Institut für Theoretische Physik

Bachelor thesis

Algorithmic Design and Documentation with Qiskit: The Quantum Tree Generator Handbook

Neele Gretchen Hinrichs

10036943

supervised by

Prof. Dr. Tobias Osborne

19th of June 2024

Abstract

This thesis deals with the solution of a well-known NP-complete problem [NC10]. This problem is the 0-1 Knapsack problem. In the paper "A quantum algorithm for the solution of the 0-1 Knapsack problem", to which this paper refers, a quantum method for solving this problem was developed [Wil+]. This quantum method is called Quantum Tree Generator. The implementation of the Quantum Tree Generator and the creation of the 0-1 knapsack algorithm in Qiskit is the goal of this thesis. In the context of this work, it has been possible to achieve this goal. The algorithm is only valid for integer inputs. It is a solution for the simple NP-problem of the 0-1 knapsack problem. The algorithm consists of the Grover algorithm [NC10], which uses the Quantum Tree Generator to prepare the initial state. In this case, the Grover algorithm is used to maximize profits. For the construction of the Quantum Tree Generator, a number of individual quantum methods are required. These include a quantum adder and a quantum subtracter, which take as input a binary number stored in a register and an already known classical number. The construction of these two quantum methods requires a quantum Fourier transform [BA13] and a number of phase patterns. The quantum Fourier transform is a unitary operator for transforming vectors of an orthonormal basis. Another quantum method needed to build the Quantum Tree Generator is the quantum mechanical comparison of a known classical number with a binary number stored in a register. This function is already built into Qiskit and has an output qubit from which the result can be read. These quantum methods are necessary to update the registers of the algorithm and thus define the possible solutions. The implementation of the Quantum Tree Generator consists of a circuit with three registers. These registers are an item register, a capacity register and a profit register. These registers are made up of a number of qubits, which must be adapted to the individual problem. The implementation of the 0-1 knapsack algorithm has the same number of qubits, plus a profit register of a doubled length and an additional qubit for the oracle of the Grover algorithm. The composite knapsack algorithm, similar to the Quantum Tree Generator, uses the method of quantum mechanical comparison. The quantum mechanical comparison is used to compare the measured results with the profit threshold. The limit is updated based on the results and leads to the finding the maximum. At the end of this thesis, the algorithms will be checked for correctness and the Grover iterations will be analysed. During this work, the limitations of the algorithm were also highlighted with regard to the following points. Qiskit can only simulate a limited number of qubits, which severely limits the simulation of the 0-1 knapsack algorithm. To implement the algorithm with six possible items and the following example values, Qiskit needs 27 qubits.

$$\text{I} = [i1, i2, i3, i4, i5, i6] \text{ (items)}, \text{ w} = [4, 1, 5, 6, 2, 3] \text{ (weights)}, \text{ p} = [2, 3, 6, 4, 7, 5] \text{ (profits)},$$
$$\text{W} = 18 \text{ (total weight)}$$

This algorithm needs some time to run on a conventional computer. It is clear that the upper limit of usable qubits on a conventional computer is not much higher [Wil+]. Unfortunately, it is not possible to simulate quantities that show an advantage over the classical solution. The paper, "A quantum algorithm for the solution of the 0-1 Knapsack problem", suggests that at least 600 variables are required [Wil+]. Overall the thesis presents the method of the solution step by step and thus serves as a guide for this procedure. For this reason, no great prior knowledge is required to develop the algorithm using this thesis. This makes it of interest to a wide range of people.

Contents

List of Figures

1 Introduction

Quantum information theory is an important current topic. The largest quantum computer currently available has 1225 qubits [TEC]. To evaluate the performance of a quantum heuristic, 100 to 10000 qubits are required [Wil+]. Therefore, it is not yet possible to the full extent to evaluate the performance in practice. The only possibility is to perform classical simulations of quantum methods. However, these simulations are currently limited to a size of 20 to 30 qubits, which is still too small to be able to assess the performance [Wil+]. The paper published on 10. October 2023, "A quantum algorithm for the solution of the 0-1 Knapsack problem", analyses the performance of the quantum heuristics of the 0-1 Knapsack problem. A high-level simulator was used to evaluate and compare it to the classical solver. The classical solver used here is called COMBO. The results of this paper were a lower memory requirement and a shorter cycle time for the methods based on a Quantum Tree Generator. These results were obtained under the assumption that bits and qubits are comparable. This suggests a possible practical quantum advantage in solving the 0-1 Knapsack problem [Wil+]. The 0-1 Knapsack problem is one of the NP-complete problems [KPP04]. The term NP-complete was introduced by Stephen A. Cook in 1971 [KPP04]. NP-complete problems are some of the hardest problems in the NP class. This is the case if the problem belongs to both the NP class and the hard NP class. The NP class is characterised by the fact that a deterministic computer needs only a polynomial time to solve the problem. The NP-hard class is characterised by an upper limit on the polynomial time. It is assumed that these problems are not efficiently solvable. This is because the solution would require a significant amount of time on real computers. In practice, however, this does not have a major impact, as there are also problems that can be solved in a reasonable time using selected methods [Har][FOR][GS08][KV18]. In the paper mentioned above, a special quantum method was developed to solve such a problem (0-1 Knapsack). It is called the Quantum Tree Generator. This is a kind of decision tree. A node of this decision tree stores the remaining weight and the current profit. If the next node (the child node) stores a weight that is less than or equal to the remaining weight, then an superposition and thus further nodes are formed by the inclusion and the exclusion of this state. The superposition of these states allows an efficient verification of all possible results. The general interest in this algorithm for the 0-1 knapsack problem on a quantum computer is based on the fact that there could be a possible quantum advantage in solving it. For the simulation of a quantum computer on the classical computer, the Qiskit open-source software development kit is highly suitable [IBMc]. The goal of this thesis is therefore to implement the Quantum Tree Generator and to create the solution algorithm for the 0-1 Knapsack problem in Qiskit.

2 Theoretical framework for solving the 0-1 knapsack problem in Qiskit

2.1 What is Qiskit?

The International Business Machines Corporation (IBM) has established Qiskit to develop software for quantum computing services [IBMc]. In 2015, IBM was the first manufacturer to launch a five-qubit quantum computer as a cloud computing service [Man][GS][Marb]. IBM is also working with superconductor qubits. Superconductors have no electrical resistance at extremely low temperatures, which is a prerequisite for the quantum state of the qubits. Microwave photons are used to enable transitions and control the behaviour of the qubits. To make these quantum computers accessible to the public, Qiskit was developed. It is an open source software development system designed to facilitate work with quantum computers. Qiskit allows the simulation of problems using a quantum computer. Users can design and run quantum circuits. In Qiskit, quantum problems can be solved, and quantum circuits and programs can be analysed. Qiskit has many methods and gates that can be used to create a circuit for any problem. Python has proven to be the best programming language for these quantum computers. In order to simulate a quantum computer with this programming language, interfaces, so-called APIs, must be created. These provide the necessary extensions to the programming language. They are usually provided by the respective manufacturers. These APIs are equipped with quantum computer simulators. These perform a simulation on a conventional computer. However, since classical computers are only partially capable of simulating quantum computers, the only way to test quantum programs is with a few qubits. The package manager "pip" is provided to integrate the additional modules necessary for Qiskit into Python [IBMc][Marb].

2.2 0-1 knapsack-problem

The knapsack problem is a well-known optimization problem. It describes the problem of finding the best combination of items, each item has its own weight and profit. In this problem there is a fixed budget or time constraint. Therefore, the best combination is found by selecting the items with the maximum profit without exceeding the budget. There are several variations of the knapsack problem. These variations differ in the parameters of the problem. For example, there is the multidimensional knapsack problem. In this problem the weight of each item is given in a multidimensional vector. Also the limit weight is given in a multidimensional vector and with these parameters the best combination has to be found. The 0-1 knapsack problem is the most common of the variations. In the 0-1 knapsack problem, each item can only be selected once. This means that the selection of items for this problem is either 0, i.e. not included, or 1, i.e. included in the combination [KPP04]. The mathematical representation of the problem is shown below [KPP04].

$$\begin{array}{l} \sum_{i=1}^{n} v_i\, x_i \text{ , has to be maximized.} \\ \sum_{i=1}^{n} w_i\, x_i \leq W, \text{ with } x_i \in \{0,1\} \end{array}$$

The 0-1 knapsack problem is interesting for computational research for several reasons. Due to the fact that this is an NP-complete problem, there is still no algorithm capable of solving the problem correctly in all cases and in polynomial time. However, there is a fully polynomial approximation

method for this problem. It is interesting to note that the input and the difficulty of the Knapsack problem are related. For integer weights and utilities it is only slightly NP-complete, but for rational numbers it is highly NP-complete. A complete polynomial approximation is also given for rational entries [Woj][KPP04].

2.3 Importance of the required algorithm registers

This section discusses the theory of the quantum mechanical solution of the 0-1 Knapsack problem and describes the function of the individual registers. Compared to the classical solver COMBO of the 0-1 Knapsack problem, the quantum solver (QTG-based method) leads to a acceleration of the running time and requires a smaller amount of memory. These facts show that a quantum advantage is possible from 600 variables [Wil+]. A solution algorithm on a quantum computer is therefore of great interest. The solution requires three registers and the tensor product between these registers [Wil+]:

$$\mathcal{H} = \mathcal{H}_{\mathcal{A}} \otimes \mathcal{H}_{\mathcal{B}} \otimes \mathcal{H}_{\mathcal{C}}.$$

These registers have different functions. In the first register $\mathcal{H}_{\mathcal{A}}$ the possible paths in a basic state $|y\rangle^A$ are represented. Each element is assigned to a qubit in this register. The current capacity of the problem is stored in the second register $\mathcal{H}_{\mathcal{B}}$. The quantum state $|x_y\rangle^B$ is used for storage. The last register $\mathcal{H}_{\mathcal{C}}$ represents the current profit of the problem. The place where the profit is stored is a quantum state called $|P_y\rangle^C$. The total number of qubits used in these registers is [Wil+]:

$$q_{total} = n + \lceil log_2 Z \rceil + \lceil log_2 P \rceil.$$

At the beginning of the algorithm, controlled Hadamard gates are used to achieve the direct exclusion of variables that exceed the capacity in advance. For variables that do not exceed the remaining capacity, this gate achieves a superposition of inclusion and exclusion of variables, so that with this algorithm it is possible to check and cover several possibilities at the same time. In this way, for each element, a comparison is made between the capacity of the element and the residual capacity. The remaining capacity is then updated by a quantum subtractor. This process is described by the following operator [Wil+]:

$$U_m^B := C_m^A SUB_{zm}$$

The next step is to update the profits for each path. In this step, the quantum adder is used. The operator for this step is defined as follows [Wil+]:

$$U_m^C := C_m^A ADD_{pm}$$

The quantum Fourier transform is used by both the quantum adder and the quantum subtractor. It allows the addition or subtraction of all the entanglements of the elements used. The objective of the algorithm is to maximise the win on the win register. To achieve this goal, it is necessary to define a fixed threshold. With this fixed threshold, each profit of a possible solution can be compared. In this way the maximum value can be determined. The Quantum Tree Generator is used as a simulator. This decision tree uses breadth search to determine the best path [Wil+].

2.4 Quantum Fourier transform

The quantum Fourier transform is a linear transformation of qubits. It is an analogue of the classical discrete Fourier Transform. It is the only known algorithm in the field of quantum computing that has an exponential advantage over the classical algorithm. The quantum Fourier transform can therefore be solved on a quantum computer in polynomial time. The number of n-qubit gates needed for this transformation can be described by $O(n^2)$ [BA13]. Uniformity is another property of the quantum Fourier transform. The mathematical definition of the transformation on the Hilbert space $\mathbb{C}^{2^n}$ is the following [BA13].

$$F\left|j\right\rangle = \frac{1}{\sqrt{2^n}} \cdot \sum_{k=0}^{2^n-1} \cdot e^{2 \cdot \pi \cdot i \cdot \frac{j \cdot k}{2^n}} \left|k\right\rangle$$

The inverse quantum Fourier transform is similarly represented as follows [BA13].

$$F\left|j\right\rangle = \frac{1}{\sqrt{2^n}} \cdot \sum_{k=0}^{2^n-1} \cdot e^{-2 \cdot \pi \cdot i \cdot \frac{j \cdot k}{2^n}} \left|k\right\rangle$$

This quantum Fourier transform is used in many algorithms in quantum computing. In the case of the 0-1 Knapsack problem it is of great importance for the quantum adder and subtracter. The quantum Fourier transform is a composition of different gates. These gates can be represented in the form of unitary matrices. The gates that make up the quantum Fourier transform are the Hadamard gate, the swap gate and the phase gates. The unitary matrices to this gates are presented below [IBMb].

$$R_k = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{\frac{2 \cdot i \cdot \pi}{2^k}} \end{pmatrix} \; Swap = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \; H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Applying the Hadamard gate to a qubit k_1 looks like this [BA13].

$$H\left|k_1\right\rangle = \frac{\left|0\right\rangle + e^{2 \cdot \pi \cdot i[0.k_1]}\left|1\right\rangle}{\sqrt{2}}$$

For $k_1 = 1$ the exponential term gives -1 and for k = 0 it gives +1. It can be seen that the Hadamard gate gives a superposition of 0 and 1. This superposition makes it possible to check several results at the same time. For this reason, the Hadamard gate is not only used in the quantum Fourier transform algorithm. If we add the phase gates R_2 to R_n to this state, we obtain an extended term [BA13].

$$H\left|k_1\right\rangle = \frac{\left|0\right\rangle + e^{2 \cdot \pi \cdot i[0.k_1 k_2 \ldots k_n]}\left|1\right\rangle}{\sqrt{2}}$$

This phase gates have the task of shifting the phase around the Z-axis depending on the choice of k. If $k = 0$ or $k = 1$ it is a rotation of $\frac{\pi}{2}$ and if $k = 2$ it is a rotation of $\frac{\pi}{4}$. The function of the swap gate is to switch between two states [IBMb].

$$S\left|\phi_i\right\rangle\left|\phi_j\right\rangle = S\left|\phi_j\right\rangle\left|\phi_i\right\rangle$$

This state change function can be implemented by three CNOT gates. Putting all the gates and their functions together in a quantum Fourier transform gives the circuit in FIG.1.

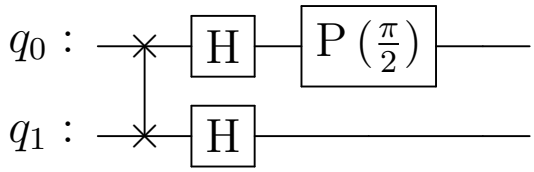


FIG. 1: Circuit of the quantum Fourier transform in Qiskit [UKb]

The application of the quantum Fourier transform to the 0-1 knapsack problem can be found in the quantum methods of the quantum adder and the quantum subtracter.

