Leiden University

Faculty of Computer Science

Solving SAT on noisy quantum computers

Name: Martijn Swenne

Date: 10/07/2019

1st supervisor: Vedran Dunjko
2nd supervisor: Alfons Laarman

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Quantum Computers are a new paradigm for computation based on quantum principles, which has the theoretical potential of substantial speed-ups over classical computers. In this thesis we explore the applications of quantum computers for boolean satisfiability problems called SAT. The topic of SAT solving has always been extremely important. SAT problems are NP-complete and all NP-problems can be reduced to another NP-complete problem. Although we expect no truly efficient algorithms for all SAT problems, since SAT is very important in Artificial Intelligence and other applications, much effort is dedicated in producing ever faster algorithms. There is a quantum algorithm, called Grover's search algorithm, that can be used to speed-up many of the classical algorithms, and it can also be directly used to solve SAT. Although Grover's search is not faster in solving 3-SAT than the fastest classical algorithm for 3-SAT, it will be faster for some large enough k-SAT, under certain complexity theoretical assumptions. The studying of Grover's algorithm is also important since Grover's algorithm will be a component of quantum-enhanced algorithms [1][2]. Unfortunately current quantum computers are very small and noisy. Current real-world quantum computers have only few quantum bits available, which means we need to find an implementation of Grover's algorithm that is as space-efficient as possible. It is also not yet possible to do fault-tolerant quantum computing, which is why researching the effects of noise will be interesting and rewarding.

In this thesis we give an introduction to quantum computing, as well as satisfiability problems. After this, we describe the quantum computing framework Qiskit and it's applications. Then we will give the basic implementation of Grover's search algorithm and apply this implementation to SAT, after which we will provide new optimizations of a space efficient implementation of Grover's search algorithm. In the last part of this thesis we will present some results on the noise effects after some noise model is applied to the space efficient Grover's search algorithm.

Contents

1 Introduction

Quantum computing is an upcoming term in our society. However researchers have already been investing this field for some decades. Quantum computing is based on quantum theory, using phenomena such as entanglement and superposition. A quantum computer can perform quantum computations. Currently, quantum computing is best known for factorization and quantum search.

Integer factorization means finding the prime factors that compose a certain number, and is often used in cryptography. Cryptography is a security mechanism that is used by for example banks or secure websites. There is currently no known classical algorithm that can solve the integer factorization problem efficiently in polynomial time. However, using quantum computing, this problem can be solved efficiently in polynomial time, which is not yet possible with classical algorithms. This means current cryptography systems can be broken with quantum computing. Besides integer factorization, quantum computing can be used for a quantum database search. An algorithm that is used for this purpose is called Grover's search algorithm and has a quadratic speedup over classical algorithms. One of the possible implementations of Grover's search algorithm is solving satisfiability problems, also called SAT problems. It is known that a lot of problems can be mapped to satisfiability problems, which makes this an important problem to have an efficient algorithm for.

Unfortunately, making a large-scale quantum computer is a big challenge that requires more research. This means there is not much working space for these algorithms yet, which is why creating space-efficient algorithms is crucial.

One of the greatest problems in quantum computing is noise. Current quantum computers are very noisy, which means there is a loss of important data. Correcting the errors that happen in a quantum computer is currently a rapidly growing field of research within quantum computing.

Space-related optimizations for Grover's search algorithm will be researched, after which the effects of noise on running a space-efficient implementation of Grover's search algorithm will be tested. The research question this thesis is going to address will be:

" What are the effects of noise on a space-efficient implementation of Grover's search algorithm solving SAT problems? "

This thesis will first give information about the basics of quantum computation and how to use simulations to implement a quantum algorithm. Next, some space-related optimizations for Grover's search algorithm will be presented. Lastly, The results of experiments will be presented using different levels of noise applied to a space-efficient implementation of Grover's search algorithm. These results will help to gain understanding of what the effects of noise are on quantum algorithms solving satisfiability problems. It will also be an indication up until which level of noise these algorithms will no longer be accurate and what different factors in these algorithms have a possible influence on the accuracy while using noise.

2 Quantum Computing

Quantum Computers have the potential of substantial speed-ups over classical computers. For example, Shor's algorithm can factor integer numbers in a polynomial time, which currently cannot be done with any known classical algorithm. This is an exponential speed-up, but it is very specific. Grover's search algorithm however, is more broadly applicable. Grover's search algorithm can brute force search an unordered data base for a target data entry with a quadratic speed-up over classical brute force. Unfortunately, current quantum computers are very small and they usually only have a few qubits you can work with (see Figure 1). They are also very noisy, which is currently a very big problem in practical quantum computing. Finding ways to combat noise or use noisy quantum computers is a central focus of research in the quantum computing community.

2.1 Quantum bits

A quantum bit (or qubit) is the basic unit of quantum information. It is the quantum version of the classical bit. A classical bit has to be in one of two states, namely 0 or 1. However, the qubit can be in a *coherent superposition* of both states simultaneously, a property which is fundamental to quantum theory.

Coherent superpositions are expressed in terms of so-called basis states of a physical system. This simply means that the value of a single qubit can be read out using a measurement device, which is always going to report a 0 or a 1. Nonetheless, before measurement, a qubit can be in a coherent superposition of the basis states, which is written as $\alpha|0\rangle + \beta|1\rangle$. The values α and β are so called amplitudes: complex numbers which satisfy a normalization condition described shortly. The values of the amplitudes dictate the probabilities of the measurement outcomes. The probability of measuring $|0\rangle$ is $|\alpha|^2$ and the probability of measuring $|1\rangle$ is $|\beta|^2$. Because the absolute square of the amplitude of each state equals to the probability of measuring that state, α and β are constrained by $|\alpha|^2 + |\beta|^2 = 1$.

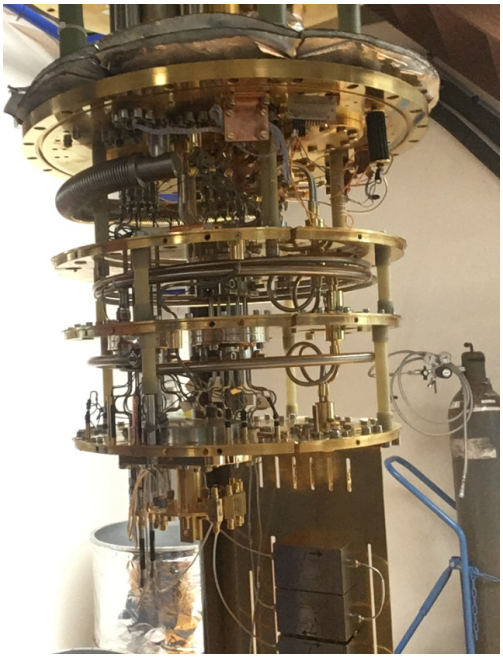


Figure 1: A particular implementation of a so-called superconducting quantum computer. Most of the Figure presents a cooling system and controls, whereas the qubits themselves are at the very tip (bottom) of the device.

Quantum computers use registers of many qubits, so we need to describe many-qubit systems. An n-qubit system has 2^n basis states: all possible combinations of zeroes and ones that could possibly be measured. Then, by the principle of superposition described above, a general state of n qubits can be written as $a_1|00\ldots0\rangle + a_2|00\ldots1\rangle + \cdots + a_{2^n}|11\ldots1\rangle$, where $\sum |\alpha_i|^2 = 1$. The n qubits can be represent by a numerical vector $[a_1, a_2, \ldots, a_{2^n}]^T$.

For example, let $n = 2$. Then the state of our qubits will be $|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$ and the state vector will be $[\alpha, \beta, \gamma, \delta]^T$, where $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$.

2.2 Quantum gates

Quantum gates can be used to manipulate the state of qubits. They usually operate on a small number of qubits. Unlike many classical gates, quantum gates must be reversible. This means that, for instance, a standard OR gate cannot be a quantum gate, because there is a loss of information. When the output of the logical OR gate is 1, there is no unique way to find the input, thus is the classical OR gate not reversible.

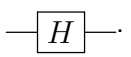
Quantum gates are represented by unitary matrices which preserve the normalization of quantum states. A square matrix U is unitary when the conjugate transpose $U^\dagger$ is equal to the inverse of U. This means that $UU^\dagger = U^\dagger U = I$, where I is the identity matrix. The number of input and output qubits for a gate must be the same.

2.2.1 The Hadamard gate

The Hadamard gate is one of the basic gates in quantum computing. The gate operates on a single qubit and maps the basis state $|0\rangle$ to $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and the basis state $|1\rangle$ to $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$, which means that measuring the output qubit will result in the outcomes 0 and 1, with equal probabilities. But note that, prior to measurement, the quantum states $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$ are distinct and, in fact, orthogonal. The Hadamard gate is represented by the Hadamard matrix:

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

When presenting the Hadamard gate using the circuit representation in the next Section, the Hadamard gate will be visualized as:



Multiplying the Hadamard matrix with the vector representation of a qubit in state $|0\rangle$, which is represented by matrix vector $[1,0]^T$, will result in:

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

Whereas multiplying the Hadamard matrix with the vector representation of a qubit in state $|0\rangle$, which is represented by matrix vector $[0,1]^T$ will result in:

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

These outcomes can be rewritten as $\frac{1}{\sqrt{2}}[1,0]^T + \frac{1}{\sqrt{2}}[0,1]^T$ and $\frac{1}{\sqrt{2}}[1,0]^T - \frac{1}{\sqrt{2}}[0,1]^T$ respectively, which can be rewritten as the states $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$, as given before.

2.2.2 The Pauli gates

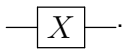
There are three Pauli gates, namely the Pauli X gate, the Pauli Y gate, and the Pauli Z gate. All three gates operate on one qubit. They represent a rotation of the state vector of a qubit around three different axes (X,Y,Z) in the space of a single qubit. Note that if the amplitudes had been real numbers, there would have been only one axis of rotation. Since they are complex numbers, we have 3 geometric axes, and the single qubit space will be represented by a sphere.

The Pauli X gate, also called the NOT gate, is a 180-degree rotation over the X-axis in the

Hilbert space of the qubit's state vector, which is the analogue of a classical NOT gate. It maps $|0\rangle$ to $|1\rangle$ and $|1\rangle$ to $|0\rangle$ and is also sometimes called the bit-flip. It is represented by the Pauli X matrix, given by:

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

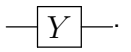
The Pauli-X gate will be visualized in the circuit representation as:



The Pauli Y gate is a 180-degree rotation over the Y-axis in the Hilbert space of the qubit's state vector. It maps $|0\rangle$ to $i\,|1\rangle$ and $|1\rangle$ to $-i\,|0\rangle$, where i stands for $\sqrt{-1}$. It is represented by the Pauli Y matrix:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}.$$

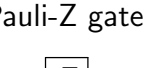
The Pauli-Y gate will be visualized in the circuit representation as:



The Pauli Z gate is a 180-degree rotation around the Z-axis in the Hilbert space of the qubit's state vector. It is a special case of a phase shift gate, which is described below, and is also sometimes called the phase-flip. The gate does not have any effect on the state of the qubit, but it flips the sign of the amplitude of that qubit. It is represented by the Pauli Z matrix:

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

The Pauli-Z gate will be visualized in the circuit representation as:

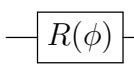


2.2.3 The Phase-shift gate

This gate works on a single qubit. It is a generalization of the Pauli-Z gate, it rotates a certain amount of degrees around the Z-axis in the Hilbert space of the qubit's state vector. This leaves the state $|0\rangle$ unchanged, but maps the state $|1\rangle$ to $e^{i\phi}\,|1\rangle$. This is special because this means that the probability of measuring $|0\rangle$ or $|1\rangle$ is unchanged after applying this gate, while it does modify the phase of the quantum state.

