



Institute for Theoretical Physics
Leibniz University Hannover

Efficiently Preparing Supports of Boolean Functions in Uniform Superposition

Bachelor's Thesis

Moritz Marwede

10046700

Supervisor:

Prof. Dr. Tobias J. Osborne

June 10, 2025

Sometimes, when I think for a while, something happens.

– *Hazem Abo Hamed*

Abstract

In quantum computation it is often required to prepare uniform superpositions over specific states of the qubit register. Given a set of integer inputs, we introduce a classical algorithm that produces the layout of quantum circuits to generate such a uniform superposition over the binary interpretations of the input. We present two heuristics for a minimization of the required gates; one greedy, oriented towards local solutions, and the other one based on a solving algorithm for the travelling salesperson problem (TSP) from **Google OR-Tools**. Comparison shows that the greedy heuristic produces results nearly as good as the TSP-based solver. Subsequently, we analyse possibilities to create superpositions over the satisfying assignments of boolean functions. We show how non-uniform superpositions for functions in disjunctive normal form can be generated, though we leave open whether they can be adapted to efficiently create uniform superpositions. For clauses of n disjunctions of literals, as they appear in formulae in the conjunctive normal form, it is possible to efficiently realize uniform superpositions.

Acknowledgement

I would like to thank Lennart Binkowski for providing the idea for this thesis and for supporting me in the process of developing it. I am grateful to him for always taking the time to plan the next steps with me and for fostering a supportive atmosphere in which I could ask any questions freely.

Contents

1	Introduction	1
2	Preliminaries	4
2.1	Description of Qubits	4
2.2	Single-qubit Gates	6
2.3	Multi-qubit Gates	8
3	An Algorithm for Preparing Integer Lists in Uniform Superposition 12	
3.1	Core Concepts of the Algorithm	12
3.2	Shortest Hamiltonian Path Problem	19
3.3	Travelling Salesperson Problem	22
3.4	Efficiency Comparison of the TSP Solver	25
4	Preparing Uniform Superpositions Conditioned on Boolean Functions 32	
4.1	General Boolean Formulae	33
4.2	Disjunctive Normal Form	38
4.3	Conjunctive Normal Form	40
4.4	Clauses of n Disjunctions	42
5	Conclusion and Outlook	46
	List of Figures	48
	Bibliography	50

Introduction

In 1982, Richard Feynman published a paper called *Simulating Physics with Computers* [1]. He proposes the idea of a particular computer to model the laws of physics. Somewhat earlier, in 1980, Soviet mathematician Yuri Manin independently proposed a similar concept [2]. On the most fundamental level, all physical phenomena are governed by quantum mechanics. According to Feynman, it is therefore only consequential to construct a computer that simulates physical systems using the laws of quantum mechanics itself. Such a computer, he envisions, would be able to not only approximate physics, as classical computers do, but due to its fundamental quantum mechanical structure, this computer could behave exactly the same way as nature itself.

While simulating quantum mechanical processes can require an exponential overhead on classical computers, a quantum computer has the potential to perform such simulations far more efficiently. *Quantum advantage* refers to the ability of a quantum computer to solve certain problems that are infeasible on classical computers, to solve them more economically, or with better quality than any classical computer can. In the last years, several research groups managed to suggest quantum advantage of computational runtimes. One of the first widely accepted demonstrations was done in 2020 with the Chinese quantum computer **Jiuzhang** that is based on photons [3]. It provided a solution for a problem which had been mathematically proven to be infeasible on classical computers, with estimated runtimes on present-days supercomputers of half the earth's lifetime.

Years before that, quantum algorithms with provable speed-ups compared to the best known classical algorithms had been introduced. In 1985, David Deutsch proposed an oracle-based algorithm [4]. While not directly useful for practical applications, it was the first to demonstrate how quantum computation can exploit the *superposition* of quantum states to gain an advantage in solving black-box problems. Seven years later, Deutsch and Jozsa generalized this idea, introducing an algorithm that provides an exponential speed-up over any classical algorithm [5].

In the nineties, Peter Shor created algorithms for the prime factorization of integers and calculating discrete logarithms in polynomial time [6] and Lov Grover introduced a quantum search algorithm with quadratic speed-up [7]. While many theoretical concepts exist of how quantum computers could work and what algorithms they could run, today's greatest issue is still the physical construction of them. The smallest computational element in a quantum computer is the *qubit*, which can be realized through various physical systems. Possible implementations include trapped ions, superconducting circuits, photons and many others [8, 9, 10]. However, when qubits interact and are used to perform computations, we still face the significant challenge of *decoherence*. The loss of coherence of quantum states through interactions with the environment introduces errors in the computation. Therefore, the usable scale of quantum computers is still highly limited. The first presented working quantum computer (1998) consisted of only two qubits [11]. Present quantum computers are made up of up to several hundreds of qubits [12].

In the future, quantum computers are expected to employ more qubits while minimizing decoherence. This could lead to all sorts of novel applications and sectors. Possible problems span from material sciences and drug discovery to cryptography and optimization problems. For many problems, it is crucial to initialize the qubits in a certain quantum state. One can think of a general algorithm that prepares a state in uniform superposition over a specific set of values as a starting point. Then it evaluates the state, so that its computation involves all input values simultaneously. Exhaustive search methods could be sped up like this. In January 2025, Lennart Binkowski introduced the classical-quantum programming language **CQ** [13]. One of its features is the ability to initialize a uniform superposition by passing a boolean function. The superposition is created over the *support*, i.e. the set of all assignments that the function evaluates to true.

In this thesis, we investigate possibilities to efficiently prepare supports of boolean functions in uniform superposition. We will implement a classical algorithm in Python that is capable of handling integer lists as an input for the creation of efficient quantum circuits and consider theoretical approaches to use boolean functions as an input. The second chapter provides preliminaries like the quantum mechanical description of qubit states as vectors in the Hilbert space and how to perform calculations with them. Furthermore, specific single-qubit and multi-qubit gates are characterized mathematically and their effect on qubits is considered. We will need them later in the construction of quantum circuits. Also, we introduce a notation to visualize quantum circuits.

In Chapter 3, we introduce the algorithm in Python. Its input is a list of integer values which the algorithm interprets as binary number strings. Each bit of a string is allocated to one qubit and the algorithm produces a circuit that creates a uniform

superposition over all bit strings. After explaining the core concepts of the algorithm, we analyse two methods for a computational speed-up; one intuitive method and one that is based on a problem known from theoretical computer science, namely the *Shortest Hamiltonian Path Problem*. Both methods improve the algorithm by adapting the order of the input list. After that, we compare these speed-ups as well as the circuit sizes that the algorithm produces.

In the fourth chapter, we investigate ideas of quantum circuits to prepare uniform superpositions over the supports of boolean functions. The aim is to avoid the necessity of computing the satisfying assignments of a boolean function in advance. Instead, we examine the functions in search of general circuit construction-rules. As we will see, constructing such circuits is not straightforward. Therefore, we focus on specific classes of logical formulae. While we do not find efficient circuits for preparing uniform superpositions over the supports of general logic formulae, we present several special cases for which such circuits can indeed be constructed.

Preliminaries

The following chapter is based on the work of Nielsen and Chuang in *Quantum Computation and Quantum Information* [14].

2.1 Description of Qubits

In quantum computation and quantum information the carrier of information is no longer the classical bit with its two possible states, 0 and 1. Instead, we use the *quantum bit*, or *qubit*, for short. The difference to a classical bit is that, apart from the classical states 0 and 1, for the state of a qubit linear combinations of those are possible. They are often called *superpositions*. We will use the Dirac notation to describe qubit states as vectors of the two-dimensional Hilbert space \mathbb{C}^2 . A *Hilbert space* \mathcal{H} is a complete, real or complex, vector space equipped with an inner product. A qubit in a state $|\psi\rangle$ generally takes the form

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

with

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{and} \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Mathematically, $|0\rangle$ and $|1\rangle$ form (one possible) orthonormal basis for \mathbb{C}^2 and are called the *computational basis*¹. When we measure the qubit, however, we will not obtain its exact form. Rather, there will be a chance of $|\alpha|^2$ that we measure 0 and $|\beta|^2$ that we measure 1. It holds that

$$\langle\psi|\psi\rangle = |\alpha|^2 + |\beta|^2 = 1$$

¹This is just one way of defining the basis states. Another equivalent basis is $|\pm\rangle = \frac{1}{\sqrt{2}}(|0\rangle \pm |1\rangle)$. Here, the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ is no superposition. Thus, we see that what defines a superposition in one basis does not have to hold in another basis.

because the probabilities sum to one. $|\psi\rangle$ is therefore a unit vector. Now we can mathematically describe one single qubit. But for quantum computation, like simulating physical systems or solving optimisation problems, more qubits are required. Thus, the description of a single qubit does not suffice. Upon measurement, a system of two qubits can collapse into one of four possible basis states, since there are four combinations for assigning the values 0 and 1 to two (qu)bits. The computational basis for two qubits then looks like this:

$$|0\rangle \otimes |0\rangle, |0\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle, |1\rangle \otimes |1\rangle =: |00\rangle, |01\rangle, |10\rangle, |11\rangle$$

and, correspondingly, a two-qubit state takes the form

$$|\psi\rangle = \alpha_{00} |00\rangle + \alpha_{01} |01\rangle + \alpha_{10} |10\rangle + \alpha_{11} |11\rangle \quad \alpha_{ij} \in \mathbb{C}, \quad i, j \in \{0, 1\}$$

with $\sum \alpha_{ij} = 1$.

To describe a system of n qubits we need to consider its state space. If the n underlying single-qubit states are given by $|\psi_0\rangle$ to $|\psi_{n-1}\rangle$, the composed state $|\Psi\rangle$ is the tensor product of the single-qubit states

$$|\Psi\rangle = |\psi_{n-1}\rangle \otimes |\psi_{n-2}\rangle \otimes \dots \otimes |\psi_0\rangle.$$

Concatenating the states in this reverse order is called *big-endian* [15]. Those states of the composed system live on the product Hilbert space

$$\mathcal{H}^{\otimes n} = \mathcal{H} \otimes \dots \otimes \mathcal{H} \cong \mathbb{C}^{2^n}.$$

We use the notation $\mathbb{b} := \{0, 1\}$ for the classical bit states. Now, we prepare n states $|x_i\rangle$ individually in 0 or 1. For $x_i \in \mathbb{b}, i \in \{0, n-1\}$ the compound state is

$$|\Psi\rangle = |x_{n-1}\rangle \otimes \dots \otimes |x_0\rangle =: |x_{n-1}\dots x_0\rangle.$$

The state $|x_{n-1}\dots x_0\rangle$ is a sequence of zeros and ones that we call *bit string*. All possible bit strings of length n together define a n -qubit computational basis:

$$\{|x\rangle : x \in \mathbb{b}^n\}. \tag{2.1.1}$$

The overall state vector for multiple qubits is given by

$$|\Psi\rangle = \sum_{x \in \mathbb{b}^n} \alpha_x |x\rangle.$$

2.2 Single-qubit Gates

On both classical and quantum computers we want to manipulate the states of the (qu)bits so that they perform the desired calculations. For this purpose, on classical computers we have classical logic gates like AND, OR, NOT. They get one or more input signals and transform them into an output signal. The NOT-gate is the only non-trivial classical gate with just one input. This is not surprising, as the only operation that can be performed on a single bit restricted to the values 0 or 1 is negation. In contrast to that, there are many quantum gates which act on just one qubit non-trivially. Qubits can be in one of an infinite amount of states and therefore infinitely many single-qubit gates are imaginable. A quantum gate in general can be described by a unitary, linear operator $U \in \mathcal{L}(\mathcal{H})$. In fact, every unitary represents a physically possible gate. We have

$$U^\dagger U = I \quad \Leftrightarrow \quad U^\dagger = U^{-1},$$

so every unitary is invertible with the inverse being its Hermitian adjoint. In the context of quantum computing this means that every quantum gate is reversible. Classical gates do not have to be reversible², but it can be shown that an arbitrary classical circuit can be simulated by an equivalent reversible circuit. Let's take a look at some quantum gates of central importance.

A set of three essential single-qubit gates is given by the *Pauli matrices*

$$X := \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \quad Y := \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad Z := \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

with circuit representations

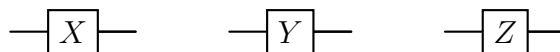


Figure 2.1: Single-qubit Pauli gates.

We will often use the X gate that acts on a single qubit as a bit flip (NOT gate on a classical computer).

$$X |0\rangle = |1\rangle, \quad X |1\rangle = |0\rangle.$$

²Consider, for example, the classical XOR gate with two inputs and one output. It produces 1 if exactly one of the inputs is 1 and results in 0 otherwise. From the output alone we have no chance to figure out in what state both inputs individually were.

Another gate we will often use is the *Hadamard* gate,

$$H := \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \quad \text{---} \boxed{H} \text{---}$$

Figure 2.2: Matrix and circuit representation of the single-qubit Hadamard gate.

Considering its action on the computational basis,

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

we can use H to gain a uniform superposition between 0 and 1 when starting in one of the computational basis states.

