed to make them work correctly.

which is larger than the cost and choose 11. For this formulation, a pairwise notation of plans per query is also suitable, such as in (2). Here, we want to penalise identical values among the plans for a query $P_{q_i}$, i.e. the inversion of the MaxCut problem. The final QUBO sum is

$$\sum_{i \leq j} w_{ij}(x_i(1-x_j)+x_j(1-x_i)) = -11(x_0(1-x_1)+x_1(1-x_0))-11(x_2(1-x_3)+x_3(1-x_2)).$$

Now, the QUBO sums can be reformulated as Hamiltonians, where we maintain the separation of the goals (1) to (3) and proceed in the same order, starting with (1):

$$H_{C_1} = -3\sigma_Z^1 - 13\sigma_Z^2 - 21\sigma_Z^3 - \sigma_Z^3$$

where we trivially replace the variables $x_i$ by $\sigma_Z^i$ operators. In our Qiskit program, the code listing 4.7 represents the steps thus far.

Listing 4.7: Qiskit code for initialisation (lines 6-8) and implementation of goal (1) as quantum circuit. On line 10, the uniform superposition is created and on lines 13 and 14, the for loop represents the Hamiltonian $H_{C_3}$. We additionally set cost, angles and scaling factors. The angles $\beta$ and $\gamma$ are explained later.

```
1  beta = 0.3
2  gamma = 0.5
3  scaling = 0.5
4  cost = [3, 13, 21, 1]
5
6  x_register = QuantumRegister(4, name='vars')
7  classical_register = ClassicalRegister(4, name='classic')
8  qc = QuantumCircuit(x_register, classical_register)
9
10 qc.h(x_register)
11 qc.barrier()
12
13 for i in range(len(cost)):
14     qc.rz(-cost[i]*scaling*gamma, i)
```

Now, we reformulate (2)

$$H_{C_2} = -\sigma_Z^2\sigma_Z^3$$

here, a discrepancy appears: the term $x_2x_3$ equals 1 iff both variables are 1, and otherwise 0, which can be considered as a $AND$ operation. The $\sigma_Z^2\sigma_Z^2$ operation (implemented by $R_{ZZ}$-gates) however acts as XOR gate[4]. As a consequence, we invert the weight of the operation, essentially creating a $XNOR$ gate that penalises different values of $x_2$ and $x_3$. The state $|x_2x_3\rangle = |00\rangle$ is however falsely rewarded as well, which we neglect in this experiment. In the Qiskit program, the goal (2) only requires one line of code:

```
1  qc.rzz(-1*gamma, 1,2)
```

Finally, we reformulate (3), where the same simplifications as with the MaxCut problem are applicable. Since we want to reward different values among plans, i.e. $|x_ix_j\rangle = \{\ |01\rangle, |10\rangle\ \}$, we choose positive weights:

$$H_{C_3} = 11\sigma_Z^1\sigma_Z^2 + 11\sigma_Z^3\sigma_Z^4$$

[4]We discovered that $CU_1$-gates had the property to act as $AND$ analogues, but were however not able to reproduce this behaviour in nontrivial quantum circuits with other gates.

In the Qiskit implementation, this Hamiltonian requires two lines of code:

```
qc.rzz(11*gamma, 0, 1)
qc.rzz(11*gamma, 2, 3)
```

For the driver Hamiltonian $H_B$ we choose

$$H_B = \sum_{i=1}^{n} \sigma_X^i = \sigma_X^1 + \sigma_X^2 + \sigma_X^3 + \sigma_X^4.$$

which is implemented by the code in listing 4.8.

Listing 4.8: Here, we implement the driver Hamiltonian $H_B$ (line 2) and the measuring of the qubits (line 6).

```
1 # Driver Hamiltonian
2 qc.rx(beta, x_register)
3 qc.barrier()
4
5 # Measuring of all qubits
6 qc.measure(x_register, classical_register)
```

For the quantum circuit, we choose the angles $\gamma = 0.5$ and $\beta = 0.3$, as they have empirically shown to produce correct results. By setting $\gamma = 0.5$ in conjunction with the prescaling of cost, we scale all the weights such that they are in the value range $[0, 2\pi]$ of $R_Z$ and $R_{ZZ}$-gates, as with the MaxCut example. The resulting quantum circuit is depicted in figure 4.8

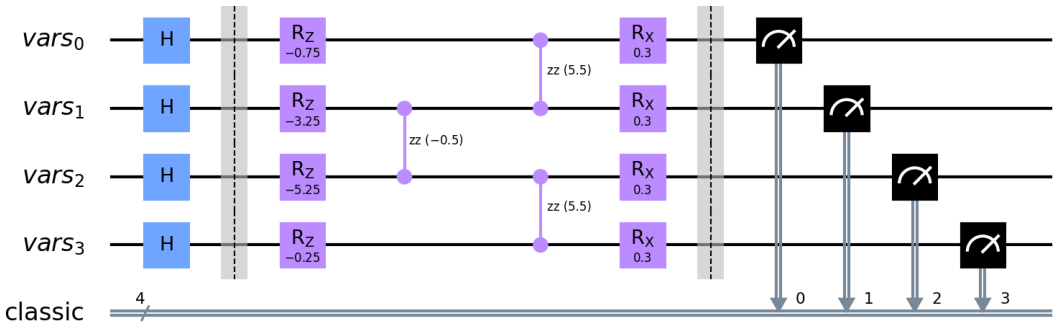
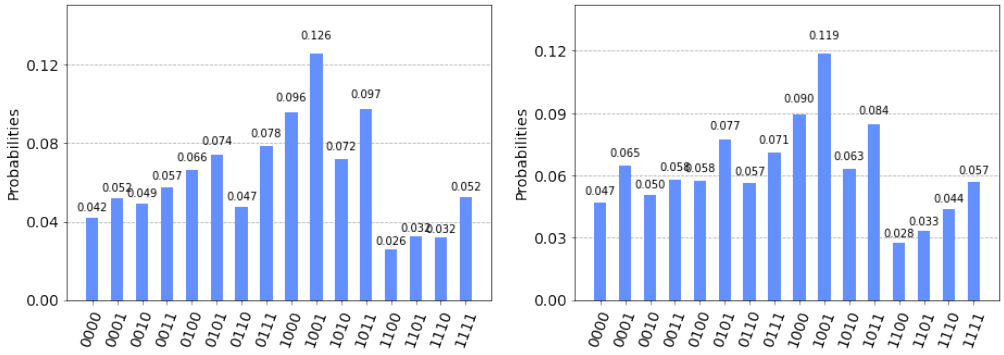



Figure 4.8: Quantum circuit of the QAOA for the MQO problem

We execute the algorithm both as simulation and with a quantum device, with the results rendered in figure 4.9.