$$R_\phi = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}.$$

where ϕ is the phase shift (or the phase) that is applied.
The phase-shift gate will be visualized in the circuit representation as:

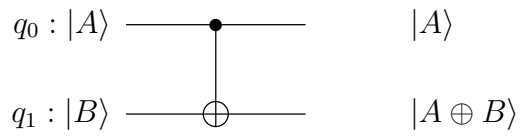


2.2.4 The controlled-NOT gate

The controlled-NOT gate, also called the CNOT gate or CX gate, is a gate that operates on two qubits. It performs a NOT gate on the second qubit (q_1), if and only if the first qubit (q_0) is in the $|1\rangle$ state. The matrix representation of this gate is as follows:

$$CNOT = CX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

The CNOT gate will be visualized in the circuit representation as:

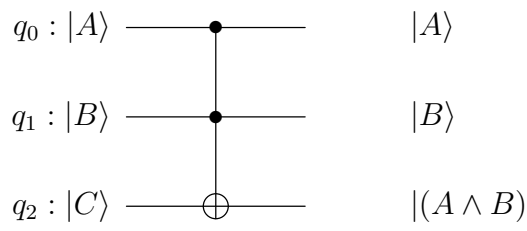


2.2.5 The Toffoli gate

The Toffoli gate, also called the controlled-controlled-NOT gate, CCNOT gate or CCX gate, is a gate that operates on three qubits. It performs a NOT gate on the third qubit (q_2), if and only if the first qubit (q_0) and the second qubit (q_1) are both in the $|1\rangle$ state. The matrix representation of this gate is as follows:

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

The Toffoli gate will be visualized in the circuit representation as:



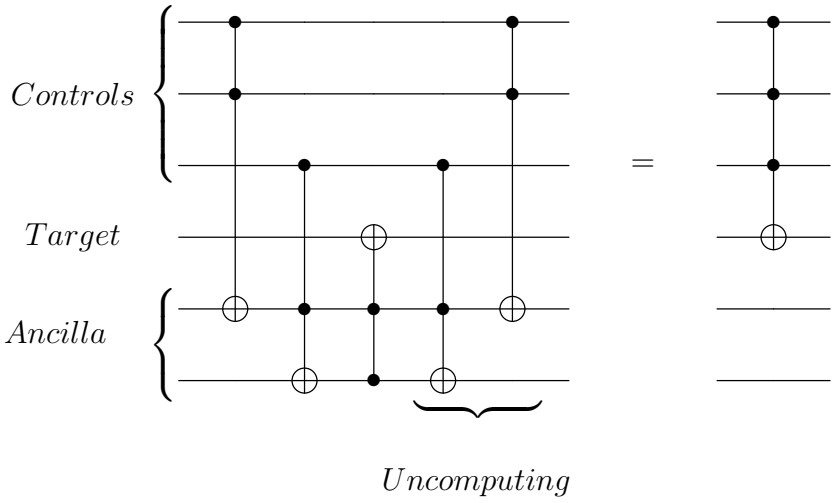
The Toffoli gate is classically universal, which means that for any desired Boolean function computation there exists a sequence of Toffoli gates that will the desired formula in a reversible manner using some ancilla bits. For instance, you can make a classical AND, XOR and NOT by initializing the qubits in the circuit representation of the Toffoli gate as follows:

1. You get $A \wedge B$ by initializing C to state $|0\rangle$.
2. You get $A \oplus C$ by initializing B to state $|1\rangle$ and $B \oplus C$ by initializing A to state $|1\rangle$.
3. You get $\neg(A \wedge B)$ by initializing C to state $|1\rangle$ and $\neg C$ by initializing both A and B to state $|1\rangle$.

2.3 Quantum Circuits

Quantum circuits can be represented using the quantum circuit representation, as already seen in Section 2.2. A quantum circuit is a model for a quantum computation, where a computation is a sequence of quantum gates. Note that the gate model is also a classical model of classical computation. These gates work on a register. Knowing how many qubits you need for your computation is very important. Additional qubits which are used to implement desired unitary transformations, are known as ancilla qubits, or ancillas.

Since quantum computers do not have many qubits to work with, it is important to recycle your ancillas. This is done with uncomputation, which is a technique for cleaning up temporary side effects on ancillas so they can be re-used. An example of this is seen in the circuit representation below:



3 Quantum Simulators

Making a large-scale quantum computer is still very difficult to do and the number of qubits is still very low. Since there are no large experimental platforms, quantum simulators are critical for testing and debugging quantum algorithms such as a SAT solving version of Grover's algorithm and investigating aspects such as noise. To classically compute the output of a quantum circuit using a quantum simulator, you can represent the state of a register of qubits as vectors (Section 2.1) and the quantum gates as unitary matrices (Section 2.2). Then computing the result of a quantum circuit can be done by multiplying the unitary matrix of each quantum gate in the circuit, in the order that is given in the circuit. Then the whole circuit can be represented by the resulting unitary matrix. A local unitary on one or more qubits actually corresponds in the linear algebraic language to a matrix matching the entire register size. This is because a local U is actually $I \otimes U \otimes I$, where I is the identity matrix. The identities signify that nothing is done on the other qubits, but the overall matrix is still 2^n dimensional. Since this grows exponentially, classical computations of quantum circuits are not efficient for circuits with a large number of qubits.

3.1 IBMQ

IBM allows on-line access to their NISQ (Noisy intermediate-scale quantum) devices [3]. IBM Q System One consists of several custom components and is the most used system at the moment.

3.2 Qiskit

The Quantum Information Science Kit, or Qiskit [4], was founded by IBM Research to allow software development for this cloud quantum computing service. The primary version of Qiskit uses Python as a programming language. Swift and Java Versions are also available. These are used to create quantum programs based on the OpenQASM[1] representation of quantum circuits. Qiskit currently contains 4 libraries, namely Terra, Aer, Aqua and Ignis:

- Terra provides the foundation for the software. It contains tools for composing quantum programs at the level of circuits and managing the execution of experiments on remote-access backends. The backend can be described as "black-box" code that can be used run simulations. It is remotely accessed with an IBM token.
- Aer provides a high-performance simulator framework for the software. It contains simulator backends for executing circuits compiled in Qiskit Terra. It also has tools for constructing noise models for performing realistic noisy simulations.
- Aqua contains a library of quantum algorithms out of which quantum applications can be built. Aqua is extensible, and quantum algorithms can easily be added.
- Ignis is a framework for understanding noise in quantum circuits and devices. It provides self-contained experiments that include tools for generating the circuits that can be executed on real backends via Terra (or on simulators via Aer) and the tools for fitting the results and analyzing the data.

3.2.1 Building circuits

To build circuits and run those on simulators, Qiskit is needed. After that, Qiskit can be imported in a Python script. To access the real quantum computers and classical simulators an API token is needed. An API token can be stored locally for later use by running the following Python code:

```
from qiskit import IBMQ
IBMQ.save_account('MY_API_TOKEN')
```

where `MY_API_TOKEN` should be replaced with the token received from IBMQ. Using this token is done by running this line:

```
IBMQ.load_accounts()
```

First `ClassicalRegister, QuantumRegister, QuantumCircuit` and `execute` need to be imported. A circuit can be made, consisting of a classical register and a quantum register. Note that extra registers can also be added later to the circuit. Some basic gates are added by executing the following functions of the circuit:

[1] Open Quantum Assembly Language

- `.x(q[_])` #NOT gate
- `.z(q[_])` #Z gate
- `.cx(q[_],q[_])` #Controlled-NOT gate
- `.ccx(q[_],q[_],q[_])` #Toffoli gate
- `.measure(q[_],c[_])` #measuring gate

One or more variables need to be given, namely in which quantum register and on which qubit of that register the gate needs to work on. Note that, when measuring a qubit in a coherent superposition, it will collapse to a basis state. Measuring a qubit will store this basis state in the classical register.

```
# Import qiskit parts
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit import Aer, IBMQ, execute

# Construct quantum circuit
q = QuantumRegister(2)
c = ClassicalRegister(2)
qc = QuantumCircuit(q,c)

# Add gates to the circuit
qc.h(q[0])
qc.cx(q[0], q[1])

# Measure the qubits
qc.measure(q[0], c[0])
qc.measure(q[1], c[1])

# Select the QasmSimulator from the Aer provider
simulator = Aer.get_backend('qasm_simulator')

# Execute and get counts
result = execute(qc, simulator).result()
counts = result.get_counts(qc)
print(counts)

```

Figure 2: Qiskit code

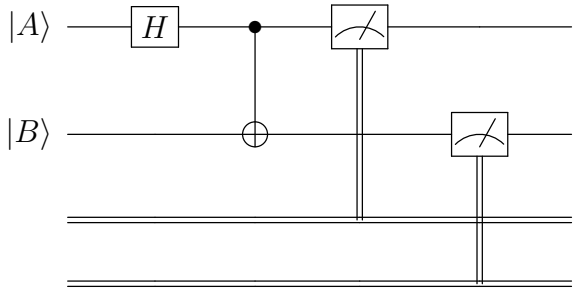
To simulate a circuit on a backend, `Aer` needs to be imported. Then:

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

will call the `qasm_simulator` backend and run the circuit. For a real quantum computer, for instance the `ibmq_5_tenerife` backend, can be called. Running the circuit on this backend is done by running this code:

```
execute(circuit, backend).result()
```

This gets outputs of the result of executing the circuit on the chosen backend, with the registers initialized in an "all zero" state. If no amount is given, the function will shoot a basic 1024 shots. The number of shots is the amount of repetitions of the circuit. The output of one shot will be a state vector corresponding to the measured register. Each qubit in the circuit will have a probability of ending up in a specific basis state. This means the measured state of each of these qubits can differ. Printing all outputs of the result will return a histogram of all states after a number of shots. If a qubit is not measured in a circuit, the result will always show a zero. The Qiskit code as seen in Figure 2 generates the circuit below. Note that we use double wires in quantum circuits to denote classical information. Running it outputs a bell-state, which means that qubits A and B are maximally entangled. The outputs will be $50\%\,|00\rangle$ and $50\%\,|11\rangle$, disregarding the noise of a quantum computer.



3.2.2 Custom gates

Since Qiskit is still in development, and the functions for adding custom gates did not work for me personally. But, it is possible to add custom gates in the code of Qiskit manually. First, the gate has to be defined and needs to be saved in the folder `qiskit/extensions/standard`.

Then the gate has to be added to the `__init__.py` file in the same folder. After that, the gate needs to be added to the `__init__` function in the file `_dagcircuit.py`.

At last, the visualization of the gate needs to be added to qiskit. In the folder `qiskit/tools/visualization` the file `_text.py` can be found, where the `build_layers` function needs to be edited. Here the visualization of the gate is set.

4 SAT problems

In computer science, the Boolean satisfiability problem, or the SAT problem, is the problem of determining if there exists an assignment of all the variables, where each variable can have the assignment `True` or `False`, such that it satisfies a given Boolean formula. If this is the case, the formula is said to be satisfiable, if not, it is unsatisfiable. A boolean formula F is defined with the following set of rules:

1. A literal l is a variable or the negation of a variable.
2. A literal l is a formula.
3. if F_1 and F_2 are formulas, then $F = (F_1 \wedge F_2)$ is a formula.
4. if F_1 and F_2 are formulas, then $F = (F_1 \vee F_2)$ is a formula.
5. if F is a formula, then $\neg F$ is a formula.

