INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN


**Bachelorarbeit**

# Encoding Strategies to solve Soduko with Quantum Computers

Maximilian Weiß

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN


**Bachelorarbeit**

# Encoding Strategies to solve Sudoku with Quantum Computers

Maximilian Weiß

| | |
|---|---|
| Aufgabensteller: | Prof. Dr. Dieter Kranzlmüller |
| Betreuer: | Sophia Grundner-Culemann<br>Tobias Guggemos<br>Korbinian Staudacher<br>Andreas Spörl (DLR) |
| Abgabetermin: | 19. Juli 2022 |

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 19. Juli 2022

.....

*(Unterschrift des Kandidaten)*

**Abstract**

Quantum computers represent a new generation of computing machines and can solve problems by exploiting quantum effects in a way that classical computers cannot. Especially for combinatorial problems, this new way of computing is relevant and could lead to significant improvements in the future. Quantum Computers use qubits instead of classical bits. Since high numbers of qubits are not yet available in current quantum computers, efficient encoding of problems is essential. A well-known and easy-to-understand combinatorial problem is Sudoku. It is an exciting and fun puzzle that can be mapped to several problem classes that are more general. This work considers Sudoku as a graph coloring and exact covering problem. Both are ways to formalize Sudoku and allow different optimization strategies to reduce the number of qubits needed. Our work presents optimizations for graph coloring and exact covering encoding. Techniques for improvement are presented in both cases by preprocessing the games and using binary encoding. Specifically, we show the impact on encoding size when the problem gets solved with Grovers's Algorithm and Quantum Approximation Optimization Algorithm (QAOA). Also, the number of qubits explicitly needed for Grover's Algorithm is improved, reducing the number of auxiliary qubits from N to log(N), by implementing a counter. Further, we present and briefly explain the software developed to calculate the number of qubits needed to encode a Soduko with a specific strategy and Algorithm.

# Inhaltsverzeichnis

# 1 Introduction

Sudoku is a popular puzzle that people of all cultures play without needing any mathematical knowledge to solve. A classic Sudoku puzzle is played on a board consisting of 81 cells that are arranged in a 9x9 square. The board is further subdivided into nine sub-areas consisting of 3x3 squares. These areas are called subfields. The game's goal is to fill all the cells with the digits 1-9. Thereby, numbers may not appear twice in a row, column, or subfield. In the start state, some cells are already pre-filled; by this default, the player can draw logical conclusions and solve the puzzle, one cell at a time. Figure 1.1 shows a detailed example by which the naming of the units can be well understood. However, Sudoku is not only an entertaining game for puzzle fans. It also engages many mathematicians with questions that the game raises on an abstract level. For example, the number of possible Sudoku games or the minimum number of starting squares that need to be filled to get a unique solution are exciting questions [Del06]. Nevertheless, simply solving the game also occupies mathematicians and computer scientists. In particular, it is interesting how solutions for such puzzles can be found as efficiently as possible. Sudoku games are special cases of more general problem classes, like Graph Coloring or Exact Covering. Our work will explain both problem classes in detail. Solving such combinatorial problems is possible on modern, classical computers, thanks to ever-increasing computing power. However, the algorithms used for this are inefficient, so a computation of a solution is not possible in a reasonable time for larger problems [Lei77]. These combinatorial problems may be suitable to be solved with quantum-based algorithms. Quantum computers do not work with classical bits but rather with qubits, enabling new algorithms and solution methods [Hom]. At the current time, the application of these algorithms in practice is limited by the hardware as the number of qubits available is very low. Therefore, our work addresses how to reduce the number of qubits needed to solve a Sudoku puzzle and thus improve the size of problems that can be solved with quantum-based algorithms.

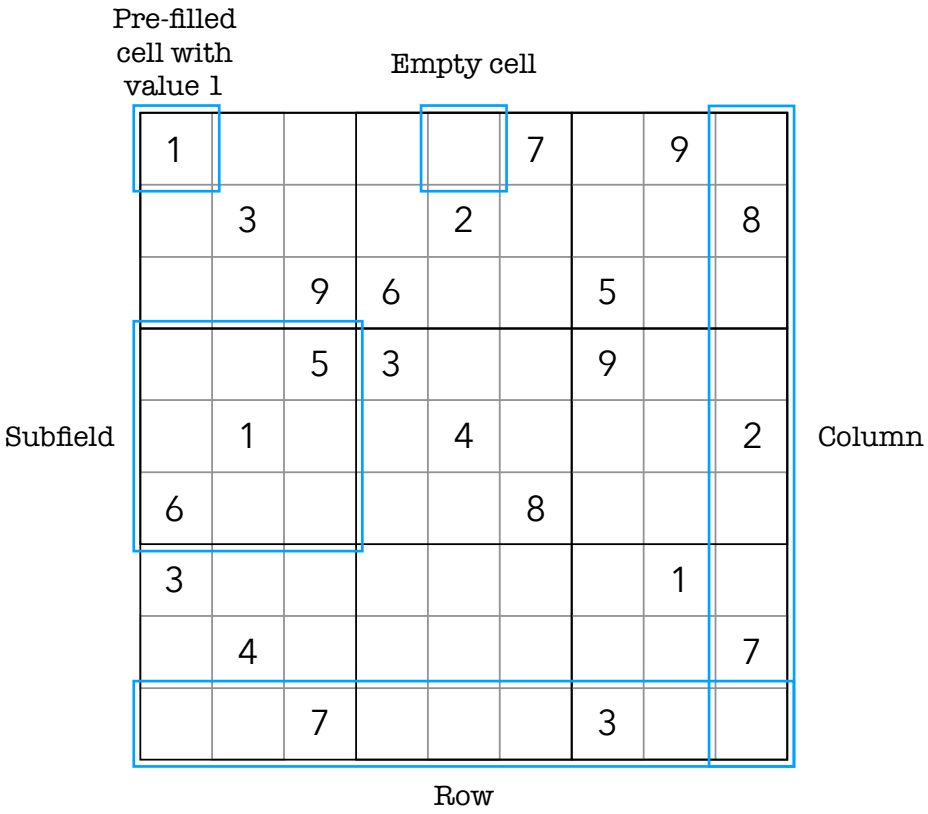


Figure 1.1: Sections of a Sudoku Board

# 2 Background

This chapter provides an overview of the problem classes, Sudoku will be mapped to in our work. Also it introduces the reader to basic concepts of quantum computing.

## 2.1 Formalization of Sudoku

Besides the standard game size, Sudoku puzzles can be played in different sizes. This work will focus on the classic version, a 9 x 9 board. For simplification and presentation of concepts, the smaller version of a 4 x 4 board is used in some places. In order to solve Sudoku puzzles with the help of computer algorithms, a formal mathematical description is necessary. Therefore a mapping of the game to known problem classes is performed. This work will discuss the mapping to a graph coloring problem and an exact covering problem.

### 2.1.1 Graph Coloring

To understand the idea of the graph coloring problem, consider an undirected graph `G(E, V)`. Here `E` represents the set of nodes and `V` the set of edges, which state connections between nodes from E. The goal of graph coloring is to assign a color `c` to each node $e \in E$ so that no two nodes connected by an edge have the same color. Formally, this mapping is described as $C : e \rightarrow c$. For the mapping of the graph to be valid, it must hold that $C(a) \neq C(b) \forall v(a, b) \in V$ [VD12]. First, a graph is constructed to map a Sudoku to a graph coloring problem. The set of nodes `E` is formed by the cells of the puzzle. Each node representing a pre-filled cell is assigned the corresponding digit/color from the start. The set of vertices `V` is formed by the relations of the cells inside the Sudoku. That means vertices are generated between all cells, which lie in the same row, column, or subunit. Thus, every node in the graph connects to every other node whose cells are in the same row, column, or subunit in the game. Now the puzzle can be solved by assigning a color to each node so that no nodes connected by an edge are assigned the same color. As shown in Figure 1.1, the notions of color and digit are interchangeable in this context. In the classical context of graph coloring, the notion of color is common, but concerning a Sudoku puzzle, digits are. Figure 2.1 shows the mapping of a 4 x 4 game onto a graph. Each digit is assigned a color to illustrate the concept. For readability, the figure shows only the edges of one node, in this case, the one representing the call `(row: 0, col: 0)`.

### 2.1.2 Exact Covering

The goal of the exact cover problem is to cover a set of elements, each of them exactly once. The set can represent any kind of element, for example, the cells of a sudoku board. For a Sudoku puzzle, the goal would be to cover all cells on the field while staying within the game's constraints. This covering is achieved by combining individual patterns for every

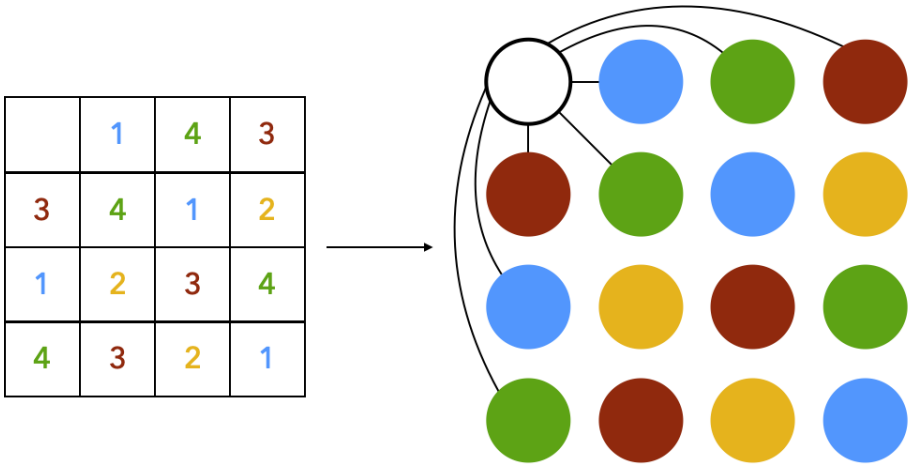


Figure 2.1: Mapping of Sudoku to a Graph Coloring problem

digit. The patterns indicate where the specific digit is positioned on the board. Figure 2.2 shows the mapping of a solved Sudoku to each digit's pattern.

This approach aims to find a pattern for every digit, achieving an exact cover. The mathematical description of the exact cover problem is: There exists a set with elements $U = \{u_1, u_2, ..., u_n\}$ and another set $C$ which has all subsets from $U$. An exact cover is achieved if a subset $S \subset C$ contains all elements from $U$ exactly once. [MO08]

In the case of a Sudoku, which is mapped to an exact cover problem, U contains all cells of the game `(row, col)`. All subsets from it are formed by C. In the case of Sudoku, a subset can be interpreted as the distribution of a single digit; that is, which cells are occupied by that digit. Since in a Sudoku, which is played on a n x n matrix, each number must occur exactly n times, the special case can be made for all sets in C that each set must contain exactly n elements to be part of a correct solution. Additional statements can be made about the set S. The subset S from C now represents a selection of subsets for each digit in the game. Since there are exactly n different digits in the game, S must also have the size n. If a valid S is found, it means that for every digit, i.e., every element in S, there exists a subset from all cells U, so that every field on the Sudoku is filled and no cell can be occupied twice. To avoid disturbing the game's constraints with respect to rows, columns, and subunits, the subsets must be preselected in C. This process is described in Chapter 4.

