

Πτυχιακή Εργασία

Solving The Sudoku Puzzle Using Quantum Annealing

Ευάγγελος Στογιάννος
ΑΜ Π2018062

Επιβλέπων:

Θεόδωρος Ανδρόνικος, Αναπληρωτής Καθηγητής
Τμήμα Πληροφορικής Ιόνιο Πανεπιστήμιο

Τριμελής Επιτροπή:

Θεόδωρος Ανδρόνικος, Αναπληρωτής Καθηγητής
Τμήμα Πληροφορικής Ιόνιο Πανεπιστήμιο

Μιχαήλ Στεφανιδάκης, Αναπληρωτής Καθηγητής
Τμήμα Πληροφορικής Ιόνιο Πανεπιστήμιο

Μάρκος Αυλωνίτης, Καθηγητής
Τμήμα Πληροφορικής Ιόνιο Πανεπιστήμιο

Corfu, June 2023

A special thanks to Julian Hatzky

for his invaluable help.

*–Evangelos Stogiannos
Corfu, August 2023*

Abstract

This thesis focuses on studying the Sudoku puzzle on D-Wave quantum systems. We have developed a new matrix formulation for the Sudoku problem, which allows for easy integration of benchmark instances in quantum platforms. Our mathematical analysis determines the exact number of constraints needed to express Sudoku puzzles as Ising and QUBO formulations. Additionally, we have created a technique for generating QUBO models for Sudoku puzzles of different sizes. Our extensive experiments show that the D-Wave Advantage_system6.2 can handle smaller Sudoku instances, but larger ones are still not possible. Furthermore, our experiments confirm that the D-Wave hybrid solvers always produce valid solutions without violating the given constraints.

Contents

1	Introduction	1
1.	Introduction to Quantum Annealing	1
1.1	What is Quantum Annealing	1
1.2	Available Systems for Quantum Annealing	2
1.3	The QUBO model for Quantum Annealing	3
2.	The Sudoku Puzzle	5
2	Related Work	7
1.	Quantum Annealing Applications	7
1.1	Machine Learning	7
1.2	Bio-informatics	8
1.3	Combinational Optimization	9
2.	Sudoku in Classical Computing	10
2.1	Backtracking Algorithm	10
2.2	Constraint Satisfaction Algorithms	10
2.3	Heuristic-Based Techniques	11
3.	Sudoku in Quantum Computing	11
3.1	Variational Quantum Algorithms	11
3.2	Quantum Walks	12
3.3	Quantum Approximate Optimization Algorithm (QAOA)	12
3.4	Quantum Annealing	13

3	Theory	14
1.	QUBO Fundamentals	14
1.1	Ising Models	14
1.2	QUBO Models	15
2.	A QUBO Model for the Sudoku Puzzle	15
2.1	Sudoku Rules as Constraint Matrices	18
3.	Solution Interpretation	28
4	Experiments	30
1.	Experiment Sets	30
2.	Approximate Experiment Size	30
3.	Solvers Used	31
5	Results and Analysis	32
1.	The 4×4 Sudoku Instance	32
1.1	Simulated Quantum Annealer Sampler Results	33
1.2	Hybrid Annealer Sampler Results	36
1.3	Quantum Annealer Sampler Results	39
2.	The 9×9 Sudoku Instance	43
2.1	Simulated Quantum Annealer Sampler Results	43
2.2	Hybrid Annealer Sampler Results	46
2.3	Quantum Annealer Sampler Results	49
6	Conclusions	50
A	Software	56

Chapter 1

Introduction

1. Introduction to Quantum Annealing

1.1 What is Quantum Annealing

In 1982 Richard Feynman introduced the idea of "parallelism" [1] for improving the solutions to computational problems. This idea is superior to its conventional parallels in many of the aforementioned problems, showcased exceptionally in Shor's[2] and Grover's [3] algorithms. These algorithms can provide better and quicker solutions than the commonly used alternatives. Quantum computers encapsulate the idea of parallelism, as they are made based on this principle from the ground up [4].

Quantum annealing is a metaheuristic especially good at solving combinatorial optimization problems. Introduced by Kadowaki and Nishimori in 1998 [5], it has many similarities to conventional simulated annealing [6], with the main distinction being that it takes advantage of the principles of quantum adiabatic computation. Like simulated annealing, the quantum counterpart produces an energy landscape from all possible states in a quantum superposition, with its ground state representing the problem's optimal solution [5]. After the metaheuristic creates the landscape, it parses over it repeatedly until it finds the optimal solution with a high enough probability. The way the landscape is parsed is what separates the two methods. In simulated annealing, the system needs to calculate each state's energy sequentially, while quantum annealing uses multi-qubit tunneling, improving performance due to the high degree of parallelism. This possibility of simultaneously analyzing all states is crucial in tackling NP-complete problems. While the advantage of quantum annealing is clear, further developments are needed for it to be useful as a precise-solving program [7].

Among the fields where quantum computers provide an advantage is the optimization problems [8, 9]. Solving an optimization problem in a quantum com-

puter requires a Hamiltonian, its ground state representing the problem's optimal solution. The Hamiltonian function acts as the multivariable objective function for quantum annealing. The quantum system starts at an equal superposition of all its states and applies the Hamiltonian. Over time the annealer develops according to the Schrödinger equation, and its energy state evolves, affected by the strength of the local transverse field and its differentiation over time. To "arrive" at the problem's solution, the system slowly weakens the transverse field, and after enough time, the system reaches the ground state. The ground state will be the solution to the problem, provided that the Hamiltonian used is correct.

The orchestration of the quantum annealing is executed through the utilization of quantum processing units (QPUs) housed within D-Wave's quantum machines. In contrast to the conventional binary entities, the constituents of QPUs encompass superconducting loops, wherein the states of minimum energy are accorded the stature of quantum bits, abbreviated as qubits for the sake of simplicity. As previously alluded to, these qubits exhibit a parallelism to conventional binary bits, as underscored in prior research [10, 11], signifying their capacity to adopt values of either 0 or 1.

However, it is imperative to discern that qubits also manifest as quantum entities, thereby affording them the capability to inhabit an additional state, as superposition, which encapsulates a probabilistic mixture of both 0 and 1. It is only upon the act of observation that qubits relinquish their superposition and assume one of the two discrete binary states.

In the orchestration of the quantum annealing process, an initial state is generated wherein all constituent qubits subsist in a state of superposition. Over the course of the process, this initial state transitions gradually into a novel configuration, engendering the eventual collapse of each qubit to a definitive binary value. Upon the culmination of this transformative progression, the resultant configuration assumes a classical state, thereby encapsulating the ultimate optimal solution to the problem at hand.

1.2 Available Systems for Quantum Annealing

In 2022 a significant announcement was made from D-Wave—a pioneering entity in the quantum computing domain—pertaining to a prototype upgrade within their extant generation of quantum computers [12]. This progression stands as a continuation of their developmental trajectory, wherein the year 2020 saw the emergence of the Advantage System computer, marked by the assimilation of the innovative Pegasus topology. A seminal feature of this progression was the substantial amplification in both the scope and caliber of their computational solvers, a transformation that materialized through the endowment of over 5000 qubits.

D-Wave later announced Advantage2 Prototype[12], characterized by a transition from the Pegasus topology to the Zephyr topology—a transition underscored

by a nuanced interplay between quantitative reduction and qualitative augmentation. While the system saw a reduction in available qubits, with the count hovering slightly above 500, this contraposition was accompanied by a remarkable refinement in the coupling dynamics intrinsic to the topology. Evidently, the inherent architecture of the Zephyr topology furnishes a heightened capacity for qubits to engage in superposition-based interactions, representing an instance where quality supersedes numerical abundance.

Specifically, the most recent instantiation of the Advantage System computer featured a coupling degree of 15, indicating that each individual qubit could establish interconnections with a maximum of 15 other qubits. In stark contradistinction, the Zephyr topology shines through with a coupling degree elevated to 20, a quantifiable enhancement that augments both the computational capabilities of the annealer and the fidelity of the resultant solutions. Acknowledging the present underdeveloped stature of quantum computers operating on the Zephyr topology, the pragmatic recourse lies in persisting with the utilization of Pegasus’s solvers, hinging on the extant computational prowess it affords.

The mapping of qubits within the Advantage Quantum Processing Unit (QPU) is orchestrated within the confines of a P16 Pegasus graph, wherein these qubits are logically allocated within a matrix structure of dimensions $15 \times 15 \times 2$, which delineates a diagonal grid. Within each unit cell of this grid, an assemblage of 24 qubits finds its abode, and each qubit is intricately entwined with another co-resident qubit within the same cell, alongside a pair of similarly aligned qubits located in adjacent cells [13]. This arrangement engenders a matrix of interconnectedness that underpins the QPU’s operational capacity.

Furthermore, within this computational endeavor, the notion of the “working graph” assumes primacy—a metric that encapsulates the proportion of qubits actively participating in the computation— as the total set of connected cells that inherently exist within the quantum processing unit’s physical infrastructure.

1.3 The QUBO model for Quantum Annealing

The landscape of options available for leveraging quantum annealers in the domain of optimization and probabilistic sampling is notably diverse. One of the most prominent players in this realm is represented by the computational platforms provided by D-Wave. The primary focus of D-Wave’s computing resources lies in the resolution of Quadratic Unconstrained Binary Optimization (QUBO) problems, a class of challenges characterized by their mathematical structure involving binary variables.

The concept of Quadratic Unconstrained Binary Optimization (QUBO) models constitutes a versatile pattern-matching technique with a wide array of applications, spanning from the realms of machine learning to the resolution of intricate optimization challenges [14]. At its core, this methodology involves the minimiza-

tion of a quadratic polynomial function across a set of binary variables [15, 16], which in itself embodies a hallmark of computational complexity, often associated with NP-hard problems [17].

The Schrödinger equation, a linear partial differential equation, serves as a fundamental descriptor for the wave or state function characterizing quantum-mechanical systems. Through the precise identification and inclusion of relevant variables within a given system, it becomes possible to construct models that faithfully represent diverse physical phenomena. Notably, the Ising model, serving as an equivalent counterpart to the QUBO model, was initially postulated in 1924 by Ernst Ising and Wilhelm Lenz. This model has subsequently undergone substantial investigation, particularly within the domain of ferromagnetism [18, 19].

QUBO models contain two types of variables, the linear variables that characterize the objective function's variables, and the quadratic variables which characterize the coefficients of quadratic terms in the objective function. In the matrix form of a QUBO, which we will be constructing in later chapters, the two types are visualized as follows:

- ▶ The diagonal elements of the QUBO matrix represent the coefficients of the linear variables.
- ▶ The off-diagonal elements represent the coefficients of the quadratic variables.

Under the assumption that qubits can be aligned with the variables intrinsic to the model, an intriguing conjecture arises: the interactions between these qubits inherently mirror the costs associated with each conceivable pair of qubits. This premise affords the opportunity to recast the model's structure into a configuration reminiscent of an undirected graph. Herein, the qubits are symbolized as vertices within this graph, while the interconnections between qubits find representation through edges.

In 2017, D-Wave introduced the quantum-hybrid system Qbsolv, designed ostensibly to grapple with problems of arbitrary scale. This innovative approach is noteworthy in that it does not exclusively rely on quantum processing but leverages a combination of classical and quantum resources. The key modus operandi involves decomposing the complete problem graph into numerous, considerably smaller subgraphs using classical central processing units (CPUs), subsequently resolving each subgraph employing a quantum processing unit (QPU). Iterative in nature, this procedure is further reinforced by a Tabu-based search strategy, which endeavors to converge towards progressively improved solutions throughout the course of iterations [20].

2. The Sudoku Puzzle

Sudoku, an intricate combinational number puzzle game that emerged in the 1970s, hails from its origins in Japan[21, 22]. Owing to its deceptively simple rules entwined with an intricate web of potential solutions, Sudoku has garnered substantial academic attention, finding itself under the scrutiny of various disciplines such as mathematics and computer science.

At its core, a typical Sudoku puzzle unfolds as a 9×9 grid, further partitioned into nine 3×3 subgrids, collectively forging the quintessential Sudoku structure. Within this intricate arrangement, each cell is designated to accommodate an integer from the range of 1 to 9, thus culminating in the stipulated range of digits. It is worth noting that although the foundational 9×9 variant is perhaps the most recognized iteration, an array of adaptations and variations exists, including the 4×4 variant, which merits detailed exploration later within the discourse. In a broader sense, a Sudoku puzzle can be conceived as an $n \times n$ grid, wherein the dimensions of each subgrid amount to $\sqrt{n} \times \sqrt{n}$, thereby establishing a more generalized schema.

Fundamentally, the Sudoku puzzle commences with a partially populated grid, with select cells preassigned with numbers, often referred to as "hints." The primary objective then evolves into the completion of the grid, adhering to a stringent set of rules which delineate the following imperatives:

- ▶ Each row within the grid must exclusively accommodate each digit exactly once.
- ▶ Correspondingly, each column must also preserve the uniqueness of every digit.
- ▶ Every 3×3 subgrid must contain only one instance of each number.

These constraints, while seemingly straightforward, synergistically conspire to confer a singular solution to every Sudoku puzzle instance. In other words, for any given set of initial hints, a solitary solution that completely satisfies all constraints unequivocally exists. Nonetheless, the task of discerning this precise solution can be intrinsically daunting, as the sheer magnitude of potential solutions can be bewildering. For instance, the classic 9×9 Sudoku configuration engenders a staggering count of $9! \times 722 \times 27 \times 27704267971 = 6670903752021072936960$ distinct, valid grids[23].

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

Example of an unsolved Sudoku puzzle

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

Example of a solved Sudoku puzzle

The popularity and complexity of Sudoku has made it alluring to many researchers, especially in the fields of mathematics and algorithms.

Chapter 2

Related Work

1. Quantum Annealing Applications

Quantum Annealing has seen tremendous development and has been an especially researched field for its applications in many problems from a plethora of fields. As stated before, the unique properties of quantum annealing allow it to easily reach the optimal solution of an optimization problem, when expressed as an objective function. Thus quantum annealing has been of interest to several fields, some of which we present below.

1.1 Machine Learning

In the realm of machine learning, quantum annealing has emerged as an intriguing paradigm that holds the potential to reshape the landscape of optimization and complex problem-solving. The principles of quantum computing, combined with machine learning's insatiable appetite for efficient solutions presents a captivating synergy that researchers are actively exploring [24, 25]. Quantum annealing's allure within the context of machine learning stems from its intrinsic ability to leverage quantum phenomena, such as superposition and entanglement, to explore an extensive array of potential solutions in parallel. This intrinsic parallelism promises to transcend the limitations of classical computing and expedite the resolution of intricate optimization tasks inherent to machine learning algorithms.

As machine learning algorithms delve into increasingly intricate and data-rich domains, the demand for efficient optimization methods becomes more pronounced. Quantum annealing's distinctive capacity to navigate complex solution landscapes by probing multiple paths in parallel has ignited interest as a potent tool to accelerate the optimization processes that underpin machine learning. Tasks like hyperparameter tuning[26], feature selection[27], and clustering [28] stand to benefit from quantum annealing's potential to swiftly explore an expansive search space and

identify optimal configurations. By harnessing quantum phenomena that classical counterparts cannot mimic, quantum annealing demonstrates a tantalizing ability to unlock new dimensions of efficiency in machine learning workflows.

However, the practical integration of quantum annealing into machine learning is not without challenges. Quantum hardware’s current limitations in terms of error rates and qubit coherence pose hurdles that must be surmounted. While the theoretical underpinnings of quantum annealing in machine learning are compelling, successful implementation hinges on the continued advancement of quantum hardware and robust error mitigation strategies. As the field of quantum computing matures, the convergence of quantum annealing with machine learning holds the promise of propelling the domain to new frontiers of computational efficiency, yielding insights and solutions that were previously unattainable using classical methods.

1.2 Bio-informatics

Quantum annealing has proven to be a promising avenue in the realm of bio-informatics, particularly in addressing complex challenges such as protein folding[29]. Protein folding is the process by which a protein adopts its native conformation, is inherently intricate due to the vast number of possible conformations. Quantum annealing offers a unique computational approach that holds potential in more accurately modeling these complex folding patterns. By encoding the intricate energy landscapes and interactions of proteins into a quantum annealing framework, researchers aim to gain a better understanding of protein folding and improve drug discovery by being able to predict future mutations on protein structures.

Protein folding involves intricate energy landscapes and interactions[30], making it a prime candidate for quantum annealing’s optimization capabilities. Quantum annealing’s intrinsic ability to explore a wide solution space simultaneously aligns with the multi-dimensional nature of protein folding landscapes. Leveraging quantum properties, such as entanglement and tunneling, quantum annealing can potentially navigate these complex energy landscapes more efficiently than classical methods.

While the application of quantum annealing to protein folding is promising, challenges remain. The current scale and precision of quantum annealers may limit their capability to accurately capture the intricate dynamics of protein molecules. Additionally, the need for error correction and increased qubit coherence in quantum hardware poses hurdles. Nonetheless, the utilization of quantum computing technology applications for bio-informatics is still very promising.

1.3 Combinational Optimization

Undoubtedly, one of the most distinguished and extensively recognized challenges in the realm of combinational optimization pertains to the Hamiltonian Cycle Problem (HCP). Its prominence is further amplified by its broader manifestation known as the Traveling Salesman Problem (TSP). These intricacies render them as highly favorable contenders for investigating their resolution potential through the utilization of quantum annealers. The objective of this study is to evaluate the efficiency exhibited by various iterations of the D-Wave quantum annealer in grappling with both the Hamiltonian Cycle Problem and the Traveling Salesman Problem.

One of the most noteworthy results of the study is an all-encompassing QUBO matrix form for TSP and HCP problems. This formalism draws its inspiration from the constructs elucidated in the work by Papalitsas [31]. It is pertinent to highlight that this endeavor also incorporates a normalization stratagem, adept at mitigating predicaments stemming from potential imbalances in the weight distribution within Traveling Salesman Problem graphs.

Empirical investigations conducted upon D-Wave’s annealing platforms furnish us with a cadre of discerning observations:

- ▶ The Advantage_system4.1 outperforms Advantage_system1.1, utilizing qubits more efficiently and consistently generating accurate solutions.
- ▶ While limitations hinder providing a “correct” solution, the small graph instances up to 17 nodes can be tackled using quantum annealers with the mentioned methods.
- ▶ Hybrid solvers consistently deliver accurate outcomes without breaching constraints.

The ramifications of these empirical revelations are of considerable moment, casting a spotlight upon the latent prowess harbored by quantum annealers in the domain of resolving the intricate challenges posed by the Traveling Salesman Problem and the Hamiltonian Cycle Problem. Remarkably, these findings coalesce harmoniously with the conjectures posited in the work by Warren [32]. Given the persistent strides witnessed in the quantum computing arena, the landscape appears auspicious for the ascendancy of quantum annealing as a credible avenue for contending with real-world instances of the Traveling Salesman Problem, a prognosis fortified by the ongoing rapid strides in the scale and scope of quantum computational resources.

The computational outcomes gained from this inquiry confer palpable validation upon the efficiency inherent in the framework proffered. The fascinating and encouraging findings catalyze a collective motivation to extend the scope of this

approach, ushering in its application to other problem classes. It is worth mentioning that this anticipated expansion primarily encompasses the gamut of TSP variations.

2. Sudoku in Classical Computing

In this chapter we shall examine the Sudoku Puzzle within the context of classical computing. We will present the most common algorithms and methodologies employed for efficient Sudoku-solving, analyzing their strengths, limitations, and computational complexities.

2.1 Backtracking Algorithm

The backtracking algorithm is one of the most common classical methods used for solving Sudoku puzzles. It employs a recursive search technique to explore possible digit assignments for each cell in the grid. Starting at the top-left cell, the algorithm tries different digits (1 to 9) one by one and checks if the placement satisfies the rules of Sudoku (no repeated digits in rows, columns, and 3x3 subgrids). If a digit violates the rules, the algorithm backtracks and tries another digit until a valid solution is found or all possibilities are exhausted. [33] Despite its simplicity and ease of implementation, the backtracking algorithm suffers from inefficiency, especially when solving large and complex Sudoku puzzles. The search space grows exponentially with the number of empty cells, leading to a combinatorial explosion of possibilities. As a result, the backtracking algorithm can take a considerable amount of time to find solutions for challenging Sudoku instances. [34]

2.2 Constraint Satisfaction Algorithms

Constraint Satisfaction Problems (CSPs) offer another class of classical approaches to solving Sudoku puzzles. CSPs involve defining a set of variables, domains, and constraints, where the goal is to find a valid assignment of values to variables that satisfies all constraints. Sudoku can be represented as a CSP, with variables being the cells, domains being the digits (1 to 9), and constraints defined by the Sudoku rules. [35] Various algorithms like the Constraint Propagation and Backjumping, as well as techniques like arc consistency, are employed in constraint satisfaction approaches to Sudoku. These methods exploit the inherent constraints in the puzzle to reduce the search space and improve solving efficiency. However, like the backtracking algorithm, the performance of constraint satisfaction approaches degrades as the puzzle complexity increases, and they may still face challenges when dealing with exceptionally difficult Sudoku instances.