Boolean formulas can be in conjunctive normal form, or CNF. This means that the literals are grouped together in clauses. In this thesis we only care about k-CNF formulas. The definition is as follows:

1. A k-clause is a conjunction of k literals.
2. A k-CNF formula is a conjunction of a set of clauses.

A common SAT problem is the 3-SAT (k-SAT) where the formula is a 3-CNF formula, where each clause is limited to at most three literals. A 3-CNF formula can be written like:

$$F(x_1 \dots x_n) = (l_1 \vee l_2 \vee l_3) \wedge \cdots \wedge (l_{m-2} \vee l_{m-1} \vee l_m)$$
where m = total number of literals and $l_i = x_j$ or $\neg x_j$ for some j $(0 < j < n)$

3-SAT problems are NP-complete, which means that a a satisfying assignment can be verified in polynomial time, but finding a satisfying assignment is believed to take exponential time. A classical brute force solves it in $O(2^n)$. However, there is a simple randomized algorithm by Schöning [5] that runs in time $O((\frac{4}{3})^n)$ (where n is the number of variables in the 3-SAT formula), and succeeds with constant probability to correctly decide 3-SAT (Note that there have been improvements on this, which we will not consider in this thesis [6]).

The simplest quantum algorithm to solve SAT is called Grover's search algorithm, that can solve SAT brute force in $O(2^{n/2})$. This is substantially faster than the classical brute force, but still slower than Schöning's algorithm. Note that, although you can quantum-speed-up Schöning's algorithm, it needs many qubits [2].

Even though there exist faster algorithms than Grover, for some k-SAT problem where k is high enough, Grover will actually be faster. Let c_k be the sequence of coefficients such that the best algorithm for k-SAT takes at least $O(2^{c_k n})$. The *Strong Exponential Time Hypothesis*, or SETH, states that the (increasing) sequence c_k converges to 1 [7]. Consequently, for any $\alpha < 1$ there exists a k such that $c_k > \alpha$. In other words, for very large k, the best algorithms are essentially brute force search, which means that for those k, Grover's search will be faster than any classical algorithm.

Also, since Grover's search algorithm is the key subroutine used to speed up many other algorithms, studying just it's performance for SAT will be beneficial.

5 Grover's search algorithm

When searching for one or more inputs to an unknown function with a desired output value, Grover's search algorithm can be used. For SAT, this means we are looking for one or more bit-strings, which renders a given boolean formula `True`. This set of bit-strings that solve our boolean formula will be referred to as ω. Although there is some structure to the function and is not just a "black-box", we do not know how to solve this problem much better than by (directed) search algorithms such as Schöning's algorithm. Grover's search algorithm is a quantum algorithm that finds ω, given there are any, with a high probability given just black-box access to the function. It does this using $O(\sqrt{N})$ evaluations of the function, where N is the size of the set of all possible bit-strings, which is exponential (2^n) to the number of inputs n.

5.1 The Algorithm

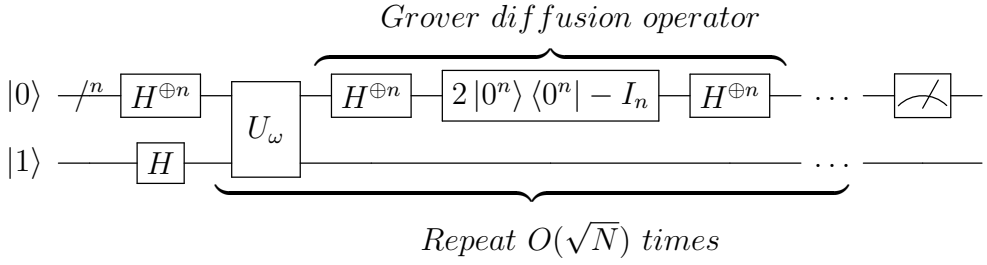
Grover's search algorithm has four basic elements:

- The *initialization* of the qubit register, described in Section 5.1.1.
- The *oracle* is the representation of the unknown function, which is in our case the boolean formula that we want to solve. This is described in Section 5.1.2.
- The *mean inversion*, also known as the Grover diffusion operator, is described in Section 5.1.3.
- The *measuring* of the qubit register.

The algorithm is seen in the circuit below and goes as follows:

1. Initialize the system to the uniform superposition over all states.
2. Perform the following "Grover iteration" $r(N)$ times, where $r(N)$ is a certain number depending on N, which we will specify later:
   (a) Apply the oracle
   (b) Apply the mean inversion
3. Perform the measurement of the qubits representing the variables.

If t is the number of bit-strings in ω and N is the number of all possible bit-strings, it can be proven that if we set $r(N)$, which is the number of iterations of (a) and (b), to be approximately $\frac{\pi\sqrt{N/t}}{4}$, then the final probability of finding ω after all iterations is lower bounded by $\frac{1}{2}$.



5.1.1 Initialization

Before running the Grover iteration, the initial state of the qubit register needs to be set. Let x be all qubits except b, which represent all the variables in our formula. The oracle qubit b is pre-set in the state $|1\rangle$. Then a Hadamard gate is applied to every qubit in x. This sets x in a uniform superposition, which means that every possible bit-string of x has an equal probability of occurring. Since b was set to state $|1\rangle$, after the Hadamard gate it will be in state $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$, denoted as $|-\rangle$.

5.1.2 Oracle

The oracle is in our case the matrix representation of our boolean formula. A function is applied on x, such that $f(x) = 1$ when $x \in \omega$ and $f(x) = 0$ when $x \notin \omega$. This can be written as $x \xrightarrow{f} f(x)$. In quantum computing this can be represented by $|x\rangle |b\rangle \xrightarrow{f} |x\rangle |b \oplus f(x)\rangle$. Since b is set to $|-\rangle$ in as described in Section 5.1.1, we can show that:

$$|x\rangle |-\rangle \xrightarrow{f} -|x\rangle |-\rangle \quad \text{for } x \in \omega \text{ and}$$
$$|x\rangle |-\rangle \xrightarrow{f} |x\rangle |-\rangle \quad \text{for } x \notin \omega$$

It results in a bit-flip on x, which is why this oracle is also called the bit-flip oracle. x flips when $f(x) = 1$, yet b is unchanged. This is called the phase-kickback. Applying the oracle on b kicks the phase (in our example a bit-flip) back on x.

Let U_ω be the matrix representation of our boolean formula, which is a function $f(x)$, that maps the variables x to 0 or 1, where $f(x) = 1$ if and only if x satisfies the criterion $(x \in \omega)$. This means that U_ω acts as follows:

$$U_\omega |x\rangle = |x\rangle \quad \text{for } x \notin \omega \quad (f(x) = 0)$$
$$U_\omega |x\rangle = -|x\rangle \quad \text{for } x \in \omega \quad (f(x) = 1)$$

5.1.3 Mean inversion

After the inversion, every amplitude of x is flipped about the mean of all amplitudes of x. This is denoted as $H^{\oplus n}(2 |0^n\rangle \langle 0^n| - I)H^{\oplus n}$. Since the oracle flipped the sign of the amplitude of each input $x \in \omega$, it means that these amplitudes are below the mean, while other inputs are above the mean. This way the amplitudes of each input $x \in \omega$ get increased, while the amplitudes of inputs $x \notin \omega$ get decreased.

5.2 Solving k-SAT using Grover's Algorithm

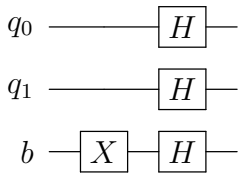
As explained in Section 5, Grover's search algorithm searches for ω. For the case of the k-SAT problem, the black box encodes the desired boolean formula. This means that every clause needs to output 1 for each input $\in \omega$. Therefore the result of every clause needs to be stored. For a given boolean k-SAT formula with n variables and m clauses, a naive implementation needs at least $n + 1 + m$ qubits. The n qubits are needed to represent the input variables, 1 qubit is needed for the phase-flip oracle and m ancillas, are needed to store the results of each clause. So instead of the phase-flip working on the n qubits, now the m qubits are used, since these are the results of each clause and thus the result of the boolean formula.

For this method, an ancilla is needed for every clause in the k-SAT formula. This means that, because there is a limited amount of qubits, there is also a limit on the number of clauses in the formula. This is not optimal, but it can be improved in a few ways.

5.3 Implementing Grover's algorithm

In this section we will exemplify how to implement a naive implementation of Grover's algorithm for the formula $(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$. With 1 satisfying assignment, namely $x_1 = \texttt{True}$ and $x_2 = \texttt{True}$, this boolean formula needs only 1 iteration, so this implementation contains one of each of the four elements of Grover's search, as described in Section 5. In the circuit representation, x_1 will be represented by q_0, x_2 will be represented by q_1, b will be the oracle qubit, a_0, a_1 and a_2 will be ancillas and c_0 and c_1 will be classical bits. Note that the register is set to an "all-zeroes" state, so in our case $|00\rangle\,|0\rangle\,|000\rangle$, before applying any of the implementations below.

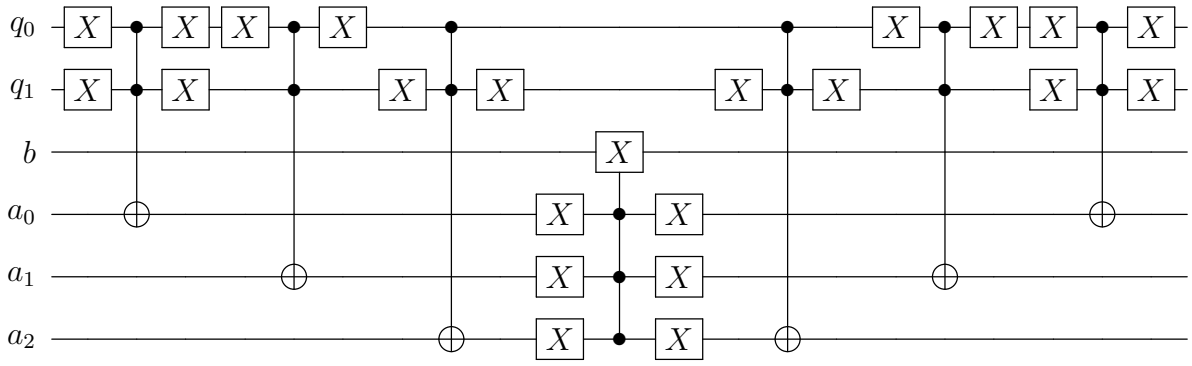
As described in Section 5.1.1, the oracle qubit b is pre-set in the state $|1\rangle$, after which a Hadamard gate is applied to every qubit. Visualizing the example boolean formula in the circuit representation, the Initialization part looks like this:



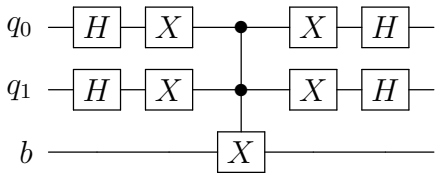
Implementing the oracle as described in Section 5.1.2 means rewriting our boolean formula to a quantum circuit representation. We sequentially check if each clause is `True` or `False`, which is saved in an ancilla. This is done by using the qubits that represent the variables in the clause. We can understand every clause as a function which yields `False` for exactly one assignment. Checking if a few bits are in one specific desired assignment, namely "all-ones" is easily checked by doing a multiply-controlled-NOT gate. NOT gates are applied to those qubits which represent variables that do not appear negated in the clause. After this a multiple-controlled-NOT gate is used, where the controls are the qubits that represent the variables in the clause, and the target is an ancilla. After this the qubits where we applied a NOT gate are uncomputed by applying a NOT gate again on these qubits.

For instance, if in our example both variables are `False`, the first clause will be `False`. If both variables are `False`, both qubits representing those variables will be in state $|0\rangle$. Because NOT gates are applied, this would mean both qubits are in state $|1\rangle$. So when using a multiple-controlled-NOT gate, the ancilla will be in state $|1\rangle$ if the clause is not satisfied.