2.5 Grover algorithm

The Grover algorithm is a search algorithm for unsorted databases with N records. The brute force algorithm has an average running time of $O(\frac{N}{2})$. In the worst case, the runtime is $O(N)$. This is significantly worse than the performance of the Grover algorithm. In this case the running time is $O(\sqrt{N})$, which corresponds to a quadratic acceleration [NC10]. The Grover algorithm was developed by Lov Grover in 1996 [Gro]. This one of the first quantum algorithms to show a quantum advantage over brute force algorithms. The Grover algorithm is a probabilistic algorithm, which has the property that the probability of the correct result can decrease when the algorithm is applied repeatedly. The main component of the Grover algorithm is the oracle. The task of the oracle is to identify and mark solutions to a problem. It is important to adapt the oracle to the specific problem being worked on. However, the basic function of the oracle remains unchanged and can be described as follows [NC10].

$$|x\rangle \ \rightarrow \mathrm{O}\,(-1)^{f(x)}\ |x\rangle$$

This shows that the solution is characterised by a phase shift. The solution to the 0-1 Knapsack problem is the one with the highest profit. Another characteristic of the oracle, and therefore of the correct solution, is the search for the maximum value. In this case the oracle is given by the comparison in the profit register. Characteristic for this comparison is the indication of a limit value. If the measured profit is greater than the current limit, the limit is updated. If this is not the case, the iteration is repeated. The Grover iteration can be described as a rotation [NC10]:

$$G = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

The Grover iteration can also be presented in a different way [NC10]:

$$G \ = \ (2\ |\psi\rangle\ \langle\psi|\ -\ I)\,\mathrm{O}.$$

Theta is the angle of rotation in the matrix representation. The Grover iteration also consists of two reflections. These are easy to see in the two-dimensional representation. The state of the final solution is shown on the y-axis. On the x-axis there is an orthogonal state. The state of all possible superimposed solutions is also rotated by the angle θ on the x-axis and then by the angle 2θ on the axis of the initial state by the operation $2\ |\psi\rangle\ \langle\psi| - I$. This process is repeated several times so that the superimposed state gets closer and closer to the solution state. In this way the solution of the algorithm is finally reached. The Grover algorithm therefore uses amplitude amplification, which consists of $Q = A \cdot S_0 \cdot A^{-1} \cdot S_f$. This amplitude amplification stretches the amplitude of the probability of the searched item. This is a Grover iteration. In the case of the 0-1 knapsack problem A is the Quantum Tree Generator G. S_0 is the reflection about the all 0 state and S_f is the oracle which flips the qubit of the right solution. The Grover algorithm is therefore used in the 0-1 Knapsack problem to select the optimal solution from the possible solutions provided by the Quantum Tree generator [Wil+][Mara][unk][Wri][IBMa]. When using the Grover algorithm to solve a problem, the number of Grover iterations is limited and depends on the specific circumstances of the problem. These are the subject of the analytical part of this thesis.

2.6 Quantum Tree Generator

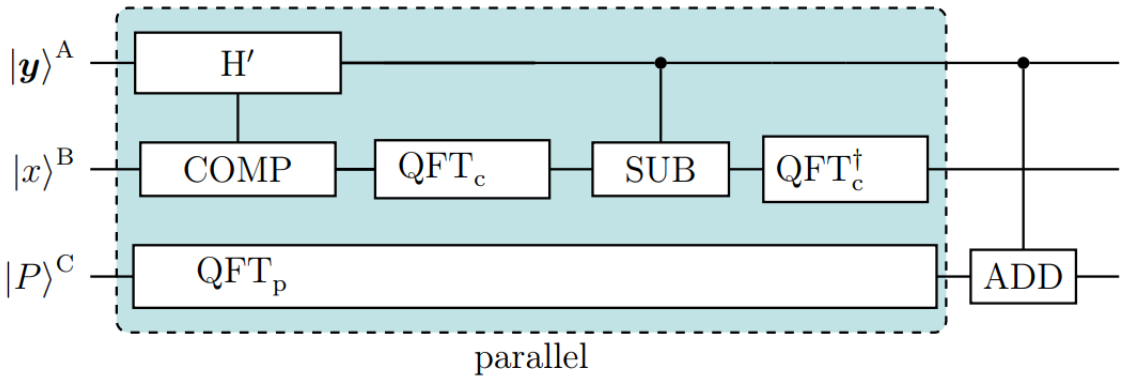


FIG. 2: Rough expansion of the circuit for a Quantum Tree Generator [Wil+]

The Quantum Tree Generator is an operator that prepares the states for the oracle. Components of this operator are Hadamard gates, a comperator, a subtractor and an adder. The rough structure is showed in FIG.2. The way the Quantum Tree Generator works can be understood by looking at the gates step by step. In every register of the items there is a controlled Hadamard gate. The Hadamard gate realises a superposition of two states [IBMb]. Therefore, if the Hadamard gate is applied to a state, a superposition of one and zero will exist. In our case this is the superposition of including or excluding the item. The Hadamard gate will be controlled by the comparison in capacity register $|c\rangle$. If there is a probability that the item will be used then there will be a subtraction in register $|c\rangle$. This subtraction subtracts the weight of the item from the total weight. The profit of the item will be recorded in the profit register $|p\rangle$. This will be done for all combinations of the item. So, only the suitable combinations will be selected [Wil+]. The gates for the comparison, adder and subtractor are programmed to have a constant component interacting with a qubit. The adder adds up a classical constant with a quantum integer as same as the subtractor and the comparison compares a quantum integer

with a classical constant. In this problem there is always a known number, so this type of comparison and operation is sufficient. The reason for this is to be found in the initial conditions associated with the problem. The following section is a detailed description of the individual components of the Quantum Tree Generator.

2.7 Quantum adder and subtractor with a classical constant

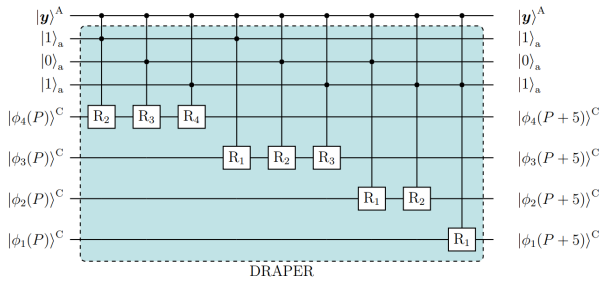


FIG. 3: Type of the draper adder [Wil+]

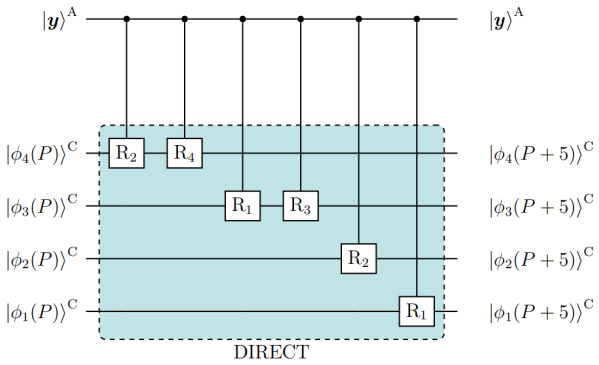


FIG. 4: Type of the direct adder [Wil+]

There are two types of adder that add a quantum integer to a classical number. There is the draper adder and the direct adder [Wil+]. The basic concept of the adders are the same. The quantum circuit only consists of a quantum register, which stores the digits of the quantum integer, and a quantum register, which represents a superposition of one and zero. This superposition makes it possible to obtain two states simultaneously, the state of adding these two numbers and the state of not adding them. The adder ends up producing both solutions simultaneously. The classical number is related to the quantum circuit by the fact that the added phases in the circuit depend on the classical number. The difference between the Draper adder and the direct adder is that in the direct adder only the phases associated with a digit one of the classical number are applied to the quantum circuit. The draper adder also uses phases that have no effect. The same applies to the subtractor. The phase gates that determine the addition can be implemented with the controlled phase gate in Qiskit. The phase gates that are added to the circuit are the same as in the quantum Fourier transform. They are always added to the quantum circuit in a certain pattern, which can be seen in FIG.3 and in FIG.4. The index k of the phase gate is counted up from one to the length of the register. For the last qubit of the register k=1 and for the first k is equal to the length of the register. Each qubit of the register is always assigned to all phase gates from one to the corresponding k. The assignment is made in such a way that the gates with the largest k to the smallest k are added first. The phase gates of the first qubit of the register are assigned to the first qubit of the register of the known number. Proceed in the same way. For this reason, both registers must be of the same size. The register of the known number is irrelevant for the type of direct addition used here, since only the places of the entries that have the value 1 are important for the following process. Only those phase gates are added that are connected to a qubit from the register of the known number that is set to 1. The quantum register should always have one more qubit than the number of digits in the classical component. This is because the quantum register stores the number to be added, but also the final result of the sum. Furthermore, the larger the classical constant, the more phase gates must be used for the addition. This shows that there is a separate quantum circuit

exists for each addition. In the implementation of the knapsack problem only the direct adder is used. Another component of the adder is the quantum Fourier transform. The quantum Fourier transform gate is pre-programmed in Qiskit and is placed before the phase gates of the adder [IBMb]. In addition, the inverse of the quantum Fourier transform must be applied at the end of the adder. The subtraction is based on the same principle and structure except for the prefix of the phase R_k. If the prefix is changed to a minus, then the addition becomes a subtraction [Wil+].

2.8 Comparison with a classical constant

Comparison with a classical constant is a pre-programmed operation in Qiskit [IBMb]. The pre-programmed operation in Qiskit is called integer comparison. This is the necessary comparison for the Knapsack problem, because for each comparison there is an already known parameter. Each element has a certain profit and a certain weight. The size of the register used for the comparison is twice the length of the binary number stored in the register. The integer comparator works in such a way that the result of the comparison is stored as zero or one in a qubit of the register and can be retrieved at any time. If the value stored in the register is higher than the classical number entered, a one is stored, otherwise a zero [UKa]. The operator performing the comparison is defined as [IBMb]:

$$|i\rangle_n \, |0\rangle \mapsto |i\rangle_n \, |i \geq L\rangle.$$

In this definition $|i\rangle_n$ is the basis state and L is the classical integer. In the Quantum Tree Generator the comparison is used to compare the given weights with the current weight stored in the register. In the Grover algorithm itself, the comparison operation is used to compare profits. There it is used as an optimisation function by comparing the gains stored in the register with the current threshold value.

3 Implementation in Qiskit

The Qiskit version used in this thesis is the following:

{'qiskit-terra': '0.25.2', 'qiskit': '0.44.2', 'qiskit-aer': '0.12.2', 'qiskit-ignis': '0.7.1', 'qiskit-ibmq-provider': '0.20.2', 'qiskit-nature': None, 'qiskit-finance': None, 'qiskit-optimization': None, 'qiskit-machine-learning': None}

In this section a connection is made between the theoretical part and the practical part through the implementation.

3.1 Required libaries

```
import qiskit
import numpy as np
import matplotlib.pyplot as plt
import math
from numpy import pi
from qiskit import *
from qiskit import QuantumCircuit, transpile
from qiskit_aer import AerSimulator
from qiskit.visualization import plot_histogram
from qiskit.circuit.library import QFT
from qiskit.circuit.library import CPhaseGate
from qiskit.circuit.library import CZGate
from qiskit.circuit.library import CSGate
from qiskit.circuit.library import DraperQFTAdder
from qiskit.circuit import QuantumRegister
from qiskit.circuit import ClassicalRegister
from qiskit import QuantumRegister, ClassicalRegister
from qiskit import QuantumCircuit, execute
from qiskit.circuit.library import IntegerComparator
from qiskit import Aer
from qiskit import QuantumCircuit
from qiskit_algorithms import AmplificationProblem
from qiskit.tools.visualization import circuit_drawer
```

Listing 1: the required libraries

These libraries are a necessary prerequisite for the implementation of the algorithm. The Aer simulator is used to solve the problem. This is a powerful simulator that includes realistic noise models [IBMb]. For the implementation of the 0-1 knapsack problem there are two useful substructures that are preconfigured in Qiskit. These are the QFT, the quantum Fourier transform, and the integer comperator. The other imports are basic structures to build a quantum circuit.

3.2 Implementation of the quantum adder

To illustrate the structure of the adder, a numerical example is shown that adds the binary number three and the non-binary number three (FIG.5). One of these numbers is a classical constant which determines the phase gates to be inserted. When the quantum register is measured, the result is divided into two solutions. In the first case the solution is three and in the second case it is six. This is due to the Hadamard gate. The Hadamard gate creates a superposition of the one and the zero. This shows that the two numbers are superimposed by adding or not adding. The implementation of this circuit in Python is described below.

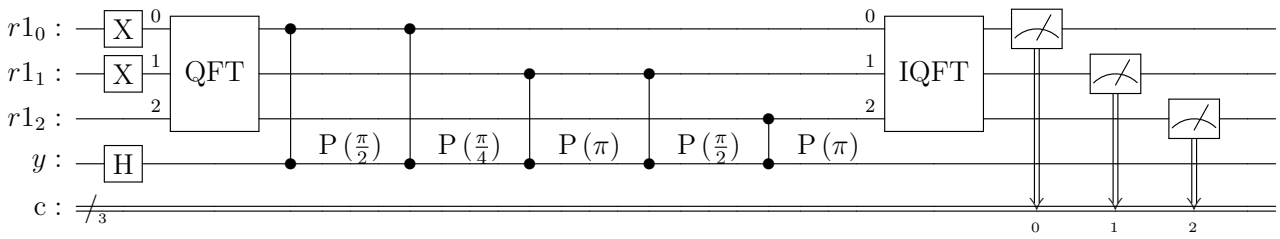


FIG. 5: Circuit of the quantum adder in Qiskit

```
#Implementation of a function which adds up two numbers, one classical and
    one quantum:
def adder(number1,number2):

    #First, the classic decimal number is converted to a binary number
    number1=dez2bin(number1)

    #The two input numbers are stored in an associated array:
    array1=[]
    array2=[]
    i=0
    j=0
    a=len(str(number1))
    b=len(str(number2))
    while i<a+1:
        array1.append(number1%10)
        number1=math.floor(number1/10)
        i=i+1
    m=0
    n=0
    while m<b+1:
        array2.append(number2%10)
        number2=math.floor(number2/10)
        m=m+1

    #This is where the arrays are aligned to a common length:
    length=len(array2)
    length2=len(array1)
    if length!=length2:
        if length2<length:
            while length2!=length:
                array1.append(0)
                length2=length2+1
        if length<length2:
            while length2!=length:
                array2.append(0)
                length=length+1

    #In this step, the array of classical numbers is inverted once:
    array1 = array1[::-1]

    #Definition of the quantum register and the quantum circuit:
    register1 = QuantumRegister(length, "r1")
    register2= QuantumRegister(1, "y")
    register4=ClassicalRegister(length,"c")
```

```
    cir = QuantumCircuit( register1,register2,register4, name="cir")

    #The Hadamard gate generates the superposition in the quantum register y
    :
    cir.h(length)

    #The quantum integer is stored in the register:
    for l in range(0,length):
        if array2[l]==1:
            cir.x(l)

    #Defining the quantum Fourier transform and inserting it into the
    circuit:
    qft = QFT(num_qubits=length).to_gate()
    k=0
    list=[]
    while k<length:
        list.append(k)
        k=k+1
    cir.append(qft, qargs=list)

    #Inserting the appropriate phase gate into the quantum circuit:
    for g in range(0,length):
        j=0
        for h in range(g,length):
            if array1[h]==1:
                phase=CPhaseGate(np.pi/(2**(j)), label=None)
                cir.append(phase,[length,g])
            j=j+1

    #Definition and application of the inverse quantum Fourier transform:
    qftinverse = QFT(num_qubits=length,inverse=True).to_gate()
    cir.append(qftinverse, qargs=list)


    #Measuring the register:
    for p in range(0,length):
        cir.measure(register1[p],register4[p])

    return cir
```

Listing 2: direct quantum adder

This code can be divided into three parts. The first part is responsible for converting the input into the desired format. The second part builds the quantum circuit with the appropriate quantum patterns and the third part performs the measurement. The first part of the code converts, the two numbers, where number 1 is the classic number, into an array. In this step, the classical input is first converted into a binary number. This conversion is done with the function dez2bin. This function is defined in the bachelor thesis textbook. The two arrays are then made equal, and the length of the register is defined as the total length plus one. The array containing the classical number is then inverted once, so that the indices are correctly aligned according to the definition of the direct adder. After these adjustments it is necessary to define the required registers. To add two numbers, two quantum registers and one classical register are required. The first register is the register that stores the quantum integer and the result at the end of the addition. The second register is used to generate the superposition of the addition by a Hadamard gate. The classical register then measures the value of the qubits included in

the measurement. The length of the register depends on whether the classical number entered is larger or smaller than the number stored in the register. If the classical number entered is smaller than the stored quantum integer, the length of the register is equal to the length of the stored quantum integer plus one. However, if the binary length of the classical number is greater than the binary length of the quantum integer, the length of the register is equal to the length of the classical number plus one. The length of the classical register is equal to the length of the first quantum register, since the result to be measured is stored in this register at the end. The second quantum register consists of a single qubit, which creates the superposition and is connected to the first quantum register by the controlled phase gates. After defining the registers, the array containing the quantum integer is first run through. An X-gate is inserted into the first register at each location where it contains a 1. This X-gate sets the state of the qubit from 0 to 1. In this way the quantum integer is transferred to the register and stored there. Next, the quantum Fourier transform is defined as a gate and inserted into the first register. The length of the quantum Fourier transform is the total length of the first register. This is followed by a double for loop. This is responsible for generating the phase gates. Since this is a direct adder and only those phase gates are added that are linked to a one in the classic number, the array is iterated which stores the classic number and each place is checked for the value 1. To do this, each qubit in the first register is compared to its position in the classical number array, and the phase gates are inserted into the circuit according to the structure defined above. Once all the required phase gates have been added, the inverse of the quantum Fourier transform is generated as a gate and inserted into the first register. The insertion of the inverse quantum Fourier transform is necessary for the transformation back to the original representation. The result stored in the first register is then measured with the classic register. These results can then be run on the selected Aer simulator to obtain the measured counts.