Based on the Pauli matrices we can create a new set of gates, called the *rotation operators* about the x , y , and z axes

$$\begin{aligned} R_x(\theta) &:= e^{-i\theta X/2} && \text{---} \boxed{R_x(\theta)} \text{---} \\ R_y(\theta) &:= e^{-i\theta Y/2} && \text{---} \boxed{R_y(\theta)} \text{---} \\ R_z(\theta) &:= e^{-i\theta Z/2} && \text{---} \boxed{R_z(\theta)} \text{---} \end{aligned}$$

Figure 2.3: Single-qubit rotation gates.

of which we are especially interested in the R_y gate. To gain a more useful form we can rewrite a general exponential function of the form e^{ixA} , with matrix A , $A^2 = I$ and $x \in \mathbb{R}$, as a power series and derive a matrix representation:

$$\begin{aligned} e^{ixA} &= \sum_{k=0}^{\infty} \frac{(ixA)^k}{k!} = I + ixA - \frac{x^2 I}{2} - \frac{ix^3 IA}{6} + \frac{x^4 II}{24} + \dots \\ &= I + ixA - \frac{x^2}{2} I - i \frac{x^3}{6} A + \frac{x^4}{24} I + \dots \\ &= \sum_{k=0}^{\infty} \left((-1)^k \frac{x^{2k}}{(2k)!} I + i(-1)^k \frac{x^{2k+1}}{(2k+1)!} A \right) \\ &= \cos(x)I + i \sin(x)A. \end{aligned}$$

With that, we easily see that

$$R_x(\theta) = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} X = \begin{bmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix} \quad (2.2.1)$$

$$R_y(\theta) = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} Y = \begin{bmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{bmatrix} \quad (2.2.2)$$

$$R_z(\theta) = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} Z = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix}. \quad (2.2.3)$$

R_y acts on the computational basis as

$$\begin{aligned} R_y(\theta) |0\rangle &= \cos \frac{\theta}{2} |0\rangle + \sin \frac{\theta}{2} |1\rangle \\ R_y(\theta) |1\rangle &= -\sin \frac{\theta}{2} |0\rangle + \cos \frac{\theta}{2} |1\rangle. \end{aligned}$$

Comparing this to the Hadamard gate, we notice a quite similar effect of creating a superposition between $|0\rangle$ and $|1\rangle$, only that R_y modulates the weighting of the amplitudes. This aspect will play a crucial role in the algorithm developed later. So, why won't we need the R_x or R_z gate as well? Our goal will be to modify the amplitudes of qubits, and in that regard, it would only complicate things to have imaginary coefficients, like those introduced by the R_x gate. R_z won't help us at all since it only applies a relative phase between the $|0\rangle$ and $|1\rangle$ components of a qubit and therefore does not create a superposition out of a computational basis state.

2.3 Multi-qubit Gates

In order to enable interactions and entanglement between qubits, we now introduce multi-qubit gates. One of the simplest two-qubit gates is the *CNOT* (Controlled NOT) gate

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = |0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes X$$

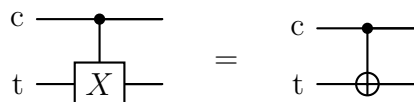


Figure 2.4: Two equivalent circuit representations of the *CNOT* gate

that flips the target qubit t on the condition that the control qubit c is $|1\rangle$. If the control qubit is $|0\rangle$, then nothing happens.

$$\begin{aligned} CNOT |00\rangle &= |00\rangle & CNOT |01\rangle &= |01\rangle \\ CNOT |10\rangle &= |11\rangle & CNOT |11\rangle &= |10\rangle \end{aligned}$$

Beyond that we are interested in the construction of an arbitrary controlled unitary gate, that allows us to perform any unitary operation on a target qubit conditioned on a control qubit being set to $|1\rangle$.

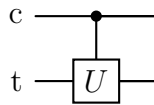


Figure 2.5: Controlled unitary gate.

Theorem 2.1. Let U be a single-qubit unitary gate. Then there exist single-qubit unitary operators A, B, C that satisfy $ABC = I$ such that U can be written in the form $U = e^{i\alpha}AXBXC$, where α is an overall phase factor.

We will use Theorem 2.1 to construct the controlled- U operator. First, we need to apply a phase of $e^{i\alpha}$ on the target qubit if the control is set.

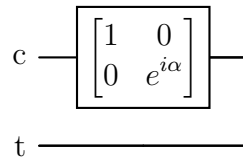


Figure 2.6: Controlled phase shift gate.

Since the two qubits compose a global state, it does not matter on which qubit the phase is applied. Therefore, the circuit in Fig. 2.6 is sufficient to perform a phase shift on the target qubit if and only if the control is set. By replacing the X gates with $CNOT$ gates, we achieve the circuit in Fig. 2.7. Let us examine its behaviour. Let the state of t be $|\phi\rangle$. First case, c is $|1\rangle$. This leads to $|\phi\rangle \rightarrow e^{i\alpha}AXBXC|\phi\rangle = U|\phi\rangle$. Second case, c is $|0\rangle$ and therefore the $CNOT$ gates have no effect, while also no phase is applied. Then $|\phi\rangle \rightarrow ABC|\phi\rangle = |\phi\rangle$. So, we have effectively obtained a controlled- U gate.

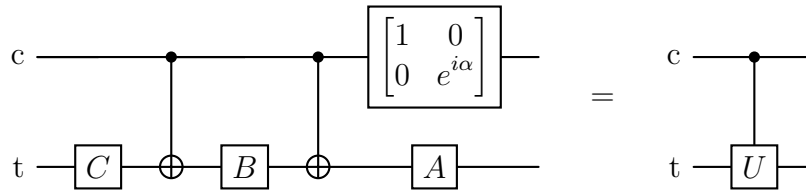


Figure 2.7: Implementation of the controlled U gate, with arbitrary unitary single-qubit operator U and α, A, B, C such that $U = e^{i\alpha}AXBXC$ and $ABC = I$.

Having explored how an arbitrary operator can be controlled by a single qubit, we now turn to the question of how to control an operator using multiple qubits. Suppose U is a single-qubit operator and V is another operator satisfying $V^2 = U$. We then can implement the circuit in Fig. 2.8 that performs the operation U on target qubit t exactly when both c_0 and c_1 are set.

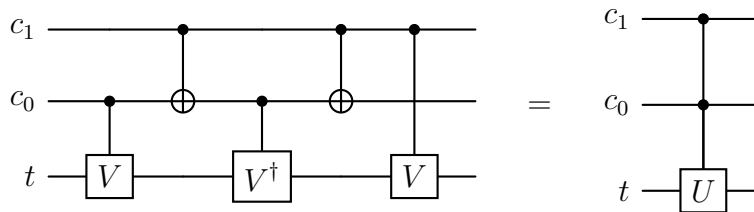


Figure 2.8: Implementation of unitary U controlled on two control qubits, with unitary V such that $V^2 = U$.

By choosing $U = X$ in Fig. 2.8, we obtain the so-called *Toffoli gate*, which serves as a fundamental building block for constructing single-qubit gates controlled by an arbitrary number of qubits. We denote the conditioning of U on n control qubits as $C_n(U)$. Fig. 2.9 shows how to construct $C_n(U)$ with Toffoli gates and a register of $n - 1$ ancilla qubits a_0, \dots, a_n , initialised in $|0\rangle$. The Toffoli gates create a stepwise descent with the help of the ancilla register, leading to a_0 only being $|1\rangle$ if all controls are in $|1\rangle$. Then, U is applied (or not) and the ancilla register is returned to its original state of $|0\rangle^{\otimes(n-1)}$.

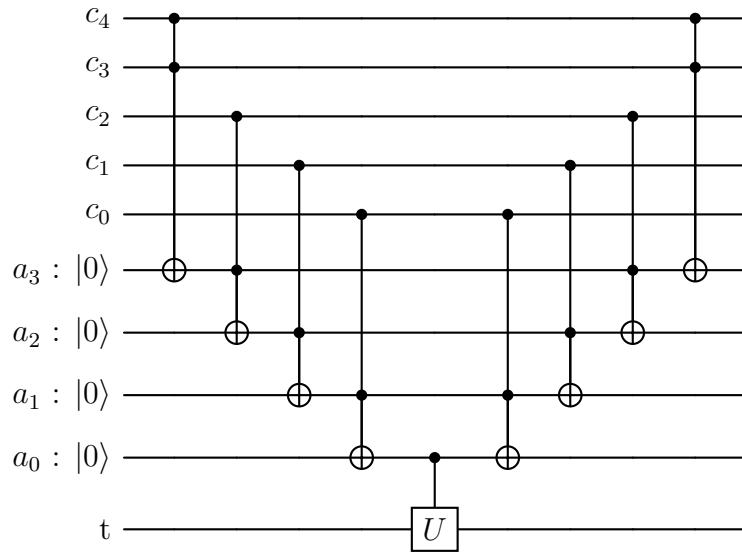


Figure 2.9: Implementation of unitary U conditioned on n control qubits ($C_n(U)$). Here, for the case $n = 5$. [14, see Figure 4.10, p. 184].

To change the condition for any conditioned operation from qubits being set to qubits not being set, we simply have to invert the value of the control and afterwards turn it back into its original using X gates (Fig. 2.10).

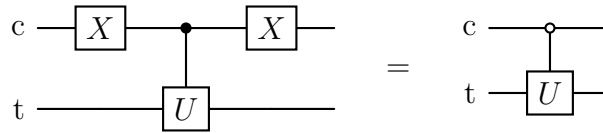


Figure 2.10: Controlled single-qubit unitary U , conditioned on the first qubit being set to zero.

An Algorithm for Preparing Integer Lists in Uniform Superposition

The main goal of this chapter is to introduce an algorithm capable of producing a quantum circuit that prepares a uniform superposition over a register of qubits for a given set of values. That is, we give a set of decimal numbers, which are then transformed into binary numbers, and after the quantum circuit is applied and we measure the global state, the output is again one of our binary input numbers, all with equal likelihood. Later, we will adapt the order in which the input values are processed in order to minimize the circuit size. Some code snippets are discussed throughout this chapter. The complete Python implementation developed for this thesis is available in the accompanying GitHub repository [16].

3.1 Core Concepts of the Algorithm

To start, we consider an input of n decimal numbers x_0 to x_{n-1} . In a first step, they are translated into binary numbers b_0 to b_{n-1} . For any binary number b_i we can insert any number of zeros up front without changing its value. So, we must determine how many (qu)bits are needed to describe every value unambiguously in binary. That is done based on how many (qu)bits are needed to describe the largest decimal input number in binary. If all input values are non-negative, we use the standard binary system called *base-2 numeral system*. It is also possible to represent negative input values. For that matter we make use of the *two's complement*, which works like this: let t be a binary representation of any (possibly negative) decimal number in the form of a k -bit string $t = t_{k-1}...t_0$ with $t_i \in \{0, 1\}$ for $i = 0, \dots, k-1$. The last bit t_0 represents a 1 in decimal, the second to last represents a 2, then 4 and so on. So far nothing special here. The difference comes with the leading bit, whose decimal equivalent now has a negative sign. Written in an equation, the decimal interpretation d of the bit

string t in two's complement is

$$d = -t_{k-1} \cdot 2^{k-1} + \sum_{i=0}^{k-2} t_i \cdot 2^i. \quad (3.1.1)$$

A sequence of n bits typically describes numbers from 0 to $2^n - 1$. In two's complement format, n bits can describe the numbers -2^{n-1} to $2^{n-1} - 1$. Thus, to find out how many qubits are needed simply for number representation, we take the amount needed to describe the highest absolute number in the base-2 numeral system. If negative numbers are also involved, we add one qubit and switch to the two's complement system. m denotes the number of qubits required to typify every input as a binary. Using this convention, we translate our input values x_0 to x_{n-1} into binary numbers whereby we only care about those bit strings and not their interpretation starting from now. This leaves us with a list B of bit strings

$$B = (b_0, \dots, b_{n-1}), \quad \text{where } b_i = (b_{i,m-1}, \dots, b_{i,0}) \quad \text{with } b_{i,j} \in \{0, 1\}$$

$$\text{for } i = 0, \dots, n-1, \quad j = 0, \dots, m-1.$$

At this point we start building the circuit. First of all, we need the register of qubits, which is split into two parts. The main quantum register consists of m qubits q_0, \dots, q_{m-1} capable of depicting each (binary) input value. We denote the state of the main register with φ . On top of that, we need $n-1$ ancilla qubits a_0, \dots, a_{n-2} , whose purpose will soon become clear. For the first binary number b_0 , we perform X operators on those qubits of the main register that correspond to the bits of b_0 that are 1. The input decimal number 13 for example is 1101 in binary, corresponding to qubits q_0, q_2 and q_3 , onto which we would therefore act with X gates. The quantum register now is in the state $\varphi = |b_0\rangle$. The next step is to apply a controlled rotation gate $CR_y(\theta_0)$, Eq. (2.2.2), on the first ancilla qubit a_0 that is conditioned on the quantum register being in the state $|b_0\rangle$ ¹. After that we compare b_0 that we have already implemented with b_1 . For every bit in which they differ we apply a CNOT gate on the corresponding qubit, conditioned by a_0 . If we were to now measure the overall state, a_0 either would still be in $|0\rangle$ and measurement of the main register would yield $|b_0\rangle$, or a_0 would have been rotated to $|1\rangle$ and the main register would be in the state $\varphi = |b_1\rangle$. In an analogous manner of how we implemented b_1 we implement all remaining b_i , $i = 2, \dots, n-1$ by first applying a $CR_y(\theta_{i-1})$ gate on a_{i-1} , conditioned on the main register being in state $|b_{i-1}\rangle$ and then applying CNOT gates onto all bits that differ between b_i and b_{i-1} . As soon as one a_i remains in $|0\rangle$ under operation $R_y(\theta_i)$, the main register remains the same and therefore no other rotation gates are activated. This is what the conditioning of the rotation gates is for. Lastly, we have a look at the angles $\theta_0, \dots, \theta_{n-2}$. Let

$$\theta_i = 2 \cdot \arccos \frac{1}{\sqrt{n-i}} \quad \text{for } i = 0, \dots, n-2, \quad (3.1.2)$$