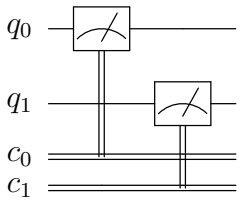
After the results of every clause is stored in ancillas, the ancillas will be in an "all-zeroes" state if all clauses are satisfied. We apply a NOT gate on every ancilla, so if all clauses are satisfied, the ancillas will now be in an "all-ones" state. Now a multiple-controlled-NOT gate is applied using all ancilla as controls and qubit b as target, which will result in the bit-flip as described in Section 5.1.2. After this we uncompute every ancilla by running the above again (without the multiple-controlled-NOT gate). Visualizing the example boolean formula in the circuit representation, the Oracle part looks like this:



The inversion as described in Section 5.1.3 is denoted by $H^{\oplus n}(2\left|0^n\right\rangle\left\langle 0^n\right| I)H^{\oplus n}$. This is implemented by first applying a Hadamard gate on every qubit in x. Because in the initialization a Hadamard gate is applied to every qubit in x, applying a Hadamard gate on every qubit in x again ensures that all qubits in x are back to state $|0\rangle$. After this, a NOT gate is applied on every qubit in x, to make sure the qubits are in an "all-ones" state. This ensures the multiple-controlled-NOT gate which follows is applied, where all qubits in x are the control qubits and b is the qubit that is acted upon. After this, x is uncomputed by again applying a Hadamard gate to every qubit in x. Visualizing the example boolean formula in the circuit representation, the mean inversion part looks like this:



The oracle and inversion implementation is repeated for $r(N)$ iterations, as described in Section 5.1, which for the example boolean formula is just 1 iteration. At last we apply a measurement gate on every qubit in x. Visualizing the example boolean formula in the circuit representation, the measurement part looks like the image below. Note that we use double wires in quantum circuits to denote bits.



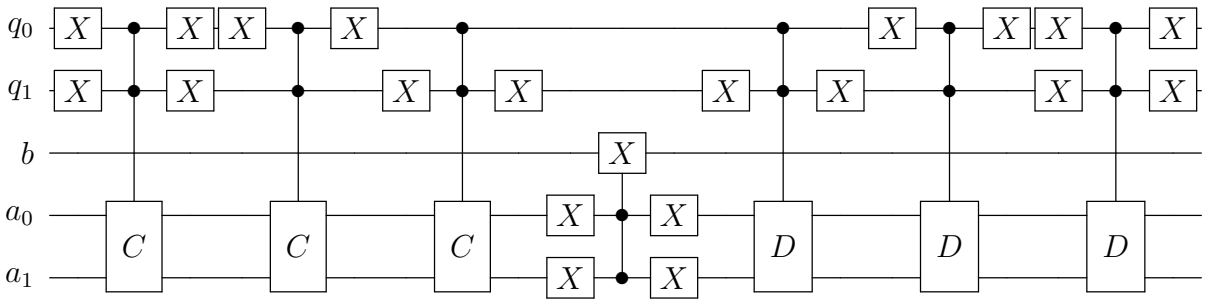
6 Results

Recall that there are currently only few quantum bits available, it means we need to find a space-efficient implementation of Grover's algorithm. Although the implementation in Section 5.3 does work, it is very naive, and can be improved to be much more space efficient. It is also not yet possible to perform quantum computing in a fault-tolerant manner (so achieving error-free results even if the system is noisy), which is why it is important to know what the effects of noise are on solving k-SAT using the space efficient Grover's search implementation.

6.1 Multiple-controlled-incrementer implementation

The number of ancillas can be reduced by implementing a multiple-controlled-incrementer. An incrementer takes as input m qubits, which represent some binary number i, denoted as $bin(i)$, and outputs m qubits, which represent the binary number, denoted as $bin(i+1) \mod bin(m)$. For example, let A be a circuit with 3 qubits initialized in the state $|101\rangle$, which is the binary representation of 5. When an incrementer takes these qubits as input, it will output the same number of qubits, now in state $|110\rangle$, which is the binary representation of 6. Note that incrementing the qubits to $bin(m)$ means that the qubits will overflow (as a result of the $\mod bin(m)$) and the result will be $bin(0)$. A multiple-controlled-incrementer only increments if and only if all controls are 1. This can be used for the clauses.

Let B be a circuit which implements Grover's search on a formula with n variables and m clauses. The number of clauses can be represented by $\lfloor log(m) \rfloor + 1$ qubits (where the log is in 2-base). B will have $n + 1 + \lfloor log(m) \rfloor + 1$ qubits. For every clause, a multiple-controlled-incrementer is needed, where the controls are the qubits that represent the variables in the clause, and the inputs of the incrementer are all ancillas. If and only if all clauses are satisfied, the ancillas will be equal to $bin(m)$. A phase flip controlled on the ancillas is done, given they represent $bin(m)$. To uncompute afterwards, we also need a multiple-controlled-decrementer. A multiple-controlled-decrementer works the same as a multiple-controlled-incrementer, only it's output qubits now represent the binary number $bin(i-1) \mod bin(m)$. Implementing Grover's algorithm with multiple-controlled-incrementers reduces the number of ancillas from m to the exponentially more efficient $\lfloor log(m) \rfloor + 1$. Note that this implementation does use more gates than the naive implementation. Given the example from Section 5.3, we can change the implementation of the Oracle using a multiple-controlled-incrementer (represented by the gate `C`) and a multiple-controlled-decrementer (represented by the gate `D`), which will look like this:



6.2 FPAA implementation

The idea is that, instead of using an incrementer, we use the state space of just a single qubit to "count" the correct clauses, by rotating a little bit. Ideally we want to check if the state $|1\rangle$ is present at all in the system at the end (meaning some rotation happened) or not at all. We could use amplitude amplification here, but unfortunately we do not know how many rotations happened on the qubit. But with Fixed-point amplitude amplification [8], or FPAA, we can fix this.

Grover's search is straightforwardly generalized to "amplitude amplification", where you start in one state and end in another state by amplifying the amplitude of the component of the target state in the initial state, provided the 2 states are not orthogonal. A general problem for Grover's search is knowing exactly how many satisfying assignments there are in ω. Without knowing this, there is no way of knowing how many iterations are needed. This leads to the so-called "soufflé problem"[8], in which iterating too little "undercooks" the result, leaving mostly inputs $\notin \omega$, and iterating too much "overcooks" the result, passing by the inputs $\in \omega$ and leaving us again with mostly inputs $\notin \omega$. An alternative approach is to construct operators that avoid overcooking by always amplifying the target state. Fixed-point amplitude amplification achieves this.

When given a unitary operator A that prepares an initial state $|s\rangle = A|0\rangle^{\oplus n}$, a target state $|T\rangle$ can be extracted with success probability $P_L \geq 1 - \delta^2$, where $\delta \in [0, 1]$ is given. The paper [8] gives a quantum circuit S_L consisting of A, $A^\dagger$, U and efficiently implementable n-qubit gates, such that:

$$P_L = 1 - \delta^2 T_L(T_{1/L}(1/\delta)\sqrt{1-\lambda})^2.$$

Here $T_L(x) = cos(L\ cos^{-1}(x))$ is the L^{th} Chebyshev polynomial of the first kind, $L-1$ is the query complexity and $\lambda \in [0, 1]$ and $\delta \in [0, 1]$ are both given. The paper also states that $L \geq \frac{log(2/\delta)}{\sqrt{\lambda}}$. While the original Grover iterate used $\alpha = \pm\pi$ and $\beta = \pm\pi$, for the generalized Grover iterate $G(\alpha, \beta)$, α and β are given by:

$$\alpha_j = -\beta_{l-j+1} = 2cot^{-1}(tan(2\pi j/L)\sqrt{1-\gamma^2}),$$
$$\text{where } \gamma^{-1} = T_{1/L}(1/\delta) = cos(acos(1/\delta)/L).$$

We adapt and choose these values based on SAT. U distinguishes the bit-string which renders a given boolean formula `True`. So U is an oracle. For every clause we rotate one ancilla by a certain rotation if it is not satisfied. So if all clauses are satisfied, this ancilla is in state $|0\rangle$, and if one or more clauses are not satisfied, the qubit will have a certain rotation $ket\phi$ bigger than zero. We choose this rotation to be $\theta = \frac{\pi}{\#clauses}$, which means that the qubit can have at most a rotation by π radians. Then we use the FPAA with $|\phi\rangle$ as initial state and $|1\rangle$ as target state, which needs only one ancilla. We define δ and λ as:

$\delta = 2^{-(0.5N+1)}$ and

$\lambda = \frac{sin(\theta)^2}{4} + \frac{(1/2 - math.cos(theta)^2}{4}$.

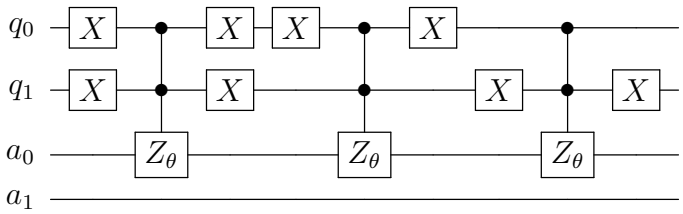
L will be defined as:

$$L = \lceil \frac{2log(2/\delta)}{\sqrt{\lambda}} \rceil \geq \frac{log(2/\delta)}{\sqrt{\lambda}}.$$

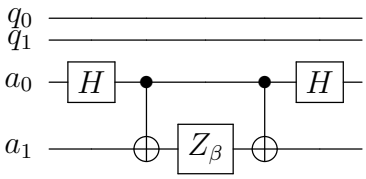
This thesis gives a short example on how to run the Generalized Grover iterate for the example boolean formula as seen in Section 5.3. Note that this is not an alteration of the Grover's search, but a general Grover iteration, which we use instead of the Oracle as seen in Section 5.3. Although only the Oracle implementation is altered, this has some consequences for the

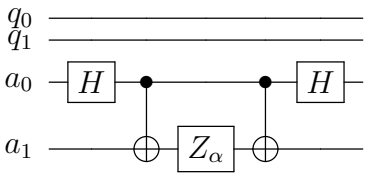
rest of the Grover's search implementation as described in Section 5.3. These changes will be described at the end of this Section.

We will need two ancillas, ancilla a_0 and ancilla a_1. Running the generalized Grover iterate is done by initializing state $|s\rangle = A|0\rangle^{\oplus n}$. A Hadamard gate is first applied to ancilla a_0, where A will work on. Then A is applied. The circuit visualization for A looks like this:



After this, the general Grover iteration is applied L times, which consists out of applying $U(\beta)$, $A^\dagger$, $U(\alpha)$ and A again. Here $A^\dagger$ is the same as the circuit above, but with $-\theta$ applied. The circuit visualizations of $U(\beta)$ and $U(\alpha)$ look like this:





Now the phase-kickback, as described in Section 5.1.2, still needs to happen. For this we put the ancilla a_0 in the $|1\rangle$ state. Since we initialized a_0 in a superposition by applying a Hadamard gate, we again apply a Hadamard gate, after which we apply a NOT gate to put a_0 in state $|1\rangle$. Then we need to put ancilla a_1 in the $|-\rangle$ state, as described in Section 5.1.1. We apply a NOT gate to put a_1 in state $|1\rangle$ and a Hadamard gate to put a_1 in state $|-\rangle$. Now we apply a CNOT gate with a_0 as control and a_1 as target qubit, which will apply the phase-kickback. We uncompute a_0 and a_1 by applying the same Hadamard and NOT gates.

Running Grover's search using the FPAA implementation is done by initializing the qubits representing all variables in a superposition state, as described in Section 5.1.1. But, since we do not have an oracle qubit, we do not need to initialize this in a $|-\rangle$ state. Now we apply the FPAA oracle as described in this Section. We uncompute the $|-\rangle$ state of a_1 described in the paragraph before, which we need for the mean inversion, as described in Section 5.3. So we need to apply a NOT gate, followed by a Hadamard gate before applying the mean inversion and uncompute this after applying the mean inversion. The measurement implementation is the same as described in Section 5.3.