3.3 Implementation of the quantum subtractor

The implementation of the subtractor is identical to that of the adder. The only difference is the sign of the phase gate. As an illustration of this implementation, the numerical example of a binary three and a non-binary three is also given here (FIG.6). The two solutions of the measurement in this case are the zero and the three due to the superposition of the subtraction and the non-subtraction.

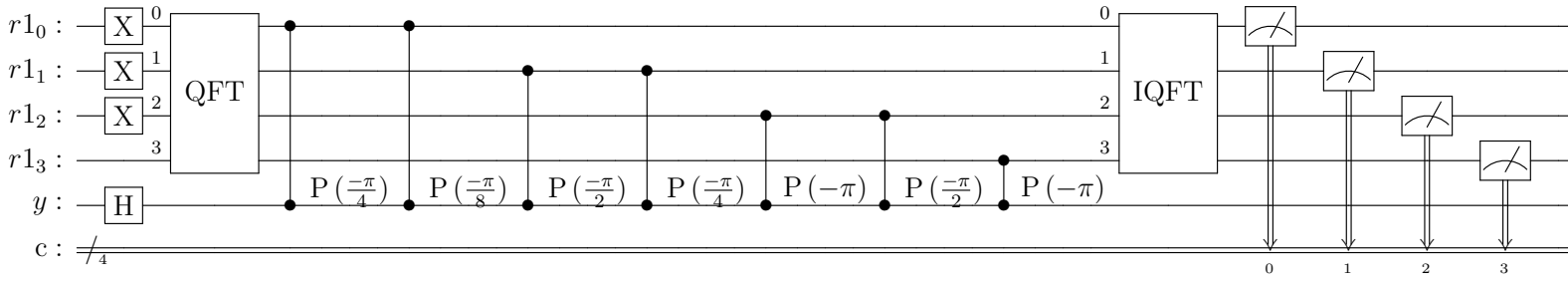


FIG. 6: Circuit of the quantum subtractor in Qiskit

```
#Implementation of a function which subtract two numbers, one classical and
    one quantum:
def sub(number1,number2):

```

```
    #First, the classic decimal number is converted to a binary number
    number1=dez2bin(number1)

    #The two input numbers are stored in an associated array:
    array1=[]
    array2=[]
    i=0
    j=0
    a=len(str(number1))
    b=len(str(number2))
    while i<a+1:
        array1.append(number1%10)
        number1=math.floor(number1/10)
        i=i+1
    m=0
    n=0
    while m<b+1:
        array2.append(number2%10)
        number2=math.floor(number2/10)
        m=m+1

    #This is where the arrays are aligned to a common length:
    length=len(array2)
    length2=len(array1)
    if length!=length2:
        if length2<length:
            while length2!=length:
                array1.append(0)
                length2=length2+1
        if length< length2:
            while length2!=length:
                array2.append(0)
                length=length+1

    #In this step, the array of classical numbers is inverted once:
    array1 = array1[::-1]

    #Definition of the quantum register and the quantum circuit:
    register1 = QuantumRegister(length, "r1")
    register2= QuantumRegister(1, "y")
    register4=ClassicalRegister(length,"c")
    cir = QuantumCircuit( register1,register2,register4, name="cir")

    #The Hadamard gate generates the superposition in the quantum register y
    :
    cir.h(length)

    #The quantum integer is stored in the register:
    for l in range(0,length):
        if array2[l]==1:
            cir.x(l)

    #Defining the quantum Fourier transform and inserting it into the
    circuit:
    qft = QFT(num_qubits=length).to_gate()
    k=0
    list=[]
    while k<length:
```

```
60          list.append(k)
61          k=k+1
62      cir.append(qft, qargs=list)
63
64      #Inserting the appropriate phase gate into the quantum circuit:
65      for g in range(0,length):
66          j=0
67          for h in range(g,length):
68              if array1[h]==1:
69                  phase=CPhaseGate(-(np.pi/(2**(j))), label=None)
70                  cir.append(phase,[length,g])
71              j=j+1
72
73      #Definition and application of the inverse quantum Fourier transform:
74      qftinverse = QFT(num_qubits=length,inverse=True).to_gate()
75      cir.append(qftinverse, qargs=list)
76
77
78      #Measuring the register:
79      for p in range(0,length):
80          cir.measure(register1[p],register4[p])
81
82      return cir
```

Listing 3: direct quantum subtractor

This function, which is a quantum subtraction, is encoded in the same way as the quantum addition. The difference in the code is that the phase set by the phase gate is negative.

3.4 Implementation of the Comparison with a classical constant

As mentioned above the preprogrammed integer comperator is needed for the comparison with the classical constant. There are two different registers that have a built-in comparison. These two comparisons have different requirements. The first register to be highlighted is the register of item weights. The integer comperator can switch between less than and greater than or equal to. This change can be achieved by changing an input parameter from True to False. In our case the True parameter is important because in both registers it is the non-binary input parameter that has to be smaller than the binary parameter that is built into the circuit by gates. The main comparison in the weight register is that the current weight on the register must not exceed the total limit weight. In the profit register, on the other hand, a comparison is made with regard to a limit value for the maximum profit. Example circuits for the integer comparison are shown in FIG.7.

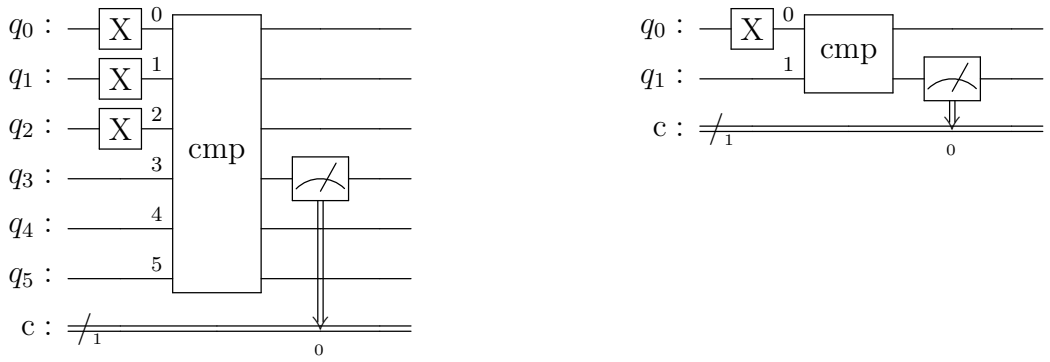


FIG. 7: Circuit of the integer comparison in Qiskit.

In the first circuit there is the comparison of seven and four which gives the result of $\{'1' : 1\}$ after measuring the qubit that stores the result. This is because four is less than seven. The second circuit compares the two numbers three and eight. Unlike the example on the right, this provides the result of $\{'0' : 1\}$ after measuring because eight is greater than three. It is important to note that the length of the comparator has to be at least twice the binary length of the number on the register. Otherwise the result will be distorted. The qubit that stores the result is located just after the one that stores the number. The Python implementation for this is as follows.

```
#Implementation of a function which compares two numbers, one classical and one quantum:
def comparison(number,classical):

    #Define the Aer simulator:
    backend = Aer.get_backend('aer_simulator')

    #The number entered in binary format is converted to an array:
    list1=[]
    a2=len(str(number))
    i=0
    while i<a2:
        list1.append(number%10)
        number=math.floor(number/10)
        i=i+1

    length=len(list1)

    #The order of the array is reversed:
    res = list1[::-1]

    #Definition of quantum registers and their assembly into a quantum circuit:
    q = QuantumRegister(length*2,'q')
    c = ClassicalRegister(1,'c')
    circuit = QuantumCircuit(q,c)

    #Definition of the integer comparator:
    comparator1 = IntegerComparator(num_state_qubits=length, value=classical, geq=True)

```

```
    #This step stores the quantum integer in the register:
    for m in range(0,length):
        if list1[m]==1:
            circuit.x(m)
    #Inserting the integer comparator into the quantum circuit:
    circuit = circuit.compose(comparator1)

    #The register is measured and the result is displayed:
    circuit.measure(q[length],c[0])
    job = execute(circuit, backend, shots=1)
    counts = job.result().get_counts()
    print(counts)


    return circuit
```

Listing 4: comperator

The implementation of the integer comparison can also be divided into three parts. The input to this function is a classical number on the one hand and a quantum integer in binary representation on the other. In the first part, the quantum integer is transferred to an array. This can be used to read the positions of the ones in this binary number. Due to the register representation, this array must be inverted before the positions can be retrieved. The second part defines the registers required for the comparison. Both a quantum register and a classical register are required. The length of the first quantum register is, as mentioned above, twice as long as the binary length of the number stored in the quantum register. In contrast the classical register is only one classical bit long. This is because the integer comperator stores the result as a truth value of zero or one on one qubit of the first register. So one classical bit is enough to measure the result. From these registers the quantum circuit is formed, which follows the definition of the integer comperator. This Qiskit preprogrammed integer comperator operation has three relevant inputs. The first input is the length of the quantum integer stored in the register in binary format. The second input is the classical number with which the quantum integer is to be compared. The third input is a Boolean value which determines the type of comparison. If the Boolean value is true, the result is one if the quantum integer is greater than or equal to the classical number. If the Boolean is false, the result is one if the quantum integer is smaller than the classical number. In this work only the case of the truth value True is used, because in the comparison on the weight register as well as on the profit register the gate controlled by the comparison result is to be added, if the classical number is smaller than the quantum integer stored on the register. This is also the desired type of comparison, since a controlled gate is only added if the controlling qubit is 1. In the next step, the array is traversed and an X-gate is attached to the corresponding qubit at each position where the array contains a 1. In this way, the quantum integer is transferred and stored in binary form in the quantum register. The integer comparator is then integrated into the quantum circuit. Subsequently, the value stored in the qubit at the exposition of the length of the stored number is measured on the classic register and can be output as such.

3.5 Implementation of the Quantum Tree Generator

By implementing the individual components, the Quantum Tree Generator can now be created. The figures FIG.8, FIG.9, FIG.10 and FIG.11 show the Quantum Tree Generator for the example with four possible elements. The individual values of the example are specified as shown below.

$$\begin{aligned} \mathrm{I} &= [i1, i2, i3, i4] \\ \mathrm{w} &= [4, 1, 5, 6] \\ \mathrm{p} &= [2, 3, 6, 4] \\ \mathrm{W} &= 13 \end{aligned}$$

Part of the structure of the Quantum Tree Generator is shown in the figure below. For each element there is an integer comparison, a quantum subtraction in the weight register and a quantum addition in the profit register. The integer comparison has to be inverted after the result has been called, so that the qubits used are released and the following subtraction does not lead to a wrong result. By using this special gate arrangement, the Quantum Tree Generator is able to select all possible solutions from all results.

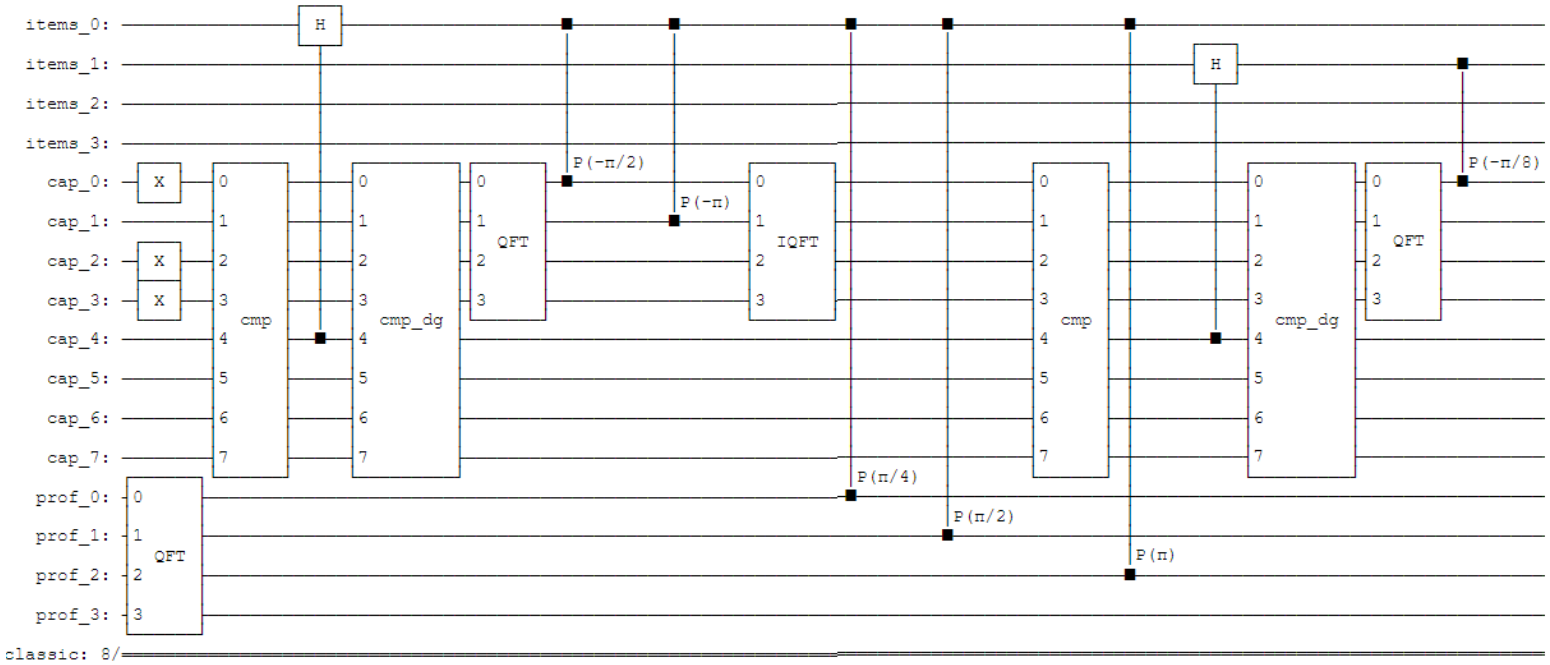


FIG. 8: First part of the circuit of the Quantum Tree Generator.