(a) Result from simulation (b) Result from quantum device

Figure 4.9: The results of the simulation and of the quantum hardware are both correct and solutions show significantly higher amplitudes than non-solutions. Although the algorithm requires four qubits, that already produced incorrect results with Grover's algorithm, the QAOA is able to keep error low.

### 4.5.3 Results

In this simple instance, the naïve classical solver performs well in terms of runtime and memory used. For complex MQO, as they are encountered in production databases, it is infeasible. As mentioned in the introduction, optimisation approaches as integer linear programming or genetic algorithms are used, and more recently, custom algorithms for MQO have been suggested. The main advantage of the QAOA against any classical algorithm is the memory and time complexity. Consider following example:

**Example.** A MQO with $n$ queries each containing $m$ plans per average produces a search space of $m^n$ admissible solutions. With every additional query, the amount of admissible solutions is multiplied by $m$. For 6 queries $n = 6$, each containing 4 plans $m = 4$, we obtain $4^6 = 1296$ solutions. The naïve algorithm evaluates the cost each of those solutions, thus experiencing a combinatorial explosion. The QAOA however only requires as many qubits as there are plans and can ideally approximate the optimal solution with $m$ qubits.

Furthermore, as low latencies most often are important for database applications, a good solution for a MQO should be found in a short time frame, with the relaxation that the solution must not be globally optimal. As the QAOA is able to approximate good solutions after few repetitions, it may be suitable for database applications. However, as the parameters defining the MQO will be available on a classical system, they would first needed to be translated into a quantum circuit, executed and then be interpreted classically, which introduces additional latency. To assess the effectiveness of this approach, further evaluation would be required.

### 4.5.4 Relation to Similar Techniques

The algorithm is a realisation of the adiabatic theorem for gate-based quantum computers. Quantum annealing (QA) is another realisation of the adiabatic theorem facilitated on quantum annealers, with *D-Wave* as prominent provider [46, 45]. There are however differences between these techniques: where QA is conceived to find the optimal solution, possibly with a very long run time, QAOA approximates the solution in a defined number of repetitions, where the optimal solution can be approximated arbitrarily close. Further comparison between QAOA, QA and classical simulated annealing (SA) is given by *Streif and Leib* [47].

## 4.6 Variational Quantum Eigensolver

The Variational Quantum Eigensolver is primarily used in physical systems where the ground energy state is desired. This state is represented by the minimum eigenvalue of the Hamiltonian describing the energy state of the system under consideration. Alternatively this may be computed by quantum phase estimation which is however infeasible by current (*NISQ*-era) quantum computers, conversely to VQE. the technique can however also be used for different applications where the eigenvector of a hermitian matrix is desired [48, 8].

With $\lambda_{min}$ being an unknown minimum eigenvalue of a hermitian matrix $H$, associated with the eigenstate $|\psi_{min}\rangle$, the VQE provides $\lambda_\theta$ as the expectation value $E(\theta)$, which is a upper bound for $\lambda_{min}$.

$$\begin{aligned} \lambda_{min} \leq \lambda_\theta = E(\theta) \\ \equiv \langle\psi(\theta)| H |\psi(\theta)\rangle \end{aligned}$$

By applying a parameterised circuit $U(\theta)$ on some arbitrary starting state $|\psi\rangle$, $\lambda_\theta$ can be iteratively optimised by a classical controller changing the value of $\theta$ as rendered in figure 4.10.

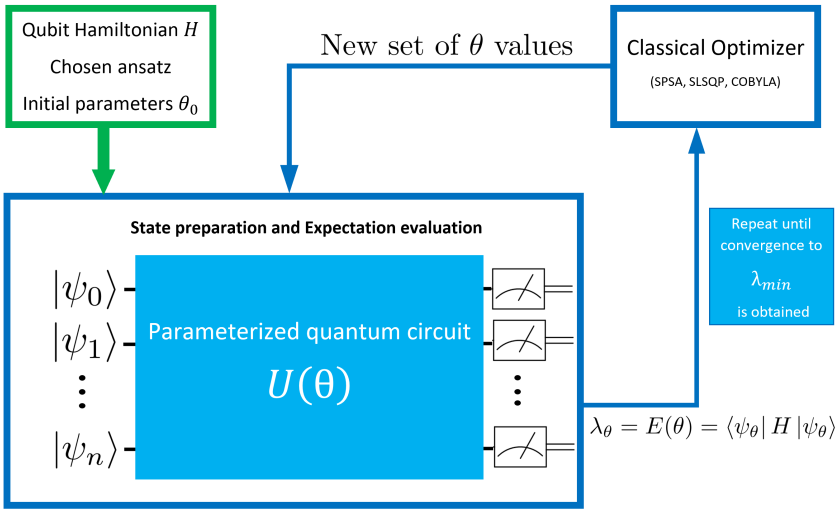



Figure 4.10: High-level overview of a VQE

### The Ansatz

The Ansatz describes the initial gate constellation for the quantum circuit. If a suitable ansatz for a problem is chosen, it should be possible to represent every necessary state to find the best solution. For example if the circuit would only consist of $R_X$-Gates we would never be able to represent all states around the Bloch sphere. Additionally the coverage of the ansatz for different entangled states of the qubits must be taken under consideration as well [49].

$R_X$, $R_Y$ or Hartree-Fock are often used as ansätze. A good ansatz should meet the following conditions:

- Covering ideally the whole space of possible states.
- Should be shallow, because the more gates are used, the higher the noise and hence the vulnerability to errors.