¹The condition is not necessary in this first step, we just use it for harmonisation reasons.

then,

$$R_y(\theta_i) |0\rangle = \cos \frac{\theta_i}{2} |0\rangle + \sin \frac{\theta_i}{2} |1\rangle = \frac{1}{\sqrt{n-i}} |0\rangle + \sqrt{1 - \frac{1}{n-i}} |1\rangle.$$

With probability $|\frac{1}{\sqrt{n-0}}|^2 = \frac{1}{n}$, a_0 is in state $|0\rangle$ and hence $\varphi = |b_0\rangle$. If $|a_0\rangle = |1\rangle$, then,

$$|a_1\rangle = \sqrt{1 - \frac{1}{n}} \cdot \frac{1}{\sqrt{n-1}} |0\rangle + \sqrt{1 - \frac{1}{n}} \cdot \sqrt{1 - \frac{1}{n-1}} |1\rangle.$$

So, we have $|a_1 a_0\rangle = |01\rangle$ with probability

$$\left| \sqrt{1 - \frac{1}{n}} \cdot \frac{1}{\sqrt{n-1}} \right|^2 = \sqrt{\frac{n-1}{n} \cdot \frac{1}{n-1}}^2 = \frac{1}{n}$$

and therefore $\varphi = |b_1\rangle$. In fact, it is easy to check that we will measure every possible state with probability $\frac{1}{n}$ and thus have created a unitary superposition over all desired states $|b_0\rangle, \dots, |b_{n-1}\rangle$. The overall final state is

$$\varphi = \frac{1}{\sqrt{n}} (|b_0\rangle |0\dots 0\rangle + |b_1\rangle |0\dots 01\rangle + \dots + |b_{n-2}\rangle |01\dots 1\rangle + |b_{n-1}\rangle |1\dots 1\rangle),$$

where the second register in each term corresponds to the ancilla qubits. One example illustrating how the algorithm operates is given in Fig. 3.1.

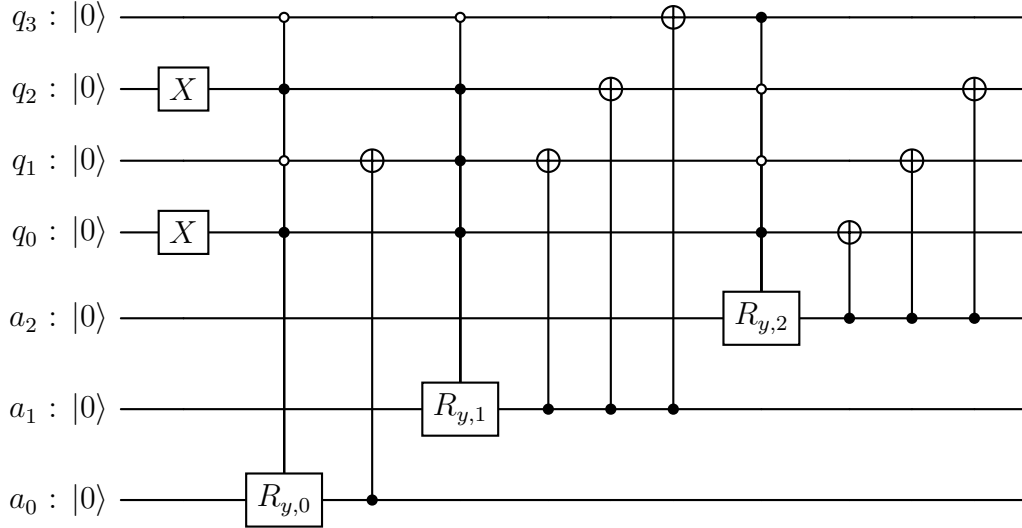


Figure 3.1: Example of the algorithm creating a circuit for decimal input [5, 7, 9, 14]. The ancilla register consists of qubits a_0 to a_2 and the main register consists of qubits q_0 to q_3 . The final state is $\varphi = \frac{1}{\sqrt{4}}(|0101\rangle |000\rangle + |0111\rangle |001\rangle + |1001\rangle |011\rangle + |1110\rangle |111\rangle)$. This result can be interpreted as a superposition between decimals [5, 7, 9, 14]. $R_{y,i} = R_y(\theta_i)$ for $i = 0, 1, 2$ and $n = 4$.

We introduce the *gate count* and *cycle count*, i.e. *circuit depth*, that measure how many gates and cycles are involved in a circuit. A *cycle* is a layer of one or more gates performed in parallel. An example would be two single-qubit gates acting on different qubits, so they can operate simultaneously. The circuit in Fig. 3.1, for example, consists of 12 gates and 11 cycles.

The algorithm described above is the one implemented in the actual code. It is also possible to reduce the number of ancilla qubits from $n - 1$ to just one, at the expense of an increased number of gates and cycles. The modified procedure works as follows: the first gates remain unchanged up to the second rotation gate. At that point, using the same control conditions as for the original second rotation gate, a multi-controlled X gate is applied to the single ancilla qubit. This is immediately followed by the controlled rotation gate, also acting on the same ancilla. The following $CNOT$ gates are naturally also controlled by this ancilla. In the same manner, all subsequent ancilla rotations are replaced by operations on this single ancilla qubit, each preceded by a multi-controlled X gate that matches the original control structure. As a result, the total number of gates and cycles both increase by $n - 2$, corresponding to the additional multi-controlled X gates introduced in this construction. The same example we saw in Fig. 3.1, now implemented with a single ancilla but a greater number of gates, is shown in Fig. 3.2.

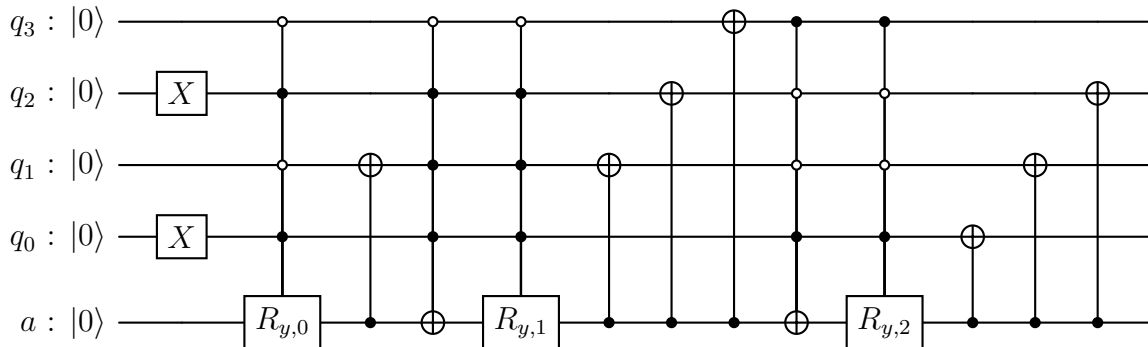


Figure 3.2: Example of the qubit saving alternative algorithm creating a circuit for decimal input [5, 7, 9, 14]. The ancilla register consists of qubit a and the main register consists of qubits q_0 to q_3 . The final state is $\varphi = \frac{1}{\sqrt{4}}(|0101\rangle |0\rangle + |0111\rangle |0\rangle + |1001\rangle |0\rangle + |1110\rangle |1\rangle)$. This result can be interpreted as a superposition between decimals [5, 7, 9, 14]. $R_{y,i} = R_y(\theta_i)$ for $i = 0, 1, 2$ and $n = 4$.

On a real-world quantum computer, one would likely use the version of the circuit that consumes fewer qubits, as qubit resources are highly limited in current setups. Both circuit versions can easily be converted into one another. In this work, we use the

gate-efficient version. Since the number of qubits and gates in the two constructions differ only by a linear factor, the results we obtain are applicable to both versions.

The implementation of the algorithm in Python is given in Fig. 3.5. We use the libraries from *Qiskit* (Quantum Information Software Kit), a software stack developed by **IBM Research** [17]. They allow us to create quantum circuits and simulate quantum computing on a classical computer. First, we introduce two auxiliary functions, which can be seen in Fig. 3.3 and Fig. 3.4.

```
1 def class_bit_flips(binary1, binary2):
2 # takes two binary strings, returns classical bit flips between
3     bit_flips = []
4     for i in range(number_of_qubits):
5         if(binary1[-(i+1)] != binary2[-(i+1)]):
6             bit_flips.append(i)
7     return bit_flips
8 # returns list of pos that need to be flipped, starting with pos 0
```

Figure 3.3: Implementation of a function to find the classical bit flips between two binary strings. **number_of_qubits** is a previously defined variable and is equal to the length of both binary input numbers for this function. In a for-loop, the characters of both bit strings are compared and every position where they differ, starting with position 0, is stored in a list **bit_flips**. This list is then returned.

```

1 def statevector_interpreter(statevector):
2     for i in range(len(statevector)):
3         if(statevector[i].real > 0.0001):
4             # only real part needed, we never have an imaginary part.
5             # if too small, it is just an error
6             result = i%(2**amount_of_qubits)
7             # i in overall state (incl. ancilla register)
8             result = dec2bin(result, 0)
9             result = bin2dec(result, signed)
10            # this way, if we have no signed numbers, they are
11            # converted into binary and back with no consequence
12            # but if should have a sign, sign is regained this way
13            print("State ", result, " with probability of ",
14                  round(statevector[i].real**2*100, 1), "%")

```

Figure 3.4: Implementation of an interpreter for state vectors. State vectors produced in **superpos** (Fig. 3.5) have the form $[a_0, \dots, a_k]$ with $k = 2^{\#qubits} - 1$ such that $\varphi = a_0 \cdot |0\dots0\rangle|0\dots0\rangle + \dots + a_k \cdot |1\dots1\rangle|1\dots1\rangle$. Most a_i are zero (or nearly zero due to computation errors), so we filter them out. Line 6 interprets the position as a decimal number by reducing i modulo $2^{\#qubits}$ to a **result**. The following steps make sure that a possible sign (stored in **signed**) is regained in the interpretation of the binary number as a decimal. The function **dec2bin** turns a decimal number into a binary number, given the **amount_of_qubits**, and **bin2dec** does the reverse. They either use the base-2 numeral system or, if **signed=True**, the two's complement. We will not go into detail regarding their construction, their implementation can be found in [16].

```

1 from qiskit import QuantumCircuit
2
3 def superpos(binlist):
4     # finds out gates to create superposition between elements of list,
5     # takes list of binary numbers
6
7     qc = QuantumCircuit(number_of_qubits+len(binlist)-1)
8     for flip in class_bit_flips(number_of_qubits*'0', binlist[0]):
9         qc.x(flip)
10
11    for list_pos in range(len(binlist)-1):
12        angle = 2*np.arccos(1/np.sqrt(len(binlist)-list_pos))
13        controlled_ry = RYGate(angle).control(
14            number_of_qubits, None, binlist[list_pos]
15        )
16        # control for the Ry gate is that the main register is
17        # in state binlist[list_pos]
18        qc.append(
19            controlled_ry, [i for i in range(number_of_qubits)]
20                + [number_of_qubits + list_pos]
21        )
22        for flip in class_bit_flips(binlist[list_pos],
23                                    binlist[list_pos+1]):
24            qc.cx(number_of_qubits + list_pos, flip)
25
26    print(qc)
27
28    statevector = Statevector(qc)
29    return statevector

```

Figure 3.5: Implementation of `superpos` with qiskit. First, a `QuantumCircuit` is generated, the argument passes the amount of qubits. All qubits are initialized in $|0\rangle$. Then, in lines 8-9, we use `class_bit_flips` (Fig. 3.3) to apply X gates so that the first value is created by the circuit. Lines 11-24 provide multi-controlled R_y gates on the ancilla register and $CNOT$ gates on the main register. The angle for R_y is given by Eq. (3.1.2) and its control is the state $|b_i\rangle$ on the main register, that is, `binlist[list_pos]`. Then, for each bit flip that is needed to convert b_i into b_{i+1} a $CNOT$ gate is applied, controlled on the ancilla we just used. Afterwards, we print the circuit and return its state vector. The state vector can then be interpreted by `statevector_interpreter` (Fig. 3.4) as a superposition of decimal numbers.

It is in our interest to minimize the gates and cycles for the implementation on an actual quantum computer for several reasons. First of all, a reduced number of cycles leads to shorter execution times, assuming the gate set remains unchanged or is diminished. Along with that comes an important criterion when working with quantum computers—

reducing decoherence. Qubits are fragile and over time they lose their quantum state due to decoherence. Shallower circuits with shorter computation times counteract this [18].

Hence, we aim to change the order of the input list so that the same algorithm **superpos** produces a circuit with fewer gates and cycles. To understand how to improve the circuit we must gain a more abstract view, which will be done in the following section.

3.2 Shortest Hamiltonian Path Problem

First, we introduce the *Hamming distance* that we will need soon.

Definition 3.1. The *Hamming distance* $d_H(a, b)$ between two strings a and b of the same length is the number of positions at which the elements in both strings differ.

In Python, we define the Hamming distance for bit strings:

```

1 def hamming_distance(bin1, bin2):
2     # takes two binary numbers represented as strings
3     bitflips = 0
4     for i in range(amount_of_qubits):
5         if(bin1[-(i+1)] != bin2[-(i+1)]):
6             bitflips += 1
7     return bitflips

```

Figure 3.6: Implementation of the Hamming distance d_H . Between two binary numbers, represented as strings, the Hamming distance is equal to the amount of positions at which both strings differ. **amount_of_qubits** is defined beforehand as the amount of qubits needed in the main qubit register as described in Section 3.1.