## 2.2 Quantum Computing

As computers get faster and better all the time, their components are getting smaller. This reduction in size leads to parts that can soon be on the order of single atoms. At this level, the laws of classical physics no longer apply, and the effects of quantum mechanics must be considered. By making use of those effects, quantum computers can perform calculations that are impossible for classical computers. The Homeister [Hom] lecture is a well-suited resource for an entry to this topic. It was used in this and the following chapters to provide an overview of the concepts.

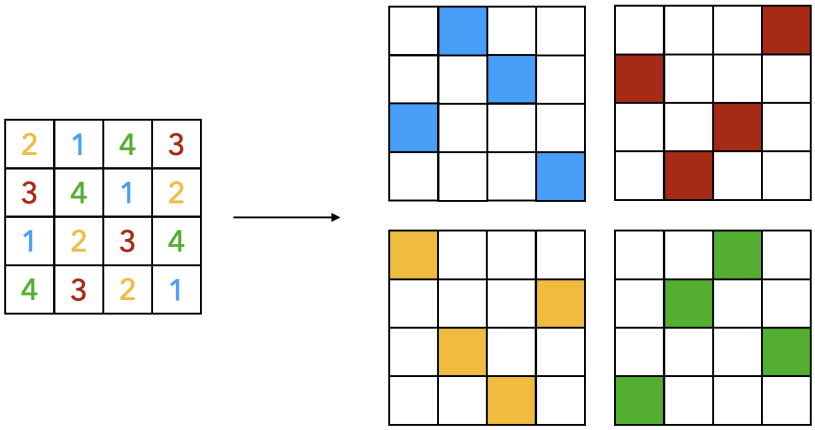


Figure 2.2: The individual pattern for each color

### 2.2.1 Quantum Parallelism and Superposition

One important effect is called quantum parallelism. A classical bit can be either in the state 0 or 1. A qubit can be in a so-called superposition over these states. To understand quantum computing, the concept of superpositions is crucial. In quantum mechanics, states are usually written in so-called *ket* notation. Thus, one would write down the classical states 0 and 1 in terms of qubits as $|0\rangle$ and $|1\rangle$. For a qubit, these are called the base states. A qubit can now take all states of the form $\alpha \cdot |0\rangle + \beta \cdot |1\rangle$, where $\alpha, \beta$ are complex numbers. The factors $\alpha, \beta$ are called amplitudes. These amplitudes express the probability of a qubit being measured in the corresponding state. The probability is calculated from the square of the amplitude. It applies $|\alpha|^2 + |\beta|^2 = 1$, which means that the probablities of measuring eighter of two states add up to 1.
A single qubit $|q\rangle$ in an equal superposition, meaning it is equally likely to be measured in the state $|0\rangle$ and $|1\rangle$, would be written as:

$$|q\rangle = \frac{1}{\sqrt{2}} \cdot |0\rangle + \frac{1}{\sqrt{2}} \cdot |1\rangle \tag{2.1}$$

The squared amplitudes would result in $\frac{1}{2}$, representing the probability of each state.
There are computational advantages due to the effect of quantum parallelism. A register with n classical bits can be in $2^n$ unique states, but only one at a time. On the other hand, a quantum register with n qubits can be in a superposition of all these states. Thus, at one time, it can be in a superposition that maps $2^n$ states, each to a certain probability.

### 2.2.2 Measurement

Unlike classical bits, the state of a qubit cannot be simply read out. For qubits, a measurement of the state is necessary. Measuring destroys the superposition, and the qubit takes the state $|0\rangle$ or $|1\rangle$. The amplitudes indicate how likely each of the two states is measured.

### 2.2.3 Quantum Circuits and Gates

Like in classical hardware circuits, several operations can be performed on a qubit. These are called gates in quantum computing. This chapter introduces the most important gates to

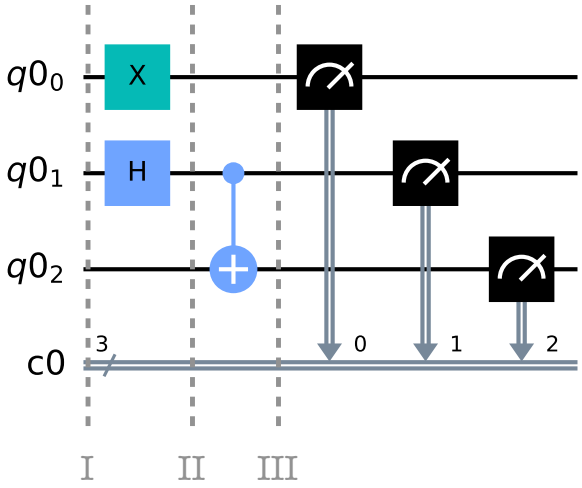


Figure 2.3: A simple Quantum Circuit

understanding the algorithms in later chapters. Figure 2.3 shows a quantum circuit. A quantum circuit consists of one or more qubits forming a quantum register and quantum gates, representing operations on the qubits. At any point in the circuit, one can calculate the state of the register, i.e., the values of the individual qubits and their probabilities. In Figure 2.3 a quantum register consisting of three qubits is initialized; these are called $q_0, q_1, q_2$. Further, there are three classical bits `c` initialized, which will later hold the measuring results of the three qubits. In the start state, each qubit is set to $|0\rangle$. So the register at the time `I` has the state:

$$|0\rangle \cdot |0\rangle \cdot |0\rangle = |000\rangle \tag{2.2}$$

First, an X-gate is applied to `q0`. This is comparable to the classical NOT. The X-Gate flips the state $|0\rangle$ to $|1\rangle$ and vice versa. On `q1`, the Hadamard gate, or H-gate for short, is executed. This H-gate is used to put qubits into superposition. If the qubit is in the $|0\rangle$ state at the start, the H-gate puts it in the equal superposition:

$$\frac{1}{\sqrt{2}} \cdot |0\rangle + \frac{1}{\sqrt{2}} \cdot |1\rangle \tag{2.3}$$

If the qubit is in the $|1\rangle$ state at the start, it is also brought into superposition, but the sign of the amplitude changes. This is not important for the probabilities as they are the square of the amplitudes, but it allows interesting benefits quantum algorithms can exploit. This will be important in later chapters as well.

$$\frac{1}{\sqrt{2}} \cdot |0\rangle - \frac{1}{\sqrt{2}} \cdot |1\rangle \tag{2.4}$$

Thus, at the time `II`, the system is in the state:

$$|1\rangle \cdot \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \cdot |0\rangle = \frac{1}{\sqrt{2}}(|100\rangle + |110\rangle) \tag{2.5}$$

To establish dependencies between multiple qubits, CNOT gates are used. These gates check whether a particular qubit is $|1\rangle$; this qubit is called the controlled qubit. If this condition is

satisfied, we apply an X-gate on a second qubit. If the controlled qubit is in the $|0\rangle$ state, the X-gate is not applied. This gate creates a permanent dependence between these two qubits, which is especially interesting when being applied to qubits in superposition. This effect is called the entanglement of two qubits. Between `q1` as well as `q2` a CNOT is applied. The superposition of the controlled qubit results in the following value of the register at the time `III`:

$$|1\rangle \cdot \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) = \frac{1}{\sqrt{2}}(|100\rangle + |111\rangle) \tag{2.6}$$

At the end of the circuit, we perform the measurement. With probability $\frac{1}{2}$ we get the results $|100\rangle$ or $|111\rangle$. This entanglement leads to matching results in the second and third qubit. So dependent on the base state the second qubit gets measured in, the third qubit will also be in the same state.

## 2.3 NISQ-Era

The state of a qubit is affected by its interaction with the environment, which is also a quantum system. Due to this effect, which is also named decoherence, errors occur, which have to be corrected. This error correction is realized by additional qubits in the circuit, which protect the information of the relevant qubits. [Hom] The currently available hardware is not yet applicable on a large scale in practice. The current state can be well illustrated by an experiment conducted by the Google AI Quantum Group in 2019. Here, a quantum system with 53 qubits was developed and programmed. The system solved a problem within minutes that would have taken the most powerful classical system to date several days. The era of current hardware is also known as NISQ. The acronym stands for Noisy Intermediate Scale Quantum. Noisy means that the systems have no error correction yet, and the qubits are error-prone. Intermediate scale refers to the size of the systems being over 50 Qubits. [Pre21] The current most powerful quantum computer is IBM's 127-qubit processor 'Eagle'. [IBM21]

## 2.4 Grover's Algorithm

The first quantum algorithm relevant to our work was published in 1996 by Lov Grover and named after its developer. This algorithm demonstrates the computational advantages of a quantum computer very clearly. It is a search algorithm. Many problems can be reduced to search problems; thus, optimization problems can also be conceived as a search for a correct, valid solution. With respect to the Sudoku game, one searches for the correct solution within all combinatorially possible assignments of individual cells. This is called a search within an unsorted database. A classical computer solves this by checking each entry in the database for correctness until it finds a correct solution. For each entry, a function is executed that checks the correctness. Checking a solution is the limiting factor in this process in terms of complexity. If the database has N entries, the computer needs N-1 calls in the worst case, one call in the best case, and (N+1)/2 calls on average. Thus, the time complexity of the algorithm is $O(N)$.
Many quantum algorithms start by mapping all possible states of a system into a superposition. Then, the goal is to amplify the amplitude of the correct solution while reducing the amplitudes of all the incorrect solutions. If this succeeds, the probability that the correct

state will be measured at the end of the algorithm increases with the amplitude. Grover proceeds as follows. First, a uniformly distributed superposition is generated, then a so-called oracle of gates is applied. The needed information to find the right state is encoded in this oracle; the reason why becomes more clear in the following example. Besides the coding bits, Grover also needs an auxiliary bit. This is brought into the superposition $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. If the oracle now finds a valid assignment, a bit flip, i.e., an X-gate is applied to the auxiliary qubit. Due to the negative amplitude of the auxiliary qubit in $|1\rangle$, the sign of the amplitude of the correct assignment thus changes with the bit flip. Grover now amplifies the amplitude by mirroring each amplitude around the mean value of all amplitudes. Due to the negative sign of the correct solution, it is amplified, while all others are reduced. The effect of mirroring is shown in Figure 2.4. Here four amplitudes are shown. Three of them are at `0.5`, and one of them is `-0.5`. The average of them is at `0.25`, so they get mirrored around this value. By doing this, the positive amplitudes become exactly `0` while the negative amplitude is `1`. Figure 2.5 shows a simple example of the application of Grover's algorithm, which is

