

Individual Diploma Thesis

**SOLVING PROPOSITIONAL SATISFIABILITY PROBLEMS ON
QUANTUM COMPUTERS**

Christos Konstantinou

UNIVERSITY OF CYPRUS



COMPUTER SCIENCE DEPARTMENT

May 2025

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

Solving Propositional Satisfiability Problems on Quantum Computers

Christos Konstantinou

Supervisor

Dr. Yannis Dimopoulos

This individual thesis was submitted in partial fulfilment of the requirements for the degree of BSc in Computer Science at the Department of Computer Science, University of Cyprus

May 2025

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Dr. Yannis Dimopoulos, for their continuous guidance, support, and invaluable feedback throughout the course of this thesis. Their expertise and encouragement were instrumental in shaping both the research and the final manuscript.

I also wish to thank the University of Cyprus and the Department of Computer Science for providing the academic environment, resources, and opportunities necessary for the completion of this work. Special thanks are extended to the EPL laboratories and the technical staff for their assistance during the experimental phases.

Furthermore, I am deeply grateful to D-Wave Systems for providing access to educational materials, software tools, and documentation that greatly enriched the practical aspects of this research.

Finally, I would like to thank my family and friends for their unwavering support, patience, and encouragement throughout my studies. Their belief in me was a constant source of motivation.

Abstract

This thesis investigates the adaptation and solution of Boolean Satisfiability (SAT) problems using quantum annealing. Beginning with a study of quantum annealing as a computing model, we describe how optimization problems are formulated physically and mathematically, focusing on the role of Hamiltonians, eigen spectra, and the transition from classical to quantum solution methods. SAT problems, expressed in Conjunctive Normal Form (CNF), are transformed into Quadratic Unconstrained Binary Optimization (QUBO) models, further processed into Binary Quadratic Models (BQM), enabling compatibility with quantum annealers such as D-Wave’s systems. The thesis explores in detail the QUBO conversion process, the mechanics of minor embedding logical graphs onto the Chimera hardware topology, and the operation of classical and hybrid solvers supporting this workflow.

Early experiments validate the end-to-end SAT-to-quantum pipeline on small instances, providing hands-on insight into CNF generation, QUBO construction, embedding, and execution on D-Wave hardware. Further experiments employ classical hybrid solvers such as QBSolv and D-Wave’s Hybrid SDK to evaluate performance on larger BQMs, alongside SAT-derived instances.

A major focus of the thesis is the experimental analysis of graph embeddings into Chimera graphs. Solver performance is benchmarked between exact methods (Clingo and MiniZinc solvers) and heuristic methods (Minorminer), revealing critical scaling limitations of constraint solvers and the robustness and speed of heuristic embedding. A detailed comparative study highlights how constraint density, clause size, number of variables, and high-frequency variable reuse in SAT instances impact embedding success and time.

Finally, the work provides a systematic characterization of how SAT formula properties influence minor embedding difficulty, emphasizing the importance of sparsity for quantum hardware compatibility. The results demonstrate that quantum annealing, while not universally faster, offers practical and scalable approaches for complex optimization problems when logical formulation and embedding strategies are carefully designed.

Contents

Chapter 1 Introduction.....	1
1.1 Introduction and Scope of the Thesis.....	1
1.2 Quantum Annealing and Optimization in Quantum Computing	2
1.3 The Mechanics of Quantum Annealing.....	4
1.4 The Role of Hamiltonians and Eigen Spectrum in Quantum Annealing.....	7
1.5 Boolean Satisfiability and Its Role in Quantum Optimization.....	10
 Chapter 2 Foundations of Quantum Annealing for SAT.....	 13
2.1 Introduction.....	13
2.2 From SAT to QUBO.....	14
2.3 Binary Quadratic Models (BQM).....	16
2.4 Minor Embedding.....	18
2.5 Chimera and Pegasus Graph Topologies.....	20
2.6 D-Wave Solvers and Tools.....	21
2.7 Summary.....	22
 Chapter 3 Early Experiments with D-Wave Quantum Annealer.....	 24
3.1 Introduction.....	24
3.2 SAT Instance Generation.....	25
3.3 Basic QUBO Submission.....	25
3.4 Full SAT-to-QPU Execution.....	29
3.5 Observations and Summary.....	31
 Chapter 4 Experimental Framework and Solver Implementation....	 33
4.1 CNF Instance Generation.....	33
4.2 Experiments with QBSolv and Classical Solvers.....	36
4.3 Hybrid Solver Evaluation.....	40