This method reduces the ancillas from $\lfloor log(m) \rfloor + 1$ to a constant 2, which is much more efficient than any of the more naive methods, and relies on simple but genuinely quantum principles. There exist other, more complicated methods which rely on classical and more algebraic properties which can be used for space-efficient formula evaluation [9], and the investigation into which methods are better suited for noisy quantum computers remains to be investigated.

6.3 Noise effects on solving k-SAT using Grover's search

This thesis gives the results of running the Grover implementation as described in Section 6.1 on the `qasm_simulator` from IBM with a modified version of the noise model provided by qiskit. We have modified this noise model, because the noise model provided by qiskit is a model based on an actual quantum computer of IBM and contains multiple noise factors. We removed all noise factors except the `T1` and `T2` noise to make the noise a scalable value; something we can work with.

When a qubit is in an exited state ($|1\rangle$) it can decay into a relaxed state ($|0\rangle$) with a time constant `T1`. A coherent state, as described in Section 2.1, can also decay, which makes the qubit fall into a basis state, with a time constant `T2`.

Although implementation as described in Section 6.2 is one of the possible optimal implementations with respect to space requirements, is not used due to time issues, because the circuit is relatively large and running the circuit is relatively slow compared to the implementation described in Section 6.1.

A threshold is set at 25% correct results from Grover's search. We expect to see an increase of the noise threshold the Grover's search can handle at most, based on the amount of gates is in the implemented circuit. We know that `T1` and `T2` have time constants, and gates take an amount of time to perform. It can be said that running circuits with more gates have a higher chance to have more noise in the system than running circuits with fewer gates. We expect to see that boolean formulas with more clauses will have a higher noise threshold, since each clause needs a certain amount of gates in the Oracle implementation, so more clauses leads to more gates in the circuit.

To show our implementation works correctly, some experiments were done without noise for every 2-SAT formula that is going to to be used during our noise experiments. The results are shown in Figure 3. As can be seen, the results are at least 95% correct for every 2-SAT formula used, so it can be said that our implementation works correct.

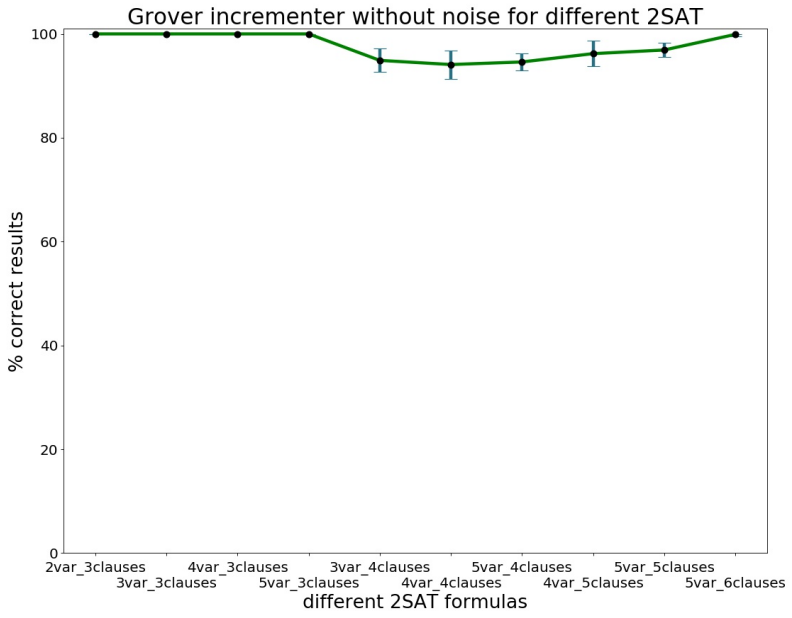


Figure 3: % of correct results without noise for each 2-SAT formula

The experiments were done with 2-SAT formulas, because of time issues. Using higher SAT problems would have taken too much time to get enough results. Note that 2-SAT is not NP hard, while higher SAT problems are NP hard, but this is merely a noise effect analysis, on which not being NP hard does not have any effect, since we solve it using brute force search methods. Each of the different 2-SAT formulas was ran on a broad noise range where the noise was incremented in equal ticks. For each measuring point, 10 runs of Grover's search were done, each with 100 shots. The first and last time the threshold is passed is highlighted. These thresholds can be seen in table 1. The results of each experiment can also be found in Appendix A.1.

2-SAT formula with:	lower bound avg threshold	upper bound avg threshold
2 variables 3 clauses	2.157	2.668
3 variables 3 clauses	2.576	2.872
4 variables 3 clauses	1.964	3.144
5 variables 3 clauses	3.523	4.155
3 variables 4 clauses	28.530	30.025
4 variables 4 clauses	21.560	23.517
5 variables 4 clauses	38.785	38.785
4 variables 5 clauses	44.251	48.081
5 variables 5 clauses	74.444	81.380
5 variables 6 clauses	105.445	122.494

Table 1: Threshold values with a broad noise range.

A more specific range of noise values was set, covering these highlighted values. Then the same boolean formulas were ran, where again the first and last time the threshold is passed is highlighted. For each point in this specific range 50 runs were done, each with 100 shots. These thresholds can be seen in table 2. The results of each experiment can also be found in Appendix A.2.

2-SAT formula with:	lower bound avg threshold	upper bound avg threshold
2 variables 3 clauses	2.649	2.887
3 variables 3 clauses	2.617	2.617
4 variables 3 clauses	2.540	2.540
5 variables 3 clauses	3.829	4.263
3 variables 4 clauses	29.082	31.231
4 variables 4 clauses	23.263	23.263
5 variables 4 clauses	37.789	37.789
4 variables 5 clauses	46.454	47.774
5 variables 5 clauses	74.088	77.554
5 variables 6 clauses	112.544	115.476

Table 2: Threshold values with a specific noise range.

The values from table 2 were then grouped together based on the amount of gates in each different circuit implementation, which were then plotted. This can be seen in Figure 4. The image shows that the threshold increases based on the amount of gates in a circuit. But this is already known, since the amount of noise in the circuit is based on time, and since gates take an amount of time to be applied, the more gates in the circuit, the more the amount of

noise in the circuit will be. But one thing that is not known, is how steep the threshold grows base on the number of circuits. So approximating this slope is actually new information. By knowing this slope, we can more precisely approximate the noise threshold that is needed for a circuit with a certain amount of gates. By doing this, the circuit can be adapted to make it more noise tolerable up to a certain noise threshold.

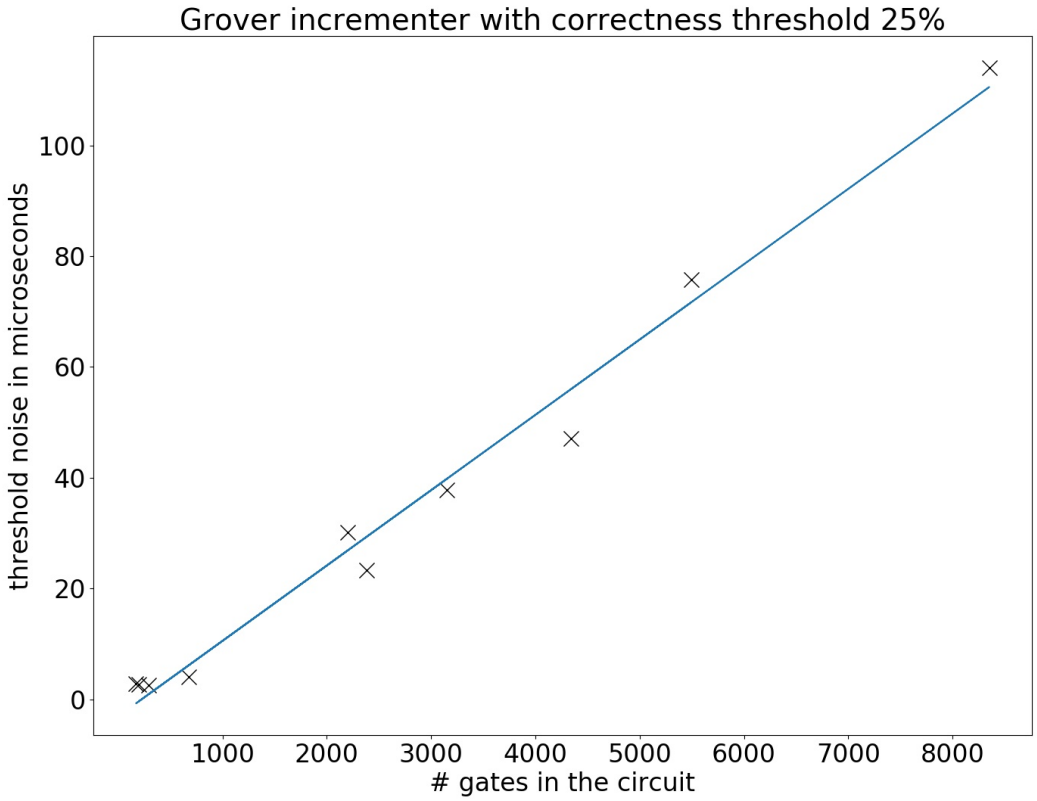


Figure 4: Threshold noise values for different amount of clauses

To summarize, this thesis explains the basics of quantum computing and gives an example of a simulator which can be used to implement quantum circuits. Then this thesis provides several improved space-efficient implementations of Grover's search. Furthermore this thesis presents the results of experiments done with the implementation described in Section 6.1, where we tested the effects of different `T1` and `T2` values on different 2-SAT formulas. The results seen in Appendix A have large standard deviations, but we assume that under uniform noise as we use, the mean is actually correct, despite the large standard deviations. These results show the influence of noise on quantum computers using Grover's search to solve SAT problems. It shows that the number of gates affects the noise threshold of our implementation of Grover's search and approximates the slope of this function that maps the number of circuits to a noise threshold.

References

[1] Vedran Dunjko, Yimin Ge, and J. Ignacio Cirac. Computational speedups using small quantum devices. *Physical Review Letters*, 121(25), dec 2018.

[2] Andris Ambainis. Quantum search algorithms. 2005.

[3] John Preskill. Quantum computing and the entanglement frontier, 2012.

[4] Gadi Aleksandrowicz et al. Qiskit: An open-source framework for quantum computing, 2019.

[5] U. Schoning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039)*. IEEE Computer Society, 1999.

[6] Timon Hertli. 3-SAT faster and simpler—unique-SAT bounds for PPSZ hold in general. *SIAM Journal on Computing*, 43(2):718–729, January 2014.

[7] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. The complexity of satisfiability of small depth circuits. In *Parameterized and Exact Computation*, pages 75–85. Springer Berlin Heidelberg, 2009.

[8] Theodore J. Yoder, Guang Hao Low, and Isaac L. Chuang. Fixed-point quantum search with an optimal number of queries. *Physical Review Letters*, 113(21), nov 2014.

[9] Alessandro Cosentino, Robin Kothari, and Adam Paetznick. Dequantizing read-once quantum formulas. 2013.