- Should not have many parameters, because more parameters make the optimisation more expensive.

### 4.6.1 VQE Algorithm

To emphasise the operating principle of the VQE, we will implement it on the example of a MaxCut problem. The MaxCut of a graph partitions its vertices into two subsets by cutting the edges with the highest weights, as introduced in the above section and rendered in figure 4.11

**Code reference.** The code used in this section is available as `Optimisation/VQE.ipynb`

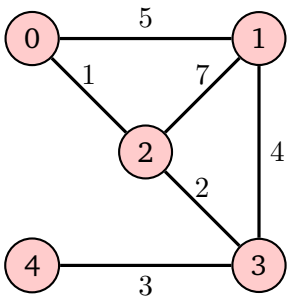



Figure 4.11: A cut through which edges gives the maximum sum?

Listing 4.9 shows a way to construct the graph in figure 4.11 in Python [50].

Listing 4.9: creating the graph

```
# Create graph
# In the form start, end, weight
V = [0, 1, 2, 3, 4]
E = [(V[0], V[1], 5),
     (V[0], V[2], 1),
     (V[1], V[2], 7),
     (V[1], V[3], 4),
     (V[2], V[3], 2),
     (V[3], V[4], 3)]

n = len(V)
G=nx.Graph()
G.add_nodes_from(V)
G.add_weighted_edges_from(E)

colors = ['r' for node in G.nodes()]
pos = {0: (0, 40), 1: (40, 40), 2: (20, 20), 3: (40, 0), 4:
   (0, 0)}

def construct_graph(G, colors, pos):
    nx.draw_networkx(G, node_color=colors, pos=pos)
    edge_labels = nx.get_edge_attributes(G, 'weight')
    nx.draw_networkx_edge_labels(G, pos=pos, edge_labels=
       edge_labels)

construct_graph(G, colors, pos)
```

### Problem formulation

The graph can be modelled as a adjacency matrix. For this, the vertices are enumerated and the weight of the edges are assigned to the row and column corresponding to the surrounding vertices. Position (0, 1) in the matrix is equal to the edge between vertice 0 and 1: 5.

The adjacency matrix can be extracted from the graph in Python as shown in listing 4.10 [50].

Listing 4.10: Deriving the adjacent matrix

```
# Computing the adjacent matrix from the graph
w = np.zeros([n,n])
for i in range(n):
    for j in range(n):
        temp = G.get_edge_data(i,j,default=0)
        if temp != 0:
            w[i,j] = temp['weight']
print(w)
```

The code yields the adjacency matrix

$$\begin{bmatrix} 0 & 5 & 1 & 0 & 0 \\ 5 & 0 & 7 & 4 & 0 \\ 1 & 7 & 0 & 2 & 0 \\ 0 & 4 & 2 & 0 & 3 \\ 0 & 0 & 0 & 3 & 0 \end{bmatrix}$$

### Mapping the adjacency matrix to a Hamiltonian

Mapping the adjacency matrix to a Hamiltonian is generally done by mapping to the Ising problem. Listing 4.11 shows the procedure to accomplish this in Python.

Listing 4.11: Mapping into Hamiltonian

```
# Mapping to the Ising problem
qubitOp, offset = max_cut.get_operator(w)
```

$$\begin{aligned} IIIZZ(2.5+0j) \\ IIZIZ(0.5+0j) \\ IIZZI(3.5+0j) \\ IZIZI(2+0j) \\ IZZII(1+0j) \\ ZZIII(1.5+0j) \end{aligned}$$

### Defining the classical optimiser to solve the problem

The optimiser varies the parameters according to the measured expectation value. This process is classical. Various optimisation techniques can be utilised depending on the problem, such as SPSA, SLSQP or COBYLA, which are commonly used as optimiser for this task [50, 49]. Those optimisers are part of the Qiskit library. Listing 4.12 shows how they can be accessed.

Listing 4.12: Choosing the classical optimiser

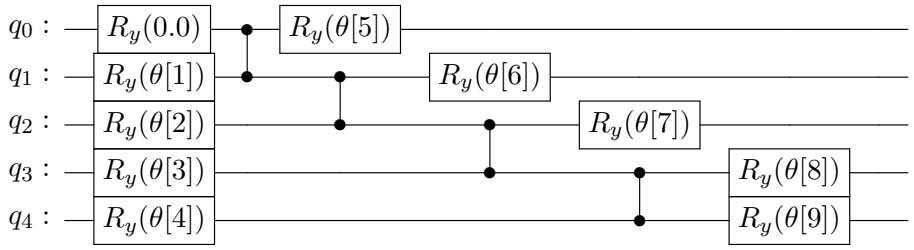
```
from qiskit.aqua.components.optimizers import SPSA, SLSQP,
    COBYLA
optimiser = SPSA()
optimiser = SLSQP()
optimiser = COBYLA()
```

### Choosing the Ansatz

The ansatz depends on the task to solve and of the quantum device the algorithm is executed. It consists of all the gates in place to perform the necessary entanglement between the qubits and allows the classical optimizer to rotate the single qubits to the optimal solution. In Qiskit, the `TwoLocal` class is meant to implement the various ansätze as shown in listing 4.13. First, the rotation gate type is chosen. The number of repetitions, `reps` defines how often the gates and the entanglement are repeated in the ansatz circuit. The various types of entanglement determine how the qubits in the circuit are entangled [49].

Listing 4.13: Choosing the Ansatz

```
ansatz = TwoLocal(qubitOp.num_qubits, ['ry', 'rz'], 'cz',
    reps=1, entanglement='sca')
ansatz = TwoLocal(qubitOp.num_qubits, 'ry', 'cz', reps=1,
    entanglement='linear')
```




### Creating a `QuantumInstance`

Now, we create a `QuantumInstance` with the device we want to use. This can either be a simulator or a real device. In this example we use Qiskit's *statevector_simulator* as shown in listing 4.14.