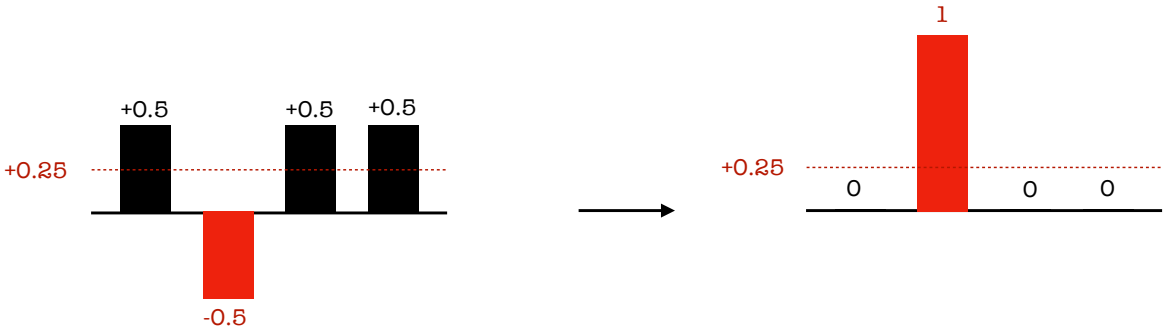


Figure 2.4: Amplitudes getting mirrored around the average.

explained below.
In the example, two independent coin tosses are to be considered. The first coin is represented by the qubit `coin1`, the second by `coin2`. Here the state $|0\rangle$ stands for heads, $|1\rangle$ for tails. This gives a database of 4 possibilities: $[00, 01, 10, 11]$. Now we are looking for the entry where both coin tosses show tails, thus 11. In step I in Figure 2.5, the input qubits `coin1, coin2` are initialized in the $|00\rangle$ state. In addition, an auxiliary bit `grover bit` starts in the $|1\rangle$ state. Step II uses H-gates to generate a superposition over all possible states:

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \cdot \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \cdot \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{2.7}$$

By transforming the equation, one gets:

$$\frac{1}{2\sqrt{2}}(|000\rangle + |010\rangle + |100\rangle + |110\rangle - |001\rangle - |011\rangle - |101\rangle - |111\rangle) \tag{2.8}$$

Step III Is also called oracle. It is used to trigger a NOT gate on `grover bit` for the sought occupancy. So in the example, the occupancy $|11\rangle$ of the qubits `coin0, coin1`. In this example, two qubits have to be controlled. This double controlling is realized with the help of a Toffoli-Gate. This is an extension of a CNOT Gate and is only executed when all controlled qubits show the state $|1\rangle$. The new state of the register will be:

$$\frac{1}{2\sqrt{2}}(|000\rangle + |010\rangle + |100\rangle + |111\rangle - |001\rangle - |011\rangle - |101\rangle - |110\rangle) \tag{2.9}$$

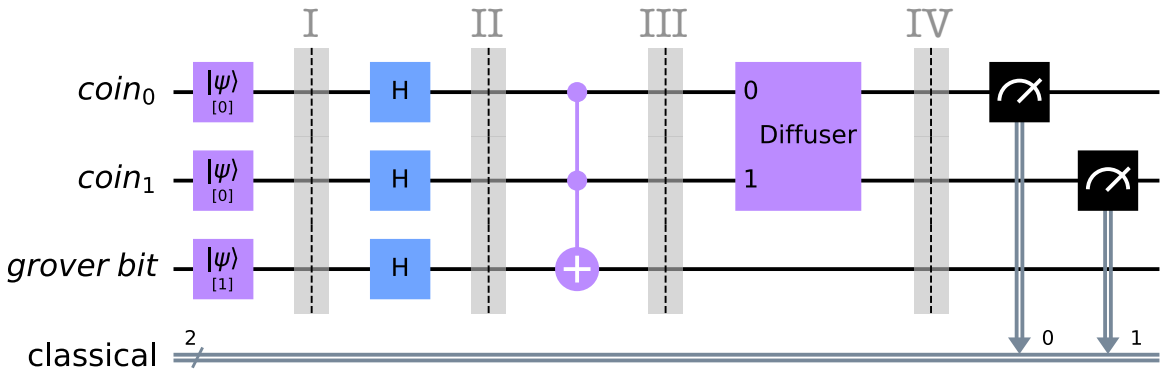


Figure 2.5: Grover circuit

If q3 is now pulled out of the bracket again, the negative amplitude can be read out more clearly:

$$\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle - |11\rangle) \cdot \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{2.10}$$

The amplitude of the searched configuration is therefore negative. In the following step IV the amplitude is amplified. This is achieved by mirroring around the mean values of the amplitudes, as already described. In the circuit, we do not consider this part more precisely since it is not necessary for understanding. The interested reader can also refer to Homeister [Hom]. After amplification of the amplitudes, the result is measured in the last step. Due to the amplitude amplification, the searched result can be measured more often than the wrong results. In this case, with four amplitudes at equal distribution, the amplitude of the correct solution is 1. This was explained before in Figure 2.4. So the solution is measured every time if the system works without errors.

## 2.5 Quantum Approximation Optimization Algorithm (QAOA)

The quantum approximation optimization algorithm (QAOA) was presented by Fahri et al. and is a metaheuristic method [FGG14, HWO+19]. This algorithm can approximate solutions of optimization problems [Qis20]. The method is especially suitable when the problem is modeled as a search for a bit string [Cer20]. A simple example makes this clear. Suppose a coin is tossed twice. Now we look for all possibilities, where heads and tails alternate. We now make two statements:

1. First toss shows heads
2. Second toss shows heads

This problem can now be expressed as a bit string with two characters. Each digit in the string now represents the truth value of the associated statement. A 0 means that the statement is not true, and a 1 that the statement is true. So in the case of coins, a 0 would be the result *tails* since the statement *heads* is false. So the bitstring `01` would state that the first toss is *tails* and the second toss shows *heads*. This problem can be formulated as a quadratic unconstrained binary optimization problem (QUBO), which is well suited to be

solved with QAOA [JN22]. A QUBO model is an optimization problem, which is to minimize the function (2.11):

$$y = x^t Q x \tag{2.11}$$

`x` is a vector that represents the variable statements, and `Q` is a square, upper triangle matrix with constant values [GKD18]. Concretely in the example of coin tosses, this would mean that the vector `x` can take all possible occupations of the two variables so that `00, 01, 10, 11` as possibilities for x exist. The QUBO matrix `Q` assigns costs to the possible states of `x` so that the solution or solutions sought have the lowest value in this function. In this example, the matrix could look like this:

$$Q = \begin{bmatrix} -1 & 2 \\ & -1 \end{bmatrix} \tag{2.12}$$

For the vectors $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ and $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, the result of equation (2.11) is -1. The vectors $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$ have the result 0. QAOA works as a hybrid variational algorithm and can approximate solutions to such problems [GKD18]. The algorithm is called a hybrid algorithm because it optimizes the parameters of operators in iterative steps. This optimization takes place through classical algorithms. The operators act on quantum systems [GP20]. A detailed explanation of the algorithm is not necessary for understanding our work since the encoding of the problems aims at a QUBO, which has been explained in this chapter. For a deeper understanding of QAOA, the original work by Fahri et al. [FGG14] is suitable. In addition, the article by Pagano et al. [GP20] represents a good read for QAOA.

# 3 Related Work

As Sudoku Games are an easy-to-grasp version of combinatorial problems, they are used in a lot of research regarding that topic as an example. This chapter overviews common ways to solve these problems on classical computers and quantum approaches.

## 3.1 Classical Approaches

One successful approach to solving Sudoku puzzles using classical computers is SAT solvers. [PKS13] A SAT solver receives a boolean formula in conjunctive normal form (CNF) as input and is asked to find an assignment such that the formula is true. A formula is in CNF if it links clauses by conjunctions (and), and the clauses consist only of literals linked by disjunction (or). [CESS08] For example, a CNF might look like this:

$$(A \vee B \vee C) \wedge (D \vee E) \tag{3.1}$$

In the case of Sudoku, these literals would represent 3-tuples containing statements about the occupancy of cells. For example, $A = (0, 0, 3)$ could represent assigning the value 3 for cell in row 0, column 0. The paper by Pfeiffer [PKS13] gives a good overview of the encoding of Sudoku puzzles for SAT solvers, as well as a selection of useful solvers for these problems. It is especially interesting to note that this approach can still efficiently solve huge Sudoku games, i.e., games of size 144 x 144.

The Tabu Search algorithm is a metaheuristic that is mainly used for combinatorial problems [SCGM+15]. The algorithm starts with a small part of the problem and finds an initial solution. It then extends this solution to larger areas. The algorithm gets its name from the memory structure, which keeps specific rules in lists. These rules tell which parts of the solution area may no longer be used, i.e., are tabus. This approach is particularly interesting in hybrid form with the alldifferent constraint, as it is described in the work of Soto [SCGM+15]. The alldifferent constraint acts as a preprocessing of the game and thus significantly speeds up the search algorithm.

An interesting way to solve Exact Covering Problems is Knuths Algorithm X. As Sudoku can be mapped to this problem, it is an appropriate method for solving these problems. To understand this Algorithm, the interested reader might study the original work of Professor Knuth [Knu00], as the concept is complex and not important for this work.

## 3.2 Quantum Approaches

For a detailed introduction and a basic understanding of quantum based methods, the work of Homeister is very suitable [Hom]. Here, the principles of quantum computing based on the special quantum effects are treated, and all relevant algorithms are taught. For a practical

introduction to quantum algorithms, the textbook from qiskit is appropriate [Qis22]. With this textbook, algorithms such as Grover and QAOA can be understood and implemented through many examples. Also, this source's reference to graph-related problems is very good for an introduction to solving Sudoku puzzles.
A tutorial from Microsoft provides a simple introduction with examples for solving Sudoku with Grover's Algorithm [Mic22]. A more detailed discussion of solving Sudoku puzzles with quantum computers is provided in the work of Pal et al. [AP]. In *Solving Sudoku Game Using Quantum Computation*, they propose an Algorithm to solve a 4 x 4 Sudoku and show how we as humans can use the concept of superposition while playing sudoku games manually.
Also, the improvement of quantum algorithms, such as Grover and QAOA represents an interesting optimization opportunity. In their work, Saha et al. [SMS+20] present the extension of Grover's algorithm to a qudit quantum system. A qudit system works with computer units that, unlike a qubit, can have more than two ground states and thus can contain and process more information per qudit. [WHSK20] The extension to these systems allows them to optimize Grover's algorithm, both in the number of qudits needed and the number of gates needed. Concerning QAOA, interesting optimizations are possible in the formulation of QUBO. The work of Nüßlein et al. shows the formulation of a QUBO for a k-SAT problem where they can reduce the growth of the matrix from O(k) to O(log(k)). [JN22]

## 3.3 Research Question

Most of the related work mentioned here regarding classical Systems focuses mainly on the question of how to solve such a fixed-size problem as efficiently as possible. The related work about quantum systems eighter implements these algorithms without further optimization or, like the optimization of Grovers Algorithm, tries to improve the algorithm in general. However, since the number of qubits in the NISQ Era is very limited, the question of how to reduce problems in size as much as possible to implement already researched algorithms on them becomes interesting. Also the efficiency of the encoding itself has an important impact. The work of Nüßlein et al. [JN22] strongly points in this direction by optimizing the encoding for QAOA approaches. Our work focuses on reducing the number of required qubits with respect to Sudoku puzzles, using both the size reduction of the problem itself and the encoding strategies. As QAOA performs a large part of the optimization with classic algorithms, our work wants to focus on Grovers Algorithm. Besides being a strict Quantum algorithm, it is less described in terms of optimizing the number of qubits needed to solve problems like Sudoku. This makes it an interesting topic for research.