A Noise results

A.1 Broad noise interval

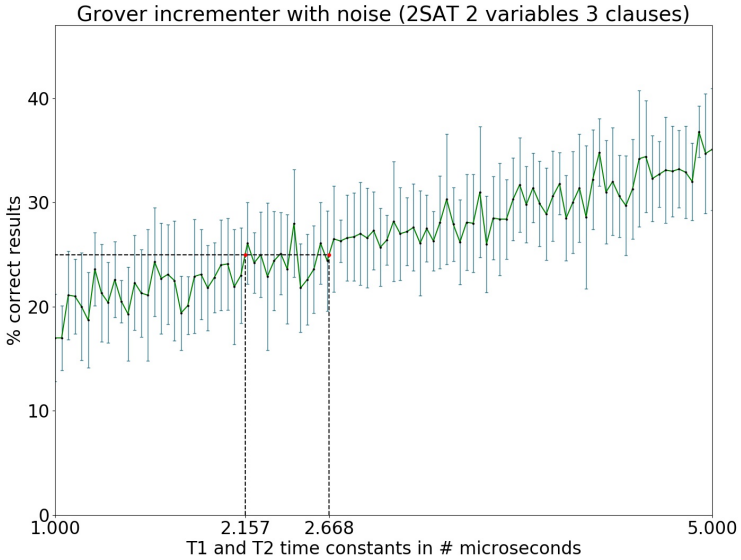


Figure 5: Broad noise interval of 2-SAT with 2 variables and 3 clauses

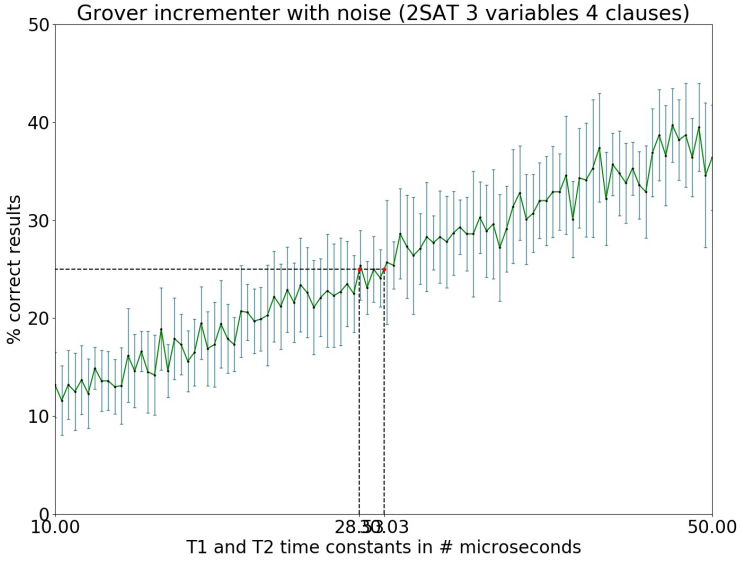


Figure 6: Broad noise interval of 2-SAT with 3 variables and 4 clauses

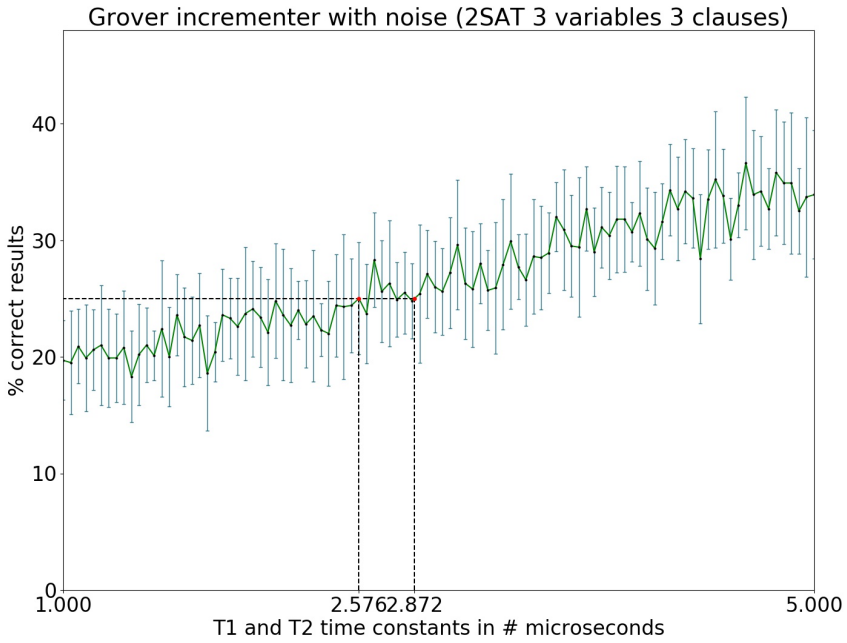


Figure 7: Broad noise interval of 2-SAT with 3 variables and 3 clauses

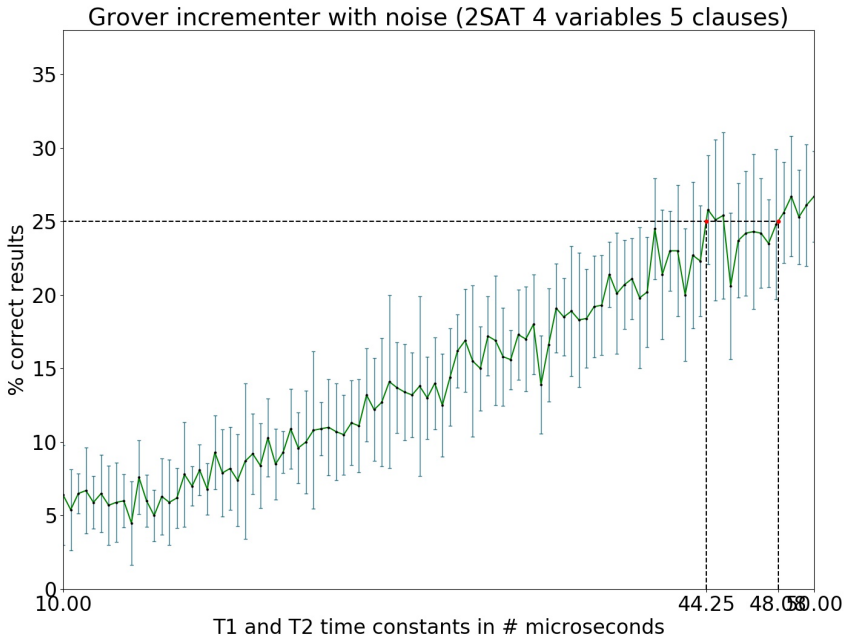


Figure 8: Broad noise interval of 2-SAT with 4 variables and 5 clauses

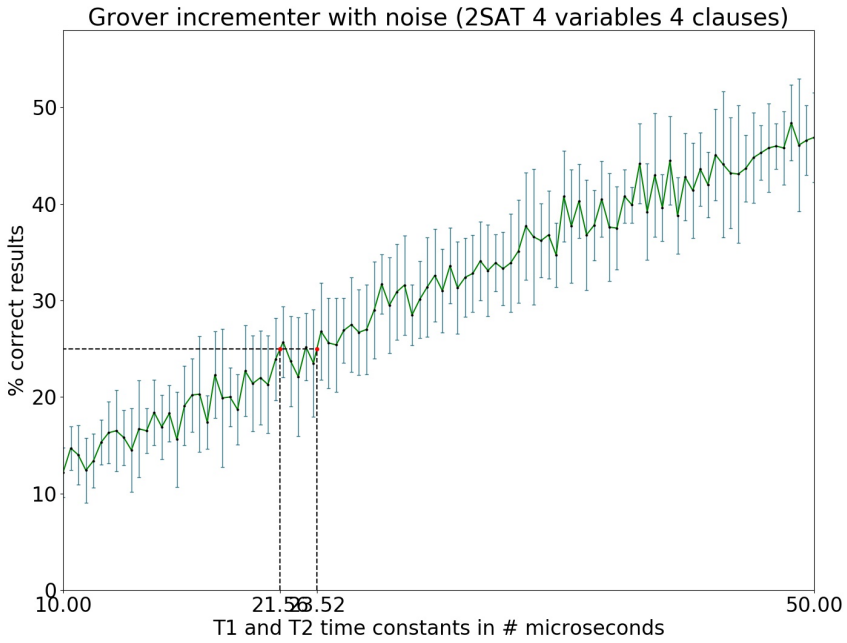


Figure 9: Broad noise interval of 2-SAT with 4 variables and 4 clauses

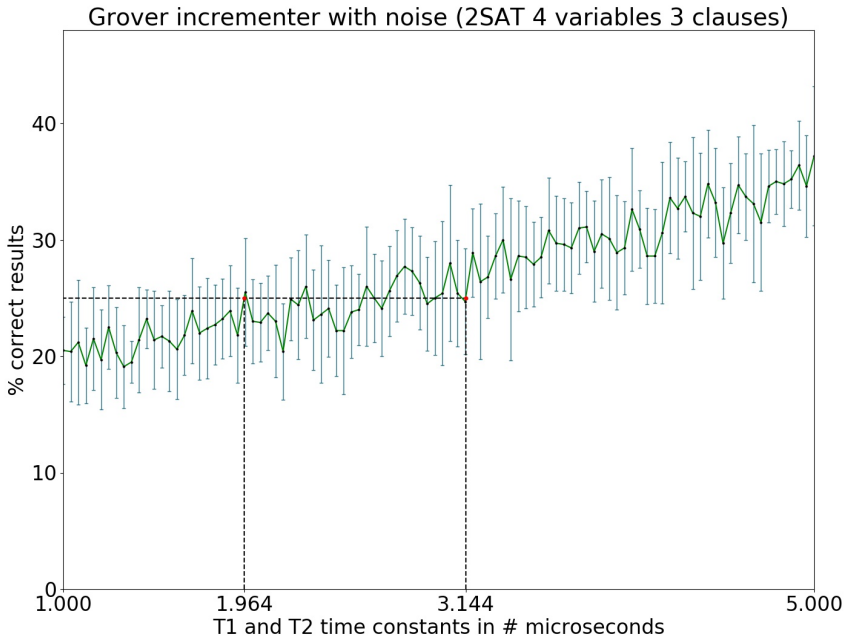


Figure 10: Broad noise interval of 2-SAT with 4 variables and 3 clauses

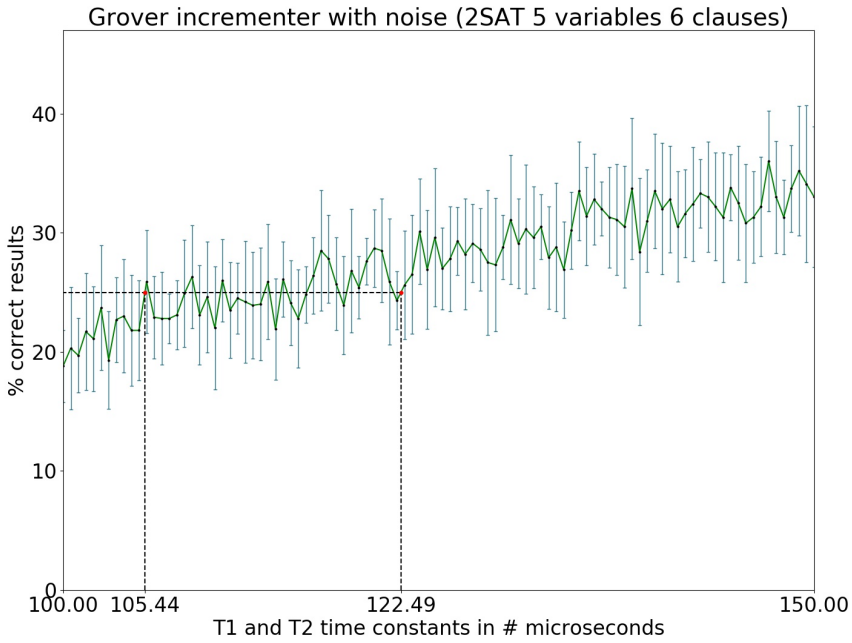