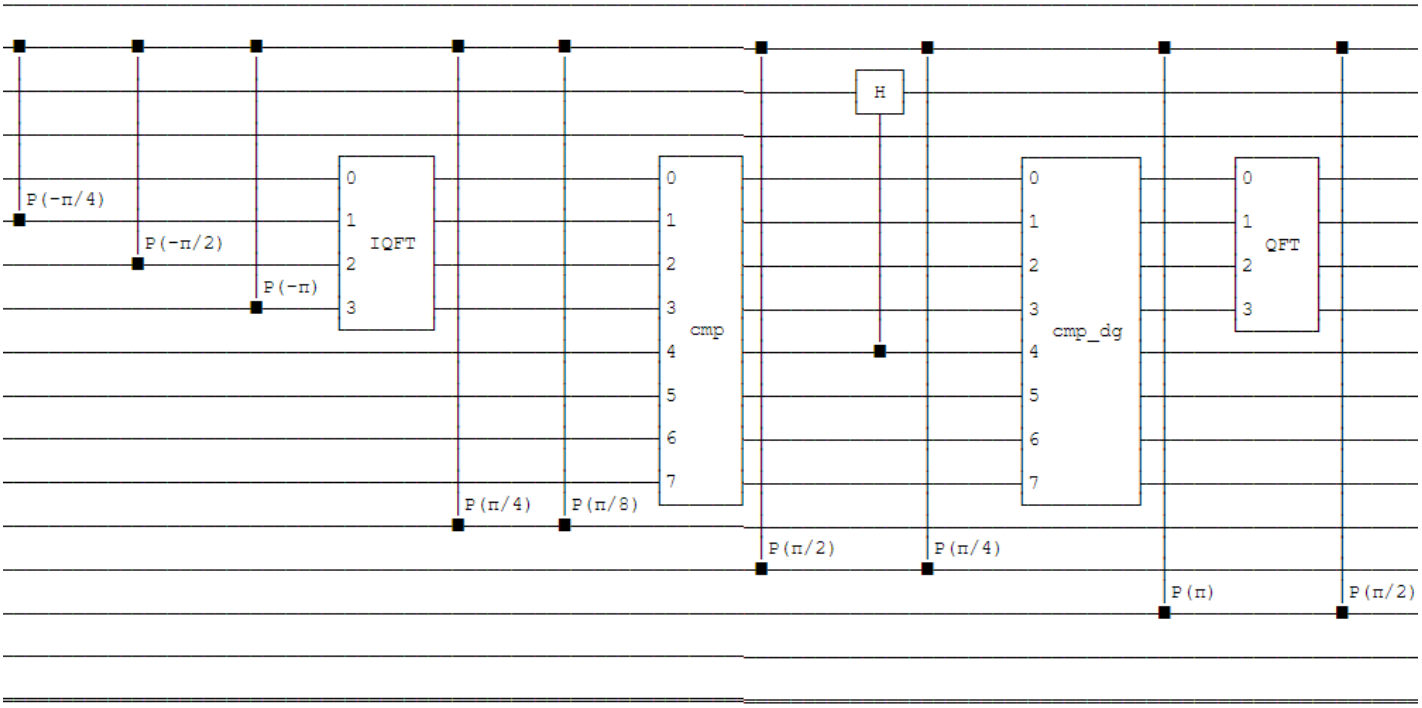


FIG. 9: Second part of the circuit of the Quantum Tree Generator.

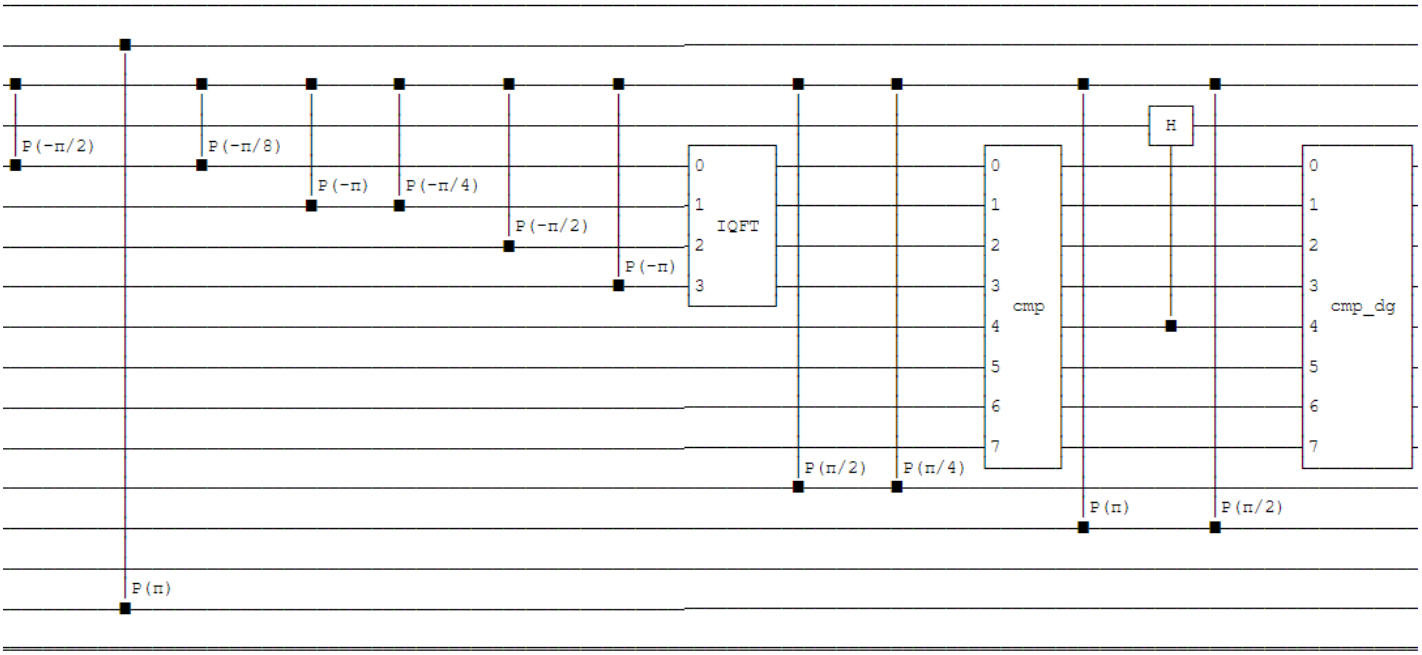


FIG. 10: Third part of the circuit of the Quantum Tree Generator.

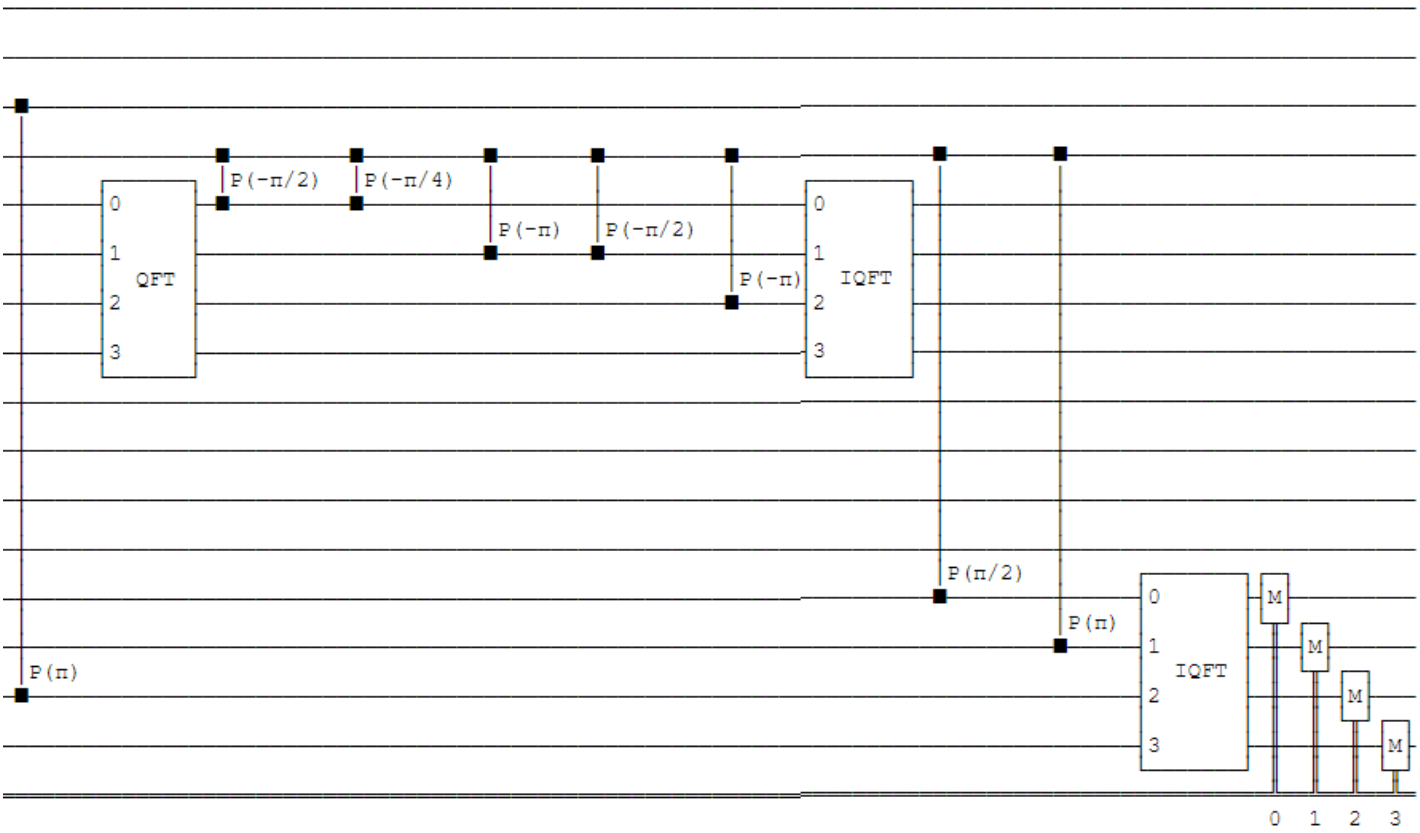


FIG. 11: Fourth part of the circuit of the Quantum Tree Generator.

```
#Implementation of a function which creates the Quantum Tree Generator:
def qtg(items, weights, profit, total_w ):

#Prepare the values:
    total_wb=dez2bin(total_w)
    total_wlen=len(str(total_wb))
    len_prof=len(profit)
    len_weight=len(weights)
    sum_prof=0

    for i in range(0,len_prof):
        sum_prof=sum_prof+profit[i]

    sum_prof_b=dez2bin(sum_prof)
    sum_prof_l=len(str(sum_prof_b))

    length_add=sum_prof_l
    array_total=[]
    m=0
    b=total_wlen

    while m<b+1:
        array_total.append(total_wb%10)
        total_wb=math.floor(total_wb/10)
        m=m+1
    length=len(array_total)-1

#Define the registers of the circuit:
    reg_i=QuantumRegister(len(items),'items')
```

```
30      reg_cap=QuantumRegister(total_wlen*2,'capacity')
31      reg_prof=QuantumRegister(length_add,'profit')
32      qtg = QuantumCircuit(reg_i,reg_cap,reg_prof)
33
34  #Define the initial state of the capacity register:
35      for j in range (0,length):
36          if array_total[j]==1:
37              qtg.x(reg_cap[j])
38
39  #Define the QFT gate of the profit register:
40      qft_add = QFT(num_qubits=length_add).to_gate()
41      k_add=0
42      list_add=[]
43
44      while k_add<length_add:
45          list_add.append(k_add)
46          k_add=k_add+1
47
48      qtg.append(qft_add,reg_prof[list_add])
49
50  #Start of the while-loop:
51      z=0
52      while z<len_prof:
53
54  #Define the actaul weights and profits after every loop:
55          weight_actual=weights[z]
56          weight_actual_b=dez2bin(weight_actual)
57          weight_actual_l=len(str(weight_actual_b))
58          prof_actual=profit[z]
59          prof_actual_b=dez2bin(prof_actual)
60          prof_actual_l=len(str(prof_actual_b))
61
62  #Define the comparison of the capacity register:
63          comparator_qtg = IntegerComparator(num_state_qubits=total_wlen,
    value=weight_actual, geq=True)
64          comparator_qtg_inverse= IntegerComparator(num_state_qubits=
    total_wlen, value=weight_actual, geq=True).inverse()
65          helper_array=[]
66          v=0
67
68          while v<(total_wlen*2):
69              helper_array.append(reg_cap[v])
70              v=v+1
71
72          qtg = qtg.compose(comparator_qtg,helper_array)
73          qtg.ch(reg_cap[total_wlen],reg_i[z])
74          qtg = qtg.compose(comparator_qtg_inverse,helper_array)
75
76  #Define the subtraction of the capacity register:
77          i_sub_actual=0
78          array1_sub_actual=[]
79          a_sub_actual= weight_actual_l
80
81          while i_sub_actual<a_sub_actual+1:
82              array1_sub_actual.append(weight_actual_b%10)
83              weight_actual_b=math.floor(weight_actual_b/10)
84              i_sub_actual=i_sub_actual+1
85
```

```
        length2_sub_actual=len(array1_sub_actual)

        if length!=length2_sub_actual:
            if length2_sub_actual<length:
                while length2_sub_actual!=length:
                    array1_sub_actual.append(0)
                    length2_sub_actual=length2_sub_actual+1

        array1_sub_actual= array1_sub_actual[::-1]
        qft = QFT(num_qubits=length).to_gate()
        u=0
        list_actual=[]

        while u<length:
            list_actual.append(u)
            u=u+1

        qtg.append(qft,reg_cap[list_actual])

        for g in range(0,length):
            f=0
            for h in range(g,length):
                if array1_sub_actual[h]==1:
                    phase=CPhaseGate(-(np.pi/(2**(f))), label=None)
                    qtg.append(phase,[reg_i[z],reg_cap[g]])
                f=f+1

        qftinverse = QFT(num_qubits=length,inverse=True).to_gate()
        qtg.append(qftinverse, reg_cap[list_actual])

#Define the addition of the profit register:
        s=0
        array1_add=[]
        a_add= prof_actual_l

        while s<a_add+1:
            array1_add.append(prof_actual_b%10)
            prof_actual_b=math.floor(prof_actual_b/10)
            s=s+1

        length2_add=len(array1_add)

        if length_add!=length2_add:
            if length2_add<length_add:
                 while length2_add!=length_add:
                        array1_add.append(0)
                        length2_add=length2_add+1

        array1_add = array1_add[::-1]

        for g_add in range(0,length_add):
            j_add=0
            for h_add in range(g_add,length_add):
                if array1_add[h_add]==1:
                    phase_add=CPhaseGate(np.pi/(2**(j_add)), label=None)
                    qtg.append(phase_add,[reg_i[z],reg_prof[g_add]])
                j_add=j_add+1
        z=z+1
```

```
144
145 #After the loop ends:
146     qftinverse_add = QFT(num_qubits=length_add,inverse=True).to_gate()
147     qtg.append(qftinverse_add, reg_prof[list_add])
148
149 #If the QTG should be returned as a gate:
150     tree=qtg.to_gate(label='QTG')
151
152
153
154     return tree
```