Listing 4.14: Choosing the QuantumInstance

```
backend = Aer.get_backend('statevector_simulator')
quantum_instance = QuantumInstance(backend)
```

### Constructing the VQE Eigensolver

We construct the VQE eigensolver with the specified parameters as shown in listing 4.15.

Listing 4.15: Creating the VQE Circuit

```
vqe = VQE(qubitOp, ansatz, optimiser, quantum_instance=
    quantum_instance)
```

#### Running the VQE

Finally, we run the VQE on the given `QuantumInstance` and print the result 4.12. To facilitate this, the code from listing 4.16 is used.

Listing 4.16: Solving with VQE

```
result = vqe.run(quantum_instance)

x = sample_most_likely(result.eigenstate)
result = max_cut.get_graph_solution(x)

print('result:', result)
print('maxcut value:', max_cut.max_cut_value(x, w))

colors = ['c' if result[i] == 0 else 'r' for i in range(n)]
construct_graph(G, colors, pos)
```

| | |
|---|---|
| result | [1. 0. 1. 1. 0.] |
| maxcut value | 19 |

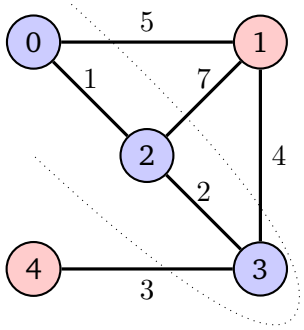



Figure 4.12: The maximum sum is achieved by cutting through the edges creating the red and blue subsets.

Using this hybrid approach, the quantum system can be used for problems where it is intrinsically superior to a classical computer, that is, the simulation of quantum systems, factoring large numbers or searching in unstructured datasets, while the classical computer solves the part where quantum computers are inferior [33].

### 4.6.2 Query optimisation using VQE and Qiskit

The query optimisation problem can also be solved using Qiskit and the VQE algorithm. In this section, this approach is described and implemented with Qiskit.

#### Problem description as QUBO

The problem can easily be described as a QUBO problem. For this only the costs of each query plan are taken into account. Here, we use plans $p_1, \ldots, p_4$ with the associated cost:

| | |
|---|---|
| $c_{p_1}$ | 2 |
| $c_{p_2}$ | 4 |
| $c_{p_3}$ | 3 |
| $c_{p_4}$ | 1 |

and for the savings $s_{p_2p_3} = 5$. Also, query $q_1$ is produced by $p_1$ and $p_2$ and query $q_2$ is produced by $p_3$ and $p_4$. The selection with the lowest total cost is $p_2p_3$ with cost of 2. Note that we name the plans as $X_1, \dots X_4$ in the following experiment. The solution is represented as bit string in the form $X_1X_2X_3X_4 = 0110$.

The QUBO is formulated as described in listing 4.17.

Listing 4.17: Formulating the query optimisation problem as QUBO

```
# execution costs
wl = 3
wm = 9

costs = [2, 4, 3, 1]
for i in range(len(costs)):
    costs[i] = costs[i] - wl

# create a QUBO
qubo = QuadraticProgram()
for i in range(len(costs)):
    qubo.binary_var('X' + str(i+1))

qubo.minimize(linear=costs, quadratic={('X2', 'X3'):-5,
                                       ('X1', 'X2'):wm,
                                       ('X3', 'X4'):wm})
print(qubo.export_as_lp_string())
```

This code yields the following problem description. Firstly, the complete cost function is given under *Minimize*. Then, the domain of the variables $x_1, \dots, x_4$ is restricted to binary values. Finally, the variables are listed.

```
Minimize
obj: - X1 + X2 - 2 X4 + [ 18 X1*X2 - 10 X2*X3 + 18 X3*X4 ]/2
Subject To

Bounds
0 <= X1 <= 1
0 <= X2 <= 1
0 <= X3 <= 1
0 <= X4 <= 1

Binaries
X1 X2 X3 X4
End
```

### Mapping to a Ising Hamiltonian

Employing Qiskit libraries, the QUBO formulation can be easily mapped to an Ising Hamiltonian as shown in listing 4.18.

Listing 4.18: Translation of the QUBO to a Hamiltonian using Qiskit libraries.

```
1 op, offset = qubo.to_ising()
2 print('offset: {}'.format(offset))
3 print('operator:')
4 print(op)
```

This yields the following form, where operators with weights are listed. Note that each row represents an operation step with weights applied to the Z-operators and each column represents a variable, or a qubit, respectively.

$$
\begin{aligned}
offset &: 2.25 \\
operator &: \\
SummedOp([ & \\
-1.75 * IIIZ, & \\
-1.5 * IIZI, & \\
-1.25 * ZIII, & \\
2.25 * IIZZ, & \\
-1.25 * IZZI, & \\
-1.0 * IZII, & \\
2.25 * ZZII & \\
]) &
\end{aligned}
$$

### Simulating the circuit

The device to run the circuit on is now defined. In this case, we use the *statevector* simulator. The required Qiskit instructions are shown in listing 4.19.

Listing 4.19: Choosing quantum instance

```
1 quantum_instance = QuantumInstance(Aer.get_backend('
    statevector_simulator'))
```

### Constructing the VQE circuit

The VQE circuit is constructed by choosing the classical optimiser and specifying the ansatz to use [50]. The way to perform this task in Qiskit is shown in listing 4.20.

Listing 4.20: Constructing the VQE

```
1 spsa = SPSA()
2 ryrz = TwoLocal(4, ['ry', 'rz'], 'cz', reps=10, entanglement=
    'linear')
3 vqe = VQE(op, ryrz, spsa, quantum_instance=quantum_instance)
```

**Running the circuit**

Finally, the circuit is executed and the result printed, as in listing 4.21.

Listing 4.21: Formulating the query optimisation problem as QUBO

```
result = vqe_mes.run(quantum_instance)
x = sample_most_likely(result.eigenstate)
print(x)
```

The following result is produced:

$$[0.1.1.0.]$$

A one in the result means that this query plan should be taken to have the least costs. Following from that, the suggested plan selection is $p_2 p_3$, which is the solution we expected.