Figure 11: Broad noise interval of 2-SAT with 3 variables and 3 clauses

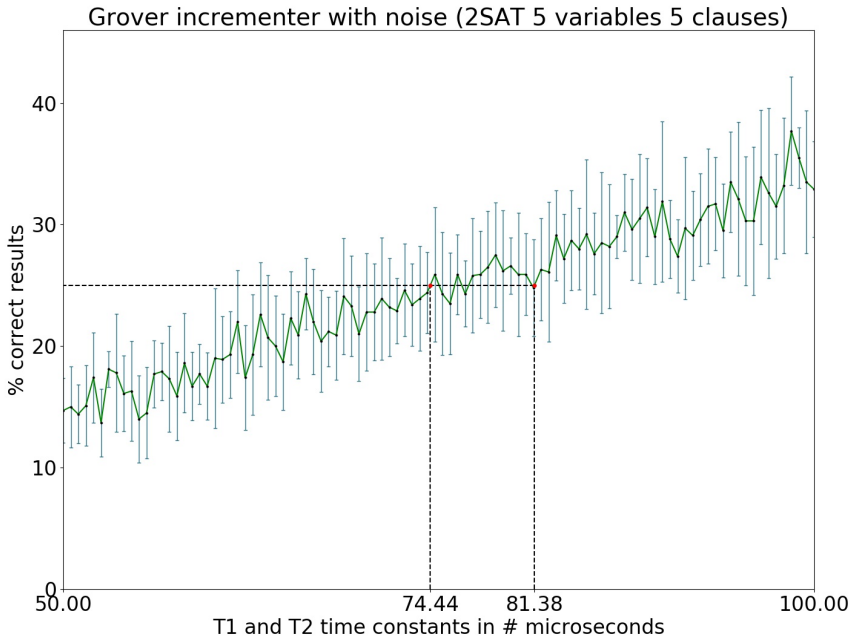


Figure 12: Broad noise interval of 2-SAT with 4 variables and 5 clauses

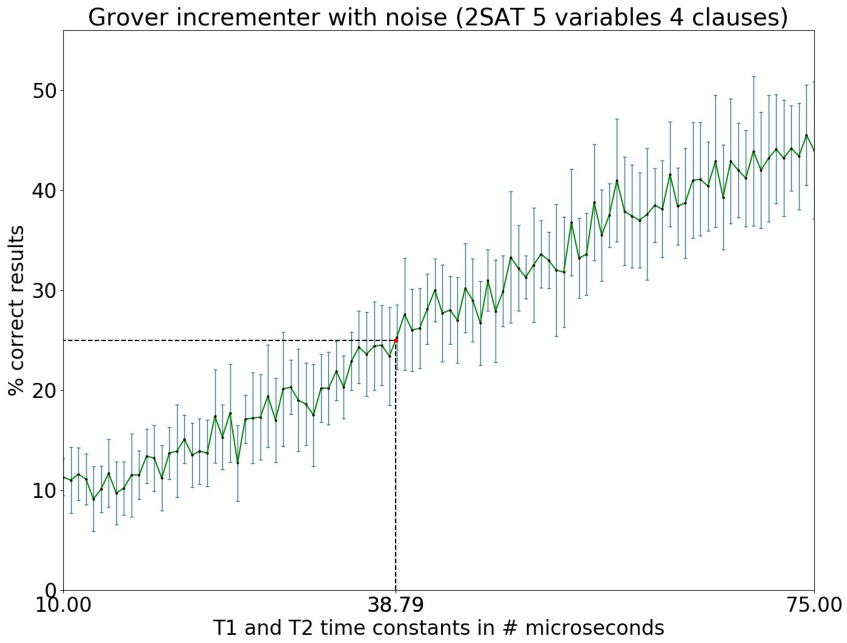


Figure 13: Broad noise interval of 2-SAT with 4 variables and 4 clauses

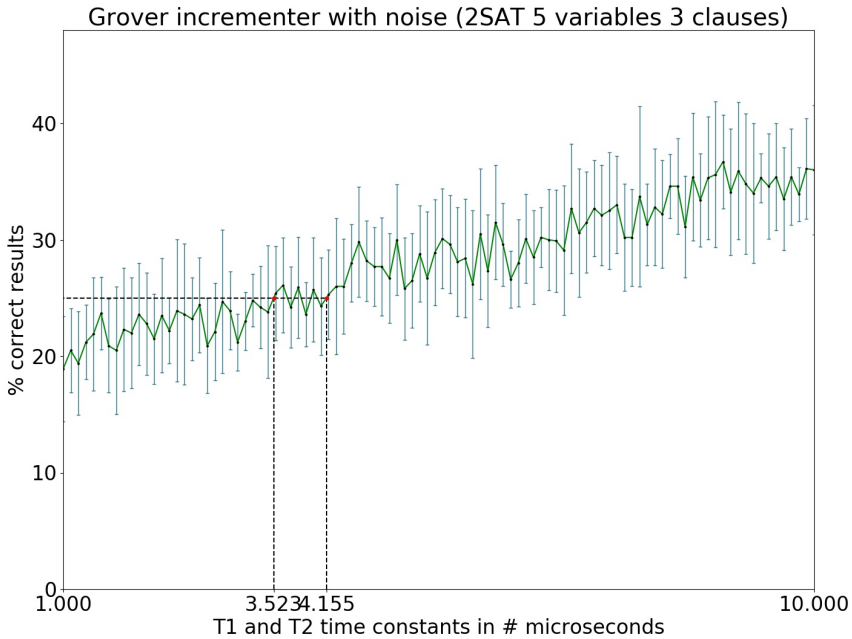


Figure 14: Broad noise interval of 2-SAT with 4 variables and 3 clauses

A.2 Specific noise interval

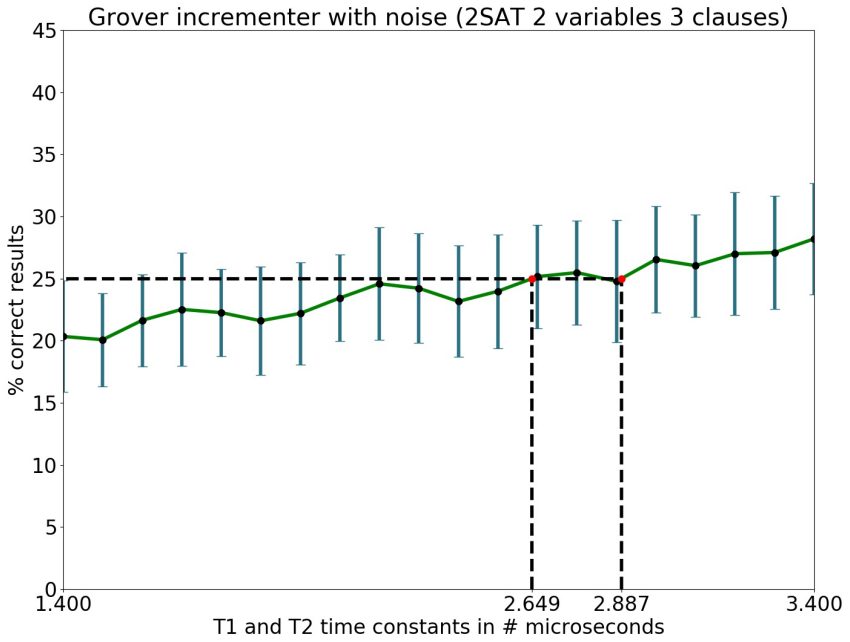


Figure 15: Specific noise interval of 2-SAT with 2 variables and 3 clauses

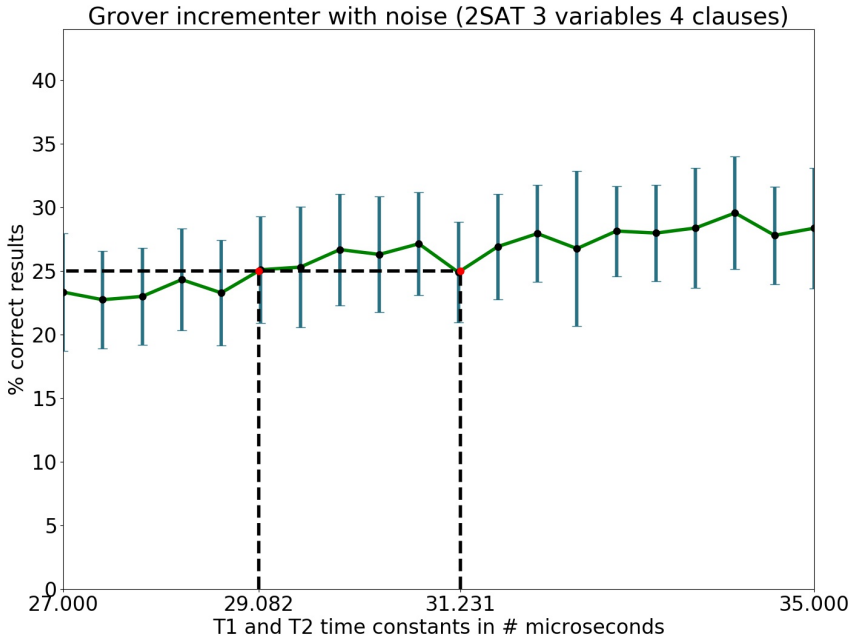


Figure 16: Specific noise interval of 2-SAT with 3 variables and 4 clauses

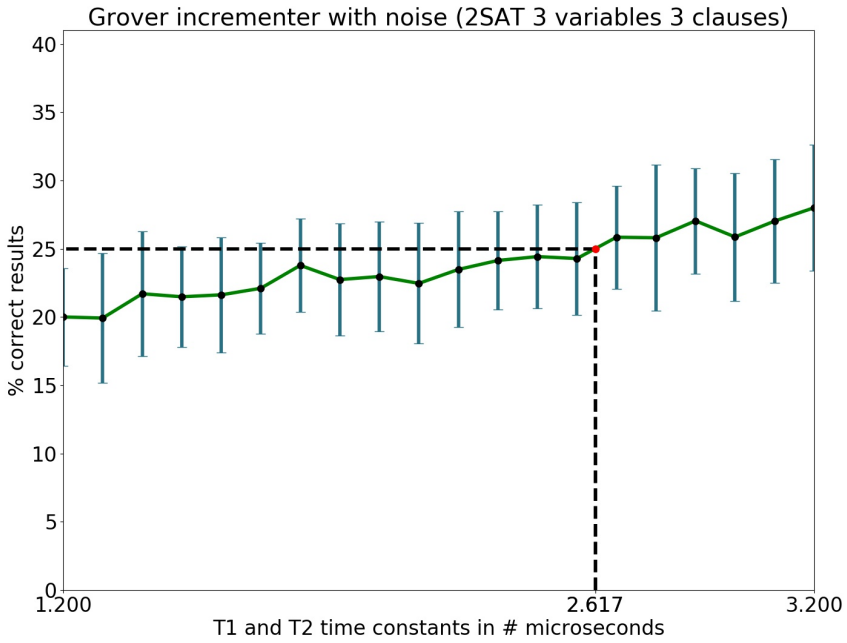


Figure 17: Specific noise interval of 2-SAT with 3 variables and 3 clauses

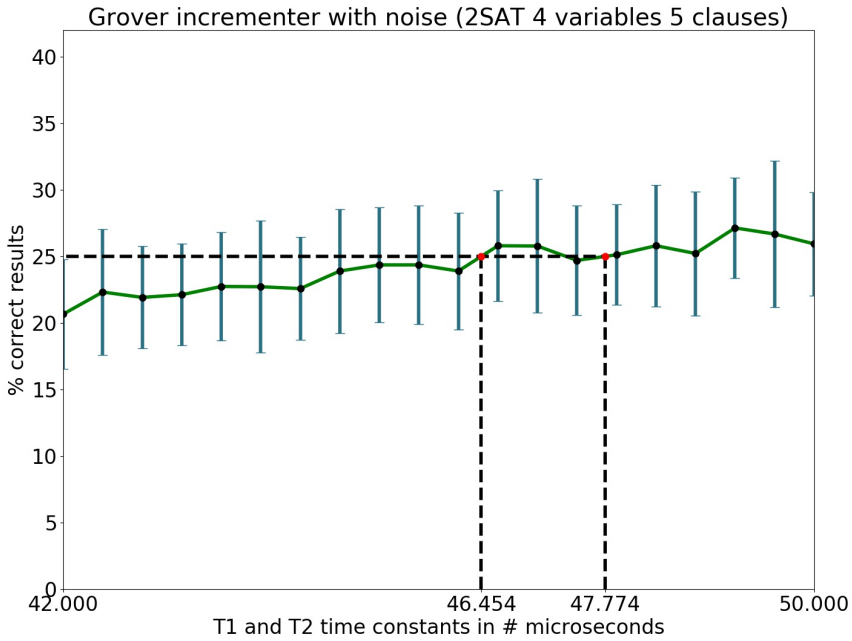