# 4 Optimization Strategies

The most considerable potential to reduce the number of qubits lies in the encoding of the problem. Our work discusses two approaches: Graph Coloring and Exact Covering, as they offer a very visual representation of the problem. Regardless of the formulation of the problem, the first step is to exclude parts of the solution space that can be found using classical computing at an insignificant computational cost. This step is referred to as preprocessing in the following. Subsequently, individual optimization is performed for different solution approaches. This is followed by mapping to an encoding that can be implemented in a Grover setup or a QUBO matrix that can be solved with QAOA.

## 4.1 General Preprocessing

The goal of preprocessing is to reduce the complexity of the problem by limiting the possible digits per cell as much as possible and thus to achieve a lower demand for qubits. The main focus during our preprocessing is that the classical computations must be efficient and fast. If this is not the case, too crucial a part of the problem solution is done with classic computation that gets inefficient at a point, and the advantage of quantum algorithms becomes insignificant. The preprocessing is divided into two steps:

1. The exclusion of digits by given constraints
2. Checking for new constraints generated by step 1

The program iterates over every row and within it over every column. Thus, each `(row, column)` cell is considered. If a cell is empty and thus no digit is given, a 3-tuple is appended to the list `open_tuple` for each possibility ranging 0 to n-1. The tuple consists of `(row, column, digit)`. However, this step additionally checks if a digit already occurs within the respective row, column, or subunit. If so, the tuple is not appended since the solution can be trivially excluded by given constraints. At the end of Step 1 there is a list containing all possible assignments for all free cells according to these constraints. In Step 2, we check whether the computations in Step 1 found unique assignments for cells that were previously empty. This is easy to detect since any `(row, column, _)` combination that is unique in `open_tuples` satisfies this case. By additionally storing this as a dictionary, the search effort can be reduced, and the computation can be done very efficiently. The cells a value was found to, form new constraints that apply to all empty cells in `open_tuples`. Thus, each tuple is iterated over, and any element that would violate new constraints is removed. This step is repeated until no more solved cells have emerged after new constraints have been applied. At the end of preprocessing is `open_tuples`, a list containing all non-trivially solvable cells with their possible assignments.

For a better understanding of the algorithm Figure 4.1 shows a given Sudoku with four prefilled cells. These are stored as `preset_tuples`. These tuples are also stored in the known format `(row, column, digit)`:

```
preset_tuples = [(0, 0, 1), (1, 1, 4), (2, 1, 2), (3, 0, 3)]
```

For the fields to be filled with a digit, another list `open_tuples` is created. This list stores one tuple for each cell for each possible assignment based on the constraints. For a better visual presentation of the list one cell per line is printed in this example:

```
open_tuples = [(0, 1, 3),
               (1, 0, 2),
               (2, 0, 4),
               (3, 1, 1),
               (0, 2, 4), (0, 2, 3), (0, 2, 2),
               (0, 3, 4), (0, 3, 3), (0, 3, 2),
               (1, 2, 3), (1, 2, 2), (1, 2, 1),
               (1, 3, 3), (1, 3, 2), (1, 3, 1),
               (2, 2, 4), (2, 2, 3), (2, 2, 1),
               (2, 3, 4), (2, 3, 3), (2, 3, 1),
               (3, 2, 4), (3, 2, 2), (3, 2, 1),
               (3, 3, 4), (3, 3, 2), (3, 3, 1)]
```

After applying Step 1 there are four positions for which only one tuple is in `open_tuples`. As there is only one possible color, these fields are known after applying Step 1. These tuples go into the third list called `fix_tuples` which is an extension to `preset_tuples`. Figure 4.1 also shows the new state of the game. Now Step 2 of the algorithm is applied, and all tuples which collide with the new elements in `fix_tuple` are removed in `open_tuple`.

```
fix_tuples = [(0, 1, 3), (1, 0, 2), (2, 0, 4), (3, 1, 1)]

open tuples = [(0, 2, 4), (0, 2, 2),
               (0, 3, 4), (0, 3, 2),
               (1, 2, 3), (1, 2, 1),
               (1, 3, 3), (1, 3, 1),
               (2, 2, 3), (2, 2, 1),
               (2, 3, 3), (2, 3, 1),
               (3, 2, 4), (3, 2, 2),
               (3, 3, 4), (3, 3, 2)]
```

Now for each cell, there is more than one element in `open_tuple`, so the algorithm terminates. If there were more single tuple fields contained, steps 1 and 2 would be performed until the algorithm terminates.

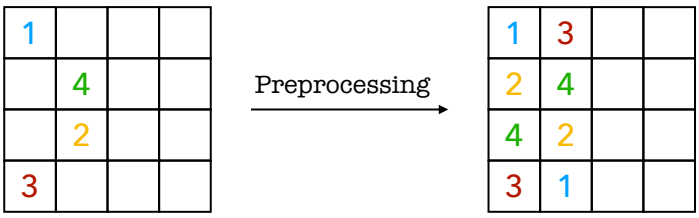


Figure 4.1: Starting state to preprocessed state

For a 4 x 4 Sudoku, this simple algorithm can solve the game in most cases if there is a unique solution. When applying the preprocessing to a 9 x 9 game, more complicated steps are involved, such as logically setting relationships between different cells to solve a game. In these cases, the algorithm for the preprocessing calculation terminates after a few iterations. These observations are described in more detail in Chapter 5. An example of a 9 x 9 Sudoku is shown in Figure 4.2 below. A setup where no fix tuple is found and the preprocessing is done after applying the `preset_tuple` constraints to `open_tuples`. This Sudoku is known as AI Escargot and is one of the hardest, if not the hardest, known board states [Sud08]. Note that other researchers also use preprocessing when optimizing these problems. A similar

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | | | | | 7 | | 9 | |
| | 3 | | | 2 | | | | 8 |
| | | 9 | 6 | | | 5 | | |
| | | 5 | 3 | | | 9 | | |
| | 1 | | | 4 | | | | 2 |
| 6 | | | | | 8 | | | |
| 3 | | | | | | | 1 | |
| | 4 | | | | | | | 7 |
| | | 7 | | | | 3 | | |

Figure 4.2: AI Escargot

approach can be seen in Soto et al. [SCGM+15], for example.

## 4.2 Graph Coloring

The graph coloring approach aims to find a selection of tuples from `open_tuple` that every node in the graph gets a single color, and all edge constraints are satisfied. Mapped back to the Sudoku this means exactly one tuple for an empty cell gets selected, and these don't violate any row, col, and subunit constraints. The question is whether the size of the generated graph can be reduced. In this work, the reduction of the number of empty cells to be solved in order to obtain the solution is investigated.

### 4.2.1 Size Reduction from $n \times n$ to $n \times (n-1)$

The special thing about solving these problems with the help of a quantum system is that there is no order in which the cells are assigned. Instead, they all are occupied at the same time, and an occupation is output as a solution that does not violate any constraints. This is a significant difference from the classical solving of Sudokus, as humans or classical algorithms do it. Humans solve cell by cell and thus create more and more new constraints, which narrow down the solution space and thus allow to solve the Sudoku. This raises the question of whether there are advantages to be gained from the quantum properties that can be exploited. One of the questions raised in this work was how this could keep the number of

qubits smaller. An interesting observation for this is that a completely solved Sudoku built by n rows and n columns remains unique if one deletes a row or column. The Sudoku is thus reduced to $n \times (n-1)$. By logically filling in the missing values, the Sudoku can be easily restored to its original $n \times n$ form. Maybe it is possible to find only a valid solution for an $n \times (n-1)$ part of the Sudoku. From this arises the question of whether an assignment valid for $n \times (n-1)$ is always a part of the correct, unique solution or whether there are also further assignments that do not violate any constraints for $n \times (n-1)$ but give a wrong solution after trivial filling up from n-1 to n rows or columns. If the partial solution always is a part of the correct assignment, it is sufficient to only solve this region to get the full solution. The following logic shows by contradiction that a valid $n \times (n-1)$ assignment must always be part of the correct solution of a unique solvable Sudoku. In this logic, the $n^{th}$ row is discarded, but the same logic applies to a removed column:

**Assumptions:**
• The Sudoku is unique, so there is only one correct solution.
• More than one solution exists for a range $n \times (n-1)$ that completely satisfies the constraints for that subrange of the board

**It follows:** Except for the one correct partial solution, after completing the $n^{th}$ row or column, there must be a solution that is not valid, otherwise the uniqueness of the puzzle is violated

**How can a Sudoku become invalid?**
• A digit occurs more than once in a row
• A digit occurs more than once in a column
• A digit occurs more than once in a subunit

**Can a digit appear more than once in a col?**
It must be true that in the partial solution in each column, another digit is missing. If this was not the case, a digit would occur at least n times. But since we fill only n-1 rows, a digit would appear at least twice in a row. The $n^{th}$ row is then filled with the missing digit because in each column, another digit is missing, so each column is valid at the end.

**Can a digit appear more than once in a row?**
Since the partial solution is valid, in each n-1 row, each digit is present only once, in the $n^{th}$ row also each digit appears only once since in each column another digit is missing. So no row can be invalid.

**Can a digit appear more than once in a subunit?**
To violate the rules of assignment in a subunit, the digit in question must already occur in one of the six filled cells of the subunit. But in the column itself, the value cannot occur yet, because in the column currently are n-1 digits, and one will add the $n^{th}$ value now. In the other two columns of the subunit, the digit must already appear in the two completely filled subunits below (i.e., the other two subunits, which are traversed by the column). Otherwise, these two subunits would not be filled with all values and the partial solution would be invalid. Thus, the digit cannot appear in the other two columns of the subunit that is only partially filled. Thus, one cannot invalidate the Sudoku with a subunit.

**From this follows:**
There is no partial solution that is correct up to n x (n-1) and becomes invalid after filling the $n^{th}$ cells. Since the Sudoku is unique, there can be no other solution, and thus $n \times (n-1)$ is unique.

Note that even if the ignored row contains any preset values, there are no constraints lost as they are already considered in the step of preprocessing when `open_tuples` are computed.