# Chapter 5

# Discussion

In chapter 3, we have shown how typical search problems can be solved with Grover's algorithm by the examples of constraint satisfaction problems (CSP) and boolean satisfiability (SAT) problems. While we discovered that Grover's algorithm is an effective method of solving such problems in theory, current quantum computers proved inadequate to solve all but trivial problems.

Grover's algorithm is currently constrained by three elements: firstly, the complete search space must be constructed and held on the quantum computer during the execution of the algorithm, similar to RAM in digital computers. This "quantum RAM" is currently restricted by the low amount of qubits available on quantum computers. Furthermore, additional qubits are often required to store intermediate results, thus making qubits even more scarce. Secondly, oracles (subroutines that represent the problem of interest) are often difficult to construct in practice. We found that their implementation is often neglected in the literature. Thirdly, Grover's algorithm is prune to error due to its long set of operations, even for simple problems.

Error is introduced by inaccuracies of individual gates, as well as by decoherence of the quantum system, which increases with the length of the execution of the algorithm. There are further error sources that are yet not understood well.

Hence, we consider Grover's algorithm as powerful search technique due to its $\mathcal{O}\sqrt{n}$ runtime, but find the algorithm to be infeasible on current quantum computers. We roughly analyse a source of error and discover that IBM quantum computers show to be less erroneous than a Rigetti quantum computer featuring more qubits. Error is the main problem of current quantum computers, therefore we examine an error-correction technique and implement it on quantum hardware. Although this method may improve stability of quantum algorithms, it requires many qubits that are currently not available.

In chapter 4, we examine two optimisation algorithms in more detail: the variational quantum eigensolver (VQE) and the quantum approximate optimisation algorithm (QAOA). Both of these hybrid (or variational) algorithms are particularly suited for current, erroneous quantum computers with parts executed on quantum computers and parameter variation on classical computers. We use the algorithms to solve combinatorial optimisation problems (CSP, an expanded variant of SAT) using VQE and QAOA and discover that the algorithms perform reliably.

The workings of the algorithms and formulation of the problem are closely related to physics, and are therefore not straightforward for computer scientists. As it is sometimes necessary to adjust parameters of the algorithms, a more thorough understanding of the

underlying quantum systems and effects is helpful.

It is possible to formulate many optimisation problems for VQE and QAOA in terms of the quadratic unconstrained binary optimisation formalism (QUBO), for which we present a general procedure. Nevertheless, we find that implementing boundary conditions require further steps and expand the QUBO term and finally, the quantum circuit. As larger quantum circuits are more prone to error, problems containing many boundary conditions are more difficult to execute due to error.

As example, we solve a multiple query optimisation problem (MQO), a database-related problem, using VQE and QAOA and compare it to a naïve classical algorithm. For the considered problem size, both algorithms produce the correct result, but for large problems, the naïve algorithm would experience an exponential growth of memory and time consumption, while the amount of qubits for VQE or QAOA would only grow linearly with the problem size.

Here, another comparison between classical and quantum approaches can be drawn: classical algorithms usually exploit the structure of specific problems to perform superior to their naïve variants. Their goal is almost always to prune the search space. Quantum algorithms, however, are often agnostic to the problem they are applied to and are therefore less "hard-coded" for particular instances. Thus, quantum algorithms are in principle more flexible than competitive classical algorithms.

When considering quantum algorithms, they should not be considered as subroutines for solving a particular problem. This is, quantum algorithms often perform better than their classical counterparts, e.g. $\mathcal{O}(\sqrt{n})$ vs. $\mathcal{O}(n)$ for unstructured search, $\mathcal{O}(\log(n))$ vs. $\mathcal{O}(n^2)$ for solving systems of linear equations. But unlike the classical counterparts, they are often constrained by various factors.

For instance, where classic *Gauss-Seidel* method for solving linear systems is generally only influenced by the matrices condition, where the quantum variant (*HHL*-algorithm) encounters three additional constrains. Knowing them, users of quantum computers can refine their problem such that it is more suitable to the algorithm. As a such, problems should be treated as a whole when considering solving them with a quantum computer. Further insight is given by *S. Aaronson* in [51].

## 5.1 Outlook

This work has shown a practical approach to the programming of selected quantum algorithms. Although we show how the algorithms perform on real quantum hardware, we provide no quantitative evaluation of their stability or statistical analysis. Currently, such analysis is lacking and methods to benchmark quantum computers are rare [23]. Hybrid algorithms as the VQE and QAOA are two of the most promising quantum algorithms as they may be used productively on short-term quantum computers. However, these algorithms are yet not very well understood, especially in practice and regarding the QAOA for multiple repetitions $p > 1$.

While algorithm design and applicable algorithms have been studied for more than 60 years in convolution with digital computer hardware, the design of quantum algorithms is different, as the hardware distinguishes. Exploiting the hardware's properties to devise new algorithms is therefore a promising field. A further topic is examining practical approaches to problems supposed to be solvable in a more efficient manner by quantum computers, such as chemistry, finance or optimisation, similar to this work.

# Bibliography


[1] G. E. Moore *et al.*, “Cramming more components onto integrated circuits,” 1965.

[2] R. P. Feynman, “Simulating physics with computers,” *Int. J. Theor. Phys*, vol. 21, no. 6/7, 1982.

[3] P. W. Shor, “Algorithms for quantum computation: discrete logarithms and factoring,” in *Proceedings 35th annual symposium on foundations of computer science*. Ieee, 1994, pp. 124–134.

[4] D. Monroe, “Neuromorphic computing gets ready for the (really) big time,” 2014.

[5] Wikipedia contributors, “Timeline of quantum computing and communication — Wikipedia, the free encyclopedia,” https://en.wikipedia.org/w/index.php?title=Timeline_of_quantum_computing_and_communication&oldid=994303604, 2020, [Online; accessed 17-December-2020].