Chapter 5 Minor Embedding Experiments and Solver Comparison..46

5.1 Experimental Scope and Objectives.....	46
5.2 Implementation Details.....	47
5.3 Solver Performance in Chimera Embedding.....	49
5.4 Minor Embedding with Extended Timeout.....	52
5.5 Heuristic Embedding Performance with Minorminer.....	56
5.6 Optimization of Heuristic Embeddings.....	59
5.7 Comparative Analysis & Discussion.....	62

Chapter 6

Impact of SAT Structure on Minor Embedding Complexity.....66

6.1 Introduction.....	66
6.2 Experimental Setup.....	67
6.3 Parameters Investigated.....	70
6.4 Results and Analysis.....	71
6.5 Conclusion.....	77

Chapter 7 Conclusions and Future Work.....78

7.1 General Conclusions.....	78
7.2 Future Work.....	79

REFERENCES.....81

APPENDIX A.....A-1

APPENDIX B.....B-1

APPENDIX C.....C-1

APPENDIX D.....D-1

APPENDIX E.....	E-1
APPENDIX F.....	F-1
APPENDIX G.....	G-1
APPENDIX H.....	H-1
APPENDIX I.....	I-1
APPENDIX J.....	J-1

Chapter 1

Introduction

1.1 Introduction and Scope of the Thesis	1
1.2 Quantum Annealing and Optimization in Quantum Computing	2
1.3 The Mechanics of Quantum Annealing	4
1.4 The Role of Hamiltonians and Eigen Spectrum in Quantum Annealing	7
1.5 Boolean Satisfiability and Its Role in Quantum Optimization	10

1.1 Introduction and Scope of the Thesis

Quantum computing is a rapidly evolving field that offers new ways to approach problems that are considered difficult or time intractable for classical computers [1]. There are different models of quantum computing, although in this thesis we focus on quantum annealing as implemented by D-Wave Systems [2]. Quantum annealing is well suited for solving optimization problems by searching for the lowest energy configuration of a system [3].

The main goal of this thesis is to study how Boolean satisfiability problems (SAT) can be adapted and solved using quantum annealing. We explore the entire pipeline required to go from a logical SAT problem, written in Conjunctive Normal Form (CNF), to a format which is compatible with quantum annealers. This process involves converting the SAT instance into a Quadratic Unconstrained Binary Optimization (QUBO) model [4], which is then translated into a Binary Quadratic Model (BQM) [5]. From there, the problem must be embedded onto the physical qubit architecture of the quantum processor, such as D-Wave's Chimera topology [6].

We will analyse these stages of the pipeline through a series of experiments. We will investigate how different characteristics of SAT problems can affect the transformation and embedding steps and also the performance when solved using classical and quantum compatible solvers like MiniZinc, Clingo and some D-Wave tools [7].

The following chapters introduce the background theory, describe the conversion and embedding processes in detail, and present experimental results. The aim is to provide insight into the practical challenges of working with quantum annealers and evaluate their potential for solving satisfiability and optimization problems.

1.2 Quantum Annealing and Optimization in Quantum Computing

The field of quantum computing is still in its early stages but is rapidly evolving, offering a range of promising approaches for building quantum systems [1]. In this thesis, we focus specifically on the quantum computers developed by D-Wave Systems, which are based on the quantum annealing approach [2]. We will explain the principles behind quantum annealing and how it compares to other models of quantum computation, such as the gate-model approach [1]. At its core, quantum annealing leverages quantum mechanics to solve specific classes of problems, particularly those related to optimization and probabilistic sampling [3]. In this work, we concentrate on the former, optimization problems, and explore how they are represented, transformed, and solved using D-Wave's quantum hardware [2].