Every element of B needs to be processed exactly once. The order of the elements is arbitrary, but the various permutations differ in the number of gates (and cycles) needed. To create the superposition we need one R_y gate between every two successive elements b_{i-1} and b_i for $i = 1, \dots, n - 1$, resulting in a constant of $n - 1$ gates. Additionally, each element b_i requires the amount of *CNOT* gates equivalent to the Hamming distance (Definition 3.1) between b_{i-1} and b_i . For b_0 the Hamming distance is that between the

zero string (a string consisting of m zeros) and itself and its qubit representation can be executed by X gates. This means that the gate count increases linearly with the summed Hamming distances for each pair (b_{i-1}, b_i) for $i = 0, \dots, n - 1$, where $b_{-1} = 0^m$. The problem at hand is to find the order of the values that minimizes the added up Hamming distances between each two successive values. It is possible to approximate the ideal order using a *greedy algorithm* that follows the heuristic of making the locally optimal choice: at each step, it selects the next element by minimizing the Hamming distance to the previous one. The Python implementation of such a greedy algorithm is shown in Fig. 3.7.

```

1 def greedy_smallest_bit_flip(binlist):
2     most_ones = 0
3     candidates = highest_hamming_to_all(binlist)
4     first = 0
5     starting_zero = '0'*len(binlist[0])
6     for i in candidates:      # "worst" value is wanted
7         current_ones = hamming_distance(i, starting_zero)
8         if current_ones > most_ones:
9             most_ones = current_ones
10            first = i
11
12    greedy_list = np.array([]) # improved sequence "greedy"
13    greedy_list = np.append(greedy_list, first)
14    for i in range(len(binlist)):
15        if binlist[i] == first:
16            binlist = np.delete(binlist, i)
17            # removing first used item from input list
18
19    while(len(binlist) > 0):
20        least_bit_flips = amount_of_qubits # searches smallest Ham.
21        next_pos = 0
22        for pos in range(len(binlist)):
23            bit_flips = hamming_distance(
24                greedy_list[-1], binlist[pos])
25            if(bit_flips < least_bit_flips):
26                next_pos = pos      # position of next element
27                least_bit_flips = bit_flips
28            greedy_list = np.append(greedy_list, binlist[next_pos])
29            binlist = np.delete(binlist, next_pos) # delete used elem.
30
31    return greedy_list

```

Figure 3.7: Implementation of a greedy sequence-finding algorithm. The separately defined function **highest_hamming_to_all** takes a list of binary numbers and returns a list of value(s) with a highest minimal Hamming distance h to all other values - such that every other value has a minimal Hamming distance of at most $h - 1$ to the rest. (For the implementation of **highest_hamming_to_all**, see [16]). First, we generate a list of all **candidates** that have the aforementioned highest minimal Hamming distance. From these, we select the one with the greatest Hamming distance to the zero string and designate it as the first element in the improved sequence. This selected value has both a high minimal Hamming distance to all other values and a high Hamming distance to the zero string. Placing it first is reasonable because its distance from zero can be realized entirely using parallelized X gates and it has only one neighbour that requires a relatively large number of gate cycles for the transition. Everywhere else it would have two neighbours with high bit differences. After setting the first position, the algorithm proceeds greedily: at each step, it selects from the remaining elements the one with the smallest Hamming distance to the previously chosen value, continuing until all elements have been placed.

Although the greedy algorithm minimizes the Hamming distance at each step, it may fail to find the globally optimal solution, as the global optimum does not necessarily consist of locally optimal choices. The goal is to minimize the total Hamming distance between all successive elements in the list. Finding the optimal ordering that achieves minimal gate and cycle counts can be formulated as a *Shortest Hamiltonian Path Problem* (SHPP). This is a problem known from graph theory and it is a variation of the *Hamiltonian Path Problem* (HPP). Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a Hamiltonian path is a path that visits each vertex exactly once and the HPP is the problem of deciding the existence of such a path. The SHPP extends the HPP by adding an optimization objective: to find the **shortest** possible Hamiltonian path. To apply the SHPP to our situation, we can create a directed graph G with the elements of B being the vertices. Between every two vertices we draw one directed edge in both directions. The weight is the Hamming distance between these two vertices. Also, we add the zero string as a starting vertex. We insert edges from zero to all b_i with weights equal to the Hamming distances between them, but no edges that end in zero. This leads us to a formal definition for the graph:

$$G = (V, E), \quad \text{where } V = \{0^m, b_0, \dots, b_{n-1}\},$$

$$\text{and } E = \{(a_i, b_j, w_{ij}) \mid a_i \in V, b_j \in V \setminus \{0^m\}, a_i \neq b_j, w_{ij} = d_H(a_i, b_j)\}$$

To find the smallest gate count in our quantum circuit is therefore equivalent to finding the shortest Hamiltonian path in G , meaning a path that visits every vertex exactly once while minimizing the cost of that travel. G is nearly *complete* (every vertex is connected to every other vertex), just the edges back to zero are missing. Due to this situation the existence of a Hamiltonian path is certain. We observe that, to provide a Hamiltonian path, such a path has to start in zero.

3.3 Travelling Salesperson Problem

In practice, the SHPP is often reformulated as a *Travelling Salesperson Problem* (TSP), for which a broader range of solving algorithms exists. The core idea of the TSP is closely related to the SHPP: while the SHPP seeks a path that visits each vertex exactly once, the TSP additionally requires the path to return to the starting vertex, forming a closed tour. The classical motivation for this problem is a travelling salesperson who must visit a list of cities exactly once and return home via the shortest possible route. Such a minimal-weight round trip in a graph is referred to as a *Hamiltonian cycle*.

The TSP is an *NP-hard* problem [19], so it is unlikely that any polynomial-time algorithms exist that solve it. Nevertheless, it is a well studied problem in combinatorial

optimization due to its manifold applications, and modern solvers are able to solve the TSP for hundreds or even thousands of instances completely. Typically, the input graph is expected to be complete, ensuring the existence of a Hamiltonian cycle and simplifying the computational process.

Transforming the SHPP into a TSP is straightforward: after completing the Hamiltonian path, we simply require the path to return to the starting vertex, 0^m , thereby turning the path into a cycle. To ensure that this return step does not affect the optimality of the original path, we insert directed edges from every vertex b_i back to 0^m with zero weight. The resulting graph is complete and directed, with different distances between some vertices depending on the direction. We have therefore created an asymmetric TSP, doubling the possible solutions compared to a symmetric TSP. It is possible to adapt the graph so that solving the TSP yields a cycle count improved solution. In order to do so, we need to set the weight from zero to every other vertex b_i to 1, because it takes only one cycle for the parallel executable X gates that create the first number, independent of that first number. The rest of the edges remain unchanged, since the Hamming distance between two binary numbers is equivalent to the amount of controlled X gates and, hence, the amount of cycles. It is not possible to parallelize any of the gates apart from the X gates of the first cycle. The R_y gates are each controlled on the whole main register, so they cannot be parallelized, leaving aside that their controls are different in different parts of the circuit. And the $CNOT$ gates between two R_y gates, acting on the main register, all have the same control ancilla, so they can not be parallelized either.

Google's open-source software suite **Google OR-Tools** provides an algorithm to efficiently solve the TSP for applications like routing problems with one or more vehicles [20]. The algorithm works on a complete graph that is implemented via a matrix, symmetric as well as asymmetric. The matrix element at position (i, j) corresponds to the distance from vertex i to vertex j . We implement the *data model*, that contains the distance matrix, for the TSP solver from OR-Tools in Python (see Fig. 3.8).

```

1 def create_data_model(binlist, min_cycles):
2 # takes binary list, boolean min_cycles
3     data = {}
4
5     data["distance_matrix"] = []
6     for i in binlist:
7         row = []
8         for j in binlist:
9             row.append(hamming_distance(i,j))
10            data["distance_matrix"].append(row)
11
12    for i in range(len(binlist)):
13        data["distance_matrix"][i][0] = 0
14
15    data["num_vehicles"] = 1
16    data["depot"] = 0
17
18    if min_cycles:
19        for i in range(1, len(binlist)):
20            data["distance_matrix"][0][i] = 1
21    return data

```

Figure 3.8: Implementation of the data model for the TSP solver from OR-Tools. The function takes a list **binlist** containing the binary strings that represent the input values over which a superposition is to be created. The first value is the zero string, which also receives a boolean **min_cycles**. The data model consists of a register "distance_matrix", "num_vehicles" and "depot". For every element in **binlist** a row is appended to the distance matrix containing the Hamming distances between that element and every other element. Then, the matrix entries for the edges from i to 0 are set to zero, as discussed above. With **data["num_vehicles"]** OR-Tools provides the option of multiple vehicles in the context of logistic problems. In our case it does not make any sense to include more than one "vehicle", because that would imply that the circuit could perform multiple operations on the same qubits at the same time. **data["depot"]** sets the starting node to position 0, i.e. the position of the zero string. Finally, the boolean **min_cycles** decides if the algorithm to come afterwards should focus on reducing the cycle count (**min_cycles = True**) or gate count (**min_cycles = False**). If **True**, the distance matrix is adapted: the weight of edges $(0,i)$ for all i are set to 1, because the implementation of the first number, no matter which is the first, costs only one cycle due to its execution through all-parallel X gates.

A construction guide for the whole TSP solver with OR-Tools' Python library is given in [21]. Apart from the distance matrix that we already developed, we mainly follow the instructions from OR-Tools. While the guide proposes a **main** method that contains the **distance_callback** function, we define **distance_callback** globally

and rename the `main` to `tsp_solver` (see Fig. 3.9). In `tsp_solver`, the boolean `min_cycles` that we use for setting the focus on either gate or cycle minimization is passed for the first time. Inside this function, `create_data_model` is addressed with the arguments `binlist` and `min_cycles`. OR-Tools offers different tools to adapt the search and optimize the results of its TSP solver ². Between the different search strategies, `guided_local_search` produces the best results in our implementation. It enables the solver to escape a local minimum, i.e. a solution for the TSP that has the lowest cost locally, but possibly not globally. After moving away from the local minimum, the solver continues the search. We implement this search strategy in a copy of our `tsp_solver` that we name `tsp_solver_guided_local_search` (see details in the code [16]). Here, only the search strategy is changed and the function receives another variable `duration` that customizes how long the solver should search for a solution.

```
1 def create_data_model(binlist, min_cycles):...
```

```
1 def print_solution(manager, routing, solution):...
```

```
1 def distance_callback(from_index, to_index):...
```

```
1 def tsp_solver(binlist, min_cycles):...
```

```
1 def tsp_solver_guided_local_search(binlist, min_cycles, duration):
```

Figure 3.9: Implementation of the TSP solver from OR-Tools. Function `create_data_model` as implemented in Fig. 3.8. The main method to find the solution is either `tsp_solver` or `tsp_solver_guided_local_search`. Both access the other three functions.

3.4 Efficiency Comparison of the TSP Solver

The TSP solvers find improved orders for the values in the input list and could theoretically pass the enhanced order to the `superpos` function to actually create and print the layout of the circuit. But, this will no longer be necessary, since we will be dealing with instances with hundreds or thousands of input values. The final circuits consist of a number of gates which is a multiple of the amount of instances. At this scale, we hardly gain any information from the circuit itself. Instead, we can

²Find a list of the options in [22].

analyse how many cycles and gates are involved in the construction, which supply a criterion to compare with. In practice, we know exactly how many gates and cycles the circuit will be comprised of just by looking at the order of the bit strings and therefore do not need **superpos**. It is more reasonable to evaluate the gate and cycle count by a separate, more economical function that does not access qiskit. We introduce **count_simulation** (Fig. 3.10).

```

1 def count_simulation(bin_list):
2   # takes binary input list
3
4   gate_count = len(bin_list)-1
5   # number of the controlled Ry gates
6   cycle_count = len(bin_list)
7   # one cycle to create first layer of X's plus number of Ry's
8
9   for i in bin_list[0]:
10    gate_count += int(i)
11   # X gates
12   for pos in range(len(bin_list)-1):
13    number1 = bin_list[pos]
14    number2 = bin_list[pos+1]
15    for i in range(amount_of_qubits):
16     if(number1[i] != number2[i]):
17      gate_count += 1
18      cycle_count += 1
19   # every CNOT gate is counted
20
21   return [gate_count, cycle_count] # returns list

```

Figure 3.10: Implementation of **count_simulation**. It calculates the number of gates and cycles required to construct a circuit in the style of **superpos**. Again, **amount_of_qubits** is a predefined variable equal to the amount of bits in each binary.

To put the gate and cycle counts we obtain into perspective, it is helpful to know the ideal values for a given input list. Unfortunately, determining which sequence(s) of the input set result in the optimal gate and cycle counts—and what those ideal counts actually are—requires checking every possible permutation of the binary input set. We can implement a function to do this, as shown in Fig. 3.11. For input lists shorter than ten elements, we observe that the regular TSP-based solver with the standard search strategy often finds an ideal succession, but not always. Using guided local search the TSP solver seems to find an ideal result for less than ten input elements every time, even with the minimum of one second search time. However, the **minimal_counts** function is impractical due to its factorial time complexity with respect to the size

of the input list. For input lists exceeding ten elements, the runtime grows rapidly and becomes impractical. Therefore, we will not use this function for comparisons involving more elements. While we could examine the behaviour and correctness of the TSP solver in more detail for small input lists, the algorithm is ultimately supposed to provide acceptable results for instances involving several hundred numbers. In the future, such sizes may be necessary for the preparation of qubit states used in subsequent quantum computations.

```
1 import itertools
2
3 def minimal_counts(bin_list):
4     # takes binary input list
5     ...
6     for permutation in itertools.permutations(
7         bin_list, len(bin_list)):
8         # produces every possible permutation of bin_list
9         ...
10    ...
11    return [best_gc_order, best_cc_order, minimal_gc, minimal_cc]
```

Figure 3.11: Implementation of function **minimal_counts**. It works with the permutation tool from the library *itertools*. Given a list of binary numbers, the function returns a list consisting of: (1) one permutation of the input list with minimal gates, (2) one potentially divergent permutation with minimal cycles, (3) the minimal gates, (4) the minimal cycles.