### 4.2.2 Field Encoding

Each cell from the area to be solved $n \times (n-1)$ must be represented in the input quantum register. At first, for each of the cells, all the possible digits are collected in a list. Every cell is now encoded by a certain number of qubits. These qubits encode the index of each digit within the list of possible values in binary format. This encoding improves the basic binary encoding, where the qubits binary value directly encodes the digit for the cell. By encoding the index of the list, one only needs enough qubits to encode the length of the list. Figure 4.3 shows this logic. In contrast to the primitive encoding without size reduction and not yet applied preprocessing, substantially fewer cells must be encoded as well as fewer possibilities per cell are considered.

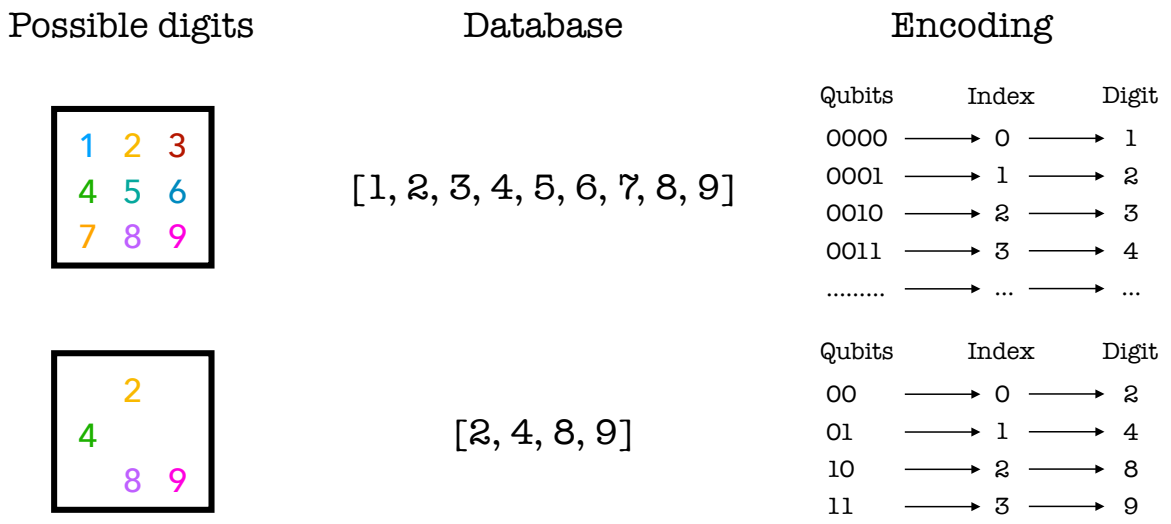


Figure 4.3: Encoding strategy with two examples

## 4.3 Exact covering

The Exact Covering Problem looks at the individual digits and their distribution in the Sudoku. For this purpose, each value is assigned an individual pattern representing its distribution on the Sudoku field. The pattern shows which cells are occupied by the digit and which are not. For an Exact Covering, a pattern must be selected for each value so that all patterns give an ideal overlay at the end. This means that all cells must be taken exactly one time. Figure Figure 4.4 shows the pattern representation of a fully solved puzzle.

In this work, a pattern is stored as a list of n elements. You can also see the list for each pattern in the example. In the list, the index within the list represents the column, and the value at that index represents the row. So for the pattern `[2, 0, 1, 3]` for digit 1 this means in column 0 it appears in row 2, and so on for every index.

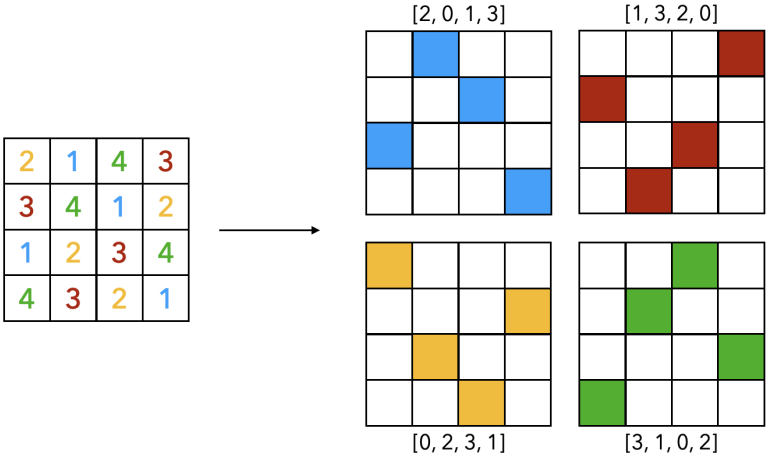


Figure 4.4: The individual pattern for each color

### 4.3.1 Pattern Generation

The number of patterns a digit can take increases exponentially with the size of the Sudoku. The exact number of patterns possible for a Sudoku is calculated with the following formula. N denotes the width of the Sudoku game, i.e., 9 for a classic 9 x 9 Sudoku, and n denotes the width of a subunit, i.e., 3 for a classic puzzle. For a detailed derivation of the formula, see the Appendix A.

$$\prod_{i=0}^{n-1}(n-i)^{2n} \tag{4.1}$$

If no given constraints were considered, a 9x9 Sudoku would already result in 46,656 possible patterns. However, this number can be narrowed down significantly by excluding patterns that are impossible due to the given assignment. This generation of patterns is solved by a tree structure. Starting from the root of the tree, a new level is inserted for each column of the puzzle. Within a level, all rows are inserted, which can be occupied for the column with the digit legal. These rows are already known from `open_tuple` as well as excluding values that are already in the trace of a node. Afterward, all possible patterns are obtained by following the path from each leaf of the tree to the root. The saving of a pattern can be realized with a list. The index within the list indicates the column, and the value at this position indicates the row. So for each column, it is stored in which row the digit is located. The following example shows the process for the board in Figure 4.5. The already computed `open_tuples` are used to build the tree structure.
From these trees, a set of possible patterns can be derived for each individual color. These patterns are stored in `patterns`, where each key represents a color and each value the set of patterns:

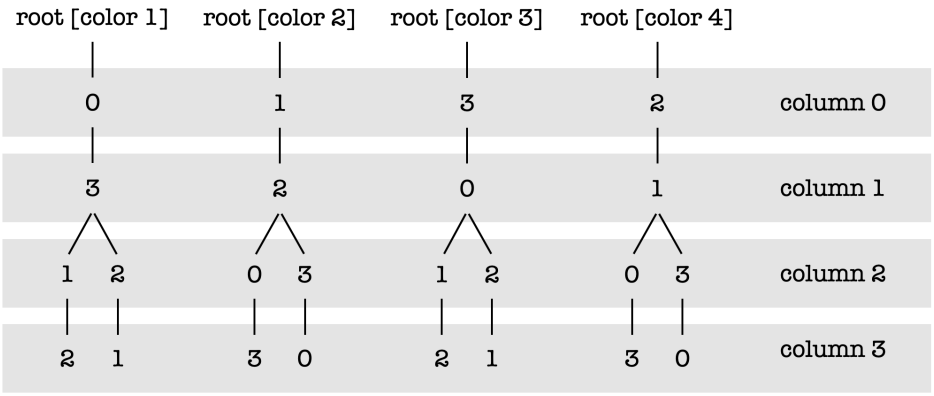


Figure 4.5: Possible patterns per color

```
patterns = {1: [[0, 3, 1, 2], [0, 3, 2, 1]],
            2: [[1, 2, 0, 3], [1, 2, 3, 0]],
            3: [[3, 0, 1, 2], [3, 0, 2, 1]],
            4: [[2, 1, 0, 3], [2, 1, 3, 0]]}
```

### 4.3.2 Pattern Encoding

For each digit, all available patterns must be encoded in this solution. This encoding is done indirectly so that a binary encoding can be applied. For each value, a list is available in `patterns`. Instead of encoding the pattern itself, one encodes its index within this list. Thus exactly so many qubits per value are needed that all indices can be mapped in binary. If one wanted to represent a digit with 82 patterns, seven qubits would be needed so that all indices could be represented within this pattern list. The coding `0000011` would then point to the pattern in the list, which is at position 3.

## 4.4 Reducing the number of auxiliary qubits in Grovers Algorithm

If more complex oracle functions are used in Grover's algorithm, additional qubits are often needed to generate the bit flip of the Grover qubit in the correct state. Here, too, a binary coding of these auxiliary bits, in the form of a counter, can achieve a significant improvement over the primitive approach.

### 4.4.1 The basic setup of auxiliary qubits

In Figure 4.6 a part of a Grover oracle is shown. There are four coding qubits in this circuit. They have been labeled `alpha, beta, gamma, delta`. These represent, for example, four coin tosses, with heads mapped to 0 and tails to 1. The oracle is now to perform the bit flip on the Grover bit in the $|1111\rangle$ state. We illustrate the use of auxiliary qubits in a short toy example. It is true that this case could be solved with a Toffoli gate, which controls all coding qubits and triggers an X-gate on the Grover bit. But the point here is to illustrate the structure of compare bits. The task of these bits, for more complex queries that cannot be mapped to the Grover bit with simple gates, is to check the satisfaction of the desired conditions for each input pair. For example, if the task were to find an assignment where

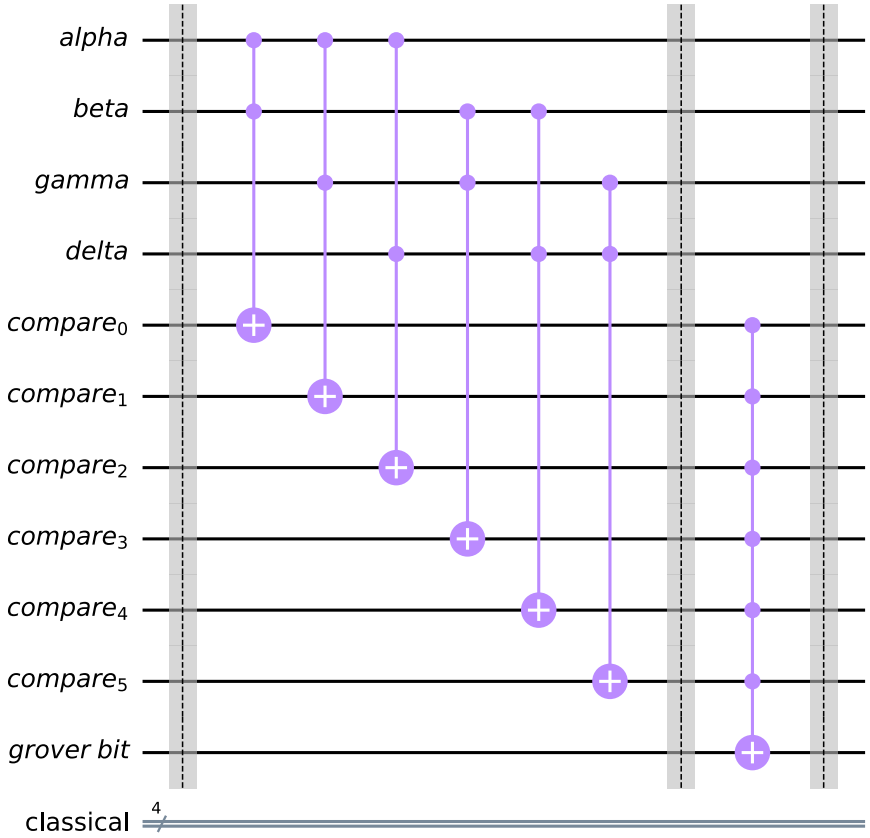


Figure 4.6: Basic setup for Compare Bit in a Grover Oracle

heads and tails alternate for each input variable, the need for these compare bits becomes logical. However, since these oracles would become very large, the example reduces the oracle function to the simplest case. The compare bits 0-5 are to show now in each case for a pair of input variables whether the desired condition is given. In this case, the state $|1\rangle$ for both variables. If each of these compare bits is in state 1, a Toffoli gate generates a bit flip on the Grover bit. The drawback of the basic compare structure is that an oracle with N input elements uses $\sum_{i=2}^{N} i$ compare bits, since each input element must be related to every other one.