2..3 Heuristic-Based Techniques

To overcome the limitations of exhaustive search algorithms like backtracking, heuristic-based techniques have been developed to guide the search towards potential solutions more efficiently. One such technique is the "Minimum Remaining Values" (MRV) heuristic, where the algorithm selects the cell with the fewest remaining possible values to be filled first. This approach narrows down the possibilities faster and can often lead to early identification of dead-end paths. [36] Another widely used heuristic is the "Degree Heuristic," which prioritizes cells with more constraints (i.e., peers) to be filled first. The idea is to tackle cells with higher interactions first, potentially reducing the branching factor and search space. [37] These heuristic-based techniques significantly improve the efficiency of solving Sudoku puzzles compared to pure backtracking algorithms. However, they still face limitations in handling puzzles with complex interlocking patterns, where the heuristic choices might not lead to immediate success.

3. Sudoku in Quantum Computing

Quantum computing, with its promise of exponential speedup for certain computational tasks, has attracted significant interest in solving combinatorial puzzles like Sudoku. In this section, we delve deeper into existing quantum algorithms proposed for Sudoku solving.

3..1 Variational Quantum Algorithms

Variational Quantum Algorithms (VQAs) have exhibited auspicious outcomes in the realm of addressing optimization quandaries through the utilization of quantum computers possessing limited computational capacities. This specific section of inquiry is dedicated to the exploration of the transmutation of VQAs for the purpose of elucidating solutions to Sudoku puzzles. In the ensuing discourse, we introduce the cardinal notion of quantum circuits serving as variational ansatzes, which are subsequently subjected to an iterative optimization process within the purview of VQAs to discern feasible Sudoku solutions.

Our inquiry embarks upon an analysis of diverse quantum circuit architectures, variance in gate sets, and assorted optimization methodologies, all within the confines of Sudoku puzzle resolution. This empirical assessment aims to ascertain disparities in performance arising from these distinct architectural choices, shedding light on their respective efficacies within the context of Sudoku solution endeavors. This exploratory undertaking is underpinned in [38].

3.2 Quantum Walks

Quantum walks manifest as quantum analogues of their classical random walk counterparts, presenting a means for effectual graph exploration. Their applicability to Sudoku puzzles introduces an innovative paradigm wherein the puzzle's grid is mapped onto a graph configuration, with individual cells manifesting as nodes and inter-cell constraints as edges. The algorithmic instantiation of the quantum walk entails the initialization of a superposition encompassing the entire gamut of conceivable cell states. Subsequent to this initialization, the quantum walker progresses across the graph lattice, facilitated by quantum operators, with the eventual quantum state measurement yielding a repertoire of potential Sudoku solutions.

Quantum walks hold intrinsic promise in the realm of Sudoku puzzle solution search due to the distinctive phenomenon of quantum interference. This attribute enables constructive and destructive interactions amongst different solution paths, thereby potentially amplifying the likelihood of identifying valid solutions in a more efficient manner. Notwithstanding these potential benefits, the practical implementation of quantum walks on near-term quantum hardware and the formulation of graph structures aptly representative of Sudoku puzzles present formidable challenges that warrant meticulous consideration. This contemplative analysis is anchored in the scholarly investigation by Rajchel et al. [39].

3.3 Quantum Approximate Optimization Algorithm (QAOA)

Sudoku puzzles are essentially combinatorial problems, thus the Quantum Approximate Optimization Algorithm (QAOA) is a great candidate for solving them. QAOA, similarly to quantum annealing solves the problem by mapping the optimization problem onto a quantum system's evolution and simultaneously examines a plethora of different solutions, using a series of parameterized quantum gates. These gates gradually optimize a quantum state to approximate the optimal solution to the problem. As shown in [40], Ankur Pal et al. developed an algorithm capable of solving a Sudoku instance using quantum circuits, and utilizing an Ising Hamiltonian model, to establish an isomorphic mapping of Sudoku's logical constraints and objectives. The experiments they performed on 4x4 Sudoku instances proved very positive, with their solver being able to outperform even classical Sudoku solvers. The quantum facet of their algorithm entails the application of QAOA to discern optimal numeral configurations within the Sudoku grid. Through iterative traversal of the Ising Hamiltonian landscape, the algorithm endeavors to minimize an objective function aligned with the puzzle's prescribed rules. The parallelism introduced by the use of quantum computers, enables the exploration of numerous potential solutions, vastly improving the solving time required to reach the optimal solution.

3.4 Quantum Annealing

To effectively utilize the potential of quantum annealing, we first need to formulate the Sudoku Puzzle into mathematical representations that can be understood and solved by quantum annealing devices. One of the most popular and well-researched formulation is the QUBO model. This translation involves mapping the Sudoku puzzle's binary variables to binary variables. Quantum-inspired techniques, including quantum-inspired constraint satisfaction, seek to take advantage of the superposition and entanglement properties of quantum systems to explore multiple possibilities concurrently. The Sudoku puzzle must be mapped onto the available qubits and expressed within the device's connectivity constraints.[38] This mapping process often involves encoding additional constraints to enforce the rules of Sudoku and minimize the number of invalid or infeasible solutions. In this thesis we will examine the quantum annealing formulations of Sudoku more closely and examine the usage of QUBO Solvers for the solving of Sudoku Puzzles.

Chapter 3

Theory

1. QUBO Fundamentals

1.1 Ising Models

Ising models are mathematical tools used to study and describe the behavior of systems with binary variables, where each variable can take values of either $+1$ or -1 . These models find wide applications in various fields, such as statistical physics, material science, and optimization problems. The Ising model's foundation lies in the notion of energy and the concept of spin interactions. The energy of an Ising model is defined as a sum over pairs of interacting spins, where each spin is represented by a binary variable. The interactions are characterized by coupling coefficients, denoted as J_{ij} , indicating the strength of the interaction between spins i and j . The objective is to find the spin configurations that minimize the system's energy, leading to the ground state of the model.

The mathematical representation of an Ising model's energy is given by the Hamiltonian as seen in 3.1, where σ_i and σ_j are the binary variables representing the spins at positions i and j , respectively. The Ising model's behavior is extensively studied using techniques from statistical mechanics, such as the partition function and free energy. Additionally, Ising models can be employed to solve combinatorial optimization problems, where the binary variables correspond to the presence or absence of certain elements or decisions. By mapping the optimization problem to an Ising model, techniques from statistical physics, like simulated annealing and quantum annealing, can be utilized to find approximate solutions.

$$H(\sigma) = -J \sum_{\langle i, j \rangle} \sigma_i \sigma_j \quad (3.1)$$

1.2 QUBO Models

Quadratic Unconstrained Binary Optimization (QUBO) models are a generalization of Ising models, allowing for more flexibility in representing problems from various domains. QUBO models extend the Ising model by allowing binary variables to take values of 0 or 1 instead of just +1 or -1. This expansion facilitates the formulation of optimization problems with discrete variables, making QUBO a crucial framework for solving combinatorial optimization problems in fields like operations research, machine learning, and artificial intelligence.

The mathematical form of a QUBO model is represented by the objective function, which seeks to minimize a quadratic form over binary variables:

$$\text{QUBO: minimize/maximize } y = x^t Q x$$

where x is a vector of binary decision variables and Q is a square matrix of constants. The Q matrix formulation can be symmetric or in upper triangular form without losing generality. This can be achieved through simple transformations as follows:

- For symmetric: Replace $Q(i, j)$ with $(Q(i, j) + Q(j, i))/2$ for all i and j except $i = j$.
- For upper triangular: Replace $Q(i, j)$ with $Q(i, j) + Q(j, i)$ for all i and j with $j > i$. Then, set all $Q(i, j)$ for $j < i$ to 0. If the matrix is already symmetric, this process will double the $Q(i, j)$ values above the main diagonal and set all values below the main diagonal to 0.

In the examples presented in the following sections we will use the full, symmetric Q matrix instead of the upper triangular form. [19]

2. A QUBO Model for the Sudoku Puzzle

In order to formulate an appropriate QUBO model for the Sudoku Puzzle we first need to translate each rule of the puzzle into a constraint to be imposed on the objective function [41].

As discussed in 2., Sudoku consists of a very specific set of rules, which we will translate into constraints. We will need to introduce some additional constraints as to make the solution completely adhere to other, not specified limitations. These are as follows:

- Each cell in the grid can only contain a single number.

- The cells already full as part of the hint for the solution cannot be changed.

After creating the correct constraints as objective functions we will need to formulate them into QUBO models by adding sums for each index in the board, and substitute the equation part with $(x - 1)$ to properly represent the function in a matrix form that will be applied on a two dimensional matrix Q , with dimensions $N^3 \times N^3$.

It's important to note that as QUBO models are only able to process binary data we need a way to equate the possible number of Sudoku, usually 1..9 in binary form. In order to achieve this we shall define the three dimensional matrix $x_{i,j,k}$ where:

- i is the row index.
- j is the column index.
- k is the number index.

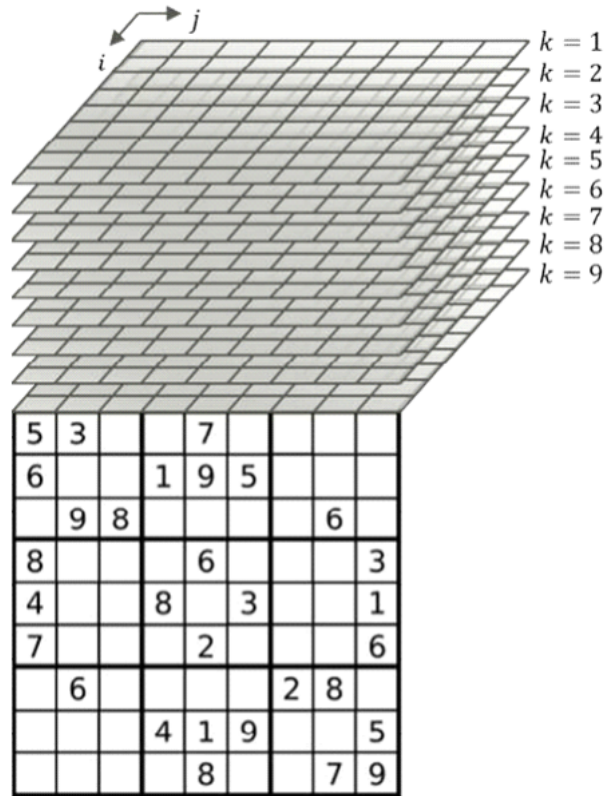


Figure 3.1: Visual representation of the binary sudoku matrix as seen in [41]

Each set of i, j, k indices in the x matrix corresponds to a set of i, j indices in the Q matrix. To calculate the corresponding sets we use the base- N representation.

Each sub matrix in x only contains binary values, with 1 symbolizing the appearance of the number in the k th matrix in the coordinates of the Sudoku matrix.

For a more illustrative example we present the matrices for Figure 3.1.

		1					
							1
		1					

$k = 1$

		1					
					1		

$k = 2$

1							
		1					
							1
				1			

$k = 3$

1							
		1					

$k = 4$

1							
				1			
							1

$k = 5$

1							
					1		
			1				
							1
	1						

$k = 6$

			1			
1						
						1

$k = 7$

		1				
1						
			1			
						1
			1			

$k = 8$

			1			
	1					
				1		
						1

$k = 9$

Thus, we are now able to properly formulate the objective functions for each needed constraint

2..1 Sudoku Rules as Constraint Matrices

As a 9×9 Sudoku would produce an immense 729×729 QUBO matrix, we will expand the equations using a smaller 4×4 instance. The constraint generation is unaffected by this change, as it is generic for any value of N .

The example Sudoku is shown in [3.2](#)

1	2		
		1	
		4	

Figure 3.2: 4×4 Sudoku example

And the binary Sudoku matrices for 3.2 are:

1			
		1	

$k = 1$

	1		

$k = 2$

$k = 3$

		1	

$k = 4$

Cell Singularity Constraint

In matrix x , the cell singularity constraint enforces that for every combination of i, j indices there can only be one non-zero value for all possible values of k . This can be achieved by depicting the constraint as:

$$\sum_{k=1}^N x_{ijk} = 1, \forall ij \in \{1..9\} \quad (3.2)$$

Due to k containing only binary values, the sum of all k matrices for each i, j is equal to the different appearances of 1 in k .

The QUBO model produced for this constraint is as follows:

$$\sum_{\substack{1 \leq i \leq N \\ N \leq j \leq N}} \left(\sum_{k=1}^N x_{i,j,k} - 1 \right)^2 \quad (3.3)$$

Where N is the maximum number accepted for the Sudoku instance. In our examples we will use a 4×4 Sudoku instance as seen in 3.2.

In order to extract the QUBO matrix from our constraint we need to take note of two extremely important factors.

Firstly, the fact that the expression 3.3 reduces into:

$$\sum_{\substack{1 \leq i \leq N \\ N \leq j \leq N}} (x_{i,j,1} + x_{i,j,2} + x_{i,j,3} + x_{i,j,4} - 1)^2 \quad (3.4)$$

Which, in turn, reduces into:

$$(x_{1,1,1} + x_{1,1,2} + x_{1,1,3} + x_{1,1,4} - 1)^2 + (x_{1,2,1} + x_{1,2,2} + x_{1,2,3} + x_{1,2,4} - 1)^2 + \quad (3.5)$$

$$(x_{1,3,1} + x_{1,3,2} + x_{1,3,3} + x_{1,3,4} - 1)^2 + (x_{1,4,1} + x_{1,4,2} + x_{1,4,3} + x_{1,4,4} - 1)^2 + \quad (3.6)$$

$$(x_{2,1,1} + x_{2,1,2} + x_{2,1,3} + x_{2,1,4} - 1)^2 + (x_{2,2,1} + x_{2,2,2} + x_{2,2,3} + x_{2,2,4} - 1)^2 + \quad (3.7)$$

$$(x_{2,3,1} + x_{2,3,2} + x_{2,3,3} + x_{2,3,4} - 1)^2 + (x_{2,4,1} + x_{2,4,2} + x_{2,4,3} + x_{2,4,4} - 1)^2 + \quad (3.8)$$

$$(x_{3,1,1} + x_{3,1,2} + x_{3,1,3} + x_{3,1,4} - 1)^2 + (x_{3,2,1} + x_{3,2,2} + x_{3,2,3} + x_{3,2,4} - 1)^2 + \quad (3.9)$$

$$(x_{3,3,1} + x_{3,3,2} + x_{3,3,3} + x_{3,3,4} - 1)^2 + (x_{3,4,1} + x_{3,4,2} + x_{3,4,3} + x_{3,4,4} - 1)^2 + \quad (3.10)$$

$$(x_{4,1,1} + x_{4,1,2} + x_{4,1,3} + x_{4,1,4} - 1)^2 + (x_{4,2,1} + x_{4,2,2} + x_{4,2,3} + x_{4,2,4} - 1)^2 + \quad (3.11)$$

$$(x_{4,3,1} + x_{4,3,2} + x_{4,3,3} + x_{4,3,4} - 1)^2 + (x_{4,4,1} + x_{4,4,2} + x_{4,4,3} + x_{4,4,4} - 1)^2 \quad (3.12)$$

As the expressions in the summed squares are almost identical we can simply examine a single one to identify the behaviour of all other squares. In the case of square $(x_{1,1,1} + x_{1,1,2} + x_{1,1,3} + x_{1,1,4} - 1)^2$ we can substitute it in the identity:

$$\begin{aligned} & (1 - a - b - c - d)^2 \\ &= 1 + a^2 + b^2 + c^2 + d^2 - 2a - 2b - 2c - 2d + 2ab + 2ac + 2ad + 2bc + 2bd + 2cd, \end{aligned} \quad (3.13)$$

Thus we reach the expression:

$$(x_{1,1,1} + x_{1,1,2} + x_{1,1,3} + x_{1,1,4} - 1)^2 = \quad (3.14)$$

$$1 + x_{1,1,1}^2 + x_{1,1,2}^2 + x_{1,1,3}^2 + x_{1,1,4}^2 - 2x_{1,1,1} - 2x_{1,1,2} - 2x_{1,1,3} - 2x_{1,1,4} \quad (3.15)$$

$$+ 2x_{1,1,1}x_{1,1,2} + 2x_{1,1,1}x_{1,1,3} + 2x_{1,1,1}x_{1,1,4} + 2x_{1,1,2}x_{1,1,3} + 2x_{1,1,2}x_{1,1,4} + 2x_{1,1,3}x_{1,1,4} \quad (3.16)$$

Secondly, as Ising and QUBO models only contain binary values (either $-1, 1$ for Ising or $0, 1$ for QUBO), we can further reduce the expression by applying the rule:

$$x_{i,j,k} = x_{i,j,k}^2 \quad (3.17)$$

Thus, the final form of $(x_{1,1,1} + x_{1,1,2} + x_{1,1,3} + x_{1,1,4} - 1)^2$ equals:

$$1 - x_{1,1,1}^2 - x_{1,1,2}^2 - x_{1,1,3}^2 - x_{1,1,4}^2 \quad (3.18)$$

$$+ 2x_{1,1,1}x_{1,1,2} + 2x_{1,1,1}x_{1,1,3} + 2x_{1,1,1}x_{1,1,4} + 2x_{1,1,2}x_{1,1,3} + 2x_{1,1,2}x_{1,1,4} + 2x_{1,1,3}x_{1,1,4} \quad (3.19)$$

By applying this conversion to the rest of our squared sums we get the following QUBO matrix:

$$\begin{aligned}
& -x_{1,1,1}^2 + 2x_{1,1,1}x_{1,1,2} + 2x_{1,1,1}x_{1,1,3} + 2x_{1,1,1}x_{1,1,4} - x_{1,1,2}^2 + 2x_{1,1,2}x_{1,1,3} + 2x_{1,1,2}x_{1,1,4} - \\
& x_{1,1,3}^2 + 2x_{1,1,3}x_{1,1,4} - x_{1,1,4}^2 - x_{1,2,1}^2 + 2x_{1,2,1}x_{1,2,2} + 2x_{1,2,1}x_{1,2,3} + 2x_{1,2,1}x_{1,2,4} - x_{1,2,2}^2 + 2x_{1,2,2}x_{1,2,3} + \\
& 2x_{1,2,2}x_{1,2,4} - x_{1,2,3}^2 + 2x_{1,2,3}x_{1,2,4} - x_{1,2,4}^2 - x_{1,3,1}^2 + 2x_{1,3,1}x_{1,3,2} + 2x_{1,3,1}x_{1,3,3} + 2x_{1,3,1}x_{1,3,4} - \\
& x_{1,3,2}^2 + 2x_{1,3,2}x_{1,3,3} + 2x_{1,3,2}x_{1,3,4} - x_{1,3,3}^2 + 2x_{1,3,3}x_{1,3,4} - x_{1,3,4}^2 - x_{1,4,1}^2 + 2x_{1,4,1}x_{1,4,2} + 2x_{1,4,1}x_{1,4,3} + \\
& 2x_{1,4,1}x_{1,4,4} - x_{1,4,2}^2 + 2x_{1,4,2}x_{1,4,3} + 2x_{1,4,2}x_{1,4,4} - x_{1,4,3}^2 + 2x_{1,4,3}x_{1,4,4} - x_{1,4,4}^2 - x_{2,1,1}^2 + 2x_{2,1,1}x_{2,1,2} + \\
& 2x_{2,1,1}x_{2,1,3} + 2x_{2,1,1}x_{2,1,4} - x_{2,1,2}^2 + 2x_{2,1,2}x_{2,1,3} + 2x_{2,1,2}x_{2,1,4} - x_{2,1,3}^2 + 2x_{2,1,3}x_{2,1,4} - x_{2,1,4}^2 - \\
& x_{2,2,1}^2 + 2x_{2,2,1}x_{2,2,2} + 2x_{2,2,1}x_{2,2,3} + 2x_{2,2,1}x_{2,2,4} - x_{2,2,2}^2 + 2x_{2,2,2}x_{2,2,3} + 2x_{2,2,2}x_{2,2,4} - x_{2,2,3}^2 + \\
& 2x_{2,2,3}x_{2,2,4} - x_{2,2,4}^2 - x_{2,3,1}^2 + 2x_{2,3,1}x_{2,3,2} + 2x_{2,3,1}x_{2,3,3} + 2x_{2,3,1}x_{2,3,4} - x_{2,3,2}^2 + 2x_{2,3,2}x_{2,3,3} + \\
& 2x_{2,3,2}x_{2,3,4} - x_{2,3,3}^2 + 2x_{2,3,3}x_{2,3,4} - x_{2,3,4}^2 - x_{2,4,1}^2 + 2x_{2,4,1}x_{2,4,2} + 2x_{2,4,1}x_{2,4,3} + 2x_{2,4,1}x_{2,4,4} - \\
& x_{2,4,2}^2 + 2x_{2,4,2}x_{2,4,3} + 2x_{2,4,2}x_{2,4,4} - x_{2,4,3}^2 + 2x_{2,4,3}x_{2,4,4} - x_{2,4,4}^2 - x_{3,1,1}^2 + 2x_{3,1,1}x_{3,1,2} + 2x_{3,1,1}x_{3,1,3} + \\
& 2x_{3,1,1}x_{3,1,4} - x_{3,1,2}^2 + 2x_{3,1,2}x_{3,1,3} + 2x_{3,1,2}x_{3,1,4} - x_{3,1,3}^2 + 2x_{3,1,3}x_{3,1,4} - x_{3,1,4}^2 - x_{3,2,1}^2 + 2x_{3,2,1}x_{3,2,2} + \\
& 2x_{3,2,1}x_{3,2,3} + 2x_{3,2,1}x_{3,2,4} - x_{3,2,2}^2 + 2x_{3,2,2}x_{3,2,3} + 2x_{3,2,2}x_{3,2,4} - x_{3,2,3}^2 + 2x_{3,2,3}x_{3,2,4} - x_{3,2,4}^2 - \\
& x_{3,3,1}^2 + 2x_{3,3,1}x_{3,3,2} + 2x_{3,3,1}x_{3,3,3} + 2x_{3,3,1}x_{3,3,4} - x_{3,3,2}^2 + 2x_{3,3,2}x_{3,3,3} + 2x_{3,3,2}x_{3,3,4} - x_{3,3,3}^2 + \\
& 2x_{3,3,3}x_{3,3,4} - x_{3,3,4}^2 - x_{3,4,1}^2 + 2x_{3,4,1}x_{3,4,2} + 2x_{3,4,1}x_{3,4,3} + 2x_{3,4,1}x_{3,4,4} - x_{3,4,2}^2 + 2x_{3,4,2}x_{3,4,3} + \\
& 2x_{3,4,2}x_{3,4,4} - x_{3,4,3}^2 + 2x_{3,4,3}x_{3,4,4} - x_{3,4,4}^2 - x_{4,1,1}^2 + 2x_{4,1,1}x_{4,1,2} + 2x_{4,1,1}x_{4,1,3} + 2x_{4,1,1}x_{4,1,4} - \\
& x_{4,1,2}^2 + 2x_{4,1,2}x_{4,1,3} + 2x_{4,1,2}x_{4,1,4} - x_{4,1,3}^2 + 2x_{4,1,3}x_{4,1,4} - x_{4,1,4}^2 - x_{4,2,1}^2 + 2x_{4,2,1}x_{4,2,2} + 2x_{4,2,1}x_{4,2,3} + \\
& 2x_{4,2,1}x_{4,2,4} - x_{4,2,2}^2 + 2x_{4,2,2}x_{4,2,3} + 2x_{4,2,2}x_{4,2,4} - x_{4,2,3}^2 + 2x_{4,2,3}x_{4,2,4} - x_{4,2,4}^2 - x_{4,3,1}^2 + 2x_{4,3,1}x_{4,3,2} + \\
& 2x_{4,3,1}x_{4,3,3} + 2x_{4,3,1}x_{4,3,4} - x_{4,3,2}^2 + 2x_{4,3,2}x_{4,3,3} + 2x_{4,3,2}x_{4,3,4} - x_{4,3,3}^2 + 2x_{4,3,3}x_{4,3,4} - x_{4,3,4}^2 - \\
& x_{4,4,1}^2 + 2x_{4,4,1}x_{4,4,2} + 2x_{4,4,1}x_{4,4,3} + 2x_{4,4,1}x_{4,4,4} - x_{4,4,2}^2 + 2x_{4,4,2}x_{4,4,3} + 2x_{4,4,2}x_{4,4,4} - x_{4,4,3}^2 + \\
& 2x_{4,4,3}x_{4,4,4} - x_{4,4,4}^2 + 16
\end{aligned}$$

Duplicates in Rows Constraint

In matrix x , the cell singularity constraint enforces that for every row can only be one non-zero value for all possible values of k . This can be achieved by depicting the constraint as:

$$\sum_{i=1}^N x_{ijk} = 1, \forall jk \in \{1..9\} \quad (3.20)$$

The QUBO model produced for this constraint is as follows:

$$\sum_{\substack{1 \leq i \leq N \\ 1 \leq k \leq N}} \left(\sum_{j=1}^N x_{i,j,k} - 1 \right)^2 \quad (3.21)$$

Where N is the maximum number accepted for the Sudoku instance.

By expanding the QUBO model we arrive at the following expression, which corresponds to the following matrix

$$\begin{aligned}
& -x_{1,1,1}^2 + 2x_{1,1,1}x_{1,2,1} + 2x_{1,1,1}x_{1,3,1} + 2x_{1,1,1}x_{1,4,1} - x_{1,1,2}^2 + 2x_{1,1,2}x_{1,2,2} + 2x_{1,1,2}x_{1,3,2} + \\
& 2x_{1,1,2}x_{1,4,2} - x_{1,1,3}^2 + 2x_{1,1,3}x_{1,2,3} + 2x_{1,1,3}x_{1,3,3} + 2x_{1,1,3}x_{1,4,3} - x_{1,1,4}^2 + 2x_{1,1,4}x_{1,2,4} + 2x_{1,1,4}x_{1,3,4} + \\
& 2x_{1,1,4}x_{1,4,4} - x_{1,2,1}^2 + 2x_{1,2,1}x_{1,3,1} + 2x_{1,2,1}x_{1,4,1} - x_{1,2,2}^2 + 2x_{1,2,2}x_{1,3,2} + 2x_{1,2,2}x_{1,4,2} - x_{1,2,3}^2 + \\
& 2x_{1,2,3}x_{1,3,3} + 2x_{1,2,3}x_{1,4,3} - x_{1,2,4}^2 + 2x_{1,2,4}x_{1,3,4} + 2x_{1,2,4}x_{1,4,4} - x_{1,3,1}^2 + 2x_{1,3,1}x_{1,4,1} - x_{1,3,2}^2 + \\
& 2x_{1,3,2}x_{1,4,2} - x_{1,3,3}^2 + 2x_{1,3,3}x_{1,4,3} - x_{1,3,4}^2 + 2x_{1,3,4}x_{1,4,4} - x_{1,4,1}^2 - x_{1,4,2}^2 - x_{1,4,3}^2 - x_{1,4,4}^2 - x_{2,1,1}^2 + \\
& 2x_{2,1,1}x_{2,2,1} + 2x_{2,1,1}x_{2,3,1} + 2x_{2,1,1}x_{2,4,1} - x_{2,1,2}^2 + 2x_{2,1,2}x_{2,2,2} + 2x_{2,1,2}x_{2,3,2} + 2x_{2,1,2}x_{2,4,2} - x_{2,1,3}^2 + \\
& 2x_{2,1,3}x_{2,2,3} + 2x_{2,1,3}x_{2,3,3} + 2x_{2,1,3}x_{2,4,3} - x_{2,1,4}^2 + 2x_{2,1,4}x_{2,2,4} + 2x_{2,1,4}x_{2,3,4} + 2x_{2,1,4}x_{2,4,4} - x_{2,2,1}^2 + \\
& 2x_{2,2,1}x_{2,3,1} + 2x_{2,2,1}x_{2,4,1} - x_{2,2,2}^2 + 2x_{2,2,2}x_{2,3,2} + 2x_{2,2,2}x_{2,4,2} - x_{2,2,3}^2 + 2x_{2,2,3}x_{2,3,3} + 2x_{2,2,3}x_{2,4,3} - \\
& x_{2,2,4}^2 + 2x_{2,2,4}x_{2,3,4} + 2x_{2,2,4}x_{2,4,4} - x_{2,3,1}^2 + 2x_{2,3,1}x_{2,4,1} - x_{2,3,2}^2 + 2x_{2,3,2}x_{2,4,2} - x_{2,3,3}^2 + 2x_{2,3,3}x_{2,4,3} - \\
& x_{2,3,4}^2 + 2x_{2,3,4}x_{2,4,4} - x_{2,4,1}^2 - x_{2,4,2}^2 - x_{2,4,3}^2 - x_{2,4,4}^2 - x_{3,1,1}^2 + 2x_{3,1,1}x_{3,2,1} + 2x_{3,1,1}x_{3,3,1} + 2x_{3,1,1}x_{3,4,1} - \\
& x_{3,1,2}^2 + 2x_{3,1,2}x_{3,2,2} + 2x_{3,1,2}x_{3,3,2} + 2x_{3,1,2}x_{3,4,2} - x_{3,1,3}^2 + 2x_{3,1,3}x_{3,2,3} + 2x_{3,1,3}x_{3,3,3} + 2x_{3,1,3}x_{3,4,3} - \\
& x_{3,1,4}^2 + 2x_{3,1,4}x_{3,2,4} + 2x_{3,1,4}x_{3,3,4} + 2x_{3,1,4}x_{3,4,4} - x_{3,2,1}^2 + 2x_{3,2,1}x_{3,3,1} + 2x_{3,2,1}x_{3,4,1} - x_{3,2,2}^2 + \\
& 2x_{3,2,2}x_{3,3,2} + 2x_{3,2,2}x_{3,4,2} - x_{3,2,3}^2 + 2x_{3,2,3}x_{3,3,3} + 2x_{3,2,3}x_{3,4,3} - x_{3,2,4}^2 + 2x_{3,2,4}x_{3,3,4} + 2x_{3,2,4}x_{3,4,4} - \\
& x_{3,3,1}^2 + 2x_{3,3,1}x_{3,4,1} - x_{3,3,2}^2 + 2x_{3,3,2}x_{3,4,2} - x_{3,3,3}^2 + 2x_{3,3,3}x_{3,4,3} - x_{3,3,4}^2 + 2x_{3,3,4}x_{3,4,4} - x_{3,4,1}^2 - \\
& x_{3,4,2}^2 - x_{3,4,3}^2 - x_{3,4,4}^2 - x_{4,1,1}^2 + 2x_{4,1,1}x_{4,2,1} + 2x_{4,1,1}x_{4,3,1} + 2x_{4,1,1}x_{4,4,1} - x_{4,1,2}^2 + 2x_{4,1,2}x_{4,2,2} + \\
& 2x_{4,1,2}x_{4,3,2} + 2x_{4,1,2}x_{4,4,2} - x_{4,1,3}^2 + 2x_{4,1,3}x_{4,2,3} + 2x_{4,1,3}x_{4,3,3} + 2x_{4,1,3}x_{4,4,3} - x_{4,1,4}^2 + 2x_{4,1,4}x_{4,2,4} + \\
& 2x_{4,1,4}x_{4,3,4} + 2x_{4,1,4}x_{4,4,4} - x_{4,2,1}^2 + 2x_{4,2,1}x_{4,3,1} + 2x_{4,2,1}x_{4,4,1} - x_{4,2,2}^2 + 2x_{4,2,2}x_{4,3,2} + 2x_{4,2,2}x_{4,4,2} - \\
& x_{4,2,3}^2 + 2x_{4,2,3}x_{4,3,3} + 2x_{4,2,3}x_{4,4,3} - x_{4,2,4}^2 + 2x_{4,2,4}x_{4,3,4} + 2x_{4,2,4}x_{4,4,4} - x_{4,3,1}^2 + 2x_{4,3,1}x_{4,4,1} - \\
& x_{4,3,2}^2 + 2x_{4,3,2}x_{4,4,2} - x_{4,3,3}^2 + 2x_{4,3,3}x_{4,4,3} - x_{4,3,4}^2 + 2x_{4,3,4}x_{4,4,4} - x_{4,4,1}^2 - x_{4,4,2}^2 - x_{4,4,3}^2 - x_{4,4,4}^2 + \\
& 16
\end{aligned}$$

Duplicates in Columns Constraint

In matrix x , the cell singularity constraint enforces that for every column can only be one non-zero value for all possible values of k . This can be achieved by depicting the constraint as:

$$\sum_{j=1}^N x_{ijk} = 1, \forall i, k \in \{1..9\} \quad (3.22)$$

The QUBO model produced for this constraint is as follows:

$$\sum_{\substack{1 \leq j \leq N \\ 1 \leq k \leq N}} \left(\sum_{i=1}^N x_{i,j,k} - 1 \right)^2 \quad (3.23)$$

Where N is the maximum number accepted for the Sudoku instance.

By expanding the QUBO model we arrive at the following expression, which corresponds to the following matrix

$$\begin{aligned}
& -x_{1,1,1}^2 + 2x_{1,1,1}x_{2,1,1} + 2x_{1,1,1}x_{3,1,1} + 2x_{1,1,1}x_{4,1,1} - x_{1,1,2}^2 + 2x_{1,1,2}x_{2,1,2} + 2x_{1,1,2}x_{3,1,2} + \\
& 2x_{1,1,2}x_{4,1,2} - x_{1,1,3}^2 + 2x_{1,1,3}x_{2,1,3} + 2x_{1,1,3}x_{3,1,3} + 2x_{1,1,3}x_{4,1,3} - x_{1,1,4}^2 + 2x_{1,1,4}x_{2,1,4} + 2x_{1,1,4}x_{3,1,4} + \\
& 2x_{1,1,4}x_{4,1,4} - x_{1,2,1}^2 + 2x_{1,2,1}x_{2,2,1} + 2x_{1,2,1}x_{3,2,1} + 2x_{1,2,1}x_{4,2,1} - x_{1,2,2}^2 + 2x_{1,2,2}x_{2,2,2} + 2x_{1,2,2}x_{3,2,2} + \\
& 2x_{1,2,2}x_{4,2,2} - x_{1,2,3}^2 + 2x_{1,2,3}x_{2,2,3} + 2x_{1,2,3}x_{3,2,3} + 2x_{1,2,3}x_{4,2,3} - x_{1,2,4}^2 + 2x_{1,2,4}x_{2,2,4} + 2x_{1,2,4}x_{3,2,4} + \\
& 2x_{1,2,4}x_{4,2,4} - x_{1,3,1}^2 + 2x_{1,3,1}x_{2,3,1} + 2x_{1,3,1}x_{3,3,1} + 2x_{1,3,1}x_{4,3,1} - x_{1,3,2}^2 + 2x_{1,3,2}x_{2,3,2} + 2x_{1,3,2}x_{3,3,2} + \\
& 2x_{1,3,2}x_{4,3,2} - x_{1,3,3}^2 + 2x_{1,3,3}x_{2,3,3} + 2x_{1,3,3}x_{3,3,3} + 2x_{1,3,3}x_{4,3,3} - x_{1,3,4}^2 + 2x_{1,3,4}x_{2,3,4} + 2x_{1,3,4}x_{3,3,4} + \\
& 2x_{1,3,4}x_{4,3,4} - x_{1,4,1}^2 + 2x_{1,4,1}x_{2,4,1} + 2x_{1,4,1}x_{3,4,1} + 2x_{1,4,1}x_{4,4,1} - x_{1,4,2}^2 + 2x_{1,4,2}x_{2,4,2} + 2x_{1,4,2}x_{3,4,2} + \\
& 2x_{1,4,2}x_{4,4,2} - x_{1,4,3}^2 + 2x_{1,4,3}x_{2,4,3} + 2x_{1,4,3}x_{3,4,3} + 2x_{1,4,3}x_{4,4,3} - x_{1,4,4}^2 + 2x_{1,4,4}x_{2,4,4} + 2x_{1,4,4}x_{3,4,4} + \\
& 2x_{1,4,4}x_{4,4,4} - x_{2,1,1}^2 + 2x_{2,1,1}x_{3,1,1} + 2x_{2,1,1}x_{4,1,1} - x_{2,1,2}^2 + 2x_{2,1,2}x_{3,1,2} + 2x_{2,1,2}x_{4,1,2} - x_{2,1,3}^2 + \\
& 2x_{2,1,3}x_{3,1,3} + 2x_{2,1,3}x_{4,1,3} - x_{2,1,4}^2 + 2x_{2,1,4}x_{3,1,4} + 2x_{2,1,4}x_{4,1,4} - x_{2,2,1}^2 + 2x_{2,2,1}x_{3,2,1} + 2x_{2,2,1}x_{4,2,1} - \\
& x_{2,2,2}^2 + 2x_{2,2,2}x_{3,2,2} + 2x_{2,2,2}x_{4,2,2} - x_{2,2,3}^2 + 2x_{2,2,3}x_{3,2,3} + 2x_{2,2,3}x_{4,2,3} - x_{2,2,4}^2 + 2x_{2,2,4}x_{3,2,4} + \\
& 2x_{2,2,4}x_{4,2,4} - x_{2,3,1}^2 + 2x_{2,3,1}x_{3,3,1} + 2x_{2,3,1}x_{4,3,1} - x_{2,3,2}^2 + 2x_{2,3,2}x_{3,3,2} + 2x_{2,3,2}x_{4,3,2} - x_{2,3,3}^2 + \\
& 2x_{2,3,3}x_{3,3,3} + 2x_{2,3,3}x_{4,3,3} - x_{2,3,4}^2 + 2x_{2,3,4}x_{3,3,4} + 2x_{2,3,4}x_{4,3,4} - x_{2,4,1}^2 + 2x_{2,4,1}x_{3,4,1} + 2x_{2,4,1}x_{4,4,1} - \\
& x_{2,4,2}^2 + 2x_{2,4,2}x_{3,4,2} + 2x_{2,4,2}x_{4,4,2} - x_{2,4,3}^2 + 2x_{2,4,3}x_{3,4,3} + 2x_{2,4,3}x_{4,4,3} - x_{2,4,4}^2 + 2x_{2,4,4}x_{3,4,4} + \\
& 2x_{2,4,4}x_{4,4,4} - x_{3,1,1}^2 + 2x_{3,1,1}x_{4,1,1} - x_{3,1,2}^2 + 2x_{3,1,2}x_{4,1,2} - x_{3,1,3}^2 + 2x_{3,1,3}x_{4,1,3} - x_{3,1,4}^2 + 2x_{3,1,4}x_{4,1,4} - \\
& x_{3,2,1}^2 + 2x_{3,2,1}x_{4,2,1} - x_{3,2,2}^2 + 2x_{3,2,2}x_{4,2,2} - x_{3,2,3}^2 + 2x_{3,2,3}x_{4,2,3} - x_{3,2,4}^2 + 2x_{3,2,4}x_{4,2,4} - x_{3,3,1}^2 + \\
& 2x_{3,3,1}x_{4,3,1} - x_{3,3,2}^2 + 2x_{3,3,2}x_{4,3,2} - x_{3,3,3}^2 + 2x_{3,3,3}x_{4,3,3} - x_{3,3,4}^2 + 2x_{3,3,4}x_{4,3,4} - x_{3,4,1}^2 + 2x_{3,4,1}x_{4,4,1} - \\
& x_{3,4,2}^2 + 2x_{3,4,2}x_{4,4,2} - x_{3,4,3}^2 + 2x_{3,4,3}x_{4,4,3} - x_{3,4,4}^2 + 2x_{3,4,4}x_{4,4,4} - x_{4,1,1}^2 - x_{4,1,2}^2 - x_{4,1,3}^2 - x_{4,1,4}^2 - \\
& x_{4,2,1}^2 - x_{4,2,2}^2 - x_{4,2,3}^2 - x_{4,2,4}^2 - x_{4,3,1}^2 - x_{4,3,2}^2 - x_{4,3,3}^2 - x_{4,3,4}^2 - x_{4,4,1}^2 - x_{4,4,2}^2 - x_{4,4,3}^2 - x_{4,4,4}^2 + 16
\end{aligned}$$

Duplicates in Subgrids Constraint

The constraint for duplicate numbers in subgrids is more complicated, and requires a new set of indices, u and v in order to "hop" between the subgrids and ensure that each constraint is applied correctly and only once. We offset each subgrid by $\sqrt{N}$ and their range is: offset..(offset + $\sqrt{N}$). This ensures that for every subgrid of i, j in for every k matrix the sum will be 1, meaning that no duplicates can exist.

$$\sum_{j=1}^{\sqrt{N}} \sum_{i=1}^{\sqrt{N}} x_{(i+u)(j+v)k} = 1, \quad \forall k \in K (K = \{1..9\}) \quad (3.24)$$

$$u, v \in \{0..N - \sqrt{N}\}, \quad \text{step} = \sqrt{N} \quad (3.25)$$

The QUBO model produced for this constraint is as follows:

$$\sum_{k=1}^N \left(\sum_{j=1}^{\sqrt{N}} \sum_{i=1}^{\sqrt{N}} x_{(i+v)(j+v)k} - 1 \right)^2 \quad (3.26)$$

$$u, v \in \{0..N - \sqrt{N}\}, \quad \text{step} = \sqrt{N} \quad (3.27)$$

Where N is the maximum number accepted for the Sudoku instance.