Listing 5: Quantum Tree Generator

Above is the Python code implementing the Quantum Tree Generator. A function was implemented that takes three arrays and an integer as input parameters. The given information consists of an array of item names, an array of item weights and an array of item profits. The integer of the input parameters is the total weight, which represents the limit of the problem. This function generates a circuit consisting of three quantum registers. The addition of a classical register, for the case that measurements shall be performed, is also possible. However, the Quantum Tree Generator circuit is represented in the code example as a gate and therefore can not contain a classical register because a gate in the circuit is a purely quantum-mechanical definition. The three quantum registers are the item register, the capacity register and the profit register. The code mainly consists of a while loop that iterates through the different items with the corresponding array positions for profit and weight, and extends the circuit based on this. Each loop run represents one item. Before this loop, it is necessary to define the parameters required by the algorithm. The preparation begins by converting the entire limit weight into a binary representation and storing the length of this binary number separately. Furthermore, the array lengths of the input arrays are needed, which are also stored separately. Then the largest possible profit is calculated by adding all possible individual profits. This result is converted into a binary number. The length of this binary number is then determined. A further preparation is to transfer and store the marginal gain in an array. The next step is to define the registers. The length of each register is crucial. The length of the item register is adjusted depending on the number of items available. The capacity register has a length equal to twice the number of qubits required to store the limit weight, due to the integer comparison required. The profit register has the size of the total profits. This is the register where the quantum addition is performed. Depending on how the output of this function is defined, it will also have a classical register. This has the size of the profit register, as this would be the only relevant measure to check whether the Quantum Tree Generator is fulfilling its function. However, this Quantum Tree Generator will be used as a gate of the circuit of the knapsack algorithm. After defining the registers in the code and generating the associated quantum circuit, the initial state is defined on the capacity register. To do this, the defined array, which stores the limit weight, is traversed in places. An X-gate is attached to the corresponding qubit in the register at each position where a 1 is present. The last step before the while loop is to define the quantum Fourier transform for the profit register. Since the addition defined in the profit register takes place in parallel and the intermediate results have no influence on the overall result, the quantum Fourier transform in the profit register is only required at the beginning and at the end and therefore takes place outside the loop. Each iteration of the loop starts with the definition of the current values. In this loop, the input arrays are scrolled through, and each time they are scrolled through, the current profit and the current weight shown by the pointer change. These values are converted to binary numbers whose length is determined and stored. The structure

of the loop code can be divided into three main parts. First there is the integer comparison on the capacity register. This is followed by the subtraction on the register and in parallel the addition on the profit register. The comparison checks whether the actual weight is less than or equal to the weight stored on the register. If the weight is less than the stored weight the Hadamard gate is applied to the item register. To ensure that the Hadamard gate is only applied to the item register if the current weight is less than or equal to the played weight, a controlled Hadamard gate connects the comparison qubit and the item qubit. This means that the comparison checks whether an item is eligible for inclusion or not. After the comparison, a subtraction is performed on the capacity register. To perform the quantum subtraction as defined above, transfer the current weight into an array and adjust the length of the array as needed. Then the steps to perform the subtraction are the same as above. The subtraction is connected with the item register. This is the reason why only the possible items for this problem will be subtracted. Due to the fact that the Hadamard gate creates a superposition of whether the item is included or not this circuit goes through all possible solutions at once. After the subtraction the capacity stored on the register has changed, so the next run is compared with a different weight stored on the register. This tries to fill the total weight as well as possible. Parallel to this process, the addition is performed in the profit register. Since the quantum Fourier transform is only performed at the beginning and at the end, only the corresponding phase gates are generated in the loop. Preparations must also be made for this step as well. The currently examined profit is transferred to an array and adjusted in length, then the already known steps follow. This addition is also tied to the item register, only the profits of the items available for the problem are added. This finishes the processes, which are different depending on the element is being viewed. This completes the implementation of the while loop. The inverse of the quantum Fourier transform is added to the profit register to complete the addition. This circuit is then converted into a gate and output. It would also be possible to measure the profit register. By measuring the superposition will be destroyed. Therefore, the measurement gives one of several solutions to the problem. The number of possible solutions displayed can be increased by increasing the number of shots in the code. This has the effect that this circuit is run several times in a row and therefore several solutions can be measured. This means that all possible solutions to the knapsack problem can be found using this function of the Quantum Tree Generator. Nevertheless, the problem is not completely solved, as it does not return an optimal solution, but all possible solutions in the set. Therefore there is a need for the Grover algorithm in combination with the Quantum Tree Generator. This will be analysed in the next section.

3.6 Implementation of the Quantum Tree Generator with Grover

This step is the final step of solving the 0-1-knapsack problem with an algorithm in Qiskit. To now find the optimal solution to this problem it is important to use the Grover Algorithm. For the Grover Algorithm the Grover operator is essential. This is a zero state reflection [NC10]. The operator is shown in FIG.12.

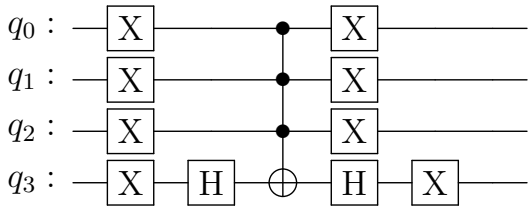


FIG. 12: Circuit of the Grover operator in Qiskit

```
#Implementation of the Grover Operator
def grover_operator(nqubits, as_gate=True):

    #A quantum circuit is created with the specified number of qubits:
    qc = QuantumCircuit(nqubits)

    #Each qubit in the register is assigned a x gate:
    for i in range(0,nqubits):
        qc.x(i)

    #A sequence consisting of a Hadamard gate, a multi-controlled X-gate and
    another Hadamard gate is attached to the last qubit:
    qc.h(nqubits-1)
    qc.mct(list(range(nqubits-1)), nqubits-1)
    qc.h(nqubits-1)

    #Each qubit in the register is assigned a x gate:
    for j in range(0,nqubits):
        qc.x(j)

    #Depending on whether it was selected when entering the function that
    the output should be a gate or a circuit, the circuit will be converted
    to a gate or output directly:
    if (as_gate==True):
        g = qc.to_gate()
        g.name = "G"
        return g
    else:
        return qc
```

Listing 6: Grover operator

The Grover Algorithm can be divided into three reflections. The Grover Operator is responsible for performing the zero state reflection. It also consists of a reflection on the initial state and a reflection by the oracle. The oracle reflection is a reflection on the x-axis. It is mirrored so that the value is negative and the possible solutions are marked. In this case, the initial state is mirrored by the Quantum Tree Generator. The oracle is constructed by a comparison in the profit register. This circuit therefore requires more qubits than the circuit for the Quantum Tree Generator. Due to the comparison of the profit register there is a need for qubits that are twice the size of the previous profit register. An additional ancilla qubit is also required for the oracle. If a solution is found, the state is reversed to the negative by the oracle. In this example, if the comparison qubit outputs a one, then an X-gate is applied to the ancilla qubit, which reverses the state. This state will be marked. The integer comparison on the profit register checks if the measured profit is greater than the current threshold. The structure of the algorithm is

given by the repeated application of a sequence consisting of the Quantum Tree Generator and the associated inversion, the comparison of the profit register with the inversion and the Grover operator. The Quantum Tree Generator generates a superposition and prepares the states for the Grover algorithm. This is followed by a measurement to obtain a reference value for the current profit. After this, a comparison is made in the profit register. This comparison is followed by the inverse of the Quantum Tree Generator and the Grover operator. Depending on the result of the comparison, a new circuit is generated, to which either another Grover iteration is added, or an update of the profit value is performed. In this way, a selection of items with the highest profit value is found. It is important to set a limit for the repeated use of this sequence. Otherwise an endless loop will be created. The limit mentioned plays a key role in obtaining a correct result. In the next section of the analysis of Grover iterations, the exact role of this limit will be discussed. This process can be seen in the circuit in FIG.13.

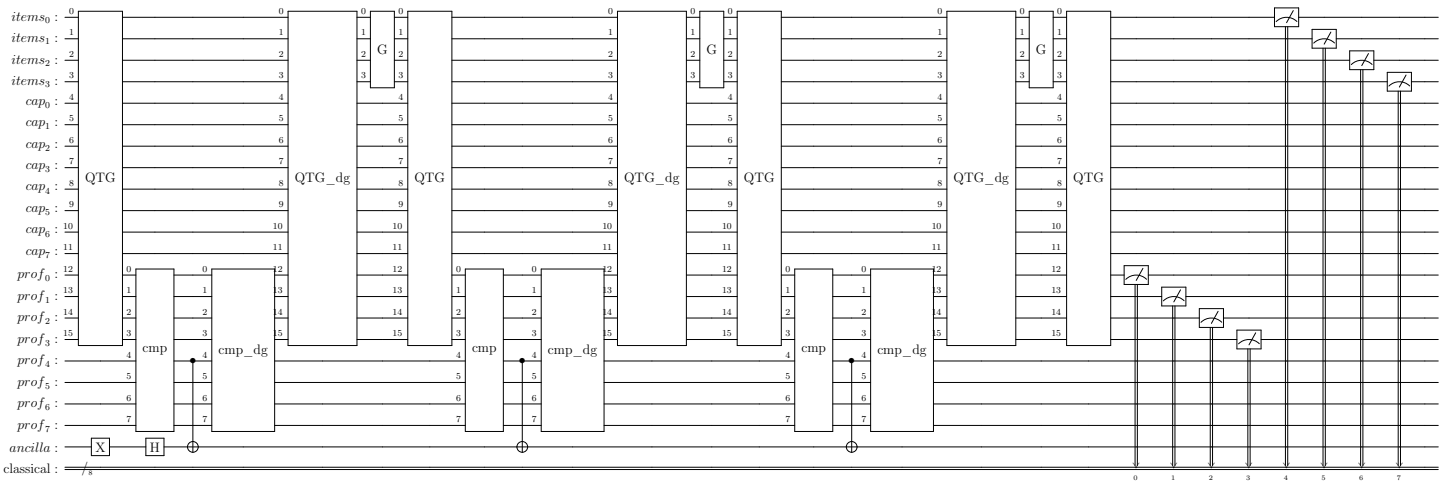


FIG. 13: Circuit of the 0-1 knapsack algorithm in Qiskit

The detailed operation of the algorithm can be described using this circuit. First, the Quantum Tree Generator is applied to all qubits except the newly added ones. The program then generates all possible solutions to the given initial conditions. One of the possible solutions is then selected by a measurement on the profit register. This value is stored. The result of this measurement determines the further procedure. First, a limit value for the prize is set, which is necessary to reach the maximum value. If the measured value is greater than the set threshold, the cycle starts again and the threshold is set to the measured value. If this is not the case, the circuit is rebuilt with another Grover iteration. It is important to rebuild the circuit after each measurement, as a measurement destroys the overlay and no correct calculation can be done afterwards. The added Grover iteration contains the integer comparator in the profit register after applying the Quantum Tree Generator. The comparison is made between the set limit value and the value in the profit register. A controlled X-gate connects the qubit on which the result of the comparison is stored to the ancilla qubit of the oracle. This is used to mark the searched solutions according to the criterion that the oracles qubit is only one if the measured result is greater than the given limit. To reset the qubits for further Grover iterations, the circuit needs to be extended to include the inverse of the comparison pattern. For the same reason, the inverse of the Quantum Tree Generator is also added. A Grover iteration ends with the application of the Grover operator to the item register. This algorithm finds the maximum

profit by repeating the comparison until a certain number of iterations is reached. Below is the corresponding algorithm implemented in Python.

```
#Implementation of the algorithm for solving the 0-1 knapsack problem:
def solution_algorithm(items_test,weights_test,profit_test,total_weight_test
    , grover_iteration_number):

    #Define the variables relevant to the register structure and the
    subsequent algorithm:
    total_wb=dez2bin(total_weight_test)
    total_wlen=len(str(total_wb))
    len_prof=len(profit_test)
    len_weight=len(weights_test)
    sum_prof=0

    for i in range(0,len_prof):
        sum_prof=sum_prof+profit_test[i]

    sum_prof_b=dez2bin(sum_prof)
    sum_prof_l=len(str(sum_prof_b))
    counter_grover_iterations=0
    selection_of_items=0
    selection_of_items_final=0

    #Definition of quantum and classical registers:
    test_item=QuantumRegister(len(items_test),'items')
    test_cap=QuantumRegister(total_wlen*2,'cap')
    test_prof=QuantumRegister(sum_prof_l*2,'prof')
    test_ancilla_oracel=QuantumRegister(1,'ancilla')
    test_cl1=ClassicalRegister(len(items_test)+sum_prof_l,'classical')
    test_circuit = QuantumCircuit(test_item,test_cap,test_prof,test_cl1,
    test_ancilla_oracel)

    #Definition of the gates to be used in the Grover algorithm:
    gatter=qtg(items_test,weights_test,profit_test,total_weight_test)
    gatter_inverse=gatter.inverse()
    grover=grover_operator(len(items_test), as_gate=True)

    #Defines the array containing the qubits of the circuit to which the
    Quantum Tree Generator is to be attached:
    over_all_length=len(items_test)+(total_wlen*2)+sum_prof_l
    arr_over_all=[]
    counter_over_all=0

    for f in range(0,over_all_length):
        arr_over_all.append(counter_over_all)
        counter_over_all=counter_over_all+1

    #Define initial conditions and simulator:
    treshold=0
    simulator_test = AerSimulator()
    test_circuit.x(test_ancilla_oracel)
    test_circuit.h(test_ancilla_oracel)

    #This is the start of the loop to generate the gate sequence in the
    circuit:
    truth_value=True

    while truth_value==True:
```

```
52          test_circuit.append(gatter,arr_over_all)
53          length_cl=len(items_test)+sum_prof_l
54          save=len(items_test)
55          diff=sum_prof_l
56          s=save
57          #This is where the profit and position register is measured. This
    provides the comparison value:
58          for p in range(0,save):
59              test_circuit.measure(test_prof[p],test_cl1[p])
60          for a in range(0,diff):
61              test_circuit.measure(test_item[a],test_cl1[s])
62              s=s+1
63          compiled_test = transpile(test_circuit, simulator_test)
64          job_test = simulator_test.run(compiled_test, shots=1)
65          result_test= job_test.result()
66          counts_test = result_test.get_counts(compiled_test)
67          solution_test=counts_test.int_outcomes()
68          solution2_test=solution_test.popitem()
69          result_of_flow_gesamt=solution2_test[0]
70          result_of_flow_bin=dez2bin(result_of_flow_gesamt)
71          result_of_flow_len=len(str(result_of_flow_bin))
72          array_solution=[]
73          m=0
74          #The measurement result is stored in an array and read out
    individually:
75          while m<result_of_flow_len:
76              array_solution.append(result_of_flow_bin%10)
77              result_of_flow_bin=math.floor(result_of_flow_bin/10)
78              m=m+1
79          while len(array_solution)<length_cl:
80              array_solution.append(0)
81          result_of_flow=0
82          one=1
83          for l in range(0,save):
84              result_of_flow=result_of_flow+array_solution[l]*one
85              one=one*10
86          one_2=1
87          selection_of_items=0
88          for u in range(save,length_cl):
89              selection_of_items=selection_of_items+array_solution[u]*one_2
90              one_2=one_2*10
91          result_of_flow=bin2dec(result_of_flow)
92          #The solution for the current measurement is displayed at this point
    :
93          print("solution:",result_of_flow)
94          print("items:",selection_of_items)
95          treshold_bin=dez2bin(treshold)
96          treshold_len=len(str(treshold_bin))
97          #This is where the branching begins, and depending on the result of
    the comparison, either a new Grover iteration is added or the algorithm
    is restarted:
98          if result_of_flow<=treshold:
99              if(counter_grover_iterations>=grover_iteration_number):
100                 truth_value=False
101                 break
102             counter_grover_iterations=counter_grover_iterations+1
103             test_circuit.clear()
104             i=0
```

```
105             test_circuit.x(test_ancilla_oracel)
106             test_circuit.h(test_ancilla_oracel)
107             print("iterations",counter_grover_iterations)
108             while(i<counter_grover_iterations):
109                 test_circuit.append(gatter,arr_over_all)
110                 comparator_qtg_prof = IntegerComparator(num_state_qubits=
    treshold_len, value=treshold, geq=True)
111                 comparator_qtg_prof_inverse = IntegerComparator(
    num_state_qubits=treshold_len, value=treshold, geq=True).inverse()
112                 hilfsarray=[]
113                 v=0
114                 while v<(treshold_len*2):
115                     hilfsarray.append(test_prof[v])
116                     v=v+1
117                 test_circuit.append(comparator_qtg_prof,hilfsarray)
118                 test_circuit.cx(test_prof[treshold_len],test_ancilla_oracel)
119                 test_circuit.append(comparator_qtg_prof_inverse,hilfsarray)
120                 print("lower->append")
121                 test_circuit.append(gatter_inverse,arr_over_all)
122                 test_circuit.append(grover,test_item)
123                 i=i+1
124
125         if result_of_flow>treshold:
126             selection_of_items_final=selection_of_items
127             print("higher->new")
128             treshold=result_of_flow
129             counter_grover_iterations=0
130             test_circuit.clear()
131             test_circuit.x(test_ancilla_oracel)
132             test_circuit.h(test_ancilla_oracel)
133
134     return test_circuit
```