Figure 18: Specific noise interval of 2-SAT with 4 variables and 5 clauses

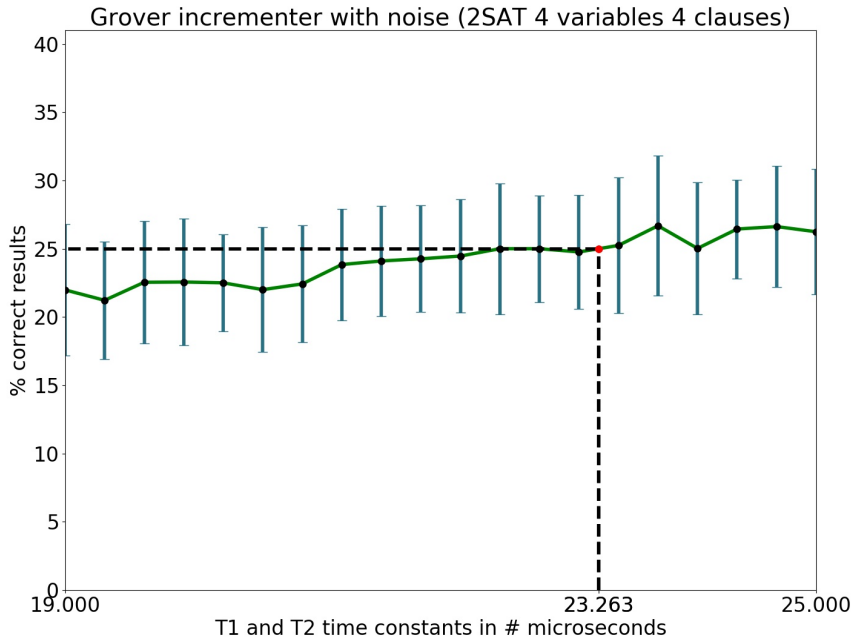


Figure 19: Specific noise interval of 2-SAT with 4 variables and 4 clauses

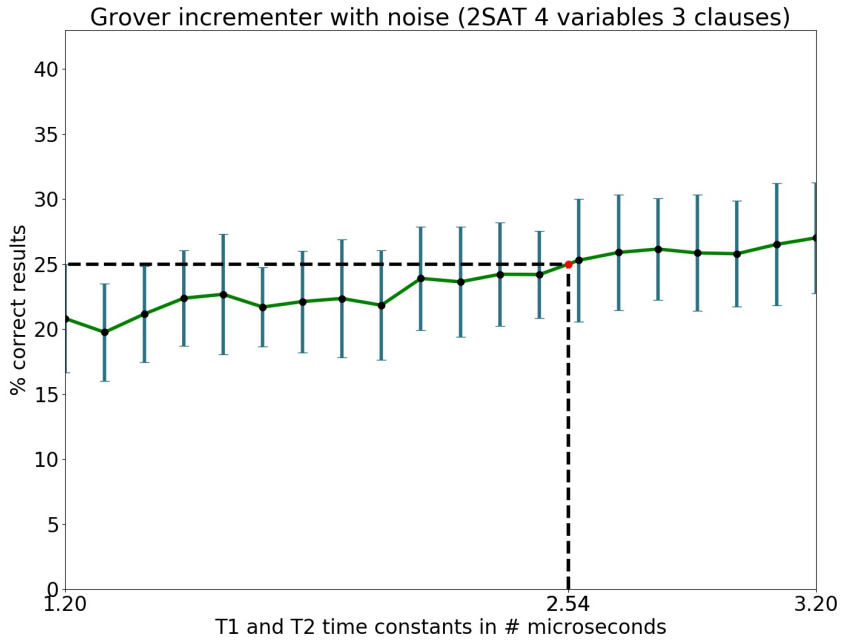


Figure 20: Specific noise interval of 2-SAT with 4 variables and 3 clauses

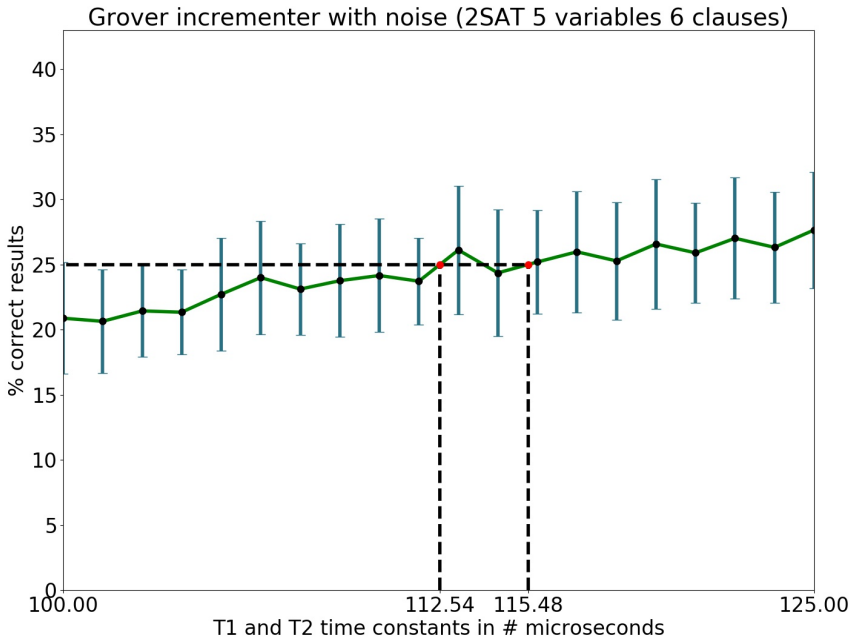


Figure 21: Specific noise interval of 2-SAT with 3 variables and 3 clauses

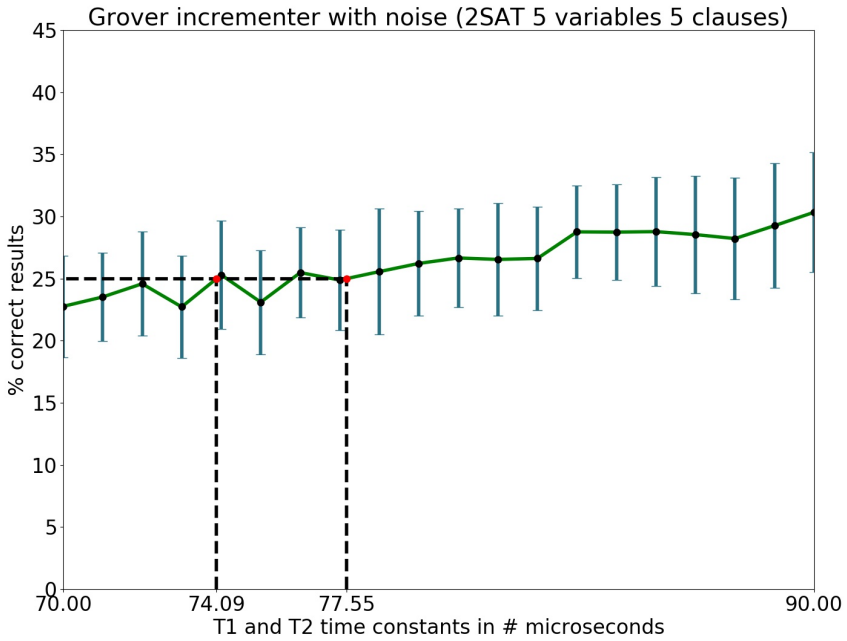


Figure 22: Specific noise interval of 2-SAT with 4 variables and 5 clauses

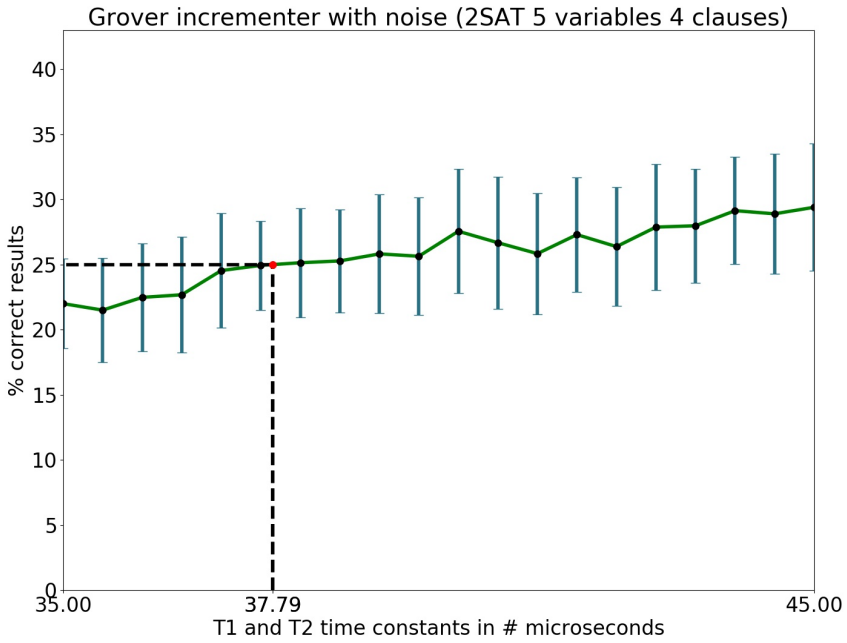


Figure 23: Specific noise interval of 2-SAT with 4 variables and 4 clauses

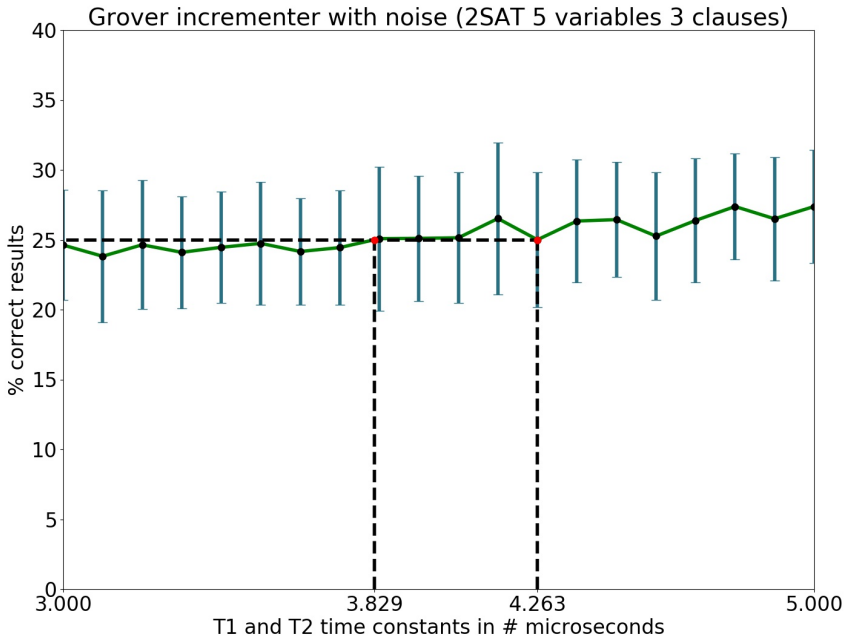


Figure 24: Specific noise interval of 2-SAT with 4 variables and 3 clauses