By expanding the QUBO model we arrive at the following expression, which corresponds to the following matrix

$$\begin{aligned} & -x_{1,1,1}^2 + 2x_{1,1,1}x_{1,2,1} + 2x_{1,1,1}x_{2,1,1} + 2x_{1,1,1}x_{2,2,1} - x_{1,1,2}^2 + 2x_{1,1,2}x_{1,2,2} + 2x_{1,1,2}x_{2,1,2} + \\ & 2x_{1,1,2}x_{2,2,2} - x_{1,1,3}^2 + 2x_{1,1,3}x_{1,2,3} + 2x_{1,1,3}x_{2,1,3} + 2x_{1,1,3}x_{2,2,3} - x_{1,1,4}^2 + 2x_{1,1,4}x_{1,2,4} + 2x_{1,1,4}x_{2,1,4} + \\ & 2x_{1,1,4}x_{2,2,4} - x_{1,2,1}^2 + 2x_{1,2,1}x_{2,1,1} + 2x_{1,2,1}x_{2,2,1} - x_{1,2,2}^2 + 2x_{1,2,2}x_{2,1,2} + 2x_{1,2,2}x_{2,2,2} - x_{1,2,3}^2 + \\ & 2x_{1,2,3}x_{2,1,3} + 2x_{1,2,3}x_{2,2,3} - x_{1,2,4}^2 + 2x_{1,2,4}x_{2,1,4} + 2x_{1,2,4}x_{2,2,4} - x_{1,3,1}^2 + 2x_{1,3,1}x_{1,4,1} + 2x_{1,3,1}x_{2,3,1} + \\ & 2x_{1,3,1}x_{2,4,1} - x_{1,3,2}^2 + 2x_{1,3,2}x_{1,4,2} + 2x_{1,3,2}x_{2,3,2} + 2x_{1,3,2}x_{2,4,2} - x_{1,3,3}^2 + 2x_{1,3,3}x_{1,4,3} + 2x_{1,3,3}x_{2,3,3} + \\ & 2x_{1,3,3}x_{2,4,3} - x_{1,3,4}^2 + 2x_{1,3,4}x_{1,4,4} + 2x_{1,3,4}x_{2,3,4} + 2x_{1,3,4}x_{2,4,4} - x_{1,4,1}^2 + 2x_{1,4,1}x_{2,3,1} + 2x_{1,4,1}x_{2,4,1} - \\ & x_{1,4,2}^2 + 2x_{1,4,2}x_{2,3,2} + 2x_{1,4,2}x_{2,4,2} - x_{1,4,3}^2 + 2x_{1,4,3}x_{2,3,3} + 2x_{1,4,3}x_{2,4,3} - x_{1,4,4}^2 + 2x_{1,4,4}x_{2,3,4} + \\ & 2x_{1,4,4}x_{2,4,4} - x_{2,1,1}^2 + 2x_{2,1,1}x_{2,2,1} - x_{2,1,2}^2 + 2x_{2,1,2}x_{2,2,2} - x_{2,1,3}^2 + 2x_{2,1,3}x_{2,2,3} - x_{2,1,4}^2 + 2x_{2,1,4}x_{2,2,4} - \\ & x_{2,2,1}^2 - x_{2,2,2}^2 - x_{2,2,3}^2 - x_{2,2,4}^2 - x_{2,3,1}^2 + 2x_{2,3,1}x_{2,4,1} - x_{2,3,2}^2 + 2x_{2,3,2}x_{2,4,2} - x_{2,3,3}^2 + 2x_{2,3,3}x_{2,4,3} - \\ & x_{2,3,4}^2 + 2x_{2,3,4}x_{2,4,4} - x_{2,4,1}^2 - x_{2,4,2}^2 - x_{2,4,3}^2 - x_{2,4,4}^2 - x_{3,1,1}^2 + 2x_{3,1,1}x_{3,2,1} + 2x_{3,1,1}x_{4,1,1} + 2x_{3,1,1}x_{4,2,1} - \\ & x_{3,1,2}^2 + 2x_{3,1,2}x_{3,2,2} + 2x_{3,1,2}x_{4,1,2} + 2x_{3,1,2}x_{4,2,2} - x_{3,1,3}^2 + 2x_{3,1,3}x_{3,2,3} + 2x_{3,1,3}x_{4,1,3} + 2x_{3,1,3}x_{4,2,3} - \\ & x_{3,1,4}^2 + 2x_{3,1,4}x_{3,2,4} + 2x_{3,1,4}x_{4,1,4} + 2x_{3,1,4}x_{4,2,4} - x_{3,2,1}^2 + 2x_{3,2,1}x_{4,1,1} + 2x_{3,2,1}x_{4,2,1} - x_{3,2,2}^2 + \\ & 2x_{3,2,2}x_{4,1,2} + 2x_{3,2,2}x_{4,2,2} - x_{3,2,3}^2 + 2x_{3,2,3}x_{4,1,3} + 2x_{3,2,3}x_{4,2,3} - x_{3,2,4}^2 + 2x_{3,2,4}x_{4,1,4} + 2x_{3,2,4}x_{4,2,4} - \\ & x_{3,3,1}^2 + 2x_{3,3,1}x_{3,4,1} + 2x_{3,3,1}x_{4,3,1} + 2x_{3,3,1}x_{4,4,1} - x_{3,3,2}^2 + 2x_{3,3,2}x_{3,4,2} + 2x_{3,3,2}x_{4,3,2} + 2x_{3,3,2}x_{4,4,2} - \\ & x_{3,3,3}^2 + 2x_{3,3,3}x_{3,4,3} + 2x_{3,3,3}x_{4,3,3} + 2x_{3,3,3}x_{4,4,3} - x_{3,3,4}^2 + 2x_{3,3,4}x_{3,4,4} + 2x_{3,3,4}x_{4,3,4} + 2x_{3,3,4}x_{4,4,4} - \\ & x_{3,4,1}^2 + 2x_{3,4,1}x_{3,4,1} + 2x_{3,4,1}x_{4,3,1} + 2x_{3,4,1}x_{4,4,1} - x_{3,4,2}^2 + 2x_{3,4,2}x_{3,4,2} + 2x_{3,4,2}x_{4,3,2} + 2x_{3,4,2}x_{4,4,2} - x_{3,4,3}^2 + 2x_{3,4,3}x_{3,4,3} + \\ & 2x_{3,4,3}x_{4,3,3} - x_{3,4,4}^2 + 2x_{3,4,4}x_{3,4,4} + 2x_{3,4,4}x_{4,3,4} + 2x_{3,4,4}x_{4,4,4} - x_{4,1,1}^2 + 2x_{4,1,1}x_{4,2,1} - x_{4,1,2}^2 + 2x_{4,1,2}x_{4,2,2} - \\ & x_{4,1,3}^2 + 2x_{4,1,3}x_{4,2,3} - x_{4,1,4}^2 + 2x_{4,1,4}x_{4,2,4} - x_{4,2,1}^2 - x_{4,2,2}^2 - x_{4,2,3}^2 - x_{4,2,4}^2 - x_{4,3,1}^2 + 2x_{4,3,1}x_{4,4,1} - \\ & x_{4,3,2}^2 + 2x_{4,3,2}x_{4,4,2} - x_{4,3,3}^2 + 2x_{4,3,3}x_{4,4,3} - x_{4,3,4}^2 + 2x_{4,3,4}x_{4,4,4} - x_{4,4,1}^2 - x_{4,4,2}^2 - x_{4,4,3}^2 - x_{4,4,4}^2 + \\ & 16 \end{aligned}$$

Enforcing Hint Compliance Constraint

The hint compliance constraint is the only one that needs an incomplete Sudoku board to implement, while the rest only needed the dimensions of the Sudoku. The hint constraint, contrary to the others, is not a negative one. Instead of the solver avoiding a solution we want to enforce, at least partly, a specific solution or the ones that fit our requirements. In this case we want the solver to adhere to the instance's hint. In order to do that we have to make that solution more "appealing". Thus, the following constraint is implemented:

$$x_{ijk} = 1, \quad \forall ijk \in \text{hint} \quad (3.28)$$

The QUBO model produced for this constraint is as follows:

$$\sum_{\text{hint}} (1 - x_{ijk}) \quad (3.29)$$

Using the example in 3.2 we get a set of ijk indices with each hint.

$$i = 1, j = 1, k = 1$$

$$i = 1, j = 1, k = 2$$

$$i = 3, j = 3, k = 1$$

$$i = 4, j = 1, k = 4$$

Substituting all the indices with the hint we get the following matrix:

$$x_{1,1,1}^2 + x_{1,1,2}^2 + x_{1,3,1}^2 + x_{4,1,4}^2$$

Final QUBO formulation

The final QUBO matrix is the result of the matrix addition of all previous QUBO matrices:

$$Q = \sum_{\substack{1 \leq i \leq N \\ N \leq j \leq N}} \left(\sum_{k=1}^N x_{i,j,k} - 1 \right)^2 + \sum_{\substack{1 \leq i \leq N \\ 1 \leq k \leq N}} \left(\sum_{j=1}^N x_{i,j,k} - 1 \right)^2 + \sum_{\substack{1 \leq j \leq N \\ 1 \leq k \leq N}} \left(\sum_{i=1}^N x_{i,j,k} - 1 \right)^2 + \quad (3.30)$$

$$\sum_{k=1}^N \left(\sum_{j=1}^{\sqrt{N}} \sum_{i=1}^{\sqrt{N}} x_{(i+v)(j+v)k} - 1 \right)^2 + \sum_{\text{hint}} (1 - x_{ijk}) \quad (3.31)$$

Where N is the maximum number accepted for the Sudoku instance.

By expanding the QUBO model we arrive at the following expression, which corresponds to the following matrix

$$\begin{aligned} & -4x_{1,1,1}^2 + 2x_{1,1,1}x_{1,1,2} + 2x_{1,1,1}x_{1,1,3} + 2x_{1,1,1}x_{1,1,4} + 4x_{1,1,1}x_{1,2,1} + 2x_{1,1,1}x_{1,3,1} + 2x_{1,1,1}x_{1,4,1} + \\ & 4x_{1,1,1}x_{2,1,1} + 2x_{1,1,1}x_{2,2,1} + 2x_{1,1,1}x_{3,1,1} + 2x_{1,1,1}x_{4,1,1} - 4x_{1,1,2}^2 + 2x_{1,1,2}x_{1,1,3} + 2x_{1,1,2}x_{1,1,4} + \\ & 4x_{1,1,2}x_{1,2,2} + 2x_{1,1,2}x_{1,3,2} + 2x_{1,1,2}x_{1,4,2} + 4x_{1,1,2}x_{2,1,2} + 2x_{1,1,2}x_{2,2,2} + 2x_{1,1,2}x_{3,1,2} + 2x_{1,1,2}x_{4,1,2} - \\ & 4x_{1,1,3}^2 + 2x_{1,1,3}x_{1,1,4} + 4x_{1,1,3}x_{1,2,3} + 2x_{1,1,3}x_{1,3,3} + 2x_{1,1,3}x_{1,4,3} + 4x_{1,1,3}x_{2,1,3} + 2x_{1,1,3}x_{2,2,3} + \\ & 2x_{1,1,3}x_{3,1,3} + 2x_{1,1,3}x_{4,1,3} - 4x_{1,1,4}^2 + 4x_{1,1,4}x_{1,2,4} + 2x_{1,1,4}x_{1,3,4} + 2x_{1,1,4}x_{1,4,4} + 4x_{1,1,4}x_{2,1,4} + \end{aligned}$$

$$\begin{aligned}
& 2x_{1,1,4}x_{2,2,4} + 2x_{1,1,4}x_{3,1,4} + 2x_{1,1,4}x_{4,1,4} - 4x_{1,2,1}^2 + 2x_{1,2,1}x_{1,2,2} + 2x_{1,2,1}x_{1,2,3} + 2x_{1,2,1}x_{1,2,4} + \\
& 2x_{1,2,1}x_{1,3,1} + 2x_{1,2,1}x_{1,4,1} + 2x_{1,2,1}x_{2,1,1} + 4x_{1,2,1}x_{2,2,1} + 2x_{1,2,1}x_{3,2,1} + 2x_{1,2,1}x_{4,2,1} - 4x_{1,2,2}^2 + \\
& 2x_{1,2,2}x_{1,2,3} + 2x_{1,2,2}x_{1,2,4} + 2x_{1,2,2}x_{1,3,2} + 2x_{1,2,2}x_{1,4,2} + 2x_{1,2,2}x_{2,1,2} + 4x_{1,2,2}x_{2,2,2} + 2x_{1,2,2}x_{3,2,2} + \\
& 2x_{1,2,2}x_{4,2,2} - 4x_{1,2,3}^2 + 2x_{1,2,3}x_{1,2,4} + 2x_{1,2,3}x_{1,3,3} + 2x_{1,2,3}x_{1,4,3} + 2x_{1,2,3}x_{2,1,3} + 4x_{1,2,3}x_{2,2,3} + \\
& 2x_{1,2,3}x_{3,2,3} + 2x_{1,2,3}x_{4,2,3} - 4x_{1,2,4}^2 + 2x_{1,2,4}x_{1,3,4} + 2x_{1,2,4}x_{1,4,4} + 2x_{1,2,4}x_{2,1,4} + 4x_{1,2,4}x_{2,2,4} + \\
& 2x_{1,2,4}x_{3,2,4} + 2x_{1,2,4}x_{4,2,4} - 4x_{1,3,1}^2 + 2x_{1,3,1}x_{1,3,2} + 2x_{1,3,1}x_{1,3,3} + 2x_{1,3,1}x_{1,3,4} + 4x_{1,3,1}x_{1,4,1} + \\
& 4x_{1,3,1}x_{2,3,1} + 2x_{1,3,1}x_{2,4,1} + 2x_{1,3,1}x_{3,3,1} + 2x_{1,3,1}x_{4,3,1} - 4x_{1,3,2}^2 + 2x_{1,3,2}x_{1,3,3} + 2x_{1,3,2}x_{1,3,4} + \\
& 4x_{1,3,2}x_{1,4,2} + 4x_{1,3,2}x_{2,3,2} + 2x_{1,3,2}x_{2,4,2} + 2x_{1,3,2}x_{3,3,2} + 2x_{1,3,2}x_{4,3,2} - 4x_{1,3,3}^2 + 2x_{1,3,3}x_{1,3,4} + \\
& 4x_{1,3,3}x_{1,4,3} + 4x_{1,3,3}x_{2,3,3} + 2x_{1,3,3}x_{2,4,3} + 2x_{1,3,3}x_{3,3,3} + 2x_{1,3,3}x_{4,3,3} - 4x_{1,3,4}^2 + 4x_{1,3,4}x_{1,4,4} + \\
& 4x_{1,3,4}x_{2,3,4} + 2x_{1,3,4}x_{2,4,4} + 2x_{1,3,4}x_{3,3,4} + 2x_{1,3,4}x_{4,3,4} - 4x_{1,4,1}^2 + 2x_{1,4,1}x_{1,4,2} + 2x_{1,4,1}x_{1,4,3} + \\
& 2x_{1,4,1}x_{1,4,4} + 2x_{1,4,1}x_{2,3,1} + 4x_{1,4,1}x_{2,4,1} + 2x_{1,4,1}x_{3,4,1} + 2x_{1,4,1}x_{4,4,1} - 4x_{1,4,2}^2 + 2x_{1,4,2}x_{1,4,3} + \\
& 2x_{1,4,2}x_{1,4,4} + 2x_{1,4,2}x_{2,3,2} + 4x_{1,4,2}x_{2,4,2} + 2x_{1,4,2}x_{3,4,2} + 2x_{1,4,2}x_{4,4,2} - 4x_{1,4,3}^2 + 2x_{1,4,3}x_{1,4,4} + \\
& 2x_{1,4,3}x_{2,3,3} + 4x_{1,4,3}x_{2,4,3} + 2x_{1,4,3}x_{3,4,3} + 2x_{1,4,3}x_{4,4,3} - 4x_{1,4,4}^2 + 2x_{1,4,4}x_{2,3,4} + 4x_{1,4,4}x_{2,4,4} + \\
& 2x_{1,4,4}x_{3,4,4} + 2x_{1,4,4}x_{4,4,4} - 4x_{2,1,1}^2 + 2x_{2,1,1}x_{2,1,2} + 2x_{2,1,1}x_{2,1,3} + 2x_{2,1,1}x_{2,1,4} + 4x_{2,1,1}x_{2,2,1} + \\
& 2x_{2,1,1}x_{2,3,1} + 2x_{2,1,1}x_{2,4,1} + 2x_{2,1,1}x_{3,1,1} + 2x_{2,1,1}x_{4,1,1} - 4x_{2,1,2}^2 + 2x_{2,1,2}x_{2,1,3} + 2x_{2,1,2}x_{2,1,4} + \\
& 4x_{2,1,2}x_{2,2,2} + 2x_{2,1,2}x_{2,3,2} + 2x_{2,1,2}x_{2,4,2} + 2x_{2,1,2}x_{3,1,2} + 2x_{2,1,2}x_{4,1,2} - 4x_{2,1,3}^2 + 2x_{2,1,3}x_{2,1,4} + \\
& 4x_{2,1,3}x_{2,2,3} + 2x_{2,1,3}x_{2,3,3} + 2x_{2,1,3}x_{2,4,3} + 2x_{2,1,3}x_{3,1,3} + 2x_{2,1,3}x_{4,1,3} - 4x_{2,1,4}^2 + 4x_{2,1,4}x_{2,2,4} + \\
& 2x_{2,1,4}x_{2,3,4} + 2x_{2,1,4}x_{2,4,4} + 2x_{2,1,4}x_{3,1,4} + 2x_{2,1,4}x_{4,1,4} - 4x_{2,2,1}^2 + 2x_{2,2,1}x_{2,2,2} + 2x_{2,2,1}x_{2,2,3} + \\
& 2x_{2,2,1}x_{2,2,4} + 2x_{2,2,1}x_{2,3,1} + 2x_{2,2,1}x_{2,4,1} + 2x_{2,2,1}x_{3,2,1} + 2x_{2,2,1}x_{4,2,1} - 4x_{2,2,2}^2 + 2x_{2,2,2}x_{2,2,3} + \\
& 2x_{2,2,2}x_{2,2,4} + 2x_{2,2,2}x_{2,3,2} + 2x_{2,2,2}x_{2,4,2} + 2x_{2,2,2}x_{3,2,2} + 2x_{2,2,2}x_{4,2,2} - 4x_{2,2,3}^2 + 2x_{2,2,3}x_{2,2,4} + \\
& 2x_{2,2,3}x_{2,3,3} + 2x_{2,2,3}x_{2,4,3} + 2x_{2,2,3}x_{3,2,3} + 2x_{2,2,3}x_{4,2,3} - 4x_{2,2,4}^2 + 2x_{2,2,4}x_{2,3,4} + 2x_{2,2,4}x_{2,4,4} + \\
& 2x_{2,2,4}x_{3,2,4} + 2x_{2,2,4}x_{4,2,4} - 4x_{2,3,1}^2 + 2x_{2,3,1}x_{2,3,2} + 2x_{2,3,1}x_{2,3,3} + 2x_{2,3,1}x_{2,3,4} + 4x_{2,3,1}x_{2,4,1} + \\
& 2x_{2,3,1}x_{3,3,1} + 2x_{2,3,1}x_{4,3,1} - 4x_{2,3,2}^2 + 2x_{2,3,2}x_{2,3,3} + 2x_{2,3,2}x_{2,3,4} + 4x_{2,3,2}x_{2,4,2} + 2x_{2,3,2}x_{3,3,2} + \\
& 2x_{2,3,2}x_{4,3,2} - 4x_{2,3,3}^2 + 2x_{2,3,3}x_{2,3,4} + 4x_{2,3,3}x_{2,4,3} + 2x_{2,3,3}x_{3,3,3} + 2x_{2,3,3}x_{4,3,3} - 4x_{2,3,4}^2 + 4x_{2,3,4}x_{2,4,4} + \\
& 2x_{2,3,4}x_{3,3,4} + 2x_{2,3,4}x_{4,3,4} - 4x_{2,4,1}^2 + 2x_{2,4,1}x_{2,4,2} + 2x_{2,4,1}x_{2,4,3} + 2x_{2,4,1}x_{2,4,4} + 2x_{2,4,1}x_{3,4,1} + \\
& 2x_{2,4,1}x_{4,4,1} - 4x_{2,4,2}^2 + 2x_{2,4,2}x_{2,4,3} + 2x_{2,4,2}x_{2,4,4} + 2x_{2,4,2}x_{3,4,2} + 2x_{2,4,2}x_{4,4,2} - 4x_{2,4,3}^2 + 2x_{2,4,3}x_{2,4,4} + \\
& 2x_{2,4,3}x_{3,4,3} + 2x_{2,4,3}x_{4,4,3} - 4x_{2,4,4}^2 + 2x_{2,4,4}x_{3,4,4} + 2x_{2,4,4}x_{4,4,4} - 4x_{3,1,1}^2 + 2x_{3,1,1}x_{3,1,2} + 2x_{3,1,1}x_{3,1,3} + \\
& 2x_{3,1,1}x_{3,1,4} + 4x_{3,1,1}x_{3,2,1} + 2x_{3,1,1}x_{3,3,1} + 2x_{3,1,1}x_{3,4,1} + 4x_{3,1,1}x_{4,1,1} + 2x_{3,1,1}x_{4,2,1} - 4x_{3,1,2}^2 + \\
& 2x_{3,1,2}x_{3,1,3} + 2x_{3,1,2}x_{3,1,4} + 4x_{3,1,2}x_{3,2,2} + 2x_{3,1,2}x_{3,3,2} + 2x_{3,1,2}x_{3,4,2} + 4x_{3,1,2}x_{4,1,2} + 2x_{3,1,2}x_{4,2,2} - \\
& 4x_{3,1,3}^2 + 2x_{3,1,3}x_{3,1,4} + 4x_{3,1,3}x_{3,2,3} + 2x_{3,1,3}x_{3,3,3} + 2x_{3,1,3}x_{3,4,3} + 4x_{3,1,3}x_{4,1,3} + 2x_{3,1,3}x_{4,2,3} - \\
& 4x_{3,1,4}^2 + 4x_{3,1,4}x_{3,2,4} + 2x_{3,1,4}x_{3,3,4} + 2x_{3,1,4}x_{3,4,4} + 4x_{3,1,4}x_{4,1,4} + 2x_{3,1,4}x_{4,2,4} - 4x_{3,2,1}^2 + 2x_{3,2,1}x_{3,2,2} + \\
& 2x_{3,2,1}x_{3,2,3} + 2x_{3,2,1}x_{3,2,4} + 2x_{3,2,1}x_{3,3,1} + 2x_{3,2,1}x_{3,4,1} + 2x_{3,2,1}x_{4,1,1} + 4x_{3,2,1}x_{4,2,1} - 4x_{3,2,2}^2 + \\
& 2x_{3,2,2}x_{3,2,3} + 2x_{3,2,2}x_{3,2,4} + 2x_{3,2,2}x_{3,3,2} + 2x_{3,2,2}x_{3,4,2} + 2x_{3,2,2}x_{4,1,2} + 4x_{3,2,2}x_{4,2,2} - 4x_{3,2,3}^2 + \\
& 2x_{3,2,3}x_{3,2,4} + 2x_{3,2,3}x_{3,3,3} + 2x_{3,2,3}x_{3,4,3} + 2x_{3,2,3}x_{4,1,3} + 4x_{3,2,3}x_{4,2,3} - 4x_{3,2,4}^2 + 2x_{3,2,4}x_{3,3,4} + \\
& 2x_{3,2,4}x_{3,4,4} + 2x_{3,2,4}x_{4,1,4} + 4x_{3,2,4}x_{4,2,4} - 4x_{3,3,1}^2 + 2x_{3,3,1}x_{3,3,2} + 2x_{3,3,1}x_{3,3,3} + 2x_{3,3,1}x_{3,3,4} + \\
& 4x_{3,3,1}x_{3,4,1} + 4x_{3,3,1}x_{4,1,1} + 2x_{3,3,1}x_{4,2,1} - 4x_{3,3,2}^2 + 2x_{3,3,2}x_{3,3,3} + 2x_{3,3,2}x_{3,3,4} + 4x_{3,3,2}x_{3,4,2} + \\
& 4x_{3,3,2}x_{4,1,2} + 2x_{3,3,2}x_{4,2,2} - 4x_{3,3,3}^2 + 2x_{3,3,3}x_{3,3,4} + 4x_{3,3,3}x_{3,4,3} + 4x_{3,3,3}x_{4,1,3} + 2x_{3,3,3}x_{4,2,3} - \\
& 4x_{3,3,4}^2 + 4x_{3,3,4}x_{3,4,4} + 4x_{3,3,4}x_{4,1,4} + 2x_{3,3,4}x_{4,2,4} - 4x_{3,4,1}^2 + 2x_{3,4,1}x_{3,4,2} + 2x_{3,4,1}x_{3,4,3} + 2x_{3,4,1}x_{3,4,4} + \\
& 2x_{3,4,1}x_{4,1,1} + 4x_{3,4,1}x_{4,2,1} - 4x_{3,4,2}^2 + 2x_{3,4,2}x_{3,4,3} + 2x_{3,4,2}x_{3,4,4} + 2x_{3,4,2}x_{4,1,2} + 4x_{3,4,2}x_{4,2,2} - \\
& 4x_{3,4,3}^2 + 2x_{3,4,3}x_{3,4,4} + 2x_{3,4,3}x_{4,1,3} + 4x_{3,4,3}x_{4,2,3} - 4x_{3,4,4}^2 + 2x_{3,4,4}x_{4,1,4} + 4x_{3,4,4}x_{4,2,4} - 4x_{4,1,1}^2 + \\
& 2x_{4,1,1}x_{4,1,2} + 2x_{4,1,1}x_{4,1,3} + 2x_{4,1,1}x_{4,1,4} + 4x_{4,1,1}x_{4,2,1} + 2x_{4,1,1}x_{4,3,1} + 2x_{4,1,1}x_{4,4,1} - 4x_{4,1,2}^2 + \\
& 2x_{4,1,2}x_{4,1,3} + 2x_{4,1,2}x_{4,1,4} + 4x_{4,1,2}x_{4,2,2} + 2x_{4,1,2}x_{4,3,2} + 2x_{4,1,2}x_{4,4,2} - 4x_{4,1,3}^2 + 2x_{4,1,3}x_{4,1,4} +
\end{aligned}$$