Listing 7: Quantum Tree Generator with Grover for a example for four items

The algorithm consists of a function that takes as input the variables of the 0-1 Knapsack problem. These variables consist of an array of elements, arrays of the respective weights and profits of the elements, a maximum weight for the problem and a number of Grover iterations. The return parameter of the function is the quantum circuit that leads to the result. The next step is to define the necessary initialisations for the algorithm. These initialisations consist of the binary conversion of the maximum weight, the length of this binary and the different lengths of the input parameters. In addition, the sum of all input wins is calculated and this value is converted to a binary number and then to the length of this binary number. A Grover iteration counter is generated and initially set to zero. This counter is incremented with each Grover iteration added to the algorithm. Another parameter, which stores the selection of elements, is also initialised with zero. The next step is to define the registers needed for the algorithm. The registers of the quantum circuit for the 0-1 knapsack problem are the same as those of the Quantum Tree Generator: an item register, a profit register, a weight register and a classical register. The only additional register compared to the Quantum Tree Generator is the ancilla-qubit register for the oracle. The lengths of the item and weight registers are the same as for the Quantum Tree Generator. The only difference is the length of the profit register. Since the algorithm requires a comparison to be performed on this register, the length of this register must be doubled compared to the length of the previous register. The register used for the oracel consists of only one qubit and therefore has a length of 1. The classic register is made up of the length of the item register and the single taken length of the profit register, as these

are the values used to measure the result. Before the algorithmic cycle described above can be started, the necessary gates have to be defined. This includes the Quantum Tree Generator, its inverse and the Grover operator. In order to insert the gate of the Quantum Tree Generator into the circuit, it is necessary to create an array with the qubit size for the gate. This size is made up of the item register, the weight register and the single taken profit register. Another important definition is the profit threshold, which is initialised at zero. Other important point is the definition of the simulators required for the measurement. The necessary gates are attached to the oracle's ancillary qubit. These consist of the X-gate, which inverts the qubit, and the Hadamard gate, which creates a superposition. These gates are attached to the ancilla qubit only once. A Boolean is then defined. This is initialised with the value True. As long as this value is true, the execution of the loop is not interrupted. The loop starts by inserting the Quantum Tree Generator into the circuit. The length of the item register and the length of the individual profit register are also stored for later division of the measurement into the measurement of items and the measurement of profits. A measurement is then performed on both the profit and item registers. The result of this measurement is then stored separately in the form of a binary number. This binary number is then transferred to an array. The first part of this array represents the result of the profit measurement and the following part represents the result of the corresponding selection of items. As the array is iterated, these results are stored in two separate variables. It is necessary to convert the binary representation to a decimal representation in order to work with the result of the profit. In order to store the binary length of this value, the current profit threshold is converted to a binary representation. Two cases are distinguished at this point in the algorithm by means of an if-condition. First, the case where the measured profit is below or equal the threshold is considered. This branch contains another branch that checks whether the number of Grover iterations has already exceeded the individually defined iteration limit. If this is the case, the truth value is set to False, the loop is terminated and the measured profit with the corresponding item selection can be output. Otherwise, the Grover iteration counter is incremented by one and the circuit is reset. Starting with the addition of the X-gate and the Hadamard gate to the ancilla qubit, the circuit is rebuilt. Then the gates are inserted into the circuit in the order Quantum Tree Generator, integer comperator, CX-gate, inverse integer comperator, inverse Quantum Tree Generator and Grover operator until the stored number of Grover iterations is reached. This process is implemented in the code by a while loop. This results in the circuit being extended by another Grover iteration. The second case covered by the code is that the measured profit is higher than the current threshold. In this case, the selection of items is saved as the final selection of items and the threshold is set to the higher measured profit. As the algorithm starts from the beginning in this case, it is necessary to reset the counter to zero and the circuit to the original settings. Then only the gates needed for the ancilla qubit are added to the circuit. If the Grover iterations are chosen correctly, this algorithm will eventually provide the desired solution. If the wrong number of Grover iterations is chosen, the algorithm may deviate from the desired solution and produce a false result. Therefore, analysis of the iteration steps is an important part of the algorithm definition.

4 Algorithm accuracy analysis

4.1 Accuracy of the Quantum Tree Generator

In order to verify the correctness of the algorithm of the Quantum Tree Generator, it is important to perform a state analysis after each addition of a new element. Ideally, this analysis should be done for both the profit and the item selection, to see which states are eliminated due to overweighting. These results are analysed in this section and compared with the expected values.

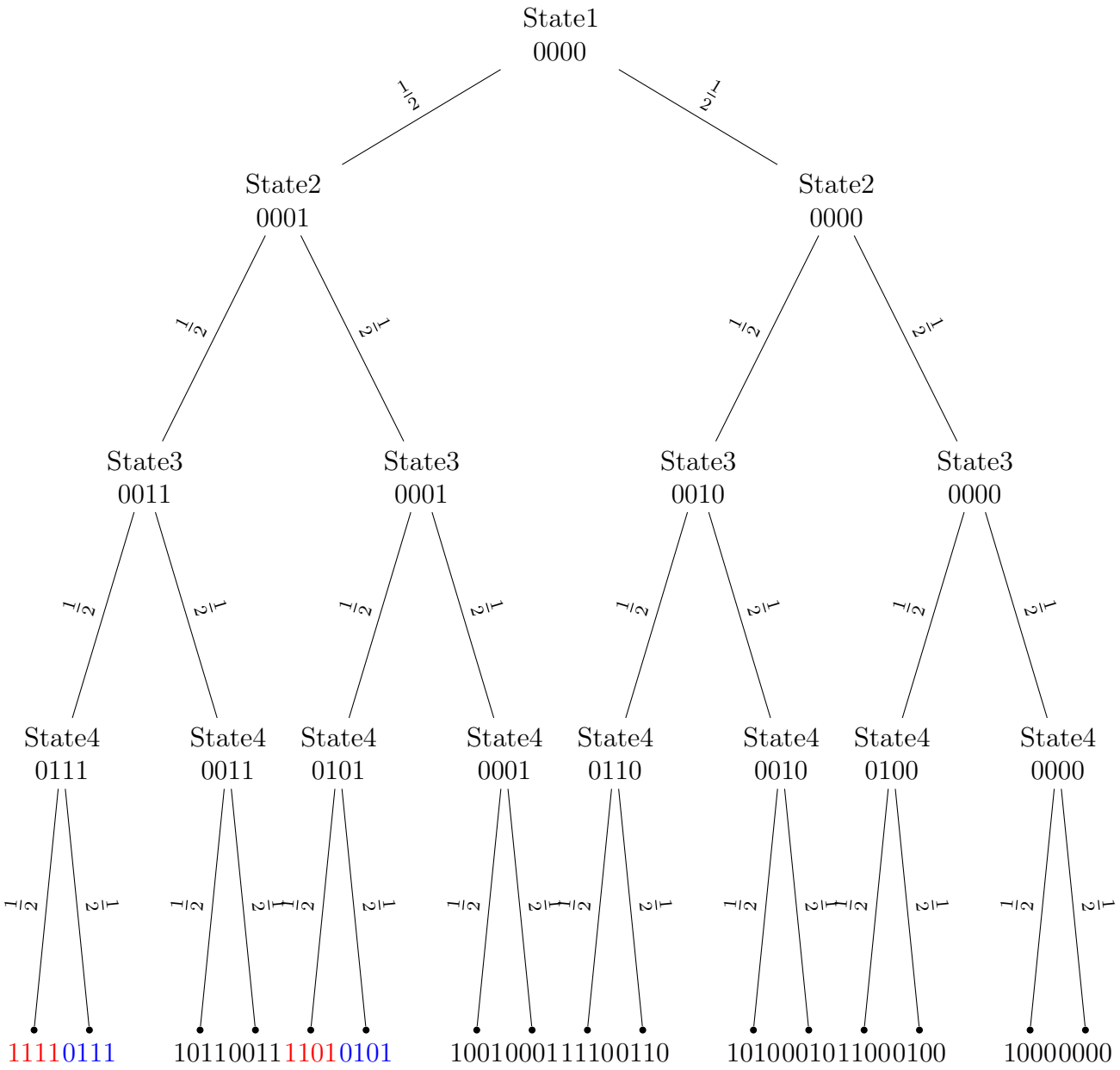
4.1.1 State representation after each added item

Since the Quantum Tree Generator acts as a search tree, it is relatively easy to examine the expected solutions. For the following example of the 0-1 knapsack problem, a step-by-step analysis is performed in this thesis:

$$\begin{aligned} \mathrm{I} &= [i1, i2, i3, i4] \\ \mathrm{w} &= [4, 1, 5, 6] \\ \mathrm{p} &= [2, 3, 6, 4] \\ \mathrm{W} &= 13 \end{aligned}$$

The following probability tree shows the possible states of the path register for the problem and the corresponding probabilities. The coloured states are of particular interest. The states marked in red are those that are excluded due to the weight limit of the problem. The states that share the same parent node with the eliminated states are marked in blue. It can be assumed that these states are more likely to occur due to the elimination of the excluded state. Using this probability tree, it is very easy to see the effects of adding the respective element on the set of states and which states are added in which step. The aim is to be able to prove these different states with the help of the algorithm and thus be able to draw conclusions about the correct functioning. For this purpose, a state measurement is performed after each addition of an element. This is done once for the path and once for the profit register.

Probability tree of the path states for an example with four items



The status measurements for the path and profit registers of the algorithm can be displayed in histograms as follows.