### 4.4.2 Auxiliary qubits set up as a counter

To reduce the number of required compare qubits, a simple counter is built. This counter should be incremented by 1 for each pair of input elements that have the desired state in relation. If this counter shows exactly $\sum_{i=2}^{N} i$ at the end, all dependencies are fulfilled, and the Grover bit can be manipulated by a bit flip. The implementation of this counter reduces the number of $\sum_{i=2}^{N} i$ to $log_2 \sum_{i=2}^{N} i + 1$ necessary qubits. Figure 4.7 shows this counter using the example from Figure 4.6. Again, the state $|1111\rangle$ is to be found. This time, however, each pair of input elements maps to a helper bit. If the condition is satisfied, the helper

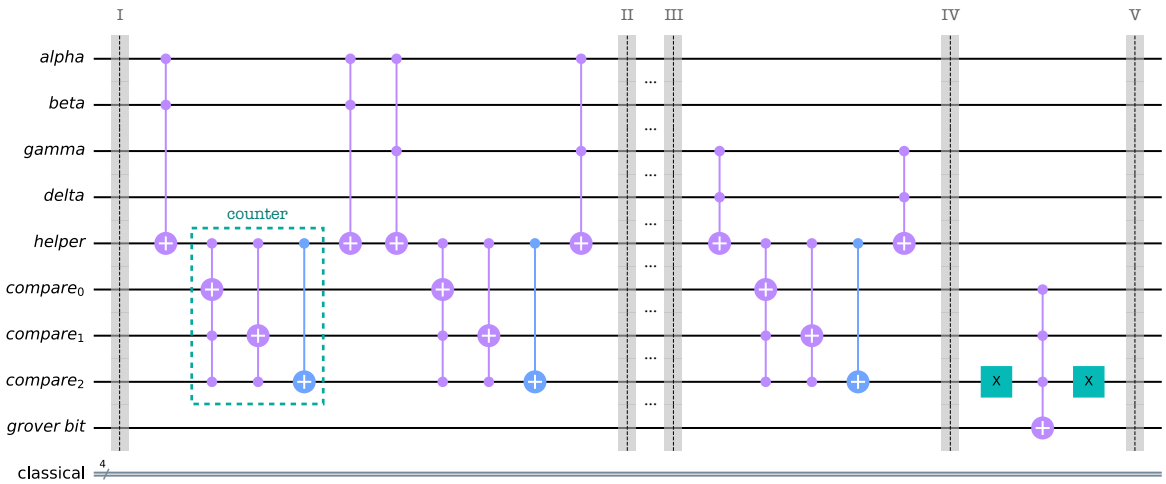


Figure 4.7: Compare Bits set up as a counter for a Grover Oracle

bit is flipped. Then, the counter includes this helper bit in the counter. At the end of the necessary comparisons, a Toffoli gate is applied to the counter bits, which flips the Grover bit with an X gate when the correct number has been counted. In this case, there are four input elements, each of which is compared to every other element, so in total six comparisons are needed. When all of them are satisfied, the counter is at $|110\rangle$. Thus, in section `IV`, the last qubit of the counter is flipped before the Toffoli Gate is ultimately applied.

# 5 Results

This Chapter shows all of our results regarding the achieved reductions of qubits needed and the tools that were developed to collect these numbers.

## 5.1 Comparing Results for different puzzles

The following section compares the number of qubits needed for puzzles of several difficulties starting from easy to Expert Level. These Sudokus were taken from the website `www.sudoku.com`.To show the impact of each optimization strategy, four difficulties were looked at. For each difficulty, five different puzzles were selected. The puzzles used for each Level are shown in the appendix. Figure 5.1 lists the percentage of puzzles that could be solved by only preprocessing the game. This shows the impact of this step when much information is provided to start with. In these cases, the simple algorithm finds the solution completely alone. For easy and medium hard start states, the solving rate by this step is pretty similar. Easy puzzles get solved in 80% of the cases while medium hard problems still are at 60%. For harder Sudoku games listed in expert or AI Escargot, there can not be computed a solution just by preprocessing the game. The rate drops to 0% in these examples.

| **Preprocessing** | Easy<br>Encoding \| Total | Medium<br>Encoding \| Total | Hard<br>Encoding \| Total | AI Escargot<br>Encoding \| Total |
|---|---|---|---|---|
| Solved by Preprocessing | 80 % | 60 % | 0 % | 0 % |

Figure 5.1: Solving Rate by Preprocessing

In the first table of results, represented by Figure 5.2, the qubits needed for the Graph coloring problem solved with Grover's algorithm are shown. In the left column, the used optimizations are listed. In the table the following keywords were used to name the different ideas:

**Primitive:** all n digits are assigned to every empty cell as possible value. No exclusion of digits due to the starting state are computed.
**Preprocessed:** the preprocessing described in Chapter 4 is applied to the free cells. So the number of possible digits per cell is significantly reduced. This step is complementary to the primitive strategy.
**OH:** one hot encoding is used to encode the options per free cell in graph coloring or per pattern in exact covering. One qubit is needed for each option, where a 1 of this qubit indicates that this option is part of the solution while 0 indictes its exclusion from the final solution.
**Binary:** for each group of options, binary encoding is used. A group of options can be a cell with its possible colors for graph coloring or a digit with its patterns for exact covering problems.

**n x (n-1):** describes the process of removing one row or column of the Soduko to reduce the number of cells to be solved as described in the 4th Chapter.

| Graph Coloring Grover | Easy<br>Encoding \| Total | Medium<br>Encoding \| Total | Hard<br>Encoding \| Total | AI Escargot<br>Encoding \| Total |
|---|---|---|---|---|
| Primitive OH | 391 \| 403 | 455 \| 467 | 529 \| 541 | 522 \| 534 |
| Preprocessed OH | 103 \| 113 | 105 \| 115 | 234 \| 245 | 224 \| 235 |
| Preprocessed Binary | 56 \| 66 | 61 \| 71 | 130 \| 141 | 128 \| 139 |
| Preprocessed n x (n-1) Binary | 45 \| 55 | 47 \| 57 | 109 \| 120 | 109 \| 120 |

Figure 5.2: Qubits needed to encode as graph coloring problem

For each strategy and every difficulty level, the number of qubits needed to encode the problem itself is given. Also, the total number of qubits to set up a Grover circuit, including the counter and the Grover bit, is provided. For comparison/counting qubits, the optimized solution from Chapter 4.4 is used. For the levels where Preprocessing has already found a solution, only the unsolved games were considered to calculate the average number of qubits. Most easy puzzles in our set started with around 44 empty cells and medium-level puzzles with around 50. Here, preprocessing accounts for a decrease in the number of qubits needed of 73% (easy) and 76% (medium). For the harder problems with an average of 59 empty cells, this number drops to 63%. The reason for this decrease is that fewer digits can be excluded per empty cell with fewer prefilled units.
The rate of improvement by binary encoding lies between 40%-45% and is not significantly dependent on the hardness of the quiz. For this step, the number of qubits per cell is reduced logarithmically, which is only in correlation with the number of options per cell, not with the number of free cells. As this number is similar for all levels, the effect stays around the same percentage. The effect is limited due to the low number of possible digits per cell, which are binary encoded. The effect of removing a row or column, we call this the *heavy line*, is at 19% and 22% for easy and medium, for expert and AI Escargot problems at 16% and 14%. A possible explanation for this could be, that in easy puzzles, after preprocessing, some areas are already solved while others cluster the unsolved cells. By this effect of clustering, the removed heavy line has a more significant effect on the reduction than with hard problems, where less of the board is solved by preprocessing and the cells to solve are more distributed.

| Exact Covering Grover | Easy<br>Encoding \| Total | Medium<br>Encoding \| Total | Hard<br>Encoding \| Total | AI Escargot<br>Encoding \| Total |
|---|---|---|---|---|
| Primitive OH | 3.410 \| 3.418 | 10.567 \| 10.575 | 35.495 \| 35.503 | 28.935 \| 28.943 |
| Preprocessed OH | 51 \| 58 | 82 \| 89 | 649 \| 657 | 280 \| 288 |
| Preprocessed Binary | 18 \| 25 | 19 \| 26 | 48 \| 56 | 44 \| 52 |

Figure 5.3: Qubits needed to encode as exact cover problem

Figure 5.3 shows the results for the same set of puzzles but when solved with the exact covering approach. An interesting difference here is that due to the way the patterns are constructed, preprocessing has a more significant effect. When cells and digits per cell are excluded in the preprocessing step, in graph coloring, these options just get added to the

sum of possible solutions. As the exact covering approach uses patterns, another effect comes into play. As more options for cells to be occupied by a digit appear, the effect of possible patterns grows through a multiplicative effect, not an additive. The number of combinatorially possible patterns grows very fast so that the reduction effect by preprocessing is above 98% for every single difficulty level. The binary encoding has a larger impact on the absolute reduction of qubits compared to the graph coloring approach. This lies in the effect of binary encoding numbers of different scales. By mapping `N` elements with binary coding, only $\log_2(N)$ qubits are needed. As Figure 5.4 shows, the absolute number of qubits saved by binary coding increases with the size of `N`. Binary coding used for the graph coloring reduces the number of qubits used per cell. However, since the amount of possibilities per field is $<= n$ and n is only 9, the absolute effect of qubits saved due to binary encoding is small. The advantage of the exact covering approach is that one has to encode the valid patterns for n numbers each. Since the number of patterns per value is often in much higher ranges (often several hundred patterns per digit possible), the reduction effect by binary coding is much larger.