$$\begin{aligned}
& 4x_{4,1,3}x_{4,2,3} + 2x_{4,1,3}x_{4,3,3} + 2x_{4,1,3}x_{4,4,3} - 4x_{4,1,4}^2 + 4x_{4,1,4}x_{4,2,4} + 2x_{4,1,4}x_{4,3,4} + 2x_{4,1,4}x_{4,4,4} - \\
& 4x_{4,2,1}^2 + 2x_{4,2,1}x_{4,2,2} + 2x_{4,2,1}x_{4,2,3} + 2x_{4,2,1}x_{4,2,4} + 2x_{4,2,1}x_{4,3,1} + 2x_{4,2,1}x_{4,4,1} - 4x_{4,2,2}^2 + 2x_{4,2,2}x_{4,2,3} + \\
& 2x_{4,2,2}x_{4,2,4} + 2x_{4,2,2}x_{4,3,2} + 2x_{4,2,2}x_{4,4,2} - 4x_{4,2,3}^2 + 2x_{4,2,3}x_{4,2,4} + 2x_{4,2,3}x_{4,3,3} + 2x_{4,2,3}x_{4,4,3} - \\
& 4x_{4,2,4}^2 + 2x_{4,2,4}x_{4,3,4} + 2x_{4,2,4}x_{4,4,4} - 4x_{4,3,1}^2 + 2x_{4,3,1}x_{4,3,2} + 2x_{4,3,1}x_{4,3,3} + 2x_{4,3,1}x_{4,3,4} + 4x_{4,3,1}x_{4,4,1} - \\
& 4x_{4,3,2}^2 + 2x_{4,3,2}x_{4,3,3} + 2x_{4,3,2}x_{4,3,4} + 4x_{4,3,2}x_{4,4,2} - 4x_{4,3,3}^2 + 2x_{4,3,3}x_{4,3,4} + 4x_{4,3,3}x_{4,4,3} - 4x_{4,3,4}^2 + \\
& 4x_{4,3,4}x_{4,4,4} - 4x_{4,4,1}^2 + 2x_{4,4,1}x_{4,4,2} + 2x_{4,4,1}x_{4,4,3} + 2x_{4,4,1}x_{4,4,4} - 4x_{4,4,2}^2 + 2x_{4,4,2}x_{4,4,3} + 2x_{4,4,2}x_{4,4,4} - \\
& 4x_{4,4,3}^2 + 2x_{4,4,3}x_{4,4,4} - 4x_{4,4,4}^2 + 64
\end{aligned}$$

As the total QUBO produces a 64×64 matrix visualizing it is quite challenging to produce a clear visualization. Thus we opted to translate each parameter in the QUBO model to a color, and recreate the QUBO matrix as an image, with each color corresponding to a certain number, as seen in Table 3.1 and shown in Figure 3.3

Number	Color
-4	Blue
0	Black
2	Yellow
4	Red

Table 3.1: Number to color association in 3.3

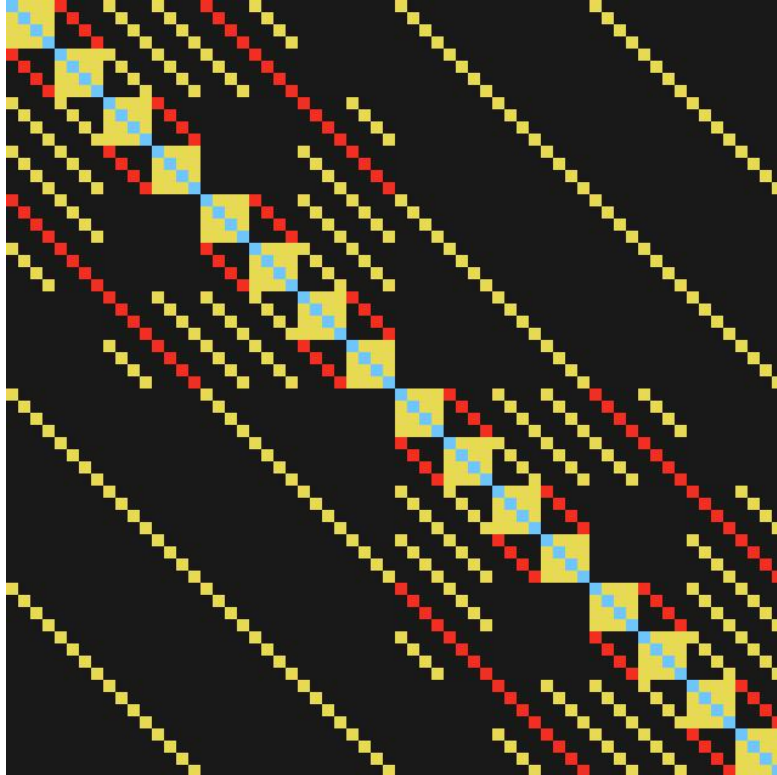


Figure 3.3: Visual Representation of the QUBO matrix for a 4×4 Sudoku

3. Solution Interpretation

Interpreting the solutions provided by D-Wave’s solvers may prove a daunting task, as we need a grounded understanding of our own QUBO model for the problem first. As we discussed in 1.3, the diagonal of the QUBO matrix contains the coefficients for each variable in the objective function, and the non-diagonal elements encapsulate the coefficients for the quadratic variables. The solutions provided by D-Wave’s solvers are always the same size of the diagonal, and contain the value for each of the objective function’s variables [42].

In our specific model, the solution string demonstrates the placement of each number in the Sudoku instance. For a more comprehensive explanation we will showcase an example solution and inspect it, and examine how each specific component correlates to a Sudoku board cell. In 3.32 we showcase a solution collected by D-Wave’s solvers for a 4×4 instance.

$$1000010000100001001000010100100000010010100001000100100000010010 \quad (3.32)$$

Now, let us delve deeper in the solution, and investigate each part. Firstly, we ought to note that the position of each element in the solution signifies it’s index, with the starting index being 0. To successfully extract the information from the solution we need to separate the linear solution index into three distinct indices, using the following algorithm, with the assumption that N is the number of our Sudoku instance’s digits:

Algorithm 1 Decoding Linear Index into Multi-variable Index

```

for  $i = 1$  to  $N$  do
  for  $j = 1$  to  $N$  do
    for  $k = 1$  to  $N$  do
      if  $(i * N^2) + (j * N) + k = index$  then
        return  $i, j, k$ 
      end
    end
  end
end
end

```

By applying this algorithm to all indices in our solution string we can ascertain that each element corresponds to a specific cell, indicated by the i and j indices, with k signifying the number it represents. Note that the range of the indices is $0 - (n - 1)$ thus, when handling them as multivariable indices we will not change their range, however, in the case of identifying the number they represent (shown by k) we will assert that their range will be $1 - n$.

Table 3.2 provides the first 8 digits in solution 3.32, alongside their corresponding multivariable indices.

Index	0	1	2	3	4	5	6	7
3D Index	[0, 0, 0]	[0, 0, 1]	[0, 0, 2]	[0, 0, 3]	[0, 1, 0]	[0, 1, 1]	[0, 1, 2]	[0, 1, 3]
Value	1	0	0	0	0	1	0	0

Table 3.2: Linear Indexes and their three-dimensional equivalents.

Visualizing the solution using our auxiliary matrices we get the following binary representation of the Sudoku in our solution string:

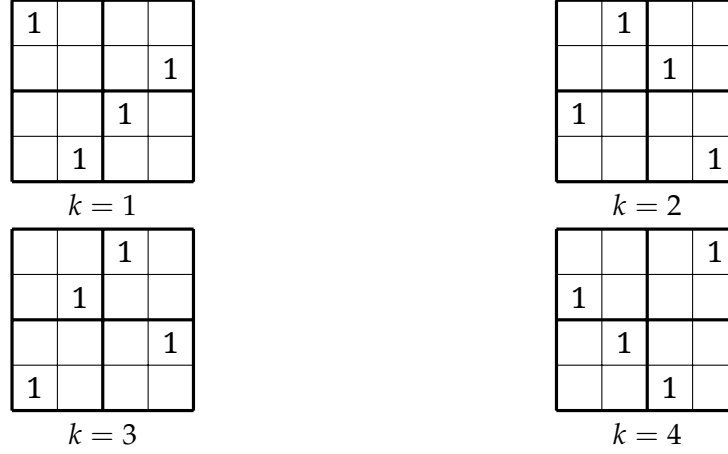


Figure 3.4

Finally, in order to get the final Sudoku we need to multiply each matrix by its respective k and sum them to our final Sudoku, presented below:

1	2	3	4
4	3	2	1
2	4	1	3
3	1	4	2

Figure 3.5: 4×4 Sudoku example solved using Quantum Annealing.

Chapter 4

Experiments

1. Experiment Sets

In pursuit of acquiring comprehensive benchmarking data pertaining to the performance of our model, we embarked upon a series of numerous and meticulously designed experiments. These experiments were executed on Sudoku puzzles of varying dimensions, specifically on grids of dimensions 9×9 and 4×4 . For each distinct puzzle variant, we deliberately handpicked a curated assortment of puzzle instances, wherein each instance contained a varying count of initial clues.

For the larger 9×9 Sudoku puzzles, we meticulously curated a selection of puzzle instances encompassing a spectrum of clue quantities, specifically 29, 30, 31, 32, 33, 34, 35, 36 and 37 clues. These instances were carefully sourced from the benchmark dataset of the open-source software project known as t-doku[43].

Similarly, with regard to the 4×4 Sudoku puzzles, we curated a collection of puzzle instances that once again contained a varying number of hints, specifically 7, 8, 9, 10 and 11. This collection was referenced from the repository[44], with some instances being generated utilizing the Arel’s Sudoku Generator[45].

2. Approximate Experiment Size

As we discussed in 2., the size of the QUBO matrix for a Sudoku instance is a $n^3 \times n^3$ matrix, containing the necessary constraints for our problem. As the number of quadratic variables corresponds to the total number of qubits needed for a QUBO model[19], we can calculate the number by adding the number of total linear and quadratic variables needed in each of our QUBO models.

Table 4.1 provides information about each variable count for every constraint we impose in a Sudoku QUBO, with m hints already on the board:

Constraint	Linear Variables	Quadratic Variables	Total Variables
QCell	n^2	n^4	n^4
QRow	n^2	n^4	n^4
QCol	n^2	n^4	n^4
QGrid	n^2	n^4	n^4
QHint	m	0	0

Table 4.1: Approximate Number of Variables in QUBO constraint matrices

Thus we can calculate that for a 4×4 Sudoku Instance we would need theoretically a total of 256 qubits, and a 9×9 instance would need 6561 total qubits.

3. Solvers Used

Each individual puzzle instance underwent rigorous evaluation across multiple solving methodologies, including Quantum Annealers, Quantum-Hybrid Annealers, and Simulated Annealing. In the case of Quantum Annealers, we employed the highly advanced Advantage_System6.2 Solver, a sophisticated offering by D-Wave. Likewise, the Quantum-Hybrid Annealer experiments leveraged the capabilities of the LeapHybridSampler, another powerful solution provided by D-Wave. Lastly, the Simulated Annealing procedure was conducted leveraging the computational resources of a Lenovo Ideapad Gaming 3 notebook.

However, while Advantage_System6.2, while containing more than 5000 qubits[46], was able to provide valid solutions for 4×4 Sudoku instances, it was unable to successfully execute experiments conducted on 9×9 Sudoku instances. While the smaller instances were successfully executed, achieving an even more efficient qubit usage than theorized, the higher logical quadratic variables are handled far less efficiently, even exceeding by magnitudes the calculated total, as noted in [47], and was unable to process the 6561 qubits needed for the instance.

Chapter 5

Results and Analysis

1. The 4×4 Sudoku Instance

As previously indicated, the initial series of experiments encompassed the utilization of a collection of Sudoku puzzles characterized by a 4×4 grid dimensions. These puzzles were deliberately selected to exhibit a diverse range of clue quantities. Each distinct puzzle instance underwent an iterative process, being subjected to approximately 25 iterations per solver implementation. This meticulous replication aimed to amass an exhaustive compendium of benchmark data points, all accomplished within the constraints of the complimentary trial duration provided by the D-Wave computing platform.

1.1.1 Simulated Quantum Annealer Sampler Results

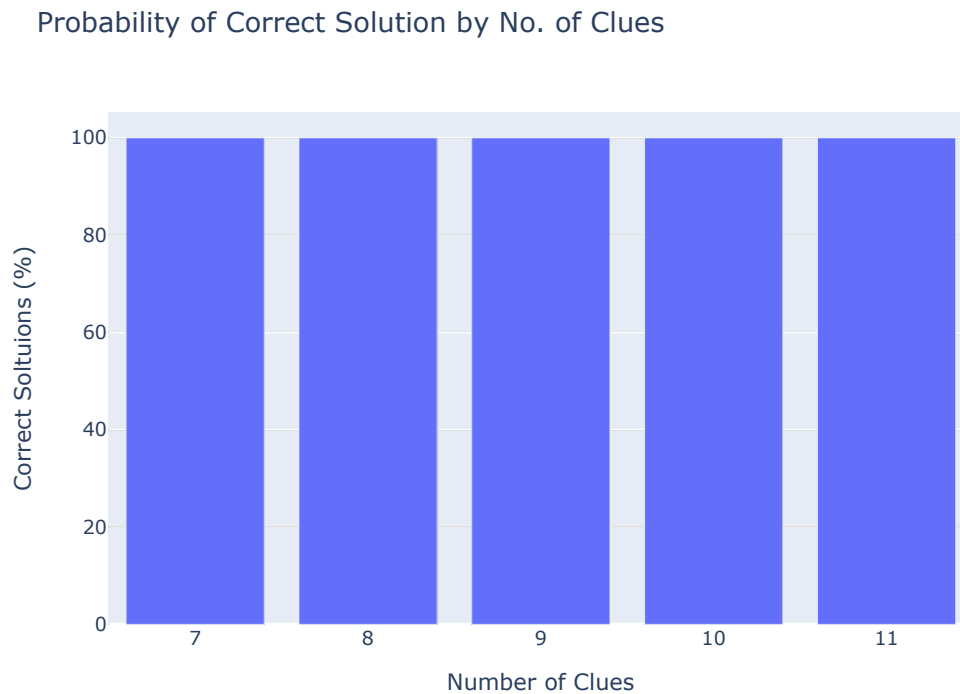


Figure 5.1: Simulated Annealing Results by Number of Clues

Figure 5.1 the solver’s ability to achieve a perfect 100.0% accuracy across all instances, which is attributed to the inherent simplicity of the problem instances being considered.

As the number of clues in the Sudoku puzzles increases incrementally from 7 to 11, the SimulatedAnnealingSampler solver consistently delivers correct solutions with unerring accuracy. This flawless performance is a direct result of the puzzles’ inherent simplicity, which allows the solver to effortlessly identify and generate accurate solutions.

While the solver’s consistent 100.0% accuracy is indeed impressive and indicative of its efficacy, it’s important to note that this level of success is specific to the simplified problem instances being considered. In more complex scenarios, the solver’s accuracy might vary. Nonetheless, the dataset provides a clear illustration of how the solver excels in solving straightforward Sudoku puzzles.