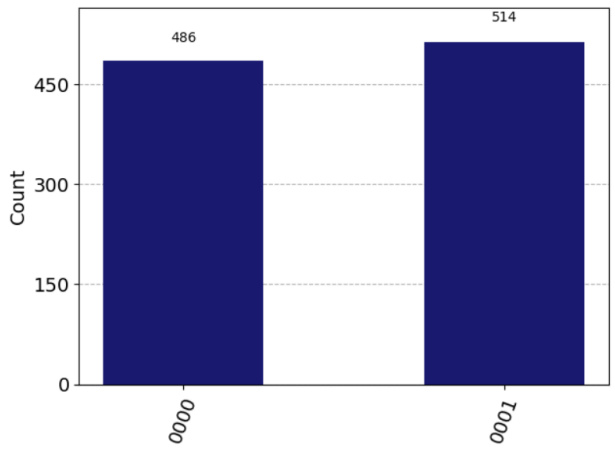


FIG. 14: Measuring station of all states of the path register with one item

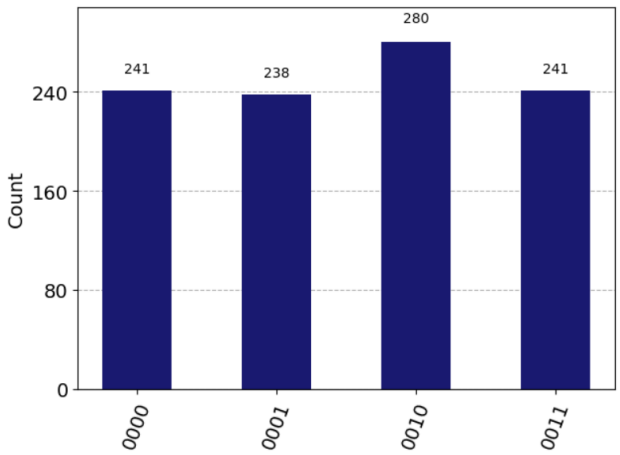


FIG. 15: Measuring station of all states of the path register with two items

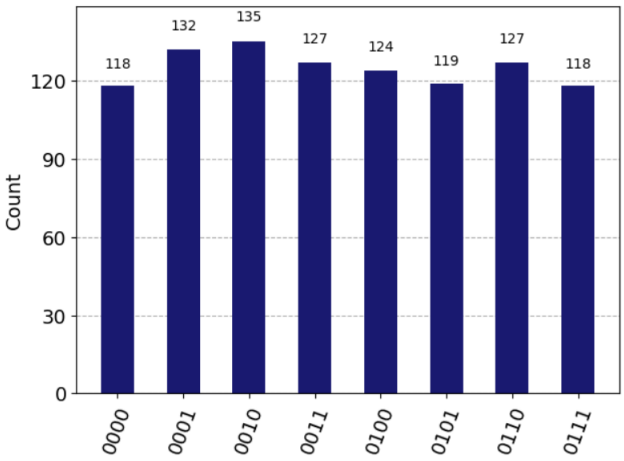


FIG. 16: Measuring station of all states of the path register with three items

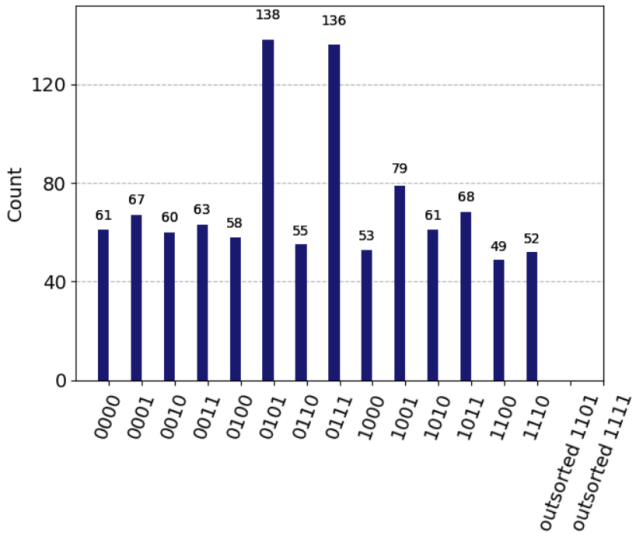


FIG. 17: Measuring station of all states of the path register with four items

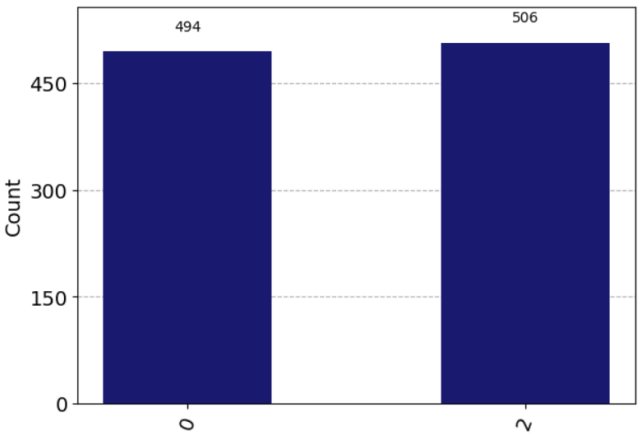


FIG. 18: Measuring station of all states of the profit register with one item

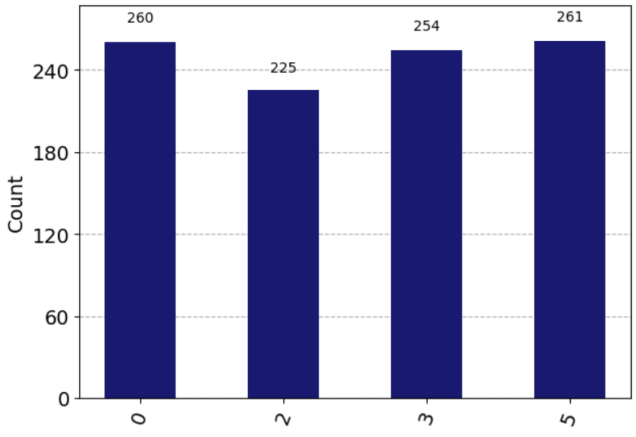


FIG. 19: Measuring station of all states of the profit register with two items

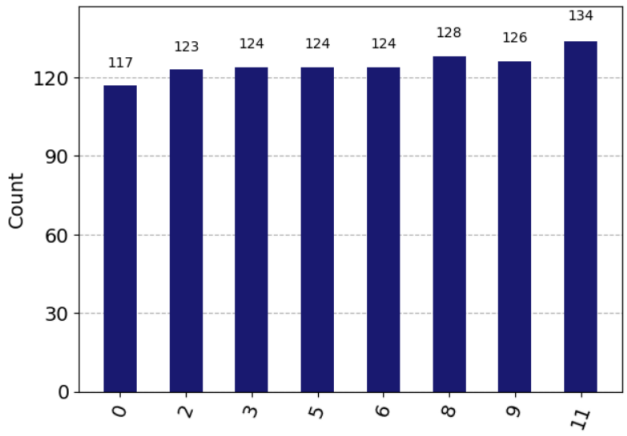


FIG. 20: Measuring station of all states of the profit register with three items

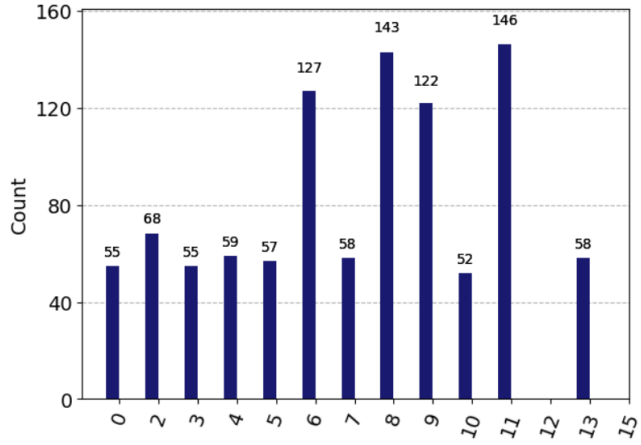


FIG. 21: Measuring station of all states of the profit register with four items

It is clear to see that the results of the probability tree and the results of the measurement stations for the algorithm are identical. The states at each state level match. The coloured states in the tree can also be seen in the histograms. The eliminated states ′1111′ (giving a profit of 15) and ′1101′ (giving a profit of 12) occur with a probability of zero. The above hypothesis for the states that have the same parent node as the eliminated states is confirmed by the measurement and the histogram. The states ′0111′ (with a profit of 11) and ′0101′ (with a profit of 8) occur with a higher probability than the others. In each state level, all states are almost equally distributed, which corresponds to the probability tree, since each element always occurs with a probability of 1/2. This is also shown by the probabilities in the histogram. However, this is not the case for the neighbouring states mentioned above, as the disappearance of states creates an imbalance. With this parallel representation it can be shown that the programmed algorithm for the Quantum Tree Generator works correctly. That is, it fulfils the task of generating all possible states under the given conditions.

4.2 Accuracy of the 0-1 knapsack algorithm

In order to check the correctness of the algorithm for the 0-1 knapsack problem, the results must be examined and compared with the expected values. This is done in this section by analysing the Grover iterations. The final states after different Grover iterations are measured and checked for correctness. In this way it can be determined whether the algorithm is working correctly and what influence the Grover iterations have on the result.

4.2.1 Correlation between probability of success and Grover iterations

Finding the optimal number of Grover iterations is not an easy task. The number of optimal iterations depends on the size of the database and the number of elements to search for. In some cases where this number is not known, it may be difficult to determine the optimal number of Grover iterations. It is important to consider the probability of finding the correct answer. The probability is calculated mathematically using the following formula [NC10]:

$$\sin^2\left((r + \tfrac{1}{2}) \cdot \Theta\right) \text{ with } \Theta = 2 \cdot \arcsin \tfrac{1}{\sqrt{N}}$$

Theta is the angle between the state vector $|s\rangle$ and the state vector after the rotation of the Grover iteration. In addition, r represents the number of Grover iterations. This can also be visualized very well in the Bloch sphere (FIG.22).

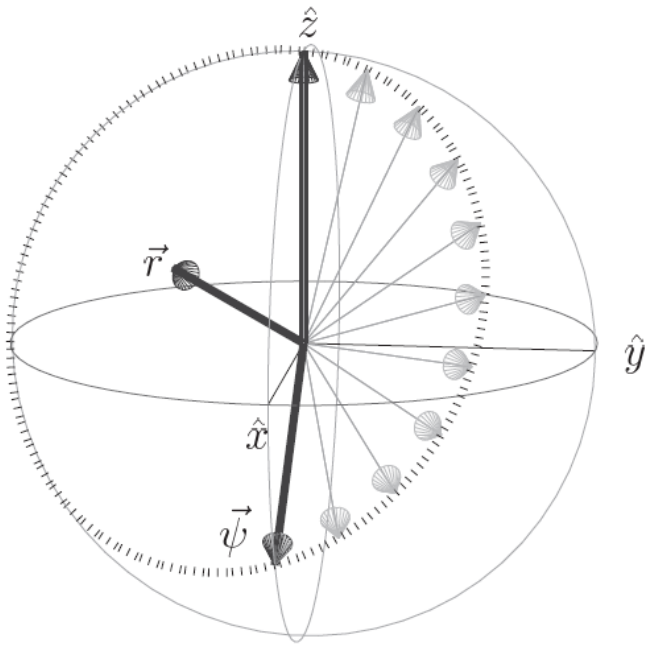


FIG. 22: This illustration shows the initial state ψ rotates around the rotation axis $\vec{r}$ to the final state z [NC10]

This visualisation clarifies that the function of the probability of finding a correct solution has several maxima and minima. It can be seen that as the iterations of Grover are varied, the results move away from and towards the desired solution. The approximate equation of Grover iterations for the first near-optimal solution is as follows [NC10]:

$$r \approx \frac{\pi}{4} \cdot \sqrt{N}.$$

In this equation, N is the number of elements. Now the Grover iterations that provide a near-optimal solution can be found exactly by these formulas. Therefore, these formulas are considered as examples for four items. The selection of four items is the same as above. The specific values of the example are listed again.

$$\begin{aligned} \mathrm{I} &= [i1, i2, i3, i4] \\ \mathrm{w} &= [4, 1, 5, 6] \\ \mathrm{p} &= [2, 3, 6, 4] \\ \mathrm{W} &= 13 \end{aligned}$$

The Quantum Tree Generator mentioned above gives us 12 solutions for the example of four items. This number of solutions is equal to N. Therefore, the individual function of the probability of finding the correct solution for this example is as shown below.

$$\sin^2\left((r + \tfrac{1}{2}) \cdot 2 \cdot \arcsin \tfrac{1}{\sqrt{12}}\right)$$

The plot of this function is shown in FIG.23.

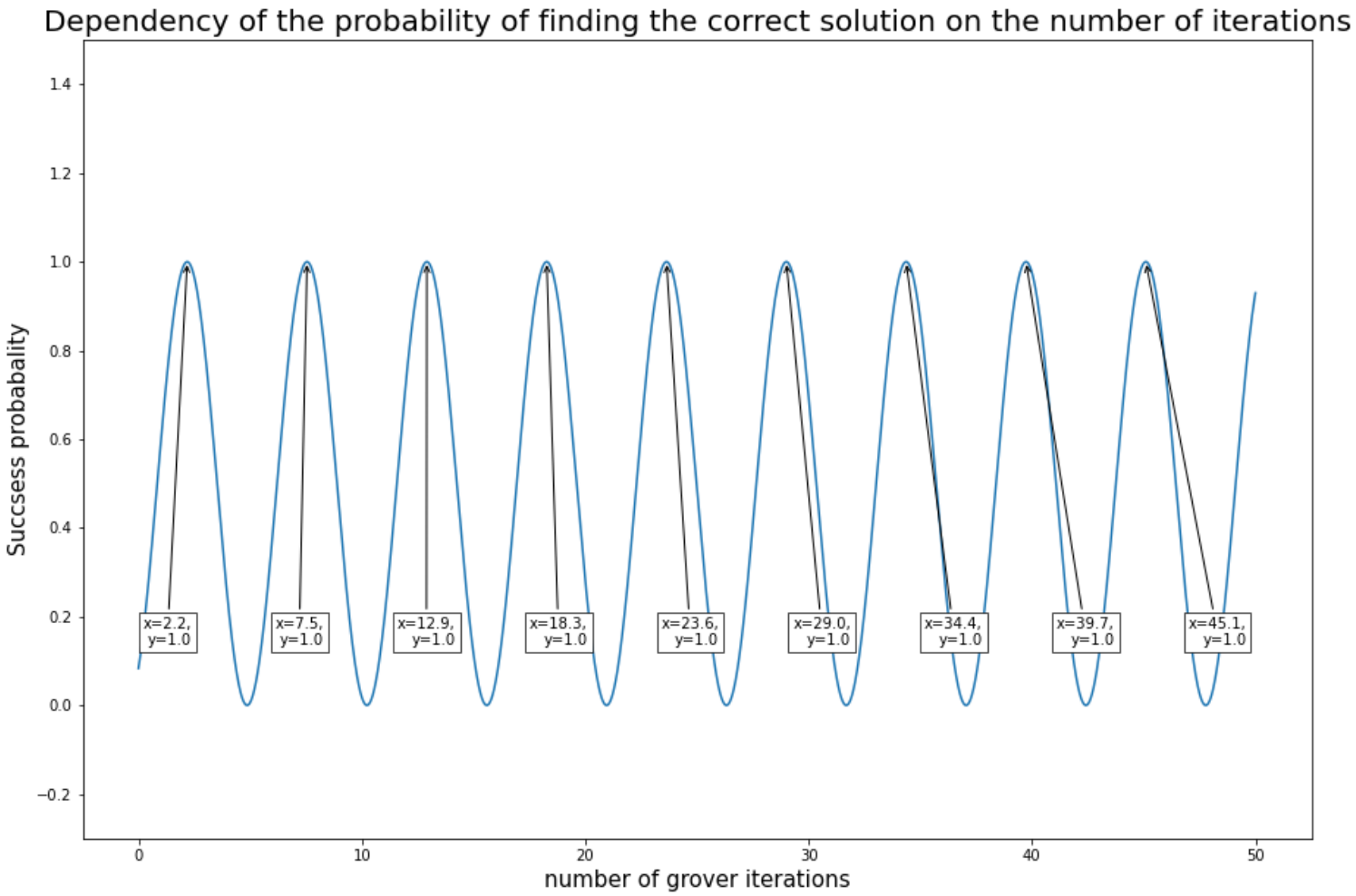


FIG. 23: This graphic shows the success probability in dependency of the iterations for the example of four items

Note that this function has a series of maxima and a series of minima. For example: 2.2, 7.5, 12.9, 18.3 Grover iterations have the highest probability of success. As only integer Grover iterations are possible, these peaks can only be approximated. In practice, the best results are achieved between two and three, between seven and eight and between twelve and thirteen. This is confirmed by the algorithm presented. If you introduce different thresholds for the Grover iterations, you get exactly this trigonometric pattern. The algorithm can help to explain this, as shown below. The result of a measurement is compared with the threshold as described. Either the threshold is overwritten and the algorithm restarted, or another Grover iteration is added. Once the algorithm has found the largest result (the one it is looking for), the further measurements will automatically become smaller and smaller than the threshold. The number of Grover iterations is also increased. Without a limit on the number of iterations, the algorithm becomes an infinite loop. For this reason setting a limit for the number of iterations is important. This will affect the result, as shown in the text previously. However, if the number of iterations is wrong, the algorithm will move away from the correct solution, as it will always move in a circle towards the desired solution. A wrong choice of the Grover iteration limit will lead to a deterioration of the final result. Depending on the example chosen, the values for the best repetitions are different. In the example, if the number of items changes, the number of possible solutions changes and so does the function defined above for the probability of success as a function of Grover iterations. For an example with five elements there are eighteen possible solutions and the function in FIG.24.

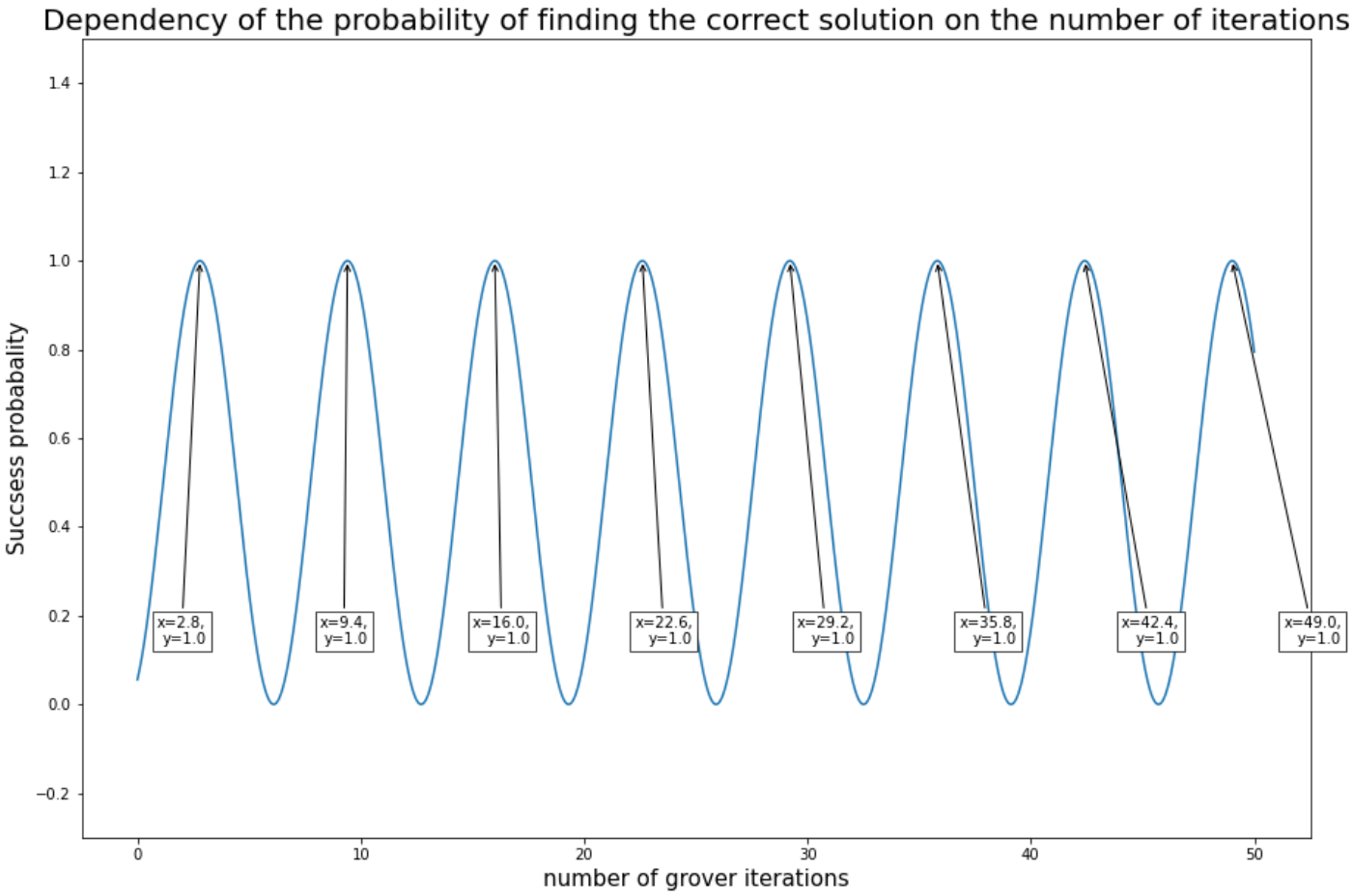


FIG. 24: This graphic shows the success probability in dependency of the iterations for the example of five items

4.2.2 State representation after a certain number of Grover iterations

To illustrate the correlation between the number of Grover iterations and the probability of results, the measured states after a given number of Grover iterations were visualised in Qiskit. Histograms were used for this visualisation. This visualisation of the states was carried out using an example with four items, which is the same example as in the previous applications. In order to get an accurate overview of the distribution of the result states after a number of Grover iterations, it is necessary to increase the number of shots in the Qiskit code when measuring the path register. The number of shots determines the number of runs of the code in which the results are to be collected. This results in a distribution as shown in the following figures.