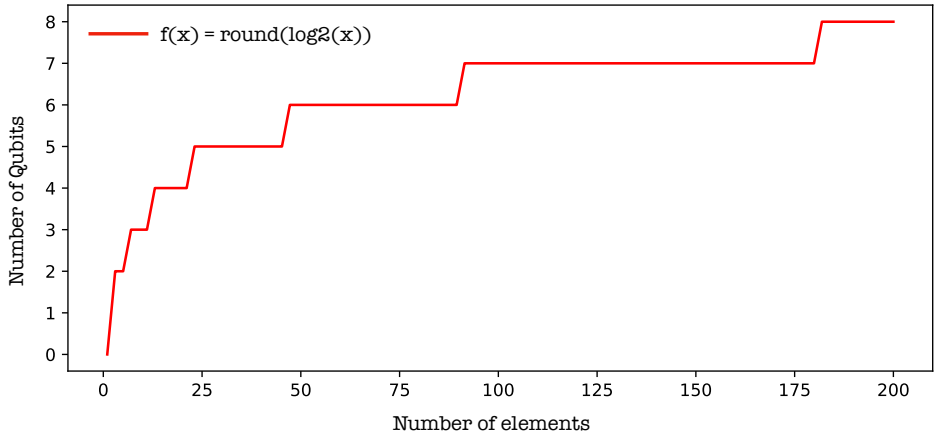


Figure 5.4: Logarithmic effect of binary representation

This optimization technique reduces the number of qubits used to encode by 64% for easy, 76% for medium, 92% for expert, and 84% with AI Escargot. The number increases with the difficulty because the number of possible patterns per digit increases by the number of free cells. The special case of AI Escargot needs fewer patterns in total than an average puzzle at the expert level. This shows that the effort to solve Sudoku by exact covering does not exactly match the effort needed to solve it with humanly used, logical operations.

| QAOA | Easy<br>Encoding \| Total | Medium<br>Encoding \| Total | Hard<br>Encoding \| Total | AI Escargot<br>Encoding \| Total |
|---|---|---|---|---|
| Graph Coloring Preprocessed n x (n-1) OH | 82 \| 82 | 81 \| 81 | 195 \| 195 | 189 \| 189 |
| Exact Covering Preprocessed OH | 51 \| 51 | 82 \| 82 | 649 \| 649 | 280 \| 280 |

Figure 5.5: Qubits needed to encode with QAOA

For the game to be solved with QAOA no extra qubits are needed. Only the qubits that are encoding the problem are necessary for the QUBO. In our work, only One Hot Encoding was

looked at for QUBO, which leads to possible optimizations in the future. A binary encoding, if possible, would significantly improve the number of variables needed. Figure 5.5 represents the number of qubits needed.

## 5.2 Developed tools

For this work, we have developed several tools. These aim to calculate the required qubits for the different approaches and visualize Sudoku games. Using a GUI, arbitrary setups can be computed, and thus the user can develop an intuition for the difficulty of puzzles for different approaches and the effect of the optimizations. The code was written in Python 3.9.13. We performed all computations on a MacBook Pro 2021 running an M1 Max chip and 64GB of RAM. The intention of this chapter is to provide an overview of the most important classes and give the interested reader an introduction to their use or further development. The code is accessible at `https://gitlab.lrz.de/lmu-gsoc/soduko-qaoa`.

### 5.2.1 Tools for preprocessing and optimization

The `SudokuTools.py` module contains classes that handle the preprocessing, the generation of the patterns, and the calculation of the required qubits.

The class `Preprocessor` receives the puzzle as a two-dimensional array. Its aim is to exclude trivial assignments for individual cells and remove the most computationally expensive row or column, as described in the Graph Coloring approach for n x (n-1). This class acts as a utility for all of the other classes of this project and is one of the main building blocks. For a deeper look into preprocessing, this class represents the most helpful starting point.

`PatternGenerator` aims to generate all legal patterns for each digit. This process is important to understand if the reader is interested in the exact covering approach and possible further optimization. For this reason, this chapter provides an explanation of it. As described in Chapter 4, the representation of a pattern is a list of n integers where n represents the size of the Sudoku field. These integers represent the n cells that this digit would occupy in this pattern. The integers themselves represent the row of the cells. The index within the list encodes the corresponding column to each cell. A tree is built as an auxiliary structure to create patterns. In Figure 4.5 the structure of these trees is shown. For each digit, a separate tree is constructed. This data structure aims to find every possible and legal pattern of this digit. The tree is built level by level, each representing a column. The set of all values a leaf and all of its parent nodes form is called a trace. Within this level, all possible row integers are attached to each leaf. Before adding, the algorithm checks whether the trace of the leave contains the row to be inserted. Further, the algorithm checks whether a conflict occurs because the respective row would lead to a conflict within a subunit. Of so, the row is not added to the leaf. After constructing the whole tree, the trace of each leaf is computed. Each trace with a length of n elements represents an individual and legal pattern. This insertion method prevents a double allocation by a digit within a column, row, or subunit.

`EncodingCalculator` is used to get the number of qubits needed for each approach. It gets the different tuple lists from `Preprocessor` as an input as well as the dict of possible patterns. From these data, the different solution paths are constructed, and the number of qubits is returned.

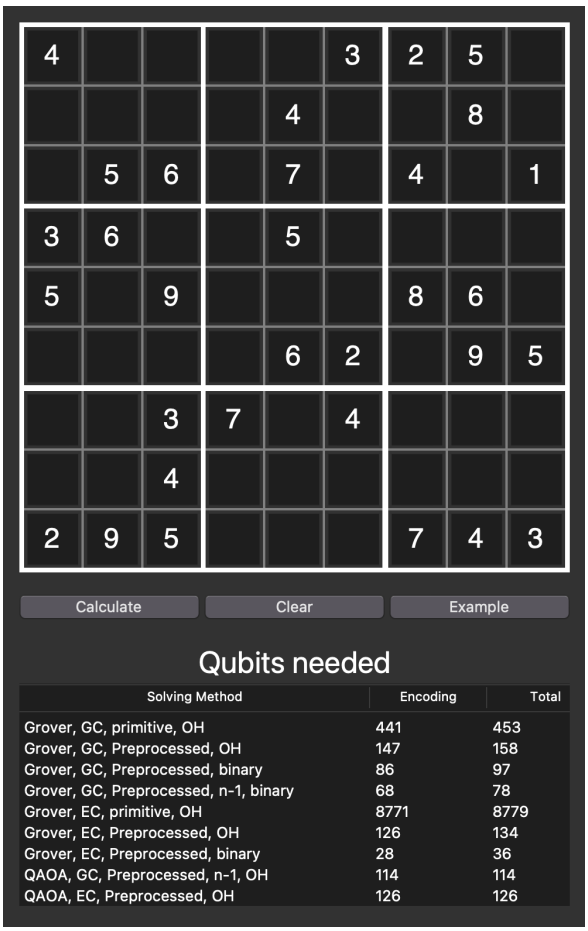


Figure 5.6: User Interface SudokuGUI

### 5.2.2 Developing a UI to calculate numbers of qubits needed

Besides the calculation of qubit numbers, the developed modules in this work were used to develop a graphical user interface. Like all other parts of the project, this was developed with Python3; the library used for this was `tkinter`. The goal of this tool is, knowing the codebase or programming itself, to be able to reproduce the results and do your own experiments. The user can run the program through the command line. For this, Python3 must be installed. By the command `python3 SudokuGUI.py`, in the folder where the files were saved, the program starts. The structure is very compact and straightforward. A screenshot from the software can be seen in Figure 5.6. The Sudoku field at the top can be edited as desired. The user can enter puzzles and then use the button *Calculate* to calculate the result, i.e., the number of qubits needed. With the button *Clear*, all cells are deleted, and a new game can be entered. If one only wants to test the program, the button *Example* can create a sample quiz. The interface is not used to solve puzzles but to calculate the qubit count. An extension with the function in the future is possible.

# 6 Conclusion

Interesting optimization possibilities arise regardless of whether Sudoku games are represented as Graph Coloring or Exact Covering problems. Both problems are very well suited to be solved with Grover's algorithm. For the graph coloring approach, preprocessing and binary coding were very helpful in reducing the number of qubits. Also, the reduction of the field by an entire row or column provides an interesting optimization. For the Exact Covering approach, the binary coding was especially crucial since the logarithmic optimization effect was beneficial due to the high number of patterns per digit. Besides the pure optimizations of the encoding of the game, the reduction of the counter bits for Grover was also crucial to keep the number of qubits needed low. Thus, the best solution to keep the qubit count as low as possible is to encode the puzzle using a binary-coded exact covering problem. Many of the reductions increase the number of quantum gates used in the circuits. For example, using a counter instead of many compare bits in a Grover oracle. The number of gates is critical to the error-proneness of NISQ Era circuits. Therefore, interesting questions arise as to what would be an optimum of the number of qubits and the width (grows with the number of gates used) of the circuit. In addition, examining an efficiently coded QUBO, which one solves with QAOA, against a solution with an efficiently coded Grover algorithm is attractive in terms of error rate and practicality. Regardless of the algorithms, it may be worthwhile to go into more detail about preprocessing. Likewise, the reduction of the area of the Sudoku to be solved can be further analyzed so that the field could become even more diminutive than n x n(-1). Should the hardware develop constantly, systems will soon be available to solve Sudoku puzzles with this setup. Regardless, the focus on efficient coding is still necessary since real-world applications will be limited by the number of qubits available for a long time.

# Abbildungsverzeichnis

# Literaturverzeichnis


[AP] Ankur Pal, Vardaan Mongia-Bikash K. Behera Prasanta K P. Sanghita Chandra C. Sanghita Chandra: *Solving Sudoku Game Using Quantum Computation*. https://www.academia.edu/39751260/Solving_Sudoku_Game_Using_Quantum_Computation. – Abgerufen am 17. Juli 2022

[Cer20] Ceroni, Jack: *Intro to QAOA*. https://pennylane.ai/qml/demos/tutorial_qaoa_intro.html. Version: 2020. – Abgerufen am 17. Juli 2022

[CESS08] Claessen, Koen ; Een, Niklas ; Sheeran, Mary ; Sorensson, Niklas: SAT-solving in practice, 2008. – ISBN 978–1–4244–2592–1, S. 61 – 67

[Del06] Delahaye, Jean-Paul: The Science behind Sudoku. In: *Scientific American* 06 (2006), Nr. 0606-80

[FGG14] Farhi, Edward ; Goldstone, Jeffrey ; Gutmann, Sam: *A Quantum Approximate Optimization Algorithm*. http://dx.doi.org/10.48550/ARXIV.1411.4028. Version: 2014

[GKD18] Glover, Fred ; Kochenberger, Gary ; Du, Yu: *A Tutorial on Formulating and Using QUBO Models*. http://dx.doi.org/10.48550/ARXIV.1811.11538. Version: 2018

[GP20] Guido Pagano, Patrick Becker-Katherine S. Collins Arinjoy De Paul W. Hess Harvey B. Kaplan Antonis Kyprianidis Wen Lin Tan Christopher Baldwin Lucas T. Brady Abhinav Deshpande Fangli Liu Stephen Jordan Alexey V. Gorshkov Christopher M. Aniruddha Bapat B. Aniruddha Bapat: Quantum approximate optimization of the long-range Ising model with a trapped-ion quantum simulator. In: *Proceedings of the National Academy of Sciences* 117 (2020), Nr. 41, 25396-25401. http://dx.doi.org/10.1073/pnas.2006373117. – DOI 10.1073/pnas.2006373117