Probability of Correct Solution by num_reads

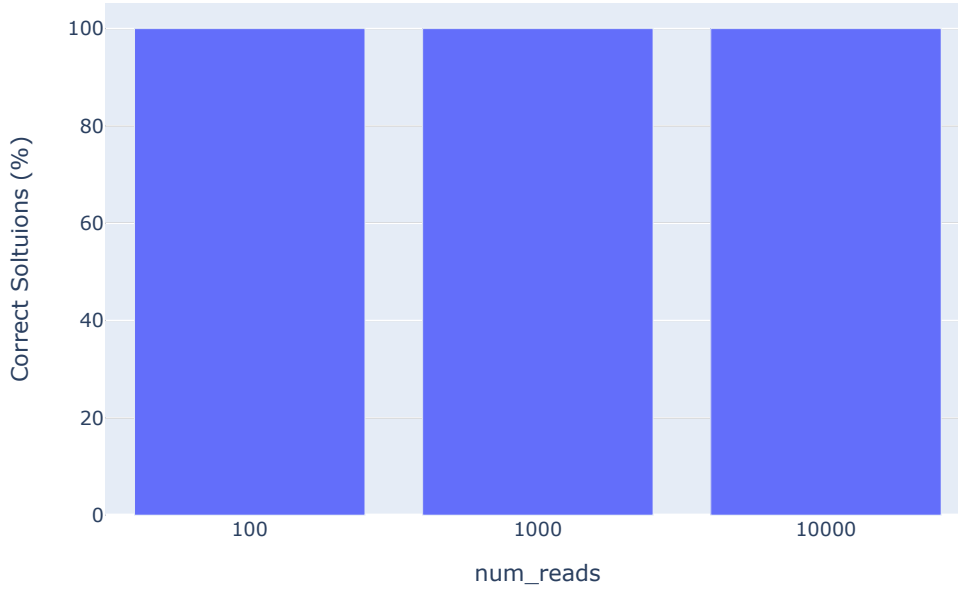


Figure 5.2: Simulated Annealing Results by Number of num_reads

Figure 5.2 highlights the solver’s consistent performance across different values of the `num_reads` parameter, demonstrating that it achieves a perfect 100.0% accuracy in all cases.

For each value of `num_reads`, including 100, 1000, and 10000, the `SimulatedAnnealingSampler` solver maintains an unswerving accuracy of 100.0%. This remarkable consistency underscores the solver’s ability to produce accurate solutions reliably, regardless of the number of reads employed during the solving process.

However, it’s important to emphasize that this remarkable accuracy is specific to the simplified problem instances considered here. When dealing with more complex Sudoku puzzles or other types of problems, the solver’s accuracy might differ. The current dataset provides a clear illustration of the solver’s efficacy in solving small-scale Sudoku instances under these specific conditions.

Probability of Correct Solution by No. of Clues and num_reads

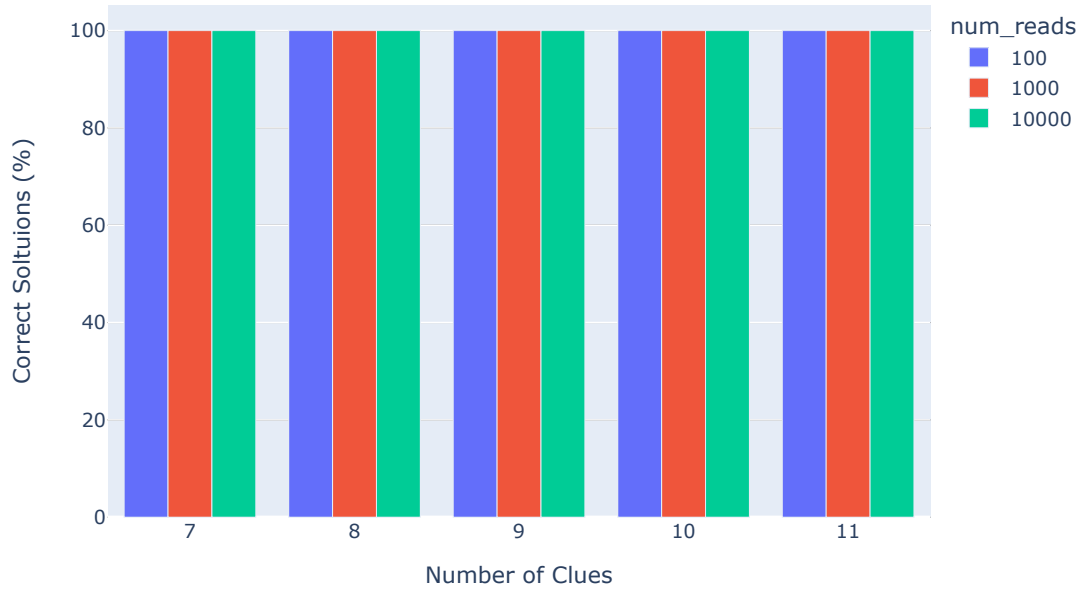


Figure 5.3: Simulated Annealing Results by Number of Clues and num_reads

Figure 5.3 continues to underscore the exceptional accuracy of the SimulatedAnnealingSampler solver in solving 4×4 Sudoku puzzles. Across varying values of both the num_reads parameter and the number of clues, the solver consistently achieves a perfect 100.0% accuracy. This persistent high accuracy can be attributed to the relatively small and straightforward nature of the problem instances being considered.

For each level of complexity, from puzzles with 7 to 11 clues, the solver maintains an unwavering accuracy of 100.0%. Even when exploring different num_reads values, including 100, 1000, and 10000, the solver's performance remains consistently perfect. This pattern reaffirms the solver's proficiency in solving Sudoku puzzles of this size, regardless of the number of reads employed during the solving process.

While the data supports the solver's remarkable accuracy in solving these specific instances, it's important to recognize that such high success rates might not generalize to more complex problem scenarios. In scenarios with larger puzzles or more intricate constraints, the solver's accuracy could potentially vary.

1.2 Hybrid Annealer Sampler Results

Probability of Correct Solution by No. of Clues

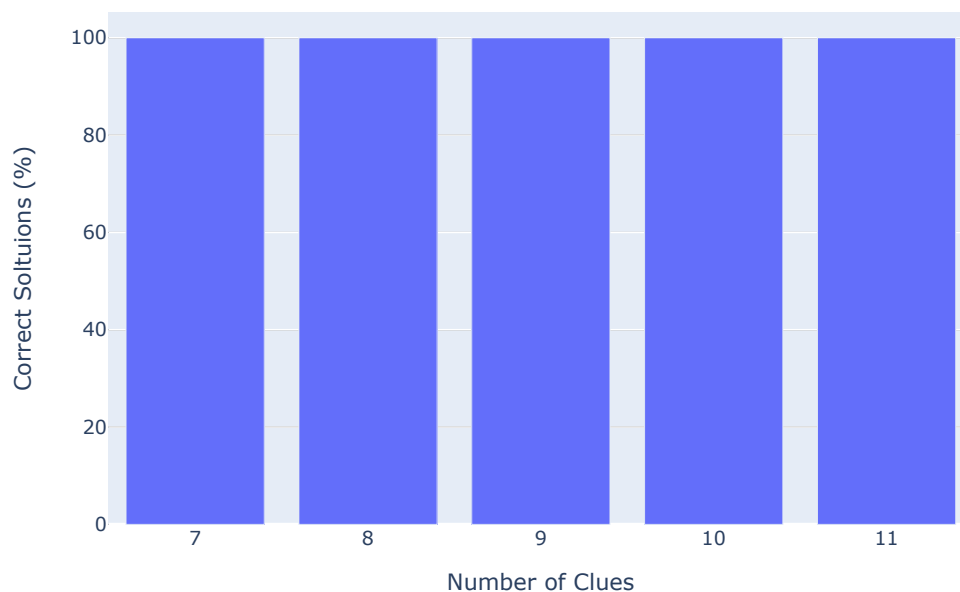


Figure 5.4: Hybrid Annealing Results by Number of Clues

In Figure 5.4 we observe the accuracy of the LeapHybridSampler in solving Sudoku puzzles. Across various levels of complexity, represented by the number of clues in the puzzles, the solver consistently achieves a perfect 100.0% accuracy.

For each puzzle instance, ranging from 7 to 11 clues, the LeapHybridSampler consistently produces accurate solutions with a flawless 100.0% success rate. This unwavering accuracy suggests that the solver effectively navigates and solves these Sudoku puzzles, regardless of their complexity.

However, it's important to note that this high accuracy might be attributed to the specific problem instances being considered, which could potentially be relatively straightforward for the solver. In more intricate or larger-scale Sudoku puzzles, the solver's accuracy might differ. The dataset presented here demonstrates the solver's efficiency in handling small-scale Sudoku instances under these particular conditions.

Probability of Correct Solution by num_reads

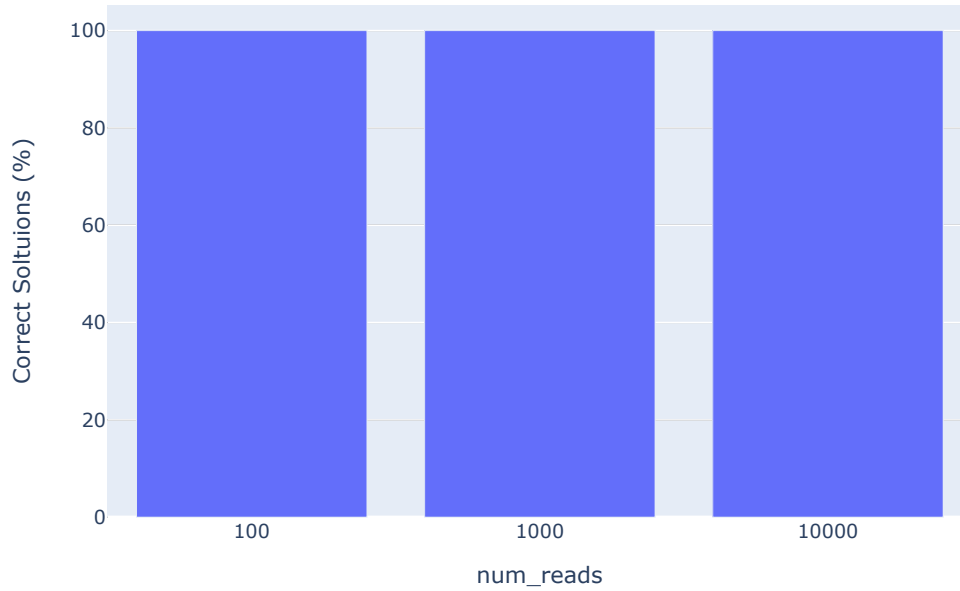


Figure 5.5: Hybrid Annealing Results by Number of num_reads

In Figure 5.5 we explore the accuracy of the LeapHybridSampler solver in solving Sudoku puzzles across various values of the num_reads parameter. In each case, the solver consistently achieves a perfect 100.0% accuracy.

Across the three values of num_reads —100, 1000, and 10000— the LeapHybridSampler solver maintains a flawless accuracy of 100.0%. This demonstrates the solver’s exceptional ability to consistently produce accurate solutions for Sudoku puzzles, regardless of the number of reads employed during the solving process.

However, it’s important to emphasize that while this dataset showcases the solver’s outstanding performance for the given problem instances, the solver’s behavior might differ when confronted with more complex or varied scenarios. The current data provides a clear illustration of the solver’s proficiency in solving Sudoku puzzles with these specific conditions.

Probability of Correct Solution by No. of Clues and num_reads

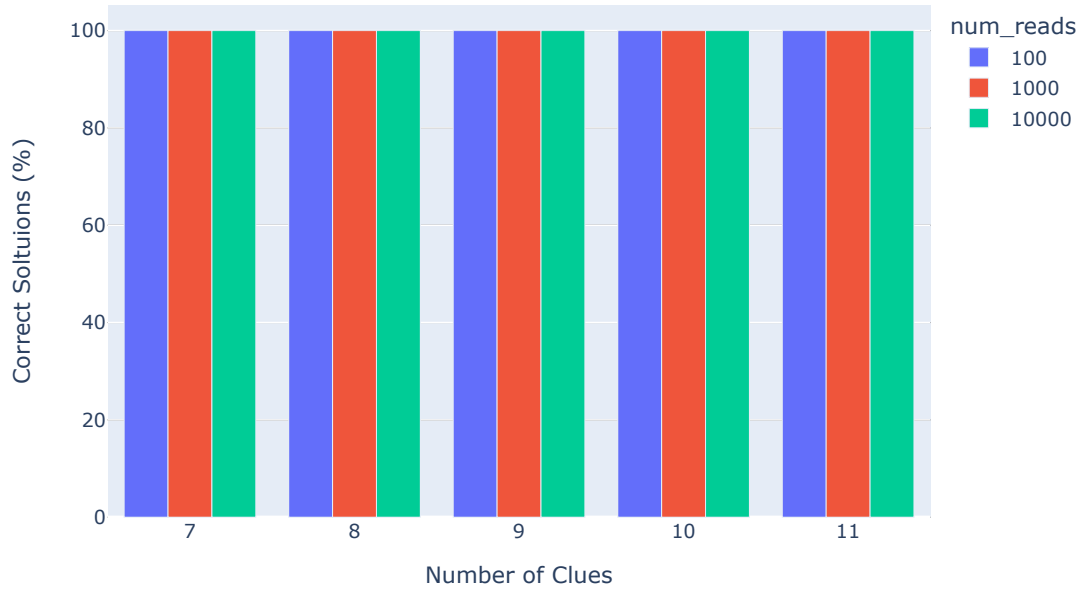


Figure 5.6: Hybrid Annealing Results by Number of Clues and num_reads

As with Figure 5.3, in Figure 5.6 we examine LeapHybridSampler’s accuracy in solving Sudoku puzzles across various values of the num_reads parameter and different levels of puzzle complexity. In each case, the solver consistently achieves a perfect 100.0% accuracy.

For every combination of the number of clues and num_reads, ranging from 7 to 11 clues and from 100 to 10000 num_reads, the LeapHybridSampler solver maintains a flawless accuracy of 100.0%. This demonstrates the solver’s capability to deliver accurate solutions reliably across a wide range of settings.

However, again, it’s important to recognize that while the solver’s performance is outstanding for the problem instances considered here, this dataset doesn’t necessarily reflect its behavior in more complex or diverse scenarios. Problem complexity, input size, and the specific characteristics of the puzzles can impact solver performance.

1.3 Quantum Annealer Sampler Results



Figure 5.7: Quantum Annealing Results by Number of Clues

Figure 5.10 presents a clear trend in the performance of the quantum annealing solver, Advantage_system6.2, when tasked with solving Sudoku instances with varying numbers of clues.

As the number of clues increases, it is evident that the quantum annealing solver exhibits a notable improvement in its performance. For instance, when confronted with puzzles containing only 7 clues, the solver achieved an accuracy of approximately 81.82% in solving them correctly. However, as the complexity of the puzzles increased with more clues, the solver's accuracy also grew significantly. The solver's success rate soared to 93.94% for puzzles with 8 clues, and further to 90.77% for puzzles with 9 clues. Notably, the solver's accuracy continued to rise, reaching an impressive 95.31% for puzzles containing 10 clues and achieving a perfect 100% accuracy for puzzles with 11 clues.

This trend underscores the solver's capacity to perform better on Sudoku instances with a greater number of clues, suggesting that the increased information provided by more clues contributes to the solver's ability to find correct solutions more reliably. The bar plot effectively illustrates this relationship, highlighting the solver's progressive accuracy enhancement as the complexity of the Sudoku puzzle

zles grows through an increase in the number of clues.

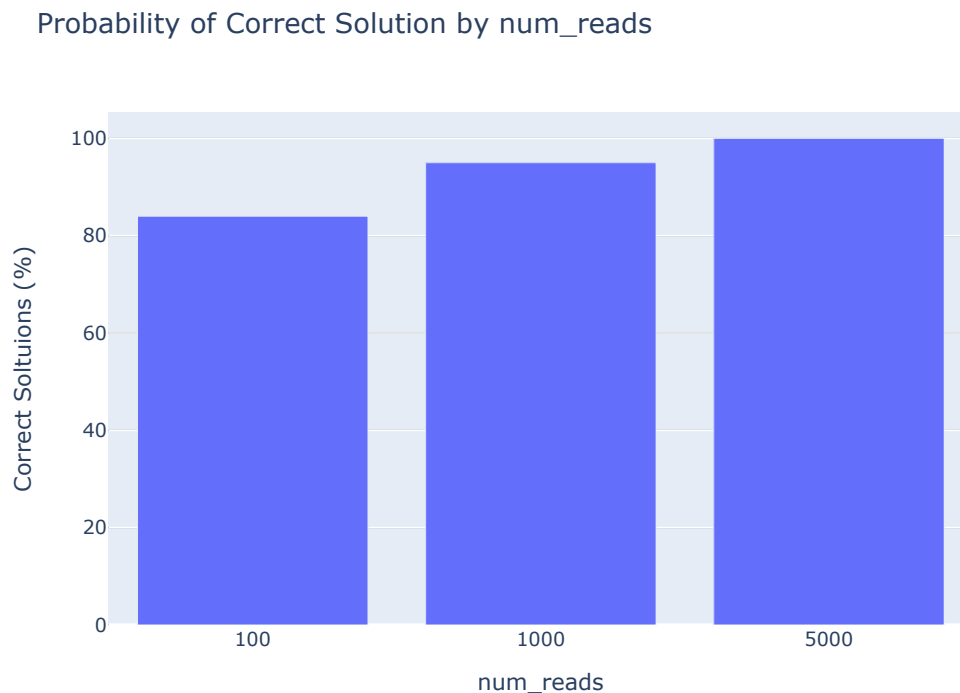


Figure 5.8: Quantum Annealing Results by Number of num_reads

Figure 5.8 presents an insightful view into the performance of the quantum annealing solver, Advantage_system6.2, across different settings of the num_reads parameter. This parameter governs the total number of passes the solver performs before concluding that it has reached the best possible solution.

As the num_reads parameter increases, we observe a distinct improvement in the solver's accuracy. When num_reads is set to 100, the solver achieves an accuracy of 84.0%, which signifies the proportion of correct solutions among the generated attempts. Upon increasing num_reads to 1000, the solver's accuracy notably rises to 95.0%. This significant enhancement in accuracy implies that the solver is not only identifying more correct solutions but also becoming more consistent in doing so.

A striking result is witnessed when the num_reads parameter is further increased to 5000, resulting in a perfect 100.0% accuracy. This suggests that the solver, at this setting, successfully identifies the correct solution for every Sudoku instance it encounters. The trend in accuracy across these different num_reads values indicates a positive correlation between the number of reads and the solver's performance, emphasizing that a higher number of reads yields more reliable and

accurate solutions.

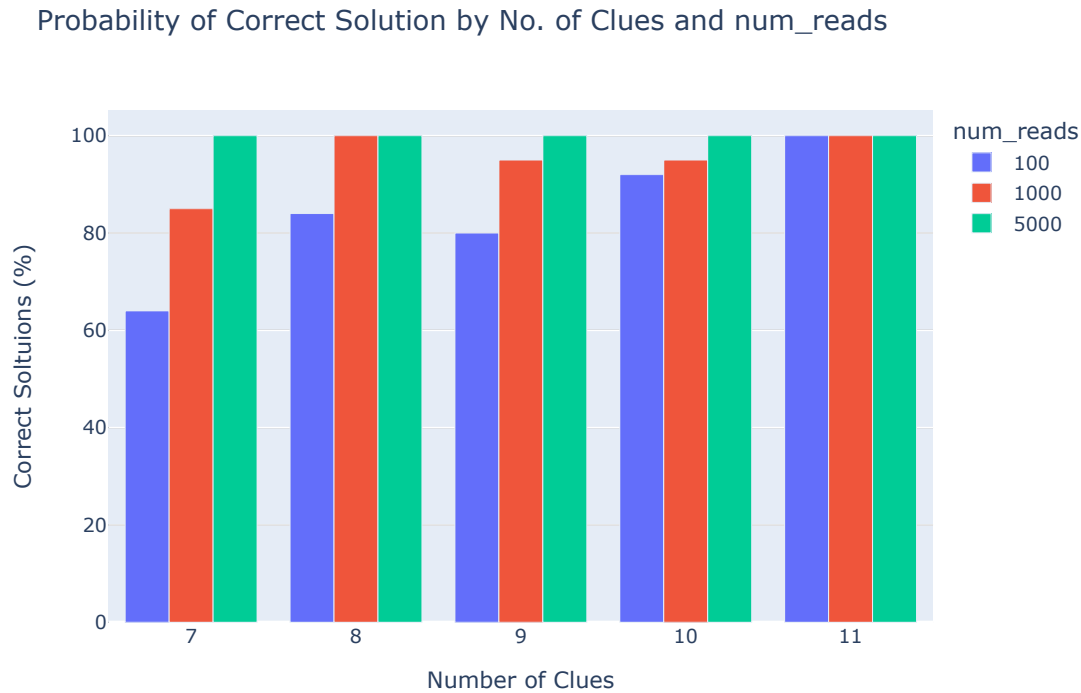


Figure 5.9: Quantum Annealing Results by Number of Clues and num_reads

Figure 5.9 contains the results for varying numbers of clues, as well as varying num_reads values. Starting with the number of clues parameter, it's evident that as the complexity of the Sudoku puzzles increases (from 7 to 11 clues), the solver's accuracy also rises. This positive correlation indicates that more clues provide the solver with a better foundation to find accurate solutions. For instance, when dealing with puzzles containing 7 clues, the solver's accuracy starts at 64.0% for 100 num_reads, which improves to 85.0% with 1000 num_reads, and reaches a perfect 100.0% with 5000 num_reads. Similarly, as the number of clues increases to 8, 9, 10, and 11, the solver's accuracy follows a similar upward trajectory, ultimately achieving 100.0% accuracy for all puzzles with 5000 num_reads.