An optimization problem is a problem where you are trying to find the most favourable solution from a set of possible candidates by either maximizing or minimizing a given objective function [8]. The goal is to identify the input values that yield the best possible outcome according to a defined criterion, often referred to as a fitness or cost function. For example, imagine designing a building under a fixed budget. There are many desirable features, but not all can be included. The challenge is to find the best possible combination of these features without exceeding the budget. Quantum annealing is well-suited in problems like this where the goal is to maximize or minimize an objective function [8].

The reason we can use physics to solve optimization problems is because we are able to frame them as a type of problem called an energy minimization problem [3]. There is a fundamental part of physics that says everything is always trying to find its minimum energy state. This can also be applied in quantum physics because quantum annealing exploits this natural behaviour, allowing quantum systems to evolve toward their minimum energy state, which corresponds to the optimal or near-optimal solution to a given problem [3].

In addition, sampling problems can also be addressed using quantum annealing. These types of problems are not aiming to find a single optimal solution but to extract representative samples from low energy regions of the solution space [2]. This can be useful in machine learning, where building and refining probabilistic models often requires insight into the structure of the underlying energy landscape.

Quantum annealing is different from other models of quantum computing, like the gate-model approach [1]. In gate-model, quantum states are precisely controlled and manipulated through a sequence of quantum gates, enabling the execution of highly sophisticated algorithms, like Shor's algorithm for factoring large numbers and Grover's algorithm for searching through databases [1]. These algorithms are way faster than anything we could possibly run on a classical machine given our current knowledge. While the gate-model approach allows for the solution of a broader range of problems, it is more technically demanding. The current state-of-the-art in gate-model quantum computing is around 10 qubits due to challenges in maintaining coherence and control. In contrast, quantum annealers like those from D-Wave have been scaled to over a thousand qubits, although they are specialized for a narrower class of problems [2].

While quantum annealing is designed for solving specific types of problems, it is part of a broader family of quantum computing approaches [3]. An example would be adiabatic quantum computing, which also relies on the principle of gradually evolving a quantum system to reach its lowest energy state [3]. In fact, adiabatic quantum computing can be seen as a more generalized or theoretical extension of quantum annealing [3].

A key difference lies in their capabilities. Adiabatic quantum computing, under certain mathematical conditions, can be equivalent in power to gate-model quantum computing [1]. This means that it can theoretically solve any problem that a gate-model quantum computer can, which is why it is considered a type of universal quantum computer [1]. In contrast, quantum annealing, as currently implemented in devices like D-Wave's systems, is not universal [2]. It is highly specialized and optimized for solving optimization problems and certain sampling tasks, but not the full range of problems that universal quantum computers are expected to handle [2].

1.3 The Mechanics of Quantum Annealing

Quantum annealing operates by gradually guiding a quantum system toward the minimum of a defined energy landscape, meaning that it finds an optimal or near-optimal solution to a problem [3]. To understand how this process works at the hardware level, we must explore the behaviour of some individual components within a quantum annealer, such as qubits, biases and couplers [2].

A qubit is the fundamental unit of quantum information. Unlike a classical bit, which exists in a state of either 0 or 1, a qubit can exist in a superposition of both states simultaneously [1]. In a quantum annealer each qubit begins in a superposition of 0 and 1 and transitions into a classical state, either 0 or 1, by the end of the annealing process [2]. This transition is influenced by the energy landscape, which is gradually shaped over the course of the computation.

The evolution of a qubit's state can be visualized using an energy diagram. Initially, a qubit exists in a single potential well representing the superposition state [2]. As the annealing process progresses, a double-well potential is formed, with each well corresponding to one of the two classical state outcomes, 0 or 1. The qubit will eventually settle into one of these two wells, but the probability