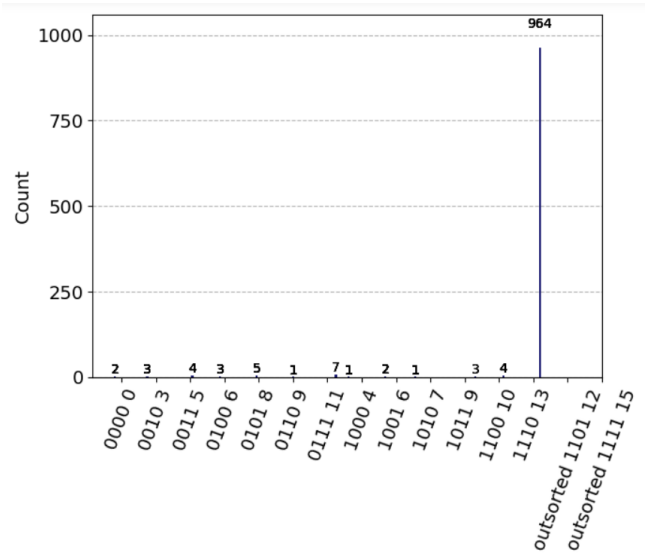


FIG. 25: Measuring station of all states of the path and profit register after 3 Grover iterations

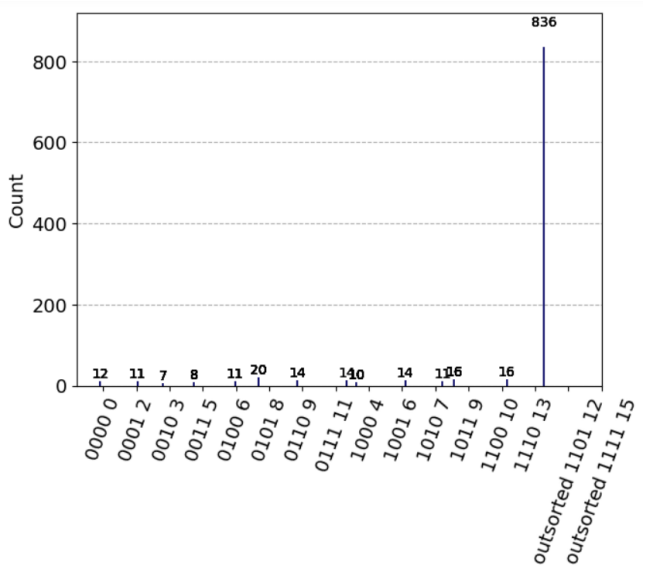


FIG. 26: Measuring station of all states of the path and profit register after 8 Grover iterations

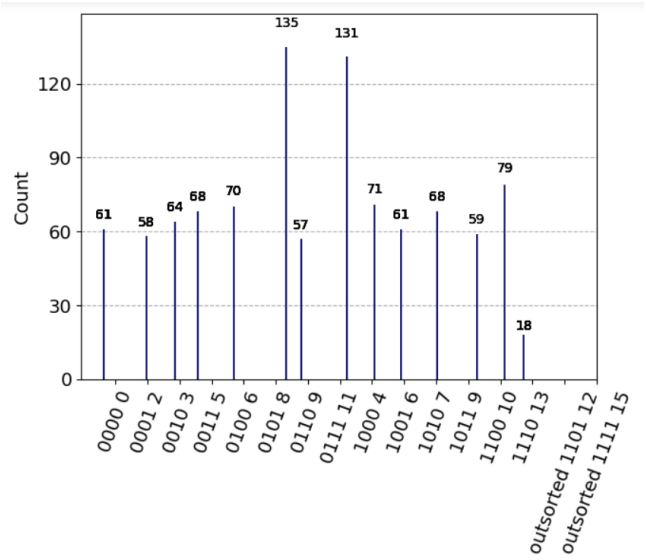


FIG. 27: Measuring station of all states of the path and profit register after 6 Grover iterations

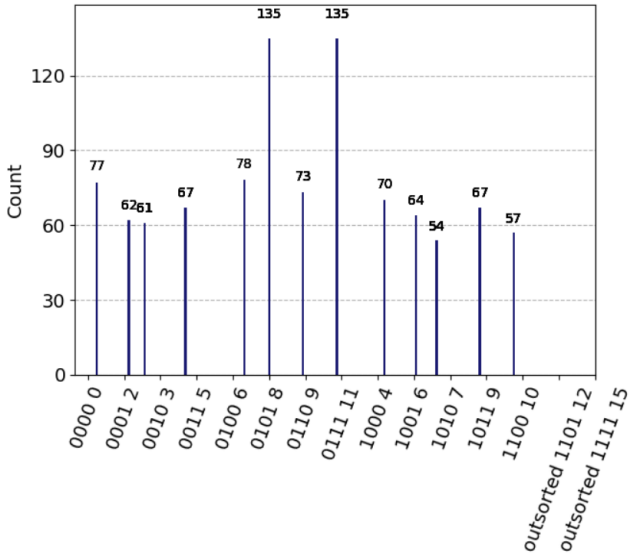


FIG. 28: Measuring station of all states of the path and profit register after 12 Grover iterations

The first figure (FIG.25) shows the measurement station of all states of the path register after three Grover iterations. The histogram shows a clear uneven distribution of the states. The state $\{'0111'\}$ is output in about 96% of all measurements. The second illustration (FIG.26), which was generated after eight iterations with Grover, shows a similar result. The state $\{'0111'\}$ is the optimal solution for the selected example. Thus, after this number of Grover iterations, a clear, optimal solution to the 0-1 Knapsack problem can be derived. The error rate is very low in both cases, 4% and 17% respectively. However, the 17% error rate of the second figure suggests that 8 is not yet optimal. The third (FIG.27) and fourth (FIG.28) figures show six and twelve Grover iterations respectively. These iterations give a different result. These histograms suggest a more even distribution. Although some amplitudes are slightly larger than others, they account for only 10% to 20% of the data. There is therefore no clear and optimal result for this number of Grover iterations. Comparing these results with Figure 15, a clear correlation can be seen. The periodic relationship shown above between the probability of a result and the number of Grover iterations is confirmed by the measurements of the path register states for different Grover iterations. The fact that the maxima and minima in Figure 15 have decimal places and that the Grover iterations can only be selected as integers makes it difficult to select both good and bad Grover iterations. The short period length of the curve makes it even more difficult, as a small change in the number of Grover iterations can have a completely opposite effect. The sine curves in Figures 15 and 16 show that, apart from the maxima on the right and left, there is no slow decrease, but a very fast transition to the minima. This problem only occurs when the element size is very small. The arc sine and thus the factor in the sin^2 curve becomes smaller the larger the item size. The smaller the factor in the sin^2 curve, the greater the scatter along the x-axis and thus along the Grover iterations. The interval between good and bad Grover iterations with high and low probabilities of success is also significantly larger for a large item size. This is particularly important from the following perspective. Since a significant quantum advantage is only achieved from an item size of significantly more than four items, the procedure is of interest in the future especially for a larger selection of variables. This makes it easier for the future applications of this algorithm to select the optimal number of

Grover iterations. This will also produce better and more accurate results for these calculations. This analysis of Grover iterations proves the correctness of the algorithm developed for the 0-1 Knapscak problem. The histograms, which represent the measuring points of the algorithm after a certain number of Grover iterations, show the expected result. This is, on the one hand, the periodic relationship between the number of Grover iterations and the probability of success and, on the other hand, the fact that an optimal result was achieved with a suitable choice of Grover iterations. These results therefore provide clear evidence of the correct functioning of the algorithm.

5 Conclusion

The purpose of this conclusion is both to summarise the results and to look into the future. This thesis deals with the implementation of the algorithm for the 0-1 knapsack problem using the Quantum Tree Generator in Qiskit. The desired result was achieved with this work in each case. The Quantum Tree Generator was created and provides all possible solutions for the boundary conditions. Furthermore, the generated algorithm is able to filter out the optimal solution and thus deliver the desired result. The functionality of the Grover algorithm can also be identified by its properties. As already analysed in the paper: "A quantum algorithm for the solution of the 0-1 Knapsack problem", to which this thesis refers, a quantum advantage is possible with this method [Wil+]. However, this quantum advantage is only achieved by using six hundred variables. The size of the sample that can be created in Qiskit does not come close to the required size. This paper is therefore less a practical demonstration than a look into the future. The purpose of this paper is to demonstrate how the new quantum method for solving the 0-1 Knapsack problem works. It is also suitable for people who have not yet dealt with the subject in detail. Qiskit gives a very simple overview of the structures of an algorithm. This paper provides instructions and illustrations for solving the 0-1 knapsack problem with a quantum advantage in mind. For the future, this thesis also has an educational function. It should enable a large number of people to engage with and understand the topic. If larger dimensions become possible in the future, this thesis will also be useful to adapt the solution of the problem to these dimensions.

References

[KPP04] Hans Kellerer, Ulrich Pferschy **and** David Pisinger. *Knapsack Problems*. Springer, 2004.

[GS08] Gawiejnowicz **and** Stanislaw. *Time-Dependent Scheduling*. Springer, 2008.

[NC10] Michael A. Nielsen **and** Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, New York, 2010.

[BA13] Yehuda B. Band **and** Yshai Avishai. *Quantum Mechanics with Applications to Nanotechnology and Information Science*. 2013.

[KV18] Bernhard Korte **and** Jens Vygen. *Kombinatorische Optimierung*. Springer, 2018.

[Bot+] Panagiotis Botsinis, Zunaira Babar, DimitriosAlanis, DaryusChandra, Hung Nguyen, SoonXin Ng **and** Lajos Hanzo. *Quantum Error Correction Protects Quantum Search Algorithms Against Decoherence*. URL: https://www.nature.com/articles/srep38095. Online; last retrieved at 10.06.2024.

[FOR] Lance FORTNOW. *The Computational Complexity Column*. URL: https://www.uni-ulm.de/fileadmin/website_uni_ulm/iui.inst.190/Mitarbeiter/toran/beatcs/column80.pdf. Online; last retrieved at 10.06.2024.

[GS] Diego García-Martín **and** Germán Sierra. *Five Experimental Tests on the 5-Qubit IBM Quantum Computer*. URL: https://www.scirp.org/journal/paperinformation?paperid=86139. Online; published July 24, 2018; last retrieved at 10.06.2024.

[Gro] Lov K. Grover. *A fast quantum mechanical algorithm for database search*. URL: https://arxiv.org/pdf/quant-ph/9605043. Online; published 1996; last retrieved at 10.06.2024.

[Har] Remis Haroon. *Decoding Complexity: A Deep Dive into NP-complete Problems*. URL: https://medium.com/nerd-for-tech/decoding-complexity-a-deep-dive-into-np-complete-problems-fe72d8efc677#:~:text=At%20its%20core%2C%20an%20NP,pieces%20can%20be%20time%2Dconsuming.. Online; published September 12, 2023; last retrieved at 10.06.2024.

[IBMa] IBM. *Grover's algorithm*. URL: https://learning.quantum.ibm.com/course/fundamentals-of-quantum-algorithms/grovers-algorithm#introduction. Online; last retrieved at 10.06.2024.

[IBMb] IBM. *IBM Quantum Documentation*. URL: https://docs.quantum.ibm.com/. Online; last retrieved at 10.06.2024.

[IBMc] IBM. *Was ist Quantencomputing?* URL: https://www.ibm.com/de-de/topics/quantum-computing. Online; last retrieved at 10.06.2024.

[Man] Ryan Mandelbaum. *Five years ago today, we put the first quantum computer on the cloud. Hereâs how we did it.* URL: https://www.ibm.com/quantum/blog/quantum-five-years. Online; published 4 May 2021; last retrieved at 10.06.2024.

[Mara] Jens Marre. *Der Grover-Algorithmus und die Suche nach dem heiligen Gral*. URL: https://www.quantencomputer-info.de/quantencomputer/grover-algorithmus/. Online; last retrieved at 10.06.2024; puplished on 11.09.2018.

[Marb] Jens Marre. *Quantencomputer programmieren mit IBMs Qiskit*. URL: https://www.quantencomputer-info.de/quantencomputer/quantencomputer-programmieren/. Online; published March 24, 2020; last retrieved at 10.06.2024.

[TEC] TECH. *IBM built the biggest, coolest quantum computer. Now comes the hard part.* URL: https://www.fastcompany.com/90992708/ibm-quantum-system-two. Online; published January 8,2024; last retrieved at 10.06.2024.

[UKa] Quantum Computing UK. *INTEGER COMPARISON IN QISKIT*. URL: https://quantumcomputinguk.org/tutorials/integer-comparison-in-qiskit. Online; published March 15,2023; last retrieved at 10.06.2024.

[UKb] Quantum Computing UK. *Quantum Fourier Transform In Qiskit*. URL: https://quantumcomputinguk.org/tutorials/quantum-fourier-transform-in-qiskit. Online; published August 20,2020; last retrieved at 10.06.2024.

[unk] unknown. *Groverâs Algorithm*. URL: https://github.com/Qiskit/textbook/blob/main/notebooks/ch-algorithms/grover.ipynb. Online; last retrieved at 10.06.2024.

[Wil+] Sören Wilkening, Andreea-Iulia Lefterovici, Lennart Binkowski, Michael Perk, Sándor Fekete **and** Tobias J.Osborne. *A quantum algorithm for the solution of the 0-1 Knapsack problem*. URL: https://arxiv.org/abs/2310.06623. Online; last retrieved at 10.06.2024.

[Woj] Dominik Wojtczak. *On Strong NP-Completeness of Rational Problems*. URL: https://arxiv.org/pdf/1802.09465. Online; published Febuary 26, 2018; last retrieved at 10.06.2024.

[Wri] John Wright. *Lecture 4: Groverâs Algorithm*. URL: https://www.cs.cmu.edu/~odonnell/quantum15/lecture04.pdf. Online; published September 21, 2015; last retrieved at 10.06.2024.