[6] H. Riel, “"quantencomputer - status und wohin geht es?",” 12 2020, talk at the "Naturwissenschaftliche Gesellschaft Winterthur", [Accessed: 11.12.2020]. [Online]. Available: https://www.youtube.com/watch?v=82GKvFw9XaA

[7] K. L. Brown, W. J. Munro, and V. M. Kendon, “Using quantum computers for quantum simulation,” *Entropy*, vol. 12, no. 11, pp. 2268–2307, 2010.

[8] H. Abraham, AduOffei, R. Agarwal, I. Y. Akhalwaya, G. Aleksandrowicz, T. Alexander, and M. Amy, “Qiskit: An open-source framework for quantum computing,” 2019.

[9] M. Homeister, *Quantum Computing verstehen*. Springer, 2008.

[10] “Bra–ket notation,” Dec 2020. [Online]. Available: https://en.wikipedia.org/wiki/Bra-ket_notation

[11] 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.

[12] C. Sinz, “Practical Applications of SAT,” [Online], 10 2005, http://www.carstensinz.de/talks/RISC-2005.pdf (Accessed 08.10.2020).

[13] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. USA: Prentice Hall Press, 2009.

[14] I. Lynce and J. Marques-Silva, “Efficient haplotype inference with boolean satisfiability,” in *National Conference on Artificial Intelligence (AAAI)*. AAAI Press, 2006.

[15] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, “Satisfiability solvers,” *Foundations of Artificial Intelligence*, vol. 3, pp. 89–134, 2008.

[16] A. Gilliam, S. Woerner, and C. Gonciulea, “Grover adaptive search for constrained polynomial binary optimization,” *arXiv preprint arXiv:1912.04088*, 2019.

[17] W. P. Baritompa, D. W. Bulger, and G. R. Wood, “Grover’s quantum algorithm applied to global optimization,” *SIAM Journal on Optimization*, vol. 15, no. 4, pp. 1170–1184, 2005.

[18] C. Gottschall, “Zentrale verarbeitung,” Apr 2020. [Online]. Available: https://www.erpelstolz.at/gateway/formular-zentral.html

[19] M. A. Nielsen and I. Chuang, “Quantum computation and quantum information,” 2002.

[20] T. I. S. Competition", “SAT Competition 2009: Benchmark Submission Guidelines,” [Online], 01 2009, http://www.satcompetition.org/2009/format-benchmarks2009.html (Accessed 15.10.2020).

[21] M. Boyer, G. Brassard, P. Høyer, and A. Tapp, “Tight bounds on quantum searching,” *Fortschritte der Physik*, vol. 46, no. 4-5, p. 493–505, Jun 1998. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1521-3978(199806)46:4/5<493::AID-PROP493>3.0.CO;2-P

[22] A. Asfaw, L. Bello, Y. Ben-Haim, S. Bravyi, N. Bronn, L. Capelluto, A. C. Vazquez, J. Ceroni, R. Chen, A. Frisch, J. Gambetta, S. Garion, L. Gil, S. D. L. P. Gonzalez, F. Harkins, T. Imamichi, D. McKay, A. Mezzacapo, Z. Minev, R. Movassagh, G. Nannicni, P. Nation, A. Phan, M. Pistoia, A. Rattew, J. Schaefer, J. Shabani, J. Smolin, K. Temme, M. Tod, S. Wood, and J. Wootton. (2020) Learn quantum computation using qiskit. [Online]. Available: http://community.qiskit.org/textbook

[23] T. J. Proctor, K. M. Rudinger, K. Young, E. Nielsen, and R. J. Blume-Kohout, “Demonstrating scalable benchmarking of quantum computers.” 11 2019.

[24] IBM Quantum team. (2020) ibmq_athens v1.3.0. Accessed 28.11.2020. [Online]. Available: https://quantum-computing.ibm.com

[25] Amazon. (2020) Amazon hardware providers: Rigetti. Accessed 28.11.2020. [Online]. Available: https://aws.amazon.com/braket/hardware-providers/rigetti

[26] B. Schroeder, E. Pinheiro, and W.-D. Weber, “Dram errors in the wild: a large-scale field study,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 193–204, 2009.

[27] R. Laflamme, C. Miquel, J. P. Paz, and W. H. Zurek, “Perfect quantum error correcting code,” *Physical Review Letters*, vol. 77, no. 1, p. 198, 1996.

[28] R. M. Karp, “Reducibility among combinatorial problems,” in *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[29] F. Black and M. Scholes, “The pricing of options and corporate liabilities,” *Journal of political economy*, vol. 81, no. 3, pp. 637–654, 1973.

[30] N. Wiebe, D. Braun, and S. Lloyd, “Quantum algorithm for data fitting,” *Physical Review Letters*, vol. 109, no. 5, Aug 2012. [Online]. Available: http://dx.doi.org/10.1103/PhysRevLett.109.050505

[31] A. W. Harrow, A. Hassidim, and S. Lloyd, “Quantum algorithm for linear systems of equations,” *Physical review letters*, vol. 103, no. 15, p. 150502, 2009.

[32] R. Weber and Y. Mehmet, "Linear regression on a quantum computer," 2020.

[33] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'brien, "A variational eigenvalue solver on a photonic quantum processor," *Nature communications*, vol. 5, p. 4213, 2014.

[34] N. Moll, P. Barkoutsos, L. S. Bishop, J. M. Chow, A. Cross, D. J. Egger, S. Filipp, A. Fuhrer, J. M. Gambetta, M. Ganzhorn *et al.*, "Quantum optimization using variational algorithms on near-term quantum devices," *Quantum Science and Technology*, vol. 3, no. 3, p. 030503, 2018.

[35] F. Malik, "What are eigenvalues and eigenvectors?" Jan 2020. [Online]. Available: https://medium.com/fintechexplained/what-are-eigenvalues-and-eigenvectors-a-must-know-concept-for-machine-learning-80d0fd330e

[36] A. Lucas, "Ising formulations of many np problems," *Frontiers in Physics*, vol. 2, p. 5, 2014.

[37] M. Born and V. Fock, "Beweis des adiabatensatzes," *Zeitschrift für Physik*, vol. 51, no. 3-4, pp. 165–180, 1928.