In absence of optimal results to compare the TSP solver's circuits with, we compare it with the layouts our initial input sequence and the greedy algorithm produce. The relevant criteria will be both the gate count and runtime. At the scales we will look at, no additional information is gained by also comparing the cycle counts, since gates and cycles only differ by a relatively small number caused by those *X* gates that form the first layer.

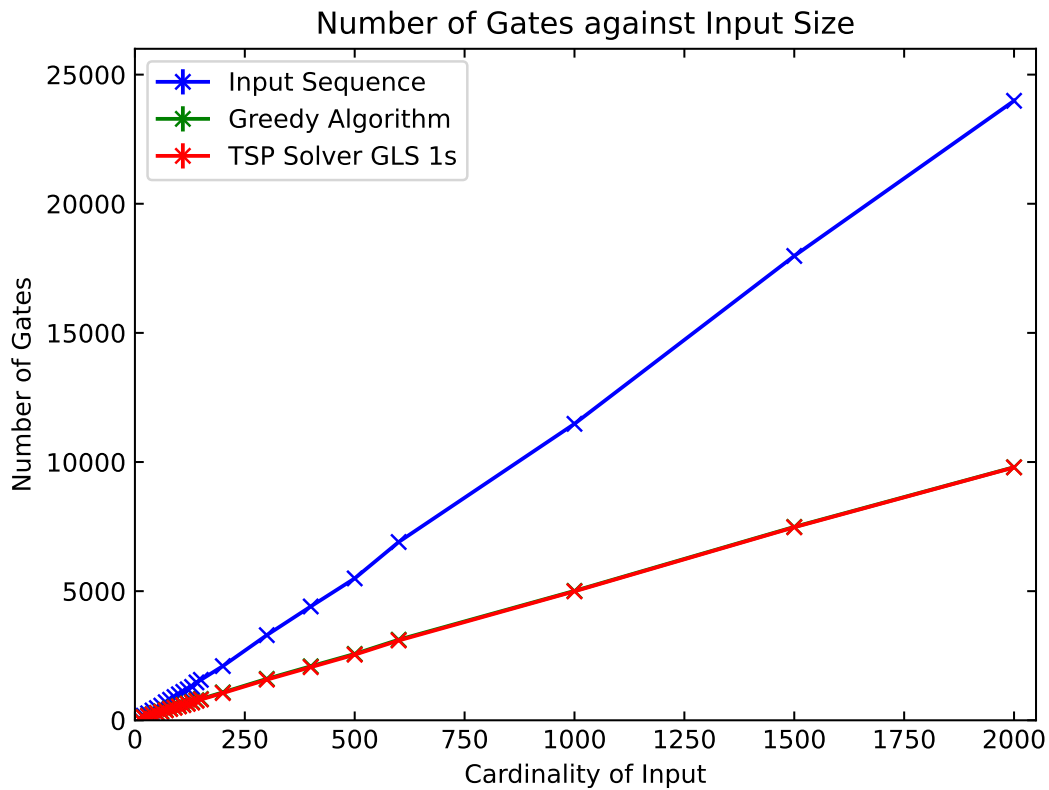


Figure 3.12: Comparison of the gate counts between the input sequence, the greedy algorithm and the TSP solver employing guided local search (gls) for 1 second. The graph shows how many gates are needed for each sequence to build a circuit with **superpos**. For each input size I , lists with unique random values between $-1000 \cdot I$ and $1000 \cdot I$ were created. The results are averaged over 50 iterations each. The data points for the greedy algorithm lie closely to the ones for the TSP algorithm.

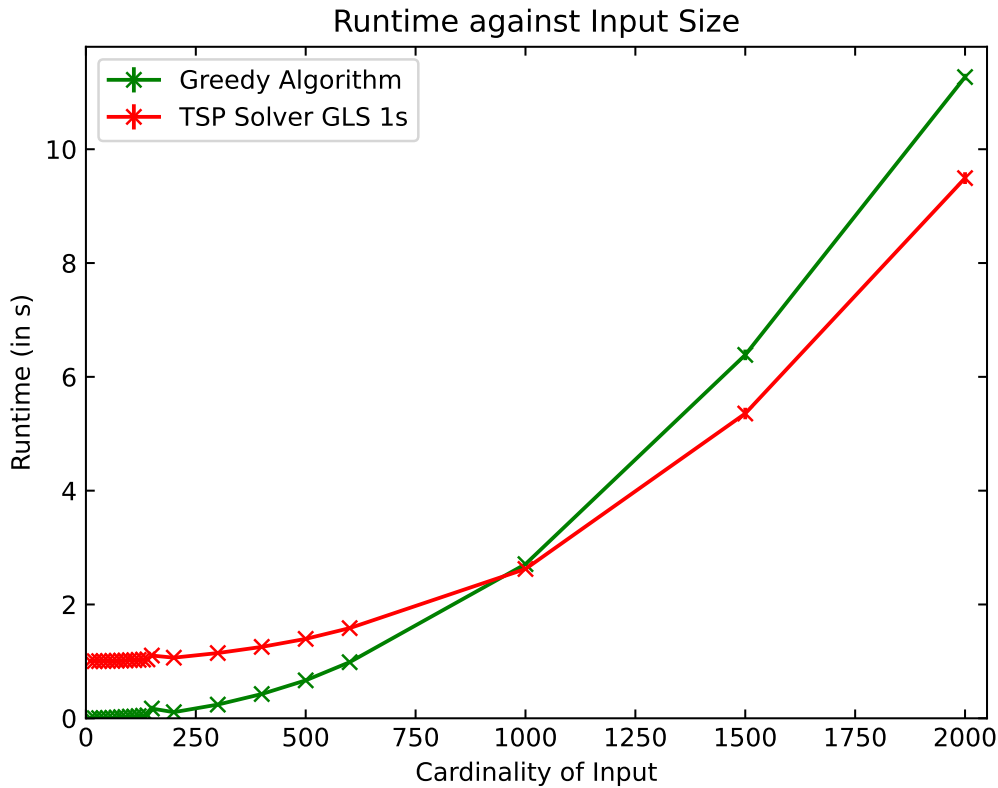


Figure 3.13: Comparison of the runtimes of the input sequence, the greedy algorithm and the TSP solver employing guided local search (gls) for 1 second. The graph depicts the time taken to compute the enhanced sequence and subsequently calculating the required gates and cycles via `count_simulation`. The data was collected concurrently with the measurements used for Fig. 3.12.

In Fig. 3.12 we see that both greedy algorithm and TSP algorithm lead to significantly improved gate counts compared to the initial random input lists. By setting the range of input values to be 1,000 times the number of inputs, we ensure that each input set represents the same relative fraction of the total range. In Fig. 3.13, the runtimes of the three variations are displayed. The original input sequence takes close to no time at all to compute because only `count_simulation` has to be run. Although we requested gls to only search for 1 second, the actual runtime can exceed this time since one full iteration of the underlying program can overrun this time. It seems like the TSP solver's runtime scales better with the cardinality of the input than greedy's runtime does. But, when comparing both runtimes, we have to take into account that both algorithms are not implemented with comparable efficiency.

While the performance-critical components of the TSP solver are written in C++ by OR-Tools, the greedy algorithm is implemented in Python. Considering this, it is not surprising that the TSP-based solver performs faster on larger instances. However, it is noteworthy that the greedy solver—although programmed in Python—has runtimes of the same order of magnitude as the TSP solver. This is a significant observation.

The greedy heuristic and OR Tools' TSP solver seem to have very similar data points in our gate count comparison. We will see another graph to compare them in more detail.

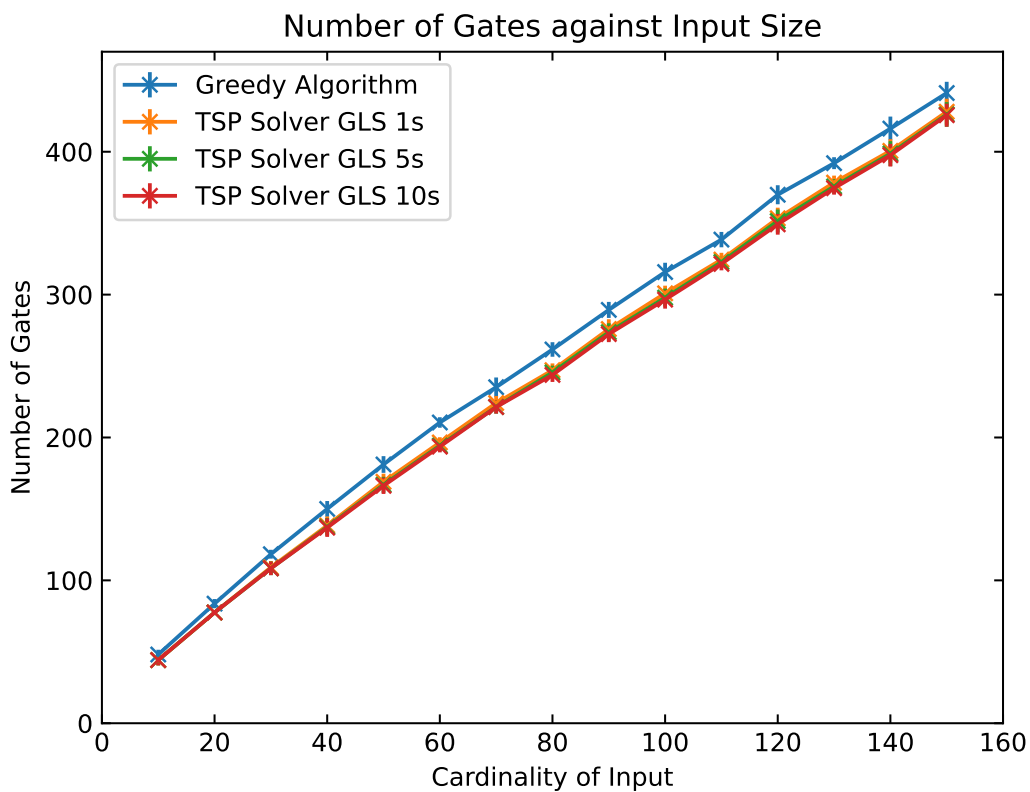


Figure 3.14: Comparison of the gate counts between the greedy algorithm and the TSP solver with gls for 1, 5 and 10 seconds runtime. The graph depicts how many gates are required to create a superposition with **superpos**. The input lists are random numbers between -1000 and 1000, each. Every data point is averaged over the results for 50 different input lists.

Fig. 3.14 shows that the gate counts for **superpos** sequences generated by the greedy solver and the TSP solver differ only slightly, with the TSP-based solver yielding

marginally better results. Furthermore, increasing the runtime of GLS does not lead to a significant reduction in gate count, indicating that the improvement is negligible. In the graph, the cardinality of the input increases while the range of values remains unchanged. This causes the curve to bend slightly downward, because the elements lie closer together on average for larger instances.

In general, the results can vary significantly depending on the characteristics of the input list. Is the allowed set of values small or large? How substantial are the Hamming distances between elements? When the occupied decimal numbers in the allowed set are closer together, the gate count tends to increase more slowly. However, we observe a clear trend of a utility in changing the random input number order. Using the TSP-based solver leads to slightly better results and, as long as we use Python to write the greedy solver, the TSP-based solver has a small advantage in runtime. Nevertheless, it is interesting to see that the straightforward approach of the greedy solver achieves competitive results when compared to the significantly more complex TSP-based approach.

In this chapter, we implemented a classical algorithm capable of generating a quantum circuit that prepares a superposition over the binary representations of a given input list. The primary objective was to minimize the number of gates and cycles required for this construction and we developed different solver strategies to optimize the circuit accordingly. In the following, we address a closely related problem. Instead of receiving explicit input lists, we now consider boolean functions as inputs.

Preparing Uniform Superpositions Conditioned on Boolean Functions

A *boolean function* f is a mapping from a binary input string to a binary output:

$$f : \{0, 1\}^n \rightarrow \{0, 1\}. \quad (4.0.1)$$

Every boolean function can be expressed as a *boolean formula* (or *logical formula*) that uses logical operators like AND (\wedge), OR (\vee), and NOT (\neg) to connect variables x_0, \dots, x_{n-1} . The aim of this chapter is to find universal rules for the construction of quantum circuits as follows: given a boolean function, a register of qubits is prepared in uniform superposition over all strings that are mapped to 1 (that is, logical *True*), without the need of identifying those strings in advance.

The following definitions will be useful.

Definition 4.1. Let φ be a boolean formula over the variables x_0, x_1, \dots, x_n . A **satisfying assignment** for φ is a bit string $\nu \in \{0, 1\}^{n+1}$, where the i -th bit represents the truth value assigned to x_i , such that φ evaluates to true under ν , i.e. $\varphi(\nu) = 1$. The **set of all satisfying assignments** of φ is denoted $Sat(\varphi)$ (**satisfiable**) and defined as

$$Sat(\varphi) := \{\nu \mid \varphi(\nu) = 1\}.$$

Every boolean formula can be written in *conjunctive normal form* and *disjunctive normal form*.