[Hom] Homeister, Matthias: *Quantum Computing Verstehen*. 5. Auflage. Springer Vieweg Wiesbaden. http://dx.doi.org/https://doi.org/10.1007/978-3-658-22884-2. http://dx.doi.org/https://doi.org/10.1007/978-3-658-22884-2

[HWO+19] Hadfield, Stuart ; Wang, Zhihui ; O'Gorman, Bryan ; Rieffel, Eleanor ; Venturelli, Davide ; Biswas, Rupak: From the Quantum Approximate Optimization Algorithm to a Quantum Alternating Operator Ansatz. In: *Algorithms* 12 (2019), feb, Nr. 2, 34. http://dx.doi.org/10.3390/a12020034. – DOI 10.3390/a12020034

[IBM21] IBM: *IBM Unveils Breakthrough 127-Qubit Quantum Processor*. https://newsroom.ibm.com/2021-11-16-IBM-Unveils-Breakthrough-127-Qubit-Quantum-Processor. Version: 2021. – abgerufen am 10. Juli 2022

[JN22] Jonas Nüsslein, Claudia Linnhoff-Popien Sebastian F. Thomas Gabor G. Thomas Gabor: *Algorithmic QUBO Formulations for k-SAT ans Hamiltonian Cycles*

[Knu00] Knuth, Donald E.: Dancing links. (2000). http://dx.doi.org/10.48550/ARXIV.CS/0011047. – DOI 10.48550/ARXIV.CS/0011047

[Lei77] Leighton, Frank T.: A Graph Coloring Algorithm for Large Scheduling Problems. In: *Journal of research of the National Bureau of Standards* 84 (1977), Nr. 6. http://dx.doi.org/10.6028/jres.084.024. – DOI 10.6028/jres.084.024

[Mic22] Microsoft: *Solving Sudoku using Grover's Algorithm*. https://docs.microsoft.com/en-us/samples/microsoft/quantum/solving-sudoku-using-grovers-algorithm/. Version: 2022. – Abgerufen am 17. Juli 2022

[MO08] Mihai Oltean, Oana M.: Exact Cover with Light. In: *New Generation Computing* 26 (2008), Nr. 4. http://dx.doi.org/10.1007/s00354-008-0049-5. – DOI 10.1007/s00354–008–0049–5

[PKS13] Pfeiffer, Uwe ; Karnagel, Tomas ; Scheffler, Guido: A Sudoku-Solver for Large Puzzles using SAT. In: Voronkov, Andrei (Hrsg.) ; Sutcliffe, Geoff (Hrsg.) ; Baaz, Matthias (Hrsg.) ; Ferm\üller, Christian (Hrsg.): *LPAR-17-short. short papers for 17th International Conference on Logic for Programming, Artificial intelligence, and Reasoning*. Bd. 13, EasyChair, 2013 (EPiC Series in Computing). – ISSN 2398–7340, 52–57

[Pre21] Preskill, John: Quantum computing 40 years later. In: Hey, Anthony J. (Hrsg.): *Feynman Lectures on Computation, 2nd edition*, 2021

[Qis20] Qiskit: *Solving combinatorial optimization problems using QAOA*. https://qiskit.org/textbook/ch-applications/qaoa.html. Version: 20. – Abgerufen am 17. Juli 2022

[Qis22] Qiskit: *Qiskit Textbook (beta)*. https://qiskit.org/learn/. Version: 2022. – Abgerufen am 19. Juli 2022

[SCGM+15] Soto, Ricardo ; Crawford, Broderick ; Galleguillos Miccono, Cristian ; Paredes, Fernando ; Norero, Enrique: A Hybrid Alldifferent-tabu Search Algorithm for Solving Sudoku Puzzles. In: *Intell. Neuroscience* 2015 (2015), 05, S. 40:40–40:40. http://dx.doi.org/10.1155/2015/286354. – DOI 10.1155/2015/286354

[SMS+20] Saha, Amit ; Majumdar, Ritajit ; Saha, Debasri ; Chakrabarti, Amlan ; Sur-Kolay, Susmita: *Asymptotically Improved Circuit for d-ary Grover's*

*Algorithm with Advanced Decomposition of n-qudit Toffoli Gate*. http://dx.doi.org/10.48550/ARXIV.2012.04447. Version: 2020

[Sud08] SudokuWiki: *Escargot*. https://www.sudokuwiki.org/Escargot. Version: 2008. – Abgrufen am 17. Juli 2022

[VD12] Vishal Donderia, Prasanta K. J.: A novel scheme for graph coloring. In: *Procedia Technology* 4 (2012), Nr. 4

[WHSK20] Wang, Yuchen ; Hu, Zixuan ; Sanders, Barry C. ; Kais, Sabre: Qudits and High-Dimensional Quantum Computing. In: *Frontiers in Physics* 8 (2020). http://dx.doi.org/10.3389/fphy.2020.589504. – DOI 10.3389/fphy.2020.589504

# Appendix A

The Derivation of equation (1)

$$\prod_{i=0}^{n-1}(n-i)^{2n} \tag{1}$$

is explained in this appendix. This equation was shown in Chapter 4. Consider a Sudoku with a field size of $n^2 \times n^2$. So a subunit has a width and height of n. In the field, there are $n \times n$ subgrids. The equation calculates the number of possible patterns for a digit on a completely empty field of size $n^2 \times n^2$. To derive that equation, one looks at the steps that are involved, to calculate this number for a puzzle step by step.

The calculation is divided into n equations, each calculating the number of possibilities for each subunit line. A subunit line is constructed of all subunits, that are stacked above on each other. So a Sudoku of size $n^2 \times n^2$ would have n of these subunit lines.

For the first line, the calculation is pretty simple. In the first column, there are n subunits with n empty cells each. So there are n x n possibilities to place the digit. In the second column, there are n-1 subunits with n empty cells left to place the digit. So the number of possibilities is $n \times (n-1)$ for the second column. With each column, the number of legal subunits to hold the digit is decreased by 1. Until in the $n^th$ column, there is only one lift with n possibilities $n \times (n-(n+1))$. So for the first subunit line, there are

$$[n*n]*[n*(n-1)]*...*[n*(1)] = n^n \prod_{i=0}^{n-1}(n-i) \tag{2}$$

possible patterns to arrange the digit in a way that doesn't hurt any constraints.

The second line can be calculated similarly. As we are in a new subunit line, every subunit is allowed again to be used. But in each of them, we already have one row that is not allowed anymore from our placing in subunit line one. So the formula for this line looks like this:

$$[(n-1)*n]*[(n-1)*(n-1)]*...*[(n-1)*(1)] = (n-1)^n \prod_{i=0}^{n-1}(n-i) \tag{3}$$

The pattern is already easy to observe. For every subunit line we calculate, the number of allowed rows per subunit decreases by one. As we fill up the stacked subunits within a subunit line, with every column, we exclude one subunit more for the next column. So for the $n^th$ subunit line, the equation looks like this:

$$[1*n]*[1*(n-1)]*...*[1*(1)] = 1^n \prod_{i=0}^{n-1}(n-i) \tag{4}$$

Note that all of theses partial products represent the options per subunit line. To get the options of patterns for the whole game, we need to multiply all of them. The term

$$\prod_{i=0}^{n-1}(n-i) \tag{5}$$

shows up for every subunit line, so n-times they are multiplied. So we can write:

$$\prod_{i=0}^{n-1}(n-i)^n \tag{6}$$

The factor in front of the term can also be represented by the product:

$$\prod_{i=0}^{n-1}(n-i)^n \tag{7}$$

When combining these terms by multiplying them, the resulting term to calculate the number of pissible patterns for an $n^2 \times n^2$ Sudoku is:

$$\prod_{i=0}^{n-1}(n-i)^{2n} \tag{8}$$

# Appendix B

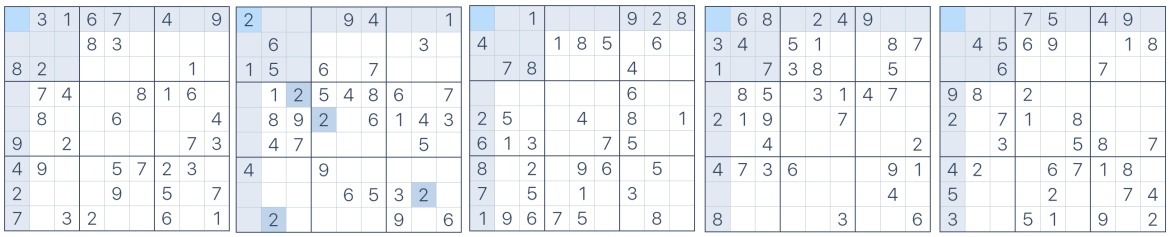

Figure 1: Easy puzzle set used

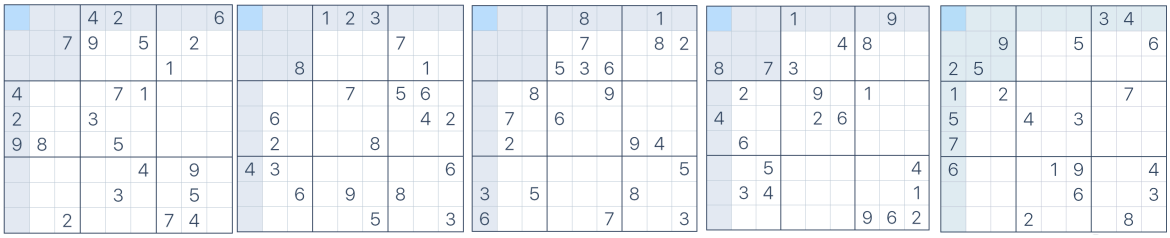

Figure 2: Medium puzzle set used

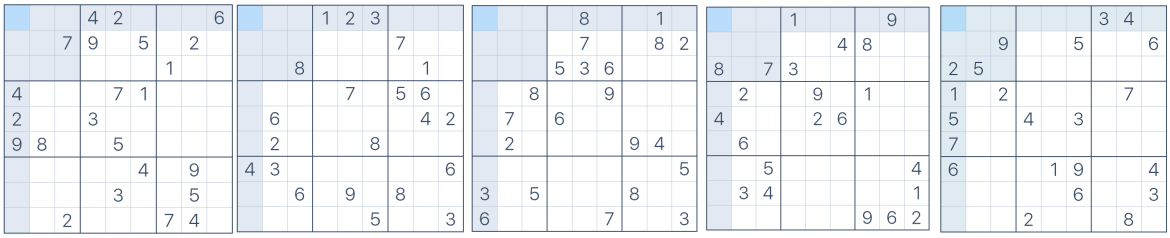

Figure 3: Expert puzzle set used