[38] J. D. Whitfield, J. Biamonte, and A. Aspuru-Guzik, "Simulation of electronic structure hamiltonians using quantum computers," *Molecular Physics*, vol. 109, no. 5, pp. 735–750, 2011.

[39] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," *arXiv preprint arXiv:1411.4028*, 2014.

[40] L. Zhou, S.-T. Wang, S. Choi, H. Pichler, and M. D. Lukin, "Quantum approximate optimization algorithm: performance, mechanism, and implementation on near-term devices," *arXiv preprint arXiv:1812.01041*, 2018.

[41] M. Stechly, "Quantum approximate optimization algorithm explained," Online, 05 2020, accessed 05.10.2020.

[42] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, 1979, pp. 23–34.

[43] T. K. Sellis, "Multiple-query optimization," *ACM Transactions on Database Systems (TODS)*, vol. 13, no. 1, pp. 23–52, 1988.

[44] I. Trummer and C. Koch, "Multi-objective parametric query optimization," *ACM SIGMOD Record*, vol. 45, no. 1, pp. 24–31, 2016.

[45] ——, "Multiple query optimization on the d-wave 2x adiabatic quantum computer," *arXiv preprint arXiv:1510.06437*, 2015.

[46] D-Wave Systems Inc. (2017) A the d-wave 2000qtm quantum computer technology overview. Accessed 05.12.2020. [Online]. Available: https://www.dwavesys.com/sites/default/files/D-Wave%202000Q%20Tech%20Collateral_0117F_0.pdf

[47] M. Streif and M. Leib, "Comparison of qaoa with quantum and simulated annealing," *arXiv preprint arXiv:1901.01903*, 2019.

[48] D. Wang, O. Higgott, and S. Brierley, "Accelerated variational quantum eigensolver," *Physical review letters*, vol. 122, no. 14, p. 140504, 2019.

[49] M. Stechly, “Recent posts,” 2019. [Online]. Available: https://www.mustythoughts.com/variational-quantum-eigensolver-explained

[50] Development Team, Qiskit, “Max-cut and traveling salesman problem¶,” Dec 2020. [Online]. Available: https://qiskit.org/documentation/tutorials/optimization/6_examples_max_cut_and_tsp.html

[51] S. Aaronson, “Read the fine print,” *Nature Physics*, vol. 11, no. 4, pp. 291–293, 2015.

# List of Figures

# List of Tables

# Appendix A

# Project Management

## A.1 Project work description

Zürcher Hochschule
für Angewandte Wissenschaften

School of Engineering

# Programmierung eines Quantencomputers - Die Zukunft der Programmierung?

## PA20_stog_01

| | |
|---|---|
| BetreuerInnen: | Kurt Stockinger, stog<br>Rudolf Marcel Füchslin, furu |
| Fachgebiete: | Datenanalyse (DA)<br>Datenbanken (DB)<br>Software (SOW) |
| Studiengang: | IT / ST / WI |
| Zuordnung: | Institut für angewandte Informationstechnologie (InIT) |
| Interne Partner: | Institut für Angewandte Mathematik und Physik (IAMP) |
| Gruppengrösse: | 2 |

**Kurzbeschreibung:**
Quantencomputer nützen die Gesetze der Quantenphysik und sind für das Lösen bestimmter Probleme viel besser geeignet als klassische Computer. Im Gegensatz zu klassischen Computern, die mit Bits arbeiten, welche entweder eine 0 oder eine 1 darstellen, operieren Quantencomputer mit Quantenbits, sogenannten Qubits. Ein Qubit repräsentiert einen Mischzustand zwischen einer 0 und einer 1. Alle Operation auf Qubits laufen parallel. Somit sind Quantencomputer hochleistungsfähige Parallelrechner.

Quantum Computing hat in den letzten Jahren rasante Fortschritte erzielt. Neben dem kanadischen Pionier D-Wave Systems, gibt es auch Entwicklungen von Google, IBM und Rigetti Computing aus Berkeley, Kalifornien. Bisher war jedoch das Programmieren von Quantencomputern nur einer geringen Anzahl an Experten vorbehalten. Seit kurzem ist es jedoch für ein breiteres Publikum möglich, erste Quantencomputer in Python zu programmieren.

Ziel dieser Arbeit ist es, ein ausgewähltes Problem aus dem Bereich Datenbanken oder Machine Learning sowohl auf einem Quantensimulator als auch auf einem echten Quantencomputer zu implementieren. Im Speziellen soll der Grover Algorithmus für derartige Probleme untersucht werden.

**Voraussetzungen:**
Freude am Programmieren. Experimentieren mit "Cutting Edge Technology" und somit das Erforschen von Unbekanntem.

**Die Arbeit ist vereinbart mit:**
Tobias Fankhauser (fankhtob)
Marc Solèr (solerma1)

**Weiterführende Informationen:**
https://www.amazon.de/Practical-Quantum-Computing-Developers-Programming/dp/1484242173/ref=sr_1_2?adgrpid=66295626426&gclid=EAIaIQobChMIrdyF6tWe5QIVB-J3Ch020wwWEAAYASAAEgJ9MPD_BwE&hvadid=332677165850&hvdev=c&hvlocphy=1003297&hvnetw=g&hvpos=1t1&hvqmt=

## A.2 Meeting notes

**Kick-off 15.09.2020**

Aufgaben Tobias und Marc:

- Klären, ob neben IBM auch andere Anbieter Zugriff auf ihre Quantenrechner zulassen (Microsoft, Amazon, Google, ...). Schnittstelle sollte Qiskit sein.
- Anwendung des Grover-Algorithmus' für Datenbanken (auch Multidimensional, Bitmap Index) klären (BA-Arbeit A. Soultanis)
- Anwendung des Grover-Algorithmus' für SAT-Probleme (Orakel)
- GitHub-Repo für Code aus Vorbereitung und Tests (eingerichtet: https://github.zhaw.ch/QC-PA-2020/)
- Folien Seminar bezüglich No-cloning-Theorem prüfen und verstehen
- Frage auf nächste Woche: Wie sind Controlled-Operationen trotz No-cloning-Theorem möglich?