Definition 4.2. A boolean formula φ is said to be in *conjunctive normal form* (CNF) if it is a conjunction of one or more *clauses*, where each clause is a disjunction of one or more *literals* (also called *atoms*) [23]. Formally, a formula in CNF has the structure:

$$\varphi = \bigwedge_i \bigvee_j (\neg)\varphi_{ij},$$

where $\bigvee_j (\neg)\varphi_{ij}$ is a clause and a negated or non-negated variable $(\neg)\varphi_{ij}$ is a literal.

Definition 4.3. A boolean formula φ is said to be in *disjunctive normal form* (DNF) if it is a disjunction of one or more clauses, where each clause is a conjunction of one or more literals [23]. Formally, a formula in DNF has the structure:

$$\varphi = \bigvee_i \bigwedge_j (\neg)\varphi_{ij},$$

where $\bigwedge_j (\neg)\varphi_{ij}$ is a clause and a negated or non-negated variable $(\neg)\varphi_{ij}$ is a literal.

4.1 General Boolean Formulae

At first, we design circuits to cover the elementary cases of AND, OR, and NOT for 1- or 2-element formulae. For the AND operator, we have

$$q_1 : |0\rangle \text{---} \boxed{X} \text{---}$$

$$q_0 : |0\rangle \text{---} \boxed{X} \text{---}$$

Figure 4.1: A circuit creating uniform superposition for an AND operator connecting two variables. The logical formula $\varphi = x_0 \wedge x_1$ accepts only one input, leading to the state $|\psi\rangle = |11\rangle$ after applying the circuit.

We introduce the notation

$$R_y(a : b) := R_y(\theta), \quad \text{where} \quad \sin\left(\frac{\theta}{2}\right) = \sqrt{\frac{a}{b}} \quad \text{with} \quad 0 \leq \theta < 2\pi. \quad (4.1.1)$$

This representation emphasises how the probabilities are distributed on the computational basis, in particular how much of the amplitude of a $|0\rangle$ state is transferred to $|1\rangle$. For example:

$$R_y(2 : 3) |0\rangle = \sqrt{1 - \frac{2}{3}} |0\rangle + \sqrt{\frac{2}{3}} |1\rangle = \sqrt{\frac{1}{3}} |0\rangle + \sqrt{\frac{2}{3}} |1\rangle$$

For the OR and NOT operator, we have

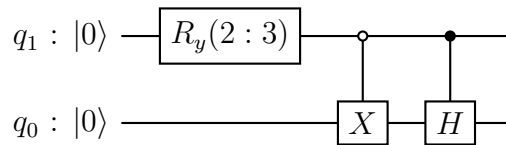


Figure 4.2: A circuit creating uniform superposition for an OR operator connecting two variables. The rotation gate prepares q_1 in $|1\rangle$ with probability $\frac{2}{3}$. A superposition is prepared for all assignments that satisfy $\varphi = x_0 \vee x_1$, namely $|01\rangle, |10\rangle, |11\rangle$.

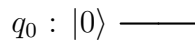


Figure 4.3: A circuit creating uniform superposition for a NOT operator on one variable. The only accepted input for $\varphi = \neg x_0$ is $|0\rangle$.

The transformation to cases with negation of one or both variables that are linked through logical conjunction or disjunction is trivial.

The next step is to take two functions that already work individually, and join them together with an AND operator to one function. Let $f(x_0, \dots, x_{n-1})$ and $g(x_n, \dots, x_{m-1})$ be two boolean functions with disjoint input sets, for which we know the quantum circuits that prepare a superposition. We label these circuits η_f and η_g , respectively. The composite function is

$$\varphi = f(x_0, \dots, x_{n-1}) \wedge g(x_n, \dots, x_{m-1}). \tag{4.1.2}$$

A uniform superposition over all satisfying assignments $Sat(\varphi)$ is prepared by the circuit

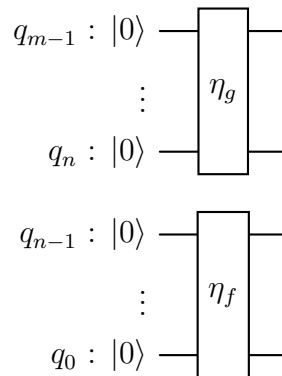


Figure 4.4: Uniform superposition for a conjunction of two functions f and g with disjoint inputs.

A composite function, as in Eq. (4.1.2), represents a special case of conjoined functions, as their input variables form disjoint sets. As we will see, finding a general solution for conjoined functions with non-disjoint input sets is significantly more challenging.

We have a look at the seemingly simply case of one shared variable in the input sets (Eq. (4.1.3)) and aim to find a circuit that prepares a superposition over $Sat(\varphi)$.

$$\varphi = f(x_0, \dots, x_n) \wedge g(x_n, \dots, x_{m-1}). \quad (4.1.3)$$

Intuitively, one might come up with the following quantum circuit:

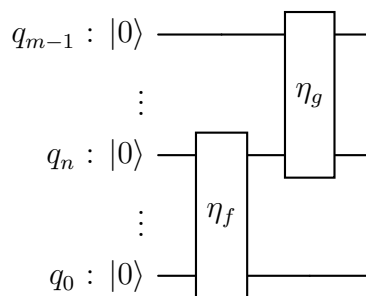


Figure 4.5: First version of a circuit to create superposition for the conjunction of two subfunctions with one shared variable in the inputs.

The problem with the circuit in Fig. 4.5 becomes evident when considering the formula

$$\varphi = f(x_0, x_1) \wedge g(x_1, x_2) = (x_0 \wedge x_1) \wedge (x_1 \wedge x_2), \quad (4.1.4)$$

for which the quantum circuit should produce $|111\rangle$. In practice, both subfunctions f and g perform X gates on both their variables, so q_1 will be subject to the effect of two X gates. Therefore, we will measure $|101\rangle$, which is not the correct state. A possible approach to address this problem is as follows. After acting with η_f on its input set of qubits, we (re)set the shared variable x_n to $|0\rangle$, regardless of whether it was previously in the state $|0\rangle$ or $|1\rangle$. Fig. 4.6 displays a suitable quantum circuit employing one ancilla qubit a .

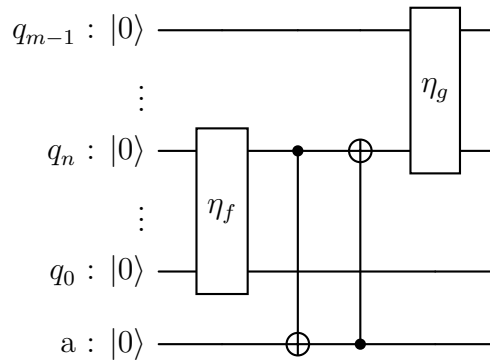


Figure 4.6: Second version of the circuit to create superposition for the conjunction of two subfunctions with one shared variable.

The previous problem is thus fixed. For a logic formula given by Eq. (4.1.4), one X gate acts on q_0 and q_2 each, and two X gates act on q_1 , but with the help of ancilla a , q_1 is set back to $|0\rangle$ in between the X gates, so the final overall state is $|111\rangle|1\rangle$. Other cases also work, where f and g are composite functions themselves, as we saw in Fig. 4.4. However, for the general construction as in Fig. 4.6, failure cases occur as well. We consider

$$\varphi = f(x_0, x_1) \wedge g(x_1, x_2) = (x_0 \wedge x_1) \wedge (x_1 \vee x_2). \tag{4.1.5}$$

The resulting superposition on the qubit register is supposed to be

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|011\rangle|1\rangle + |111\rangle|1\rangle),$$

but with the given circuit we get

$$|\psi\rangle = \frac{1}{\sqrt{3}}(|011\rangle|1\rangle + |101\rangle|1\rangle + |111\rangle|1\rangle).$$

The circuit works reliably only if q_n satisfies both f and g only with the same assignment, either $|0\rangle$ or $|1\rangle$. In such cases, the composite function can be expressed using a single

third of the amplitude is associated to inputs that satisfy both f and g . But, if f for example is satisfied by q_0, \dots, q_{n-1} and the remaining qubits q_n, \dots, q_{m-1} are in uniform superposition over all assignments, there is a chance that they also satisfy g . Thus, the amplitudes are not equal and the general superposition is non-uniform. Also, f and g might have different amounts of satisfying assignments and may cause the imbalance to intensify.

Averaging the amplitudes could be performed by R_y rotation gates. For now, we set aside the question of where exactly the gates are introduced. In any case, their angles depend on how many satisfying assignments each subfunction has, or rather, how great the proportion of satisfying the subfunctions in the set of all satisfying assignments is. The problem of counting the number of satisfying assignments is called $\#SAT$ (spoken *Sharp-SAT*). SAT (Satisfiability Problem) is the problem that asks whether a logical formula is satisfiable; it lies in NP-complete [24], i.e. it is improbable that polynomial-time solutions exist. The counting version of it, $\#SAT$, is at least as hard as SAT , since it not only asks whether, but rather how many satisfying assignments exist. $\#SAT$ lies in a complexity class called $\#P$ -complete (*Sharp-P-complete*) [25]. There exist no known polynomial-time algorithms for $\#P$ -complete problems.

While we may tolerate the inefficiency in calculating satisfying assignments if the produced quantum circuits work resource and time efficient, the circuit shown in Fig. 4.7 possesses a fundamental limitation: it can only generate a superposition over the satisfying assignments of a disjunction of functions with disjoint input sets. Extending this approach to functions with non-disjoint inputs is non-trivial and not straightforward. Therefore, in the next section, we turn our attention to general formulae in disjunctive normal form (DNF). We will demonstrate a method for preparing a uniform superposition over their satisfying assignments. However, this comes at the expense of extremely poor scalability in circuit size.

4.2 Disjunctive Normal Form

Every boolean function can be expressed by a formula in DNF (see Definition 4.3). So, finding a solution for the preparation of superposition for the DNF effectively yields a solution for all boolean functions. A possible construction of suitable circuits is shown in Example 1 and it works as follows. First, rotation gates regulate the amplitudes of a register of ancilla qubits. The amplitudes for the possible ancilla states correspond to the amount of satisfying assignments of every clause of conjunctions. Then, each controlled by one of those ancilla states, the clauses are satisfied. For each clause, a uniform superposition is prepared over all satisfying assignments of it, using X and

H gates. One satisfying assignment could be covered in multiple parts of the circuit and thus have a higher amplitude than the others. To fix this, additional ancilla qubits are introduced that are set conditioned on a doubly satisfied assignment, and partly reassign the amplitude from one such assignment inside one clause to all other satisfying assignments of that clause.

Example 1. Let φ be

$$\varphi = (x_0 \wedge x_1) \vee (x_2 \wedge x_3 \wedge \neg x_4). \quad (4.2.1)$$

Then, the circuit preparing superposition over $Sat(\varphi)$ is

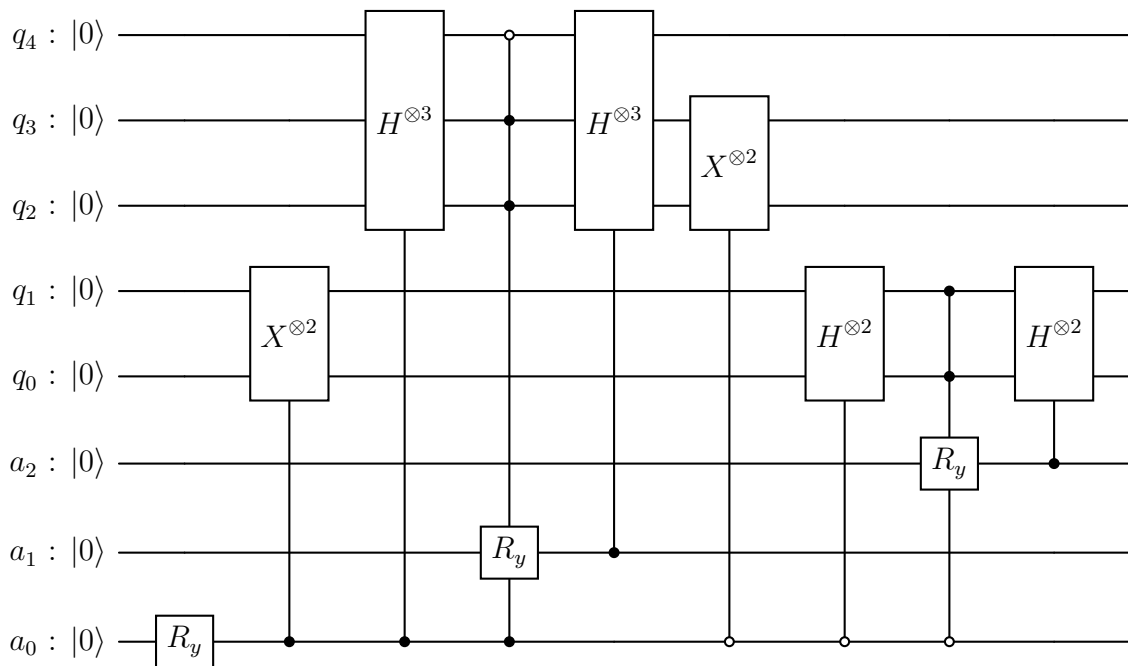


Figure 4.8: Example of a circuit for the formula in Eq. (4.2.1). a_0 controls that either the first or the second clause is satisfied. The assignment $|01111\rangle$ of the main register is satisfied twice. The Hadamard gates controlled by a_1 and a_2 flatten the amplitude of $|01111\rangle$ while raising the amplitude of all other output states. With suitable angles in the rotation gates, the circuit prepares a uniform superposition.