Shifting our focus to the num_reads parameter, we observe that it plays a pivotal role in fine-tuning the solver's accuracy. For each set of clues, as num_reads increases, the solver's accuracy consistently improves. This demonstrates that increasing the number of reads enhances the solver's capability to converge to correct solutions, regardless of the complexity of the Sudoku puzzles.

In essence, the bar plot showcases a dual relationship. Firstly, the more clues given in the puzzle, the higher the solver's accuracy tends to be. Secondly, a higher

value of num_reads results in better accuracy regardless of the puzzle’s complexity. These findings underscore the importance of both parameters in influencing the solver’s performance. It is clear that a combination of more clues and increased num_reads can significantly boost the solver’s accuracy, leading to a greater likelihood of solving Sudoku puzzles correctly.

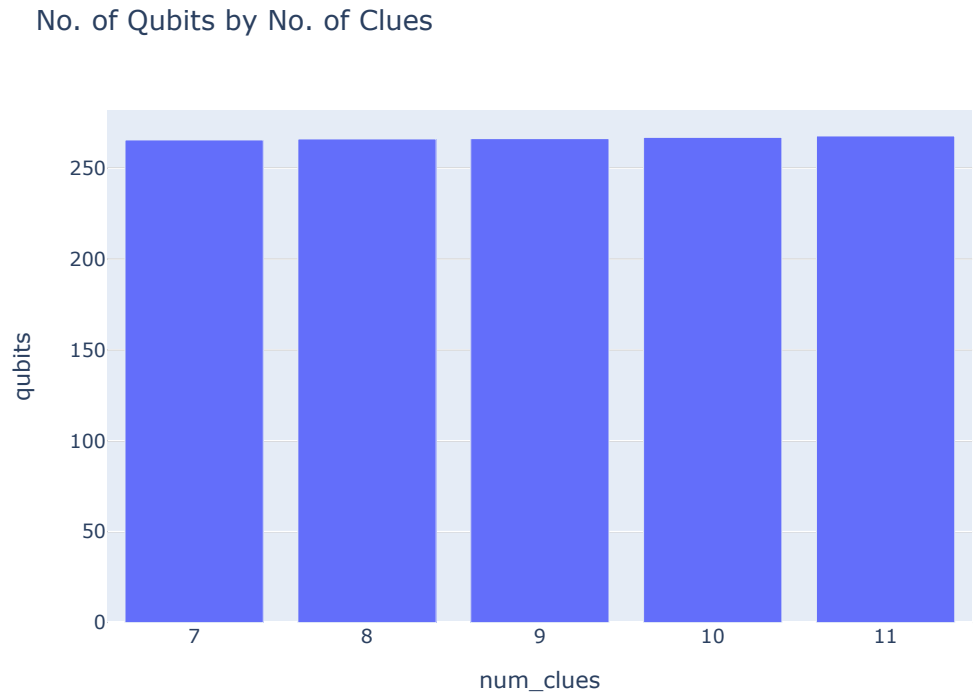


Figure 5.10: Qubit Usage by Number of Clues

Figure 5.10 reveals an interesting relationship between the number of clues in a Sudoku puzzle and the corresponding number of qubits utilized by the Advantage_system6.2 solver. Across the spectrum of puzzle complexities, from 7 clues to 11 clues, the solver’s qubit requirement demonstrates a consistently marginal difference. For Sudoku puzzles with 7 clues, the solver necessitates approximately 265.53 qubits. As the number of clues increases incrementally to 8, 9, 10, and 11, the corresponding number of required qubits experiences only slight fluctuations. The solver’s qubit requirement stands at around 266.08 qubits for 8 clues, approximately 266.25 qubits for 9 clues, roughly 266.92 qubits for 10 clues, and about 267.72 qubits for puzzles with 11 clues.

This remarkably consistent trend underscores the solver’s capacity to handle varying puzzle complexities with minimal adjustments in its qubit requirement. The minor differences observed in the number of qubits across different levels of puzzle complexity highlight the solver’s efficiency in adapting its resources to

address a range of problem intricacies. The findings emphasize that, with Advantage_system6.2, the solver’s qubit demand remains relatively stable despite fluctuations in puzzle complexity.

2. The 9×9 Sudoku Instance

2.1 Simulated Quantum Annealer Sampler Results

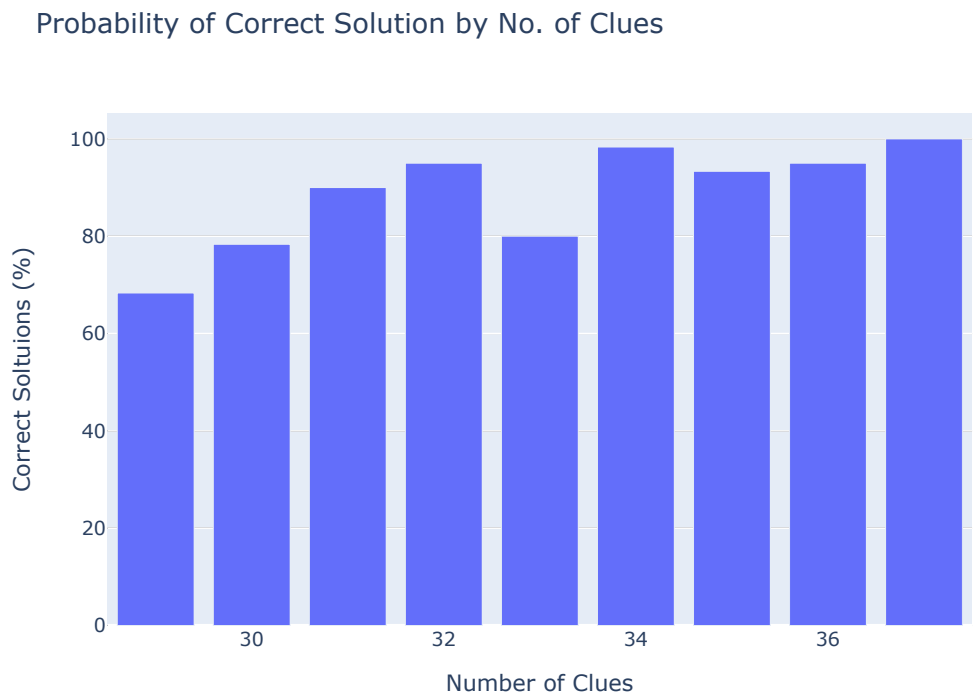


Figure 5.11: Simulated Annealing Results by Number of Clues

The data presented in Figure 5.11 outlines the performance of the SimulatedAnnealingSampler solver in solving 9x9 sudoku instances, focusing on the correlation between the number of clues provided in the puzzles and the achieved correct solutions percentage. Each entry in the chart corresponds to a different number of initial clues and indicates the solver’s success rate in percentage terms.

Analyzing the data, a noticeable trend emerges: as the number of clues increases, the solver’s effectiveness in finding correct solutions also improves. For instance, puzzles with a lower number of clues, such as those with 29 or 30 initial clues, exhibit success rates of 60% and 78.33% respectively. However, as the

number of clues grows, so does the solver’s proficiency. This is evident in the data points with 31 to 37 clues, where the success rates steadily increase from 86.67% to a perfect 100%.

This pattern suggests that the SimulatedAnnealingSampler solver benefits from a greater quantity of initial clues, as these clues likely provide more essential starting information for the solver’s search process. The gradual rise in success rates underscores the solver’s adaptability and capability to navigate the puzzle space more effectively as it gains more initial information to work with.

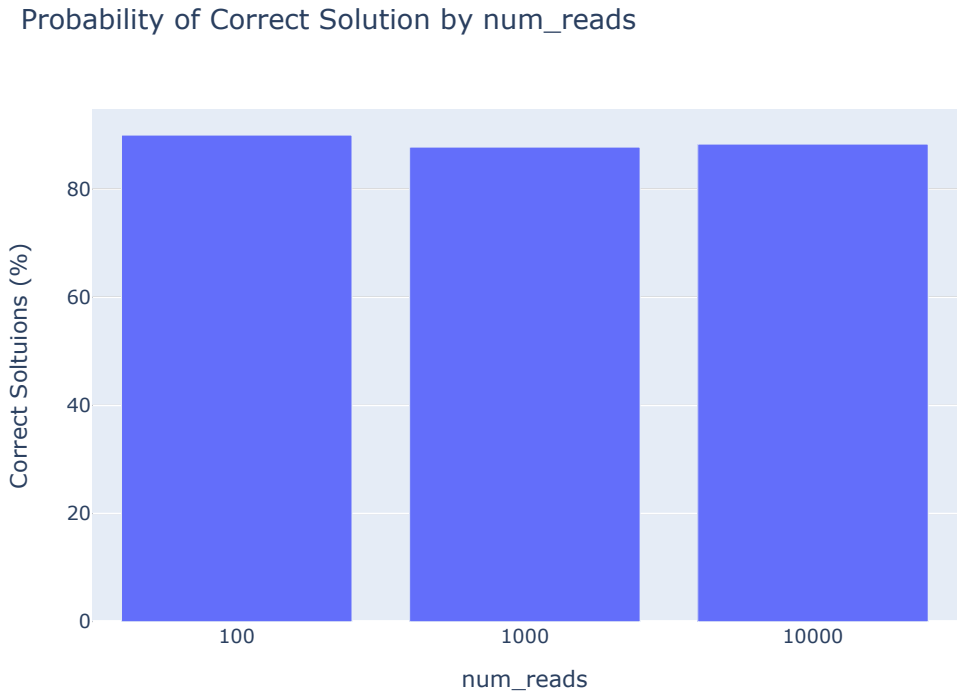


Figure 5.12: Simulated Annealing Results by Number of num_reads

Figure 5.12 showcases the relationship between the num_reads parameter and the achieved correct solutions percentage by the SimulatedAnnealingSampler. Upon examination of the data, an interesting pattern becomes evident: as the num_reads parameter increases, the solver’s accuracy in finding correct solutions also experiences a moderate improvement. For instance, when num_reads is set to 100, the solver achieves a success rate of 86.11%. This success rate gradually increases to 87.78% when num_reads is raised to 1000, and further improves to 88.33% when num_reads is elevated to 10000.

This trend highlights the impact of the num_reads parameter on the solver’s performance. A larger value of num_reads allows the solver to explore the solution

space more extensively, increasing the likelihood of identifying valid solutions. The incremental enhancement in success rates with higher num_reads values suggests that the solver benefits from additional iterations, allowing it to refine its search and select more accurate solutions.



Figure 5.13: Simulated Annealing Results by Number of Clues and num_reads

Figure 5.13 illustrates the performance of the `SimulatedAnnealingSampler` solver on solving 9x9 sudoku instances, considering different combinations of the number of clues and the `num_reads` parameter.

Firstly, as the number of clues increases, the solver’s success rate also tends to rise. This indicates that puzzles with more initially filled cells (clues) tend to be easier for the solver to tackle successfully. This aligns with the intuitive understanding that a higher number of clues provides the solver with more starting information, making it more likely to find a valid solution.

Secondly, there’s a clear impact of the `num_reads` parameter on the solver’s performance. Higher values of `num_reads` lead to more accurate and refined solutions. This is particularly evident when observing the data points with the same number of clues but varying `num_reads` values. As `num_reads` increases, the percentage of correct solutions generally improves, indicating that a larger number of iterations allows the solver to explore the solution space more comprehensively

and settle on better solutions.

2..2 Hybrid Annealer Sampler Results

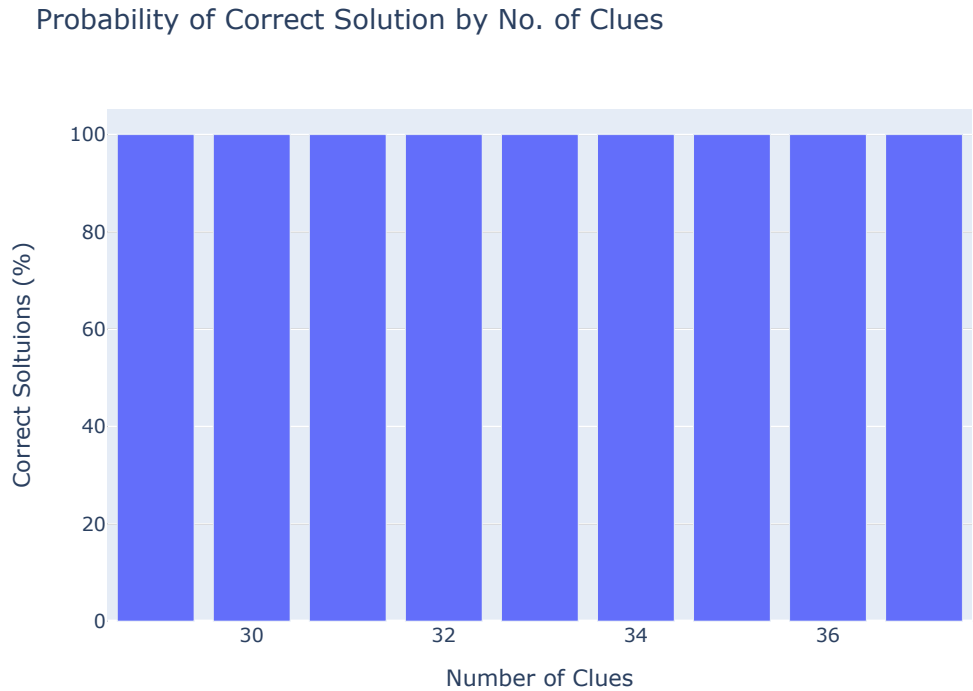


Figure 5.14: Hybrid Annealing Results by Number of Clues

The provided data presents the performance outcomes of the LeapHybridSampler solver when applied to solve 9x9 sudoku instances. The data emphasizes the relationship between the number of initial clues in the puzzles and the achieved correct solutions percentage. Each entry in the dataset corresponds to a distinct number of initial clues, and the solver has achieved a perfect 100% success rate for all cases.

Upon analysis of the data, a remarkable and consistent pattern emerges: regardless of the number of initial clues present in the puzzles (ranging from 29 to 37 clues), the LeapHybridSampler solver has succeeded in finding correct solutions in every instance, resulting in a flawless 100% success rate.

This consistent and optimal performance across all levels of initial clue counts underscores the solver’s robustness and reliability. The solver’s ability to consistently yield correct solutions, irrespective of the puzzle’s complexity as defined by the number of initial clues, suggests a high level of sophistication and efficiency in navigating the sudoku puzzle space.

Probability of Correct Solution by num_reads

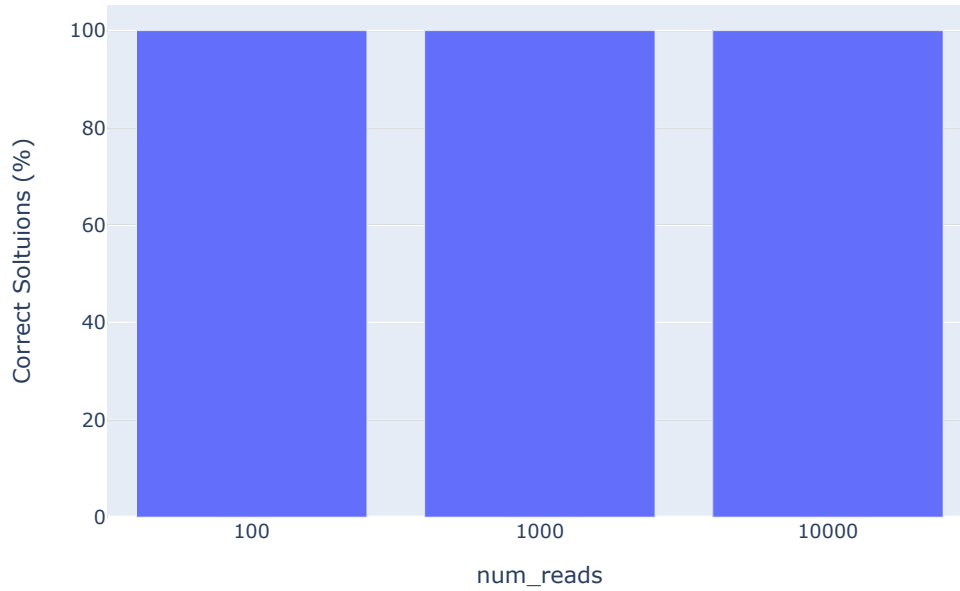


Figure 5.15: Hybrid Annealing Results by Number of num_reads

Figure 5.15 outlines the performance results of the LeapHybridSampler solver when applied to solving a specific type of problem. The data specifically focuses on the interaction between the num_reads parameter and the achieved correct solutions percentage. Each entry in the dataset corresponds to a different value of the num_reads parameter, and the solver has achieved a perfect 100% success rate for all cases.

Upon analyzing the data, a notable pattern becomes evident: regardless of the num_reads value, the LeapHybridSampler solver consistently attains correct solutions. When num_reads is set to 100, 1000, or 10000, the solver achieves a perfect success rate of 100% in all instances.

This consistent performance highlights the solver's robustness and reliability in producing correct solutions under various parameter settings. The solver's ability to consistently yield accurate solutions showcases its efficiency and effectiveness in solving the specific type of problem it is designed for.

Probability of Correct Solution by No. of Clues and num_reads



Figure 5.16: Hybrid Annealing Results by Number of Clues and num_reads

Figure 5.16 showcases the performance outcomes of the LeapHybridSampler solver in solving 9x9 sudoku puzzles. The data focuses on the interaction between the number of initial clues, the num_reads parameter, and the resultant correct solutions percentage. Each entry in the dataset pertains to a specific number of initial clues and num_reads value, and in each case, the solver has achieved a perfect 100% success rate.

Upon analyzing the data, an evident pattern emerges: the LeapHybridSampler solver consistently attains correct solutions for all tested scenarios, irrespective of the number of initial clues or the num_reads parameter. Regardless of the level of puzzle complexity, as indicated by the number of initial clues, and regardless of the number of iterations (num_reads), the solver achieves an unblemished success rate of 100

This remarkable performance across the board underscores the solver's exceptional capabilities and reliability, and is in accordance with D-Wave's assertion that it can handle an arbitrarily big problems [20]. The solver's ability to produce perfect solutions consistently across different puzzle configurations and parameter settings speaks to its advanced optimization and problem-solving abilities.

2.3 Quantum Annealer Sampler Results

As previously stated, the 9x9 Sudoku instance demands a substantial quadratic unconstrained binary optimization (QUBO) matrix of dimensions 729×729 , thereby necessitating a minimum allocation of 531441 qubits to effectively address the computational demands inherent to its resolution. Regrettably, this formidable quantum resource requirement surpasses the current capacity of our hardware infrastructure. Consequently, we find ourselves precluded from executing any experimental endeavors employing a Quantum Annealer to tackle this particular problem instance.

Chapter 6

Conclusions

In this work, we have considered various implementations, computational analyses, and comparisons of different of available annealers for the Sudoku problem. We provide a complete matrix formulation for the Sudoku problem, with generic, n -independent formulation, without the need for different formulation for each value of n . We also document a step-by-step instructions in replicating the model, as well as the needed python code to use it for any Sudoku puzzle. Additionally, we executed several experiments using D-Wave's available annealers and came to the following conclusions:

- ▶ It is possible to run a 4×4 Sudoku instance on a quantum annealer using the aforementioned methods,;
- ▶ D-Wave's Advantage_system6.2 is as efficient in its consumption of qubits, no matter how complex or hard a Sudoku instance is.
- ▶ 4×4 Sudoku instances are especially simple for Simulated and Quantum-Hybrid annealers to provide significant performance data.
- ▶ It is consistently more likely for the solver to correctly solve instances with a higher clue count.
- ▶ Using higher number of states a solver reads (num_reads) consistently provides better solutions.
- ▶ Hybrid solvers always provide a correct solution to a problem and never break the constraints of a QUBO model, even for arbitrarily big problems.