**Sitzung 22.09.2020**

Aufgaben:

- Rigetti Comp., IonQ für Account anfragen
- *Trummer und Koch 2016* lesen (auf Repository abgelegt)
- Beispiele für Grover / SAT programmieren. Z.B. aus *Soultanis 2019*
- Orakel mit logischer Bedingung implementieren (z.B. zahlentheoretisches Problem, Hashfunktion)

**Sitzung 29.09.2020**

Aufgaben:

- E-Mail mit Anfrage für vorläufiges Budget von $100 (O. Stern), und einem Academic Zugriff auf Amazon Braket (C. Marti)
- Prüfen, ob Optimierungsprobleme (à la MQO mit D-Wave) auf allgemeinen (Gate-basierten) QC umsetzbar sind
- Reduktion von allgemeinen Funktionen in SAT-ähnliche Probleme studieren (interessant: ist Reduktion in P möglich?)

**Sitzung 06.10.2020**

**Aufgaben:**

- Dokument ergänzen mit
  - allgemeiner Umwandlung von booleschen Ausdrücken in Quantenschaltkreis (praktisches Vorgehen)
  - Kurzer Einführung in Quantengatter
- Versuche mit **Amazon Braket**

**Sitzung 13.10.2020**

**Aufgaben:**

- Thema SAT
  - Korrektur der Indizes im Dokument
  - Beispiele (Code und Theoretisch) einfügen (teils aus bestehendem Dokument)
  - Algorithmus implementieren
  - Wenn fertig, Optimierung weiter untersuchen
- E-Mail an O. Stern bez. Englischem Dokument, Deutsche Zwischenbesprechungen und Präsentation
- *Gottlob et al.* [1] studieren
- Anwendungen des SAT Problems überlegen und programmieren (Logische Programmierung, [1])
- Versuche mit Amazon Braket

**Anhänge:**

Gottlob et al: Ontological Queries: Rewriting and Optimization

**Sitzung 20.10.2020**

**Aufgaben:**

- Thema SAT
  - Korrektur der Indizes im Dokument (übernommen von letzter Woche)
  - Code mit Kommentaren ergänzen
  - Anwendungen genauer und aus praktischer Sicht beschreiben (damit Vergleich / Kategorisierung möglich)
  - Tests mit Simulator und Quantumhardware
- Thema Query-Optimierung
  - Grundlagen (d.h. *VQE*, *QAOA*) studieren
  - Diese Optimierung mit *Grover* vergleichen
- Weitere Versuche mit Amazon Braket

**Sitzung 27.10.2020**

**Aufgaben:**

- Thema SAT / CSP
  - Anwendungen genauer und aus praktischer Sicht beschreiben (damit Vergleich / Kategorisierung möglich)
  - Tests mit Simulator und Quantumhardware
  - Beispiel mit Fehlerkorrektur umsetzen

- Thema (Query-)Optimierung
  - Konkretes Query-Optimierungsproblem formulieren
  - Naiven, klassischen Ansatz vorschlagen und einfache Verbesserung einsetzen
  - Umsetzung mit *VQE, QAOA* (*Trummer und Koch* hilfreich)

**Sitzung 10.11.2020**

**Aufgaben:**

- Thema SAT / CSP
  - Fehlerkorrektur umsetzen
  - QC-lösbare SAT Instanz finden
  - Anwendungsbeispiel
  - Ursachen für Fehler in QC analysieren
- Thema (Query-)Optimierung
  - Konkretes Query-Optimierungsproblem formulieren
  - Naiven, klassischen Ansatz vorschlagen und einfache Verbesserung einsetzen
  - Umsetzung mit *VQE, QAOA* (*Trummer und Koch* hilfreich)

**Sitzung 17.11.2020**

**Aufgaben:**

- Thema SAT / CSP
  - CSP-Solver mit generischem Algorithmus für *Rigetti* QC implementieren
  - *Amazon* Support wegen `None object` Problem anfragen
  - Anwendungsbeispiel für SAT / CSP beschreiben und lösen
  - Ursachen für Fehler in QC analysieren und beschreiben
- Thema (Query-)Optimierung
  - Zusammenhang Eigenvektoren / Matrizen und VQE und QAOA verstehen (Allenfalls BA der Vorgänger konsultieren)
  - Naiven, klassischen Ansatz vorschlagen und einfache Verbesserung einsetzen
  - Konkretes Query-Optimierungsproblem formulieren und lösen

**Sitzung 01.12.2020**

**Allgemein:**

- Thema SAT / CSP
  - Anwendungsbeispiel für SAT für Datenbanken
- Thema (Query-)Optimierung

  - Query-Optimierung: Naiven, klassischen Ansatz vorschlagen und einfache Verbesserung einsetzen
  - Query-Optimierung abschliessen (VQE und QAOA)

**Dokument**:

- Einführung
  - Nur nötige Gates und ihre **Anwendung** beschreiben
  - Kreuz $\oplus$ und Punkt beim CNOT Gate
  - T-Gate entfernen
  - Dagger † beschreiben
  - Weniger QC-Algorithmen, dafür genauer beschreiben
- CSP
  - Kommentar für Lösungen des CSP (Simulation, QC)
- Grover generisch
  - Bilder kommentieren (Resultate SAT Grover generisch)
  - SAT Grover: Zu Beginn SAT und Grover beschreiben und mit Code ausführen
  - Diskussion zur Anzahl Lösungen bei Grover
  - "Tractable" SAT Instanz zu Beginn
  - *Rigetti* Resultate gleichzeitig mit IBM
  - Error Correction kurz beschreiben (Code vorhanden)
- Optimierung
  - Behandelte Probleme eingangs einführen
- Kosmetik: Seitenrand für Code (zuletzt)

**Sitzung 08.12.2020**

**Allgemein:**

- Herleitung des Query-Optimierungsproblems (Tabellen, Joins etc.)

**Dokument**:

- Gemäss Sitzungsnotizen vom 01.12.2020
- Installationsanleitung zum Ausführen des Codes (inkl. Zugriffs-API für IBM / Amazon)
- Code auf Github bereitstellen