The circuit structure can be extended in the same pattern with more ancilla qubits to control more clauses and to align the amplitudes of more multiply satisfied assignments. One issue with this kind of circuit design is the necessity of knowing not only the amount of satisfying assignments for each clause, but also the exact assignments that occur several times. Maybe it is possible to compare sub-formulae to find those multiply

covered states without computing every state in advance. But an even bigger issue with the circuit is that it scales proportionally with the number of shared variables between clauses. In general, the latter scale exponentially with the number of variables involved in the logical formula. Therefore, this circuit is inefficient.

4.3 Conjunctive Normal Form

Every boolean function can also be written in conjunctive normal form (Definition 4.2). To develop an understanding of how a circuit processing formulae in CNF might be constructed, we will study simple examples. Consider the formula

$$\varphi = (x_0 \vee x_1) \wedge (x_0 \vee x_2). \tag{4.3.1}$$

In Fig. 4.2 we presented a way to process the OR operator connecting two variables. Executing it twice in a row with adjusted angles for the rotation gates yields a circuit that processes φ :

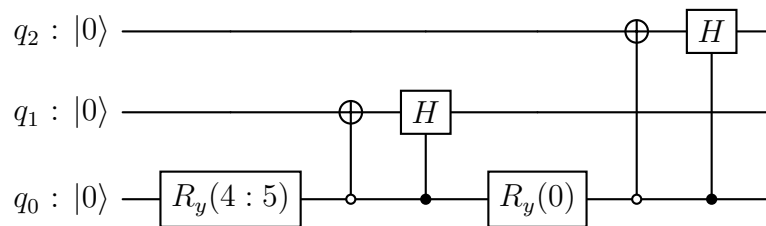


Figure 4.9: A circuit preparing a uniform superposition for the logic formula φ in Eq. (4.3.1). $R_y(0)$ is equal to the identity operator.

The final state is $|\psi\rangle = \frac{1}{\sqrt{5}}(|001\rangle + |011\rangle + |101\rangle + |110\rangle + |111\rangle)$.

To explore whether this scheme can extend to more complex formulae, we examine

$$\varphi = (x_0 \vee x_1) \wedge (x_0 \vee x_2) \wedge (x_1 \vee x_2) \tag{4.3.2}$$

The corresponding circuit is

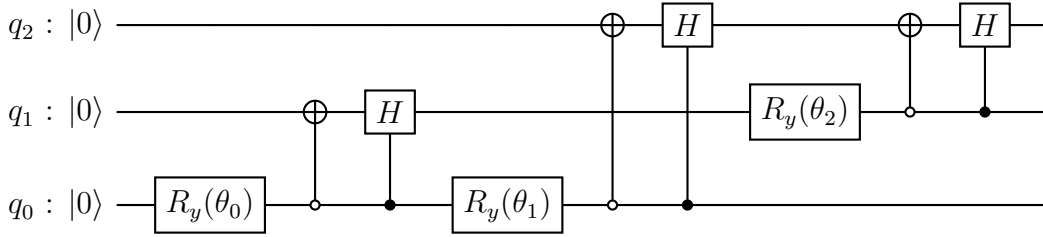


Figure 4.10: Attempt of a circuit processing the formula φ as given in Eq. (4.3.2). The angles θ_0, θ_1 and θ_2 are yet to be determined.

The resulting state of Fig. 4.10 is

$$|\psi\rangle = a_0 |000\rangle + a_1 (|001\rangle + |101\rangle) + a_2 (|010\rangle + |110\rangle) + a_3 (|011\rangle + |111\rangle), \quad (4.3.3)$$

where a_0, a_1, a_2 and a_3 are scalar coefficients that depend on θ_0, θ_1 and θ_2 . Regardless of the choice of a_0, a_1, a_2, a_3 , Eq. (4.3.3) fails to yield the desired superposition

$$|\psi\rangle = \frac{1}{\sqrt{4}} (|011\rangle + |101\rangle + |110\rangle + |111\rangle). \quad (4.3.4)$$

Therefore, we propose a more complex version of the circuit in Fig. 4.2 that handles the disjunction of two variables:

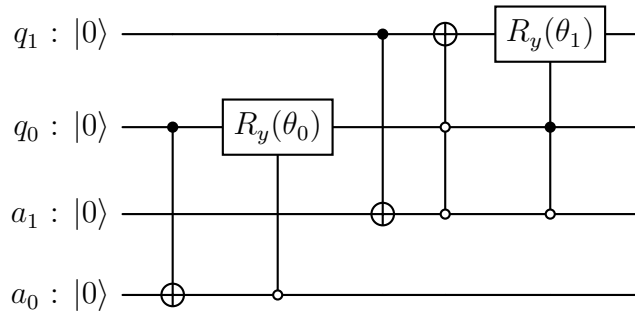


Figure 4.11: Composable circuit generating a uniform superposition for the disjunction of two variables; it can be reused to construct larger circuits. Two ancilla qubits are required for each of these circuits. The value of q_0 is communicated to a_0 , and if q_0 is already in $|1\rangle$, the rotation is omitted. A similar control is introduced for q_1 : If it is already set, no further operations are performed. This prohibits that a qubit prepared in $|1\rangle$ by previous operations is set back to $|0\rangle$. Also, the Hadamard gate is changed for a rotation gate to enable more precise control of the amplitudes.

To create a uniform superposition over $Sat(\varphi)$, φ as in Eq. (4.3.2), we introduce a register of three main qubits and six ancilla qubits, two for each sub-circuit as presented

in Fig. 4.11. All qubits are initialized in $|0\rangle$. The construction of the circuit is similar to Fig. 4.10, with additional interactions between the main and ancilla qubits. Denoting the rotation gate angles by θ_i for $i = 0, \dots, 5$ and with $c_i := \cos(2\theta_i)$ and $s_i := \sin(2\theta_i)$, the resulting quantum state is

$$\begin{aligned} |\psi\rangle = & (c_0 s_2 c_3 c_5 + s_0 c_1 c_3 c_4 c_5 + s_0 s_1 c_3 c_5) |011\rangle \\ & + (s_0 c_1 c_3 c_4 + s_0 c_1 s_3) |101\rangle \\ & + (c_0 c_2) |110\rangle \\ & + (c_0 s_2 (c_3 s_5 + s_3) + s_0 c_1 c_3 c_4 s_5 + s_0 s_1 (c_3 s_5 + s_2)) |111\rangle. \end{aligned}$$

Proper values for the angles produce a uniform superposition as in Eq. (4.3.4). If the Hadamard gates had not been replaced with rotation gates in Fig. 4.11, the system of equations would have been overdetermined with only three controllable angles and would not have been solvable.

The underlying way of how we have created circuits that process formulae in CNF is no longer sufficient for special cases involving negations. Consider

$$\varphi = (x_0 \vee x_1) \wedge (x_2 \vee x_3) \wedge (\neg x_0 \vee \neg x_2). \quad (4.3.5)$$

Circuits that work through the formula from left to right will produce the state $|0101\rangle$ after the first two clauses, since it satisfies both of them. But this state does not satisfy the third clause. To eliminate the possibility for the assignment $|0101\rangle$, its amplitude would have to be distributed among all other allowed states. To achieve this, the information from the third clause needs to affect the assignments processed before. The construction of such a circuit is not trivial.

4.4 Clauses of n Disjunctions

When working with logical formulae in CNF (Definition 4.2), we have to deal with disjunctions of one or more literals. The disjunction of one literal is the trivial case that this one literal is fulfilled. We will treat a general sequence of n disjunctions linking $n + 1$ literals of pairwise different variables. Our goal is to design a quantum circuit that creates uniform superposition over all satisfying assignments for such a clause of $n + 1$ literals. Since at least one of the literals has to evaluate to true, there are $2^{n+1} - 1$ satisfying assignments and only one non-satisfying assignment. The idea is to use R_y gates that give certain probabilities to one qubit being either $|0\rangle$ or $|1\rangle$ and operate them in a kind of cascade formation down the qubits, such that at least one literal is always true.

First, we recall the simple case of a disjunction of two variables: $\varphi = x_0 \vee x_1$ (Fig. 4.12). The starting point for a quantum circuit is a quantum register of two qubits q_0 and q_1 , both initialised in $|0\rangle$. The rotation gate transforms q_1 into a superposition assigning the probabilities of $\frac{1}{3}$ and $\frac{2}{3}$ to both outcomes $|0\rangle$ and $|1\rangle$ respectively upon measurement. If q_1 is zero, q_0 is set and otherwise q_0 itself gets transformed into a uniform superposition by H . This way we receive the outcome state $\varphi = \frac{1}{\sqrt{3}}(|01\rangle + |10\rangle + |11\rangle)$ as planned.

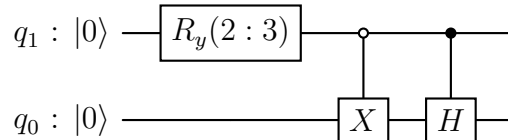


Figure 4.12: A circuit creating uniform superposition over all assignments that satisfy $\varphi = x_0 \vee x_1$, namely $|01\rangle, |10\rangle, |11\rangle$.

This scheme can be continued in a recursive way, as illustrated in Fig. 4.13. We imagine to add a new qubit $q_n : |0\rangle$ on the top layer and act on it with a rotation gate. Controlled by q_n being in the state $|0\rangle$, we then apply the whole circuit that we had previously, ensuring that at least one qubit is non-zero upon measurement. After that we attend to the case $q_n : |1\rangle$, which allows all other qubits to have arbitrary states. Acting with Hadamard gates controlled by q_n on every other qubit will do the job and create a uniform superposition between them. Finally, we have created a uniform superposition over all assignments that satisfy $\varphi = x_0 \vee \dots \vee x_n$.

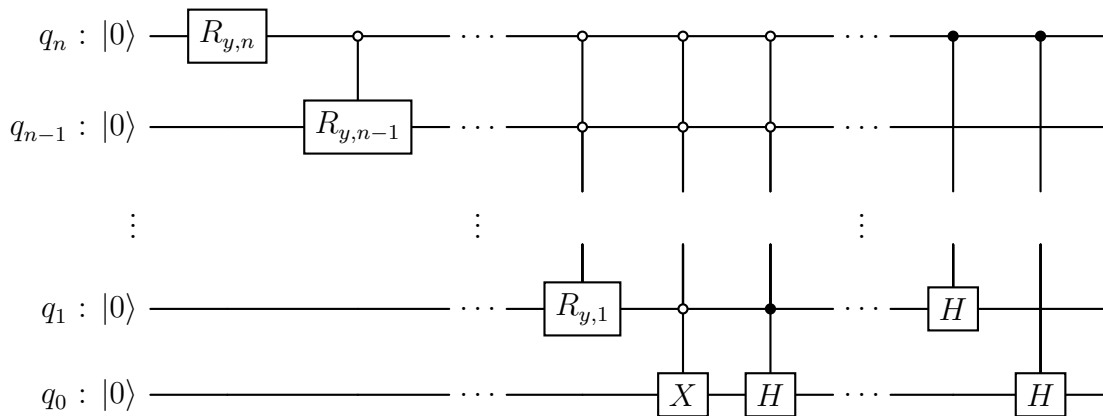


Figure 4.13: A circuit creating uniform superposition over all assignments that satisfy $\varphi = \bigvee_{i=0}^n x_i = x_0 \vee \dots \vee x_n$.

The formal representation for that circuit is

$$OR^0 = X \quad (4.4.1)$$

$$OR^n = C_n(H_{n-1} \otimes \dots \otimes H_0) \neg C_n(OR^{n-1}) R_{y,n}(2^n : 2^{n+1} - 1). \quad (4.4.2)$$

Until now, we have exclusively focused on variables, that is, literals without negation. Suppose that one or more literals of the boolean expression are negated. The probabilities should be distributed the same way as before, only the outcome has to be adapted. In practice, we just have to operate with one X gate on the qubit in question, and that has to be introduced behind the R_y gates and the controls for H gates, so that it has no effect on the probability distribution. Considering this, in the recursive setup the X gate will not interfere with any other parts of the circuit since it will be conditionally activated, assuming all prior qubits are in the zero state. If that is not the case, then the negative-controlled X gate will not be activated and thus never interfere with any other gate.

To take the negative literals into account, we extend Eq. (4.4.1) and Eq. (4.4.2) by a Pauli X if and only if literal x_i is negated. Otherwise, I is applied, leaving the expression untouched. The notation A_k for any gate A indicates that it acts on the k -th qubit. C is the control.