The above conclusions are particularly significant, as they indicate that quantum annealers are able to correctly solve Sudoku puzzles using our QUBO model. With the current advancements in quantum computing, the prospect of using a quantum annealer for solving 9×9 puzzles looks very promising, especially if quantum computers continue to scale at the rate we see today. The results from our experimental analysis seem to indicate that our method is very promising. We plan

on continuing this line of research, extending its application to a variety of computational puzzles, such as Latin Squares, which have more practical real-world applications.

Bibliography

- [1] R. P. Feynman, “Simulating physics with computers,” *International Journal of Theoretical Physics Int J Theor Phys*, vol. 21, no. 6-7, pp. 467–488, 1982.
- [2] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer,” *SIAM Review*, vol. 41, no. 2, pp. 303–332, 1999.
- [3] L. Grover, “A fast quantum mechanical algorithm for database search,” in *Proc. of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, 1996*, 1996.
- [4] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge University Press, 2010.
- [5] T. Kadowaki and H. Nishimori, “Quantum annealing in the transverse ising model,” *Physical Review E*, vol. 58, no. 5, p. 5355, 1998.
- [6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, “Optimization by simulated annealing,” *science*, vol. 220, no. 4598, pp. 671–680, 1983.
- [7] S. Pakin, “Performing fully parallel constraint logic programming on a quantum annealer,” *Theory and Practice of Logic Programming*, vol. 18, no. 5-6, pp. 928–949, 2018.
- [8] C. Papalitsas, P. Karakostas, T. Andronikos, S. Sioutas, and K. Giannakis, “Combinatorial GVNS (general variable neighborhood search) optimization for dynamic garbage collection,” *Algorithms*, vol. 11, p. 38, mar 2018.
- [9] Y. Dong and Z. Huang, “An improved noise quantum annealing method for tsp,” *International Journal of Theoretical Physics*, vol. 59, no. 12, pp. 3737–3755, 2020.
- [10] V. Choi, “Minor-embedding in adiabatic quantum computation: I. the parameter setting problem,” *Quantum Information Processing*, vol. 7, no. 5, pp. 193–209, 2008.
- [11] K. Boothby, P. Bunyk, J. Raymond, and A. Roy, “Next-generation topology of d-wave quantum processors,” tech. rep., D-Wave Systems, 2019.

- [12]
- [13] D-Wave, “D-wave qpu architecture: Topologies.” https://docs.dwavesys.com/docs/latest/c_gs_4.html#pegasus-couplers. Accessed: 2022-01-03.
- [14] H. Ushijima-Mwesigwa, C. F. Negre, and S. M. Mniszewski, “Graph partitioning using quantum annealing on the d-wave system,” in *Proceedings of the Second International Workshop on Post Moores Era Supercomputing*, pp. 22–29, ACM, 2017.
- [15] F. Glover and G. Kochenberger, “A tutorial on formulating qubo models,” *arXiv preprint arXiv:1811.11538*, 2018.
- [16] C. Silva, A. Aguiar, P. Lima, and I. Dutra, “Mapping a logical representation of tsp to quantum annealing,” *Quantum Information Processing*, vol. 20, no. 12, p. 386, 2021.
- [17] E. Boros, P. L. Hammer, and G. Tavares, “Local search heuristics for quadratic unconstrained binary optimization (qubo),” *Journal of Heuristics*, vol. 13, no. 2, pp. 99–132, 2007.
- [18] G. F. Newell and E. W. Montroll, “On the theory of the ising model of ferromagnetism,” *Reviews of Modern Physics*, vol. 25, no. 2, p. 353, 1953.
- [19] A. Lucas, “Ising formulations of many np problems,” *Frontiers in Physics*, vol. 2, p. 5, 2014.
- [20] F. Neukart, G. Compostella, C. Seidel, D. Von Dollen, S. Yarkoni, and B. Parney, “Traffic flow optimization using a quantum annealer,” *Frontiers in ICT*, vol. 4, p. 29, 2017.
- [21] G. Santos-García and M. Palomino, “Solving sudoku puzzles with rewriting rules,” *Electronic Notes in Theoretical Computer Science*, vol. 176, no. 4, pp. 79–93, 2007.
- [22] T. Yato and T. Seta, “Complexity and completeness of finding another solution and its application to puzzles,” *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 86, no. 5, pp. 1052–1060, 2003.
- [23] B. Felgenhauer and F. Jarvis, “Mathematics of sudoku i,” 2006.
- [24] J. Biamonte, P. Wittek, N. Pancotti, P. Rebentrost, N. Wiebe, and S. Lloyd, “Quantum machine learning,” *Nature*, vol. 549, no. 7671, p. 195, 2017.
- [25] R. Y. Li, R. Di Felice, R. Rohs, and D. A. Lidar, “Quantum annealing versus classical machine learning applied to a simplified computational biology problem,” *NPJ quantum information*, vol. 4, no. 1, p. 14, 2018.

- [26] A. Sagingalieva, A. Kurkin, A. Melnikov, D. Kuhmistrov, M. Perelshtein, A. Melnikov, A. Skolik, and D. Von Dollen, “Hyperparameter optimization of hybrid quantum neural networks for car classification,” *arXiv preprint arXiv:2205.04878*, 2022.
- [27] R. Nembrini, M. Ferrari Dacrema, and P. Cremonesi, “Feature selection for recommender systems with quantum computing,” *Entropy*, vol. 23, no. 8, p. 970, 2021.
- [28] K. Kurihara, S. Tanaka, and S. Miyashita, “Quantum annealing for clustering,” *arXiv preprint arXiv:1408.2035*, 2014.
- [29] A. Irbäck, L. Knuthson, S. Mohanty, and C. Peterson, “Folding lattice proteins with quantum annealing,” *Physical Review Research*, vol. 4, no. 4, p. 043013, 2022.
- [30] A. Perdomo-Ortiz, N. Dickson, M. Drew-Brook, G. Rose, and A. Aspuru-Guzik, “Finding low-energy conformations of lattice protein models by quantum annealing,” *Scientific reports*, vol. 2, no. 1, pp. 1–7, 2012.
- [31] C. Papalitsas, T. Andronikos, K. Giannakis, G. Theocharopoulou, and S. Fanarioti, “A QUBO model for the traveling salesman problem with time windows,” *Algorithms*, vol. 12, p. 224, Oct 2019.
- [32] R. H. Warren, “Solving combinatorial problems by two dwave hybrid solvers: a case study of traveling salesman problems in the tsp library,” 2021. *arXiv:2106.05948*.
- [33] D. Job and V. Paul, “Recursive backtracking for solving 9* 9 sudoku puzzle,” *Bonfring International Journal of Data Mining*, vol. 6, no. 1, pp. 7–9, 2016.
- [34] P. Berggren and D. Nilsson, “A study of sudoku solving algorithms,” *Royal Institute of Technology, Stockholm*, 2012.
- [35] H. Simonis, “Sudoku as a constraint problem,” in *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*, vol. 12, pp. 13–27, Citeseer, 2005.
- [36] S. Abuluaiah, A. Mohamed, M. Annamalai, and H. Iida, “Fog of search resolver for minimum remaining values strategic colouring of graph,” in *Soft Computing in Data Science: 4th International Conference, SCDS 2018, Bangkok, Thailand, August 15-16, 2018, Proceedings 4*, pp. 201–215, Springer, 2019.
- [37] B. Crawford, M. Aranda, C. Castro, and E. Monfroy, “Using constraint programming to solve sudoku puzzles,” in *2008 Third International Conference on Convergence and Hybrid Information Technology*, vol. 2, pp. 926–931, IEEE, 2008.
- [38] M. Weiß, *Encoding strategies to solve Sudoku with Quantum Computers*. PhD thesis, Ludwig-Maximilians-Universität, 2022.

- [39] G. Rajchel-Mieldzioć, “Quantum mappings and designs,” *arXiv preprint arXiv:2204.13008*, 2022.
- [40] A. Pal, S. Chandra, V. Mongia, B. K. Behera, and P. K. Panigrahi, “Solving sudoku game using a hybrid classical-quantum algorithm,” *Europhysics Letters*, vol. 128, no. 4, p. 40007, 2020.
- [41] J. Hatzky, “Solving sudoku with a quantum power-up.” <https://towardsdatascience.com/solving-sudoku-with-a-quantum-power-up-5bb4f64f3944>, Mar 2021.
- [42] “Constraints example: Problem formulation.” https://docs.dwavesys.com/docs/latest/c_gs_6.html.
- [43] “T-doku dataset.” <https://github.com/t-dillon/tdoku>. Accessed: 2023-08-23.
- [44] “4x4-sudoku-dataset.” <https://github.com/Black-Phoenix/4x4-Sudoku-Dataset>. Accessed: 2023-08-23.
- [45] “Arel’s sudoku generator.” <https://www.ocf.berkeley.edu/~arel/sudoku/main.html>. Accessed: 2023-08-23.
- [46] P. F. Catherine McGeoch, “The d-wave advantage system: An overview,” tech. rep., D-Wave Systems, 09 2020.
- [47] E. Stogiannos, C. Papalitsas, and T. Andronikos, “Experimental analysis of quantum annealers and hybrid solvers using benchmark optimization problems,” *Mathematics*, vol. 10, no. 8, p. 1294, 2022.

Appendix A

Software

```
from math import sqrt

import dimod
import dwave.inspector
import numpy as np
import qubovert as qv
from dwave.system import DWaveSampler, EmbeddingComposite, LeapHybridSampler
from neal import SimulatedAnnealingSampler as SAS

def print_list(l, delimiter="0"):
    """
    Prints a list of lists with elements rounded to two decimal places.

    Parameters:
        l (list of lists): The input list of lists containing numeric elements.
        delimiter (str, optional): The character to use as a delimiter for empty elements.

    Example:
        >>> my_list = [[1.234, 2.345, 0], [3.456, 0, 4.567]]
        >>> print_list(my_list)
        1.23  2.35  0
        3.46  0    4.57
    """
    msg = ""
    for i in l:
        row = ""
        for c in i:
            if c != 0:
                row += str(round(c,2))+""
            else:
                row += f"{delimiter}"
        print(msg)

def decode_var_labels(index, N=4):
    """
    Decodes an index value into three-dimensional coordinates using base-N representation.
```

Parameters:

index (int): The index value to be decoded into coordinates.

N (int, optional): The base value used for decoding the index (default is 4).

Returns:

list: A list containing three integers representing the three-dimensional coordinates.

Example:

```
>>> decode_var_labels(12)
```

```
[0, 1, 2]
```

```
"""
```

```
for i in range(N):
```

```
    for j in range(N):
```

```
        for k in range(N):
```

```
            if (i*N*N)+(j*N)+k == index:
```

```
                return[i,j,k]
```

```
def pretty_solution(solution, value, n):
```

```
    """
```

Formats the solution dictionary into a pretty path matrix.

Parameters:

solution (dict): A dictionary with sequential keys representing the path positions.

The values in the dictionary are the ordered elements of the path.

value: The value associated with the solution (not used in this function).

n (int): The number of elements per row in the path matrix.

Returns:

list: A formatted path matrix with elements from the solution dictionary.

```
    """
```

```
    path_matrix = []
```

```
    flag = 0
```

```
    for i in range(0, max(solution.keys()), n):
```

```
        path_matrix.append([])
```

```
        for j in range(i, i+n):
```

```
            path_matrix[flag].append(solution[j])
```

```
        flag += 1
```

```
    return path_matrix
```

```
def encode_board_to_binary(board, n):
```

```
    """
```

Encodes a sudku board into a binary representation.

Parameters:

board (list of lists): The game board represented as a 2D list containing integers.

n (int): The dimension of the board (n x n). The board should have values between 1 and

Returns:

list of lists: A 3D binary representation of the board, with a depth of n.

Each cell in the binary representation is either 0 or 1, representing th

Example:

```
>>> game_board = [[1, 2, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 4, 0]]
```

```
>>> encode_board_to_binary(game_board, 4)
```

```
[[[1, 0, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]],
```

```

[[0, 1, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0]]]

"""
binary_board= np.zeros((n, n, n)).tolist()

for row_index, row in enumerate(board):
    for column_index, cell in enumerate(row):
        if cell>0:
            binary_board[cell][row_index][column_index] = 1
return binary_board

def decode_board_from_binary(binary_board):
    """
    Decodes a 3D binary representation of a game board into the original game board.

    Parameters:
        binary_board (list of lists of lists): A 3D binary representation of the game board.
            Each cell in the binary representation is either
            representing the presence of a value.

    Returns:
        list of lists: The original game board represented as a 2D list containing integers.

    Example:
    >>> bianry_board = [
        [[1, 0, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 0, 0]],
        [[0, 1, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
        [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]],
        [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0]]
    ]
    >>> decode_board_from_binary(binary_board)
        [[1, 2, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 4, 0]]

    """
    board= np.zeros((4, 4))

    for k, color in enumerate(binary_board):
        for i, row in enumerate(color):
            for j, cell_value in enumerate(row):
                if cell_value>0:
                    board[i][j]+=int(k)
                else:
                    board[i][j]+=int(0)
    return board

def print_board(board):
    """
    Prints the game board in a human-readable format.

    Parameters:
        board (list of lists): The game board represented as a 2D list containing integers.

```

Example:

```
>>> game_board = [[1, 2, 0, 0], [0, 0, 0, 0], [0, 0, 1, 0], [0, 0, 4, 0]]
>>> print_board(game_board)
  1  2 |  0  0
  0  0 |  0  0
  0  0 0 0 | 0 0 0 0
  0  0 |  1  0
  0  0 |  4  0
"""
rows = 'abcd'
cols = '1234'
boxes = [[(f"{r}{c}")] for c in cols] for r in rows]
for row, _boxes in enumerate(boxes):
    if row and row % 2 == 0:
        print(' '*6+"|"+" '*6)
    for col, box in enumerate(_boxes):
        if col and col % 2 == 0:
            print('|', end='')
        print('{' + box + '}'.format((int(board[row][col]) or ' ')), end='')
    print()
print()

def run_on_dwave(qubo, token = None, solver_type = "hybrid"):
    """
    Solves the given QUBO problem on D-Wave or simulated annealing sampler.

    Parameters:
        qubo (QUBO object): The quadratic unconstrained binary optimization (QUBO) problem to be solved.
                           The QUBO object contains the Q matrix representing the optimization problem.
        token (str, optional): The D-Wave API token for accessing D-Wave quantum processing units.
                              Required only when solver_type is 'qpu' or 'hybrid'.
        solver_type (str, optional): The type of solver to use. It can be 'qpu' for D-Wave quantum processing units,
                                    'hybrid' for the Leap hybrid solver, or 'simulated' for the simulated annealing sampler.

    Returns:
        dimod.Response: The result of the QUBO problem solved using the specified solver.

    """
    if solver_type == 'qpu':
        if token != None:
            sampler = DWaveSampler(token=token)
            sampler_name = sampler.solver.name
            composite = EmbeddingComposite(sampler)
            res = composite.sample(dimod.BQM.from_qubo(qubo.Q), num_reads=1000, annealing_time=1000)
        else:
            raise TypeError("You have not provided a D-Wave API token")

    elif solver_type == 'hybrid':
        if token != None:
            sampler = LeapHybridSampler(token=token, num_reads=10000)
            sampler_name = sampler.solver.name
            res = sampler.sample(dimod.BQM.from_qubo(qubo.Q)) # Create sampler object
        else:
            raise TypeError("You have not provided a D-Wave API token")

    elif solver_type == 'simulated':
```

```

        res = SAS().sample_qubo(qubo.Q, n_reads=100000, num_sweeps=1000) # Create sampler object
    return res

def generate_cell_constraint(n):
    """
    Generates a cell constraint matrix for the QUBO formulation of the Sudoku problem.

    Parameters:
        n (int): The dimension of the Sudoku board (n x n). The total number of cells will be n**2.

    Returns:
        numpy.ndarray: A cell constraint matrix represented as a 2D NumPy array.
    """
    qcell = np.zeros((n**3, n**3), dtype=int)
    for i in range(n):
        for j in range(n):
            for k in range(n):
                x = i*(n**2) + j*n + k
                for k2 in range(n):
                    y = i*(n**2) + j*n + k2
                    if x == y:
                        qcell[x][y] += 1
                    else:
                        qcell[x][y] += 2
    return qcell

def generate_column_constraint(n):
    """
    Generates a column constraint matrix for the QUBO formulation of the Sudoku problem.

    Parameters:
        n (int): The dimension of the Sudoku board (n x n). The total number of cells will be n**2.

    Returns:
        numpy.ndarray: A column constraint matrix represented as a 2D NumPy array.
    """
    qcol = np.zeros((n**3, n**3), dtype=int)
    for k in range(n):
        for j in range(n):
            for i in range(n):
                x = i*(n**2) + j*n + k
                for i2 in range(n):
                    y = i2*(n**2) + j*n + k
                    if x == y:
                        qcol[x][y] += 1
                    else:
                        qcol[x][y] += 2
    return qcol

def generate_row_constraint(n):
    """
    Generates a row constraint matrix for the QUBO formulation of the Sudoku problem.

    Parameters:
        n (int): The dimension of the Sudoku board (n x n). The total number of cells will be n**2.

```

Returns:

numpy.ndarray: A row constraint matrix represented as a 2D NumPy array.

"""

```
qrow = np.zeros((n**3,n**3), dtype=int)
for k in range(n):
    for i in range(n):
        for j in range(n):
            x = i*(n**2) + j*n + k
            for j2 in range(n):
                y = i*(n**2) + j2*n + k
                if x == y:
                    qrow[x][y] += 1
            else:
                qrow[x][y] += 2
return qrow
```

def generate_subgrid_constraint(n):

"""

Generates a subgrid constraint matrix for the QUBO formulation of the Sudoku problem.

Parameters:

n (int): The dimension of the Sudoku board (n x n). The total number of cells will be n

Returns:

numpy.ndarray: A subgrid constraint matrix represented as a 2D NumPy array.

"""

```
qgrid = np.zeros((n**3, n**3), dtype=int)
for u in (0, 2):
    for v in (0,2):
        for k in range(n):
            for j1 in range(int(sqrt(n))):
                for i1 in range(int(sqrt(n))):
                    x = (i1+u)*(n**2) + (j1+v)*n + k
                    for j2 in range(int(sqrt(n))):
                        for i2 in range(int(sqrt(n))):
                            y = (i2+u)*(n**2) + (j2+v)*n + k
                            if x == y:
                                qgrid[x][y] = 1
                            else:
                                qgrid[x][y] = 2
return qgrid
```

def generate_hint_constraint(n, board):

"""

Generates a hint constraint matrix for the QUBO formulation of the Sudoku problem.

Parameters:

n (int): The dimension of the Sudoku board (n x n). The total number of cells will be n
board (list of lists): The initial game board represented as a 2D list containing integ

Returns:

numpy.ndarray: A hint constraint matrix represented as a 2D NumPy array.

Note:

- Each hint constraint is represented by a diagonal element ($\square 4$) in the QUBO matrix.
- The diagonal elements correspond to the cells that already have hints (non-zero value)
- The $\square 4$ value in the diagonal enforces that the variable corresponding to the hint cell
- The resulting matrix is used in the QUBO formulation of the Sudoku problem.

"""

```
binary_board = encode_board_to_binary(board, n)
```

```
qhint = np.zeros((n**3,n**3), dtype=int)
```

```
for k, color in enumerate(binary_board):
```

```
    for i, row in enumerate(color):
```

```
        for j, cell in enumerate(row):
```

```
            if cell>0:
```

```
                x = i*(n**2) + j*n + k
```

```
                qhint[x][x] +=  $\square 4$ 
```

```
return qhint
```

```
if __name__ == '__main__':
```

```
    n = 4
```

```
    board = ((1,2,0,0),
```

```
              (0,0,0,0),
```

```
              (0,0,1,0),
```

```
              (0,0,4,0),
```

```
)
```

```
qhint = generate_hint_constraint(n, board)
```

```
qcell = generate_cell_constraint(n)
```

```
qcol = generate_column_constraint(n)
```

```
qrow = generate_row_constraint(n)
```

```
qgrid = generate_subgrid_constraint(n)
```

```
q = qhint + qcell + qgrid + qrow + qcol
```

```
q = np.triu(q)
```

```
qubo = qv.utils.matrix_to_qubo(q)
```

```
# Use DWave's simulated annealer
```

```
TOKEN = "DWAVEAPITOKEN"
```

```
res = run_on_dwave(qubo, TOKEN)
```

```
solution = pretty_solution(res.first.sample, res.first.energy, n)
```

```
binary_solution_board = np.zeros((n,n,n), dtype = int)
```

```
for index in res.first.sample:
```

```
    value = res.first.sample[index]
```

```
    if isinstance(index, int):
```

```
        board_index = decode_var_labels(index, N=n)
```

```
        binary_solution_board[board_index[2]][board_index[0]][board_index[1]] = int(value)
```

```
print("Hint")
```

```
print_board(board)
```

```
solution_board = np.zeros((n, n), dtype = int)
for k, color in enumerate(binary_solution_board):
    print(k+1)
    print_board(color)
    solution_board += color * (k+1)

print_board(solution_board)
```