Theorem 4.4. Given a logical n -OR, expressed by the boolean function $\varphi = \bigvee_{i=0}^n \ell_i$ with $\ell_i \in \{x_i, \neg x_i\}$, where x_i is a variable, a uniform superposition over all satisfying assignments is created by a circuit OR^n .

$$OR^0 = X^a X \quad (4.4.3)$$

$$OR^n = X^a C_n(H_{n-1} \otimes \dots \otimes H_0) \neg C_n(OR^{n-1}) R_{y,n}(2^n : 2^{n+1} - 1) \quad (4.4.4)$$

$$\text{with } a = \begin{cases} 0, & \text{if } \ell_i = x_i \\ 1, & \text{if } \ell_i = \neg x_i \end{cases}.$$

Proof. Let ν_k denote the satisfying assignment for ℓ_k . ν_k is 1 if $\ell_k = x_k$ and 0 if $\ell_k = \neg x_k$. Moreover, let $L_k = \text{Sat}(\bigvee_{i=0}^k \ell_i)$, the set of all bit strings satisfying $\ell_0 \vee \dots \vee \ell_k$. We carry out the proof using induction. The base case is obvious:

$$OR^0 |0\rangle = |\nu_0\rangle$$

Assuming the hypothesis holds for $(n-1)$, we have

$$OR^{n-1} |0\rangle^{\otimes n} = \frac{1}{\sqrt{2^n - 1}} \sum_{x \in L_{n-1}} |x\rangle.$$

For clarity, we split the induction step $(n - 1) \rightarrow n$ into two steps. First, we consider the effect of $R_{y,n}$ on the $|0\rangle^{\otimes(n+1)}$ state.

$$\begin{aligned} R_{y,n}(2^n : 2^{n+1} - 1) |0\rangle^{\otimes(n+1)} &= \sqrt{\frac{2^n}{2^{n+1} - 1}} |\nu_n\rangle |0\dots 0\rangle + \sqrt{1 - \frac{2^n}{2^{n+1} - 1}} |\neg\nu_n\rangle |0\dots 0\rangle \\ &= \sqrt{\frac{2^n}{2^{n+1} - 1}} |\nu_n\rangle |0\dots 0\rangle + \sqrt{\frac{2^n - 1}{2^{n+1} - 1}} |\neg\nu_n\rangle |0\dots 0\rangle \end{aligned}$$

Now, given OR^{n-1} , we can operate the whole OR^n circuit on $|0\rangle^{\otimes(n+1)}$:

$$\begin{aligned} OR^n |0\rangle^{\otimes(n+1)} &= \sqrt{\frac{2^n}{2^{n+1} - 1}} \sqrt{\frac{1}{2^n}} \sum_{x \in \mathbf{b}^n} |\nu_n\rangle |x\rangle + \sqrt{\frac{2^n - 1}{2^{n+1} - 1}} \sqrt{\frac{1}{2^n - 1}} \sum_{y \in L_{n-1}} |\neg\nu_n\rangle |y\rangle \\ &= \frac{1}{\sqrt{2^{n+1} - 1}} \left(\sum_{x \in \mathbf{b}^n} |\nu_n\rangle |x\rangle + \sum_{y \in L_{n-1}} |\neg\nu_n\rangle |y\rangle \right) = \frac{1}{\sqrt{2^{n+1} - 1}} \sum_{x \in L_n} |x\rangle \end{aligned}$$

□

Conclusion and Outlook

In this thesis, methods to create superpositions for explicit integer lists and for the satisfying assignments of boolean functions were proposed. Given a list of integer inputs, we introduced an algorithm that generates quantum circuits whose qubit register scales proportionally with the number of bits required to represent each input unambiguously in binary representation. Only one additional ancilla qubit is required. The number of gates scales proportionally with the input size, assuming that the input size scales proportionally with the set of possible input values. To improve the gate count, we introduced a greedy heuristic that re-sorts the order of elements with locally optimal solutions. That is, each element is followed by the element to which it has the smallest Hamming distance, i.e. the least required gates. Moreover, we implemented a sorting method based on solving the TSP to find the (approximately) ideal order to minimize the gate count. The cycle count differs therefrom only by a constant number.

With both solving heuristics, we were able to clearly decrease the required amount of gates in the circuit construction. Interestingly, the straight-forward greedy algorithm produces comparable results to the TSP-based solver. Even when considering the runtimes, both methods deviate only slightly from one another. **OR-Tools** provides search routines to improve the results. On the device this program was tested on, the improvement acquired by extended runtimes is negligible. Since both the greedy and TSP method are implemented in Python but the performance-relevant parts of the TSP solver are written in C++ by developer **Google OR-Tools**, the implementation of the greedy heuristic in C could make it faster than TSP. Then, one could decide between faster (greedy) or higher quality (TSP) results, depending on context.

The presented algorithm requires us to know the exact input set. To handle boolean functions, we first attempted searching for fundamental recursive rules that produce quantum circuits. It is possible to cover simple OR, AND and NOT operators for literals as well as the conjunctions of them as long as their input sets are disjoint. We were able to produce non-uniform superpositions over the disjunction of subfunctions with disjoint input sets. Since the problem of averaging the amplitudes can be reduced to a #P-complete problem, there exist no known polynomial-time algorithms to solve

it. Therefore, homogenizing the superposition seems to be inefficient in this case. Also, we were not able to handle disjunctions of subfunctions with non-disjoint inputs. It seems as if the processing of conjunctions of subfunctions is more intuitive. Future efforts in their implementation could yield efficiently working circuits.

Boolean formulae in DNF can be handled intuitively with circuits that generate non-uniform superpositions. However, homogenizing the superposition is inefficient, as it leads to an expected exponential growth in circuit complexity with the number of variables. If there is a way to exclude the overlapping assignments from different parts of the formula, the DNF is a promising form of boolean functions to work with. Otherwise, a fundamentally new approach is required to handle the DNF.

It is possible to take care of simple circuits in CNF with adapted, reusable circuits for the single-OR operation. They can be melted together to form circuits for at least triple conjunctions of disjunctions of two variables each. Nevertheless, the angles for the rotation gates involved are non-trivial and the way we approached the CNF breaks down when considering negated variables.

Finally, we presented a procedure to efficiently realize uniform superpositions for clauses of n disjunctions of literals. Since functions in CNF consist of these clauses, this is a significant result. It is promising that the n -OR as well as simple conjunctions of disjunctions can be dealt with by the presented circuits. Considering larger formulae may ultimately result in general rules for the construction of circuits that handle functions in DNF. Maybe patterns or even formulae will be found determining the angles of the rotation gates. There may also be a way to also include negations, for example by employing more ancilla qubits that hold various information about the logical formula. Apart from that, completely different approaches to deal with both DNF and CNF, as well as general functions are possible. Also, other normal forms could be examined, for example the *algebraic normal form*.

Like many other aspects of quantum theory, dealing with qubits is sometimes counter-intuitive, and one must acquire a certain "quantum-intuition" before finding working circuits. The results of this thesis should be considered as reference points on how and how not to build circuits that generate superposition over supports of boolean functions. Further efforts will likely produce significant results that contribute meaningfully to quantum computing.

List of Figures

2.1	Single-qubit Pauli gates	6
2.2	Single-qubit Hadamard gate	7
2.3	Single-qubit rotation gates	7
2.4	<i>CNOT</i> gate	8
2.5	Controlled unitary gate	9
2.6	Controlled phase shift gate	9
2.7	Controlled U gate	10
2.8	Doubly-controlled unitary	10
2.9	Unitary U controlled by n qubits	11
2.10	Negative-controlled unitary U	11
3.1	Exemplary presentation of the algorithm	14
3.2	Exemplary presentation of qubit-saving algorithm	15
3.3	Implementation of function for bit flips between two binary strings	16
3.4	Implementation of an interpreter for state vectors	17
3.5	Implementation of superpos with qiskit	18
3.6	Implementation of the Hamming distance d_H	19
3.7	Implementation of a greedy sequence-finding algorithm	21
3.8	Implementation of the data model for the TSP solver from OR-Tools	24
3.9	Implementation of the TSP solver from OR-Tools	25
3.10	Implementation of count_simulation	26
3.11	Implementation of function minimal_counts	27
3.12	Gate comparison	28
3.13	Runtime comparison	29
3.14	Gate comparison with different evolution times	30
4.1	Uniform superposition for AND operator	33
4.2	Uniform superposition for OR operator	34
4.3	Uniform superposition for NOT operator	34
4.4	Uniform superposition for a conjunction of functions with disjoint input	35
4.5	First version of superposition for conjunction with one shared variable	35
4.6	Second version of superposition for conjunction with one shared variable	36
4.7	Non-uniform superposition for disjunction of functions with disjoint inputs	37
4.8	Example of a circuit for a formula in DNF	39

List of Figures

4.9	Uniform superposition for $(x_0 \vee x_1) \wedge (x_0 \vee x_2)$	40
4.10	Attempt of a circuit for $(x_0 \vee x_1) \wedge (x_0 \vee x_2) \wedge (x_1 \vee x_2)$	41
4.11	Composable circuit: Uniform superposition for OR	41
4.12	Uniform superposition for OR operator	43
4.13	Uniform superposition for clause of n disjunctions	43

Bibliography

- [1] Richard P. Feynman. “Simulating Physics with Computers”. In: *International Journal of Theoretical Physics* 21.6 (June 1, 1982), pp. 467–488. ISSN: 1572-9575.
- [2] Yuri I. Manin. *Computable and Noncomputable*. Moscow: Sovetskoye Radio, 1980.
- [3] Philip Ball. “Physicists in China Challenge Google’s ‘Quantum Advantage’”. In: *Nature* 588.7838 (2020), pp. 380–380.
- [4] D. Deutsch. “Quantum Theory, the Church-Turing Principle and the Universal Quantum Computer”. In: *Proceedings of the Royal Society of London Series A* 400 (July 1, 1985), pp. 97–117. ISSN: 0080-46301364-5021.
- [5] David Deutsch and Richard Jozsa. “Rapid Solution of Problems by Quantum Computation”. In: *Proceedings of the Royal Society of London Series A* 439 (Dec. 1, 1992), pp. 553–558. ISSN: 0080-46301364-5021.
- [6] Peter W. Shor. “Polynomial Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer”. In: *SIAM Journal on Computing* 26.5 (Oct. 1997), pp. 1484–1509. ISSN: 0097-5397, 1095-7111. arXiv: [quant-ph/9508027](https://arxiv.org/abs/quant-ph/9508027).
- [7] Lov K. Grover. *A Fast Quantum Mechanical Algorithm for Database Search*. arXiv.org. May 29, 1996. URL: <https://arxiv.org/abs/quant-ph/9605043v3> (visited on 06/09/2025).
- [8] J. I. Cirac and P. Zoller. “Quantum Computations with Cold Trapped Ions”. In: *Physical Review Letters* 74.20 (May 15, 1995), pp. 4091–4094.
- [9] Chao Song et al. “10-Qubit Entanglement and Parallel Logic Operations with a Superconducting Circuit”. In: *Physical Review Letters* 119.18 (Nov. 3, 2017), p. 180511. ISSN: 0031-9007, 1079-7114. arXiv: 1703.10302 [quant-ph].
- [10] E. Knill, R. Laflamme, and G. Milburn. *Efficient Linear Optics Quantum Computation*. June 20, 2000. arXiv: [quant-ph/0006088](https://arxiv.org/abs/quant-ph/0006088). URL: <http://arxiv.org/abs/quant-ph/0006088> (visited on 06/09/2025). Pre-published.
- [11] Isaac L. Chuang, Neil Gershenfeld, and Mark Kubinec. “Experimental Implementation of Fast Quantum Searching”. In: *Physical Review Letters* 80.15 (Apr. 13, 1998), pp. 3408–3411.

- [12] Tsubasa Ichikawa et al. “A Comprehensive Survey on Quantum Computer Usage: How Many Qubits Are Employed for What Purposes?” In: *Nature Reviews Physics* 6.6 (May 31, 2024), pp. 345–347. ISSN: 2522-5820. arXiv: 2307.16130 [quant-ph].
- [13] Lennart Binkowski and Heribert Vollmer. *CQ: A High-Level Imperative Classical-Quantum Programming Language*. Bachelor’s Thesis. Leibniz University Hannover. 2025.
- [14] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Higher Education from Cambridge University Press. Dec. 9, 2010. URL: <https://doi.org/10.1017/CB09780511976667> (visited on 05/31/2025).
- [15] D. Cohen. “On Holy Wars and a Plea for Peace”. In: *Computer* 14.10 (Oct. 1981), pp. 48–54. ISSN: 1558-0814.
- [16] Moritz Marwede. *Preparing-Integer-Lists-in-Uniform-Superposition*. 2025.
- [17] Ali Javadi-Abhari et al. *Quantum Computing with Qiskit*. June 19, 2024. arXiv: 2405.08810 [quant-ph]. URL: <http://arxiv.org/abs/2405.08810> (visited on 05/30/2025). Pre-published.
- [18] Abdullah Ash Saki, Mahabubul Alam, and Swaroop Ghosh. *Study of Decoherence in Quantum Computers: A Circuit-Design Perspective*. Apr. 8, 2019. arXiv: 1904.04323 [cs]. URL: <http://arxiv.org/abs/1904.04323> (visited on 05/31/2025). Pre-published.
- [19] Bernhard Korte and Jens Vygen. *Combinatorial Optimization*. Vol. 21. Algorithms and Combinatorics. Berlin, Heidelberg: Springer, 2008. ISBN: 978-3-540-71843-7.
- [20] *OR-Tools*. Google for Developers. URL: <https://developers.google.com/optimization> (visited on 05/31/2025).
- [21] *Traveling Salesperson Problem | OR-Tools*. Google for Developers. URL: <https://developers.google.com/optimization/routing/tsp> (visited on 06/01/2025).
- [22] *Routing Options | OR-Tools | Google for Developers*. URL: https://developers.google.com/optimization/routing/routing_options (visited on 06/02/2025).
- [23] Dirk Van Dalen. *Logic and Structure*. Universitext. Berlin, Heidelberg: Springer, 1994. ISBN: 978-3-540-57839-0 978-3-662-02962-6.
- [24] Stephen A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. New York, NY, USA: Association for Computing Machinery, May 3, 1971, pp. 151–158. ISBN: 978-1-4503-7464-4.
- [25] L. G. Valiant. “The Complexity of Computing the Permanent”. In: *Theoretical Computer Science* 8.2 (Jan. 1, 1979), pp. 189–201. ISSN: 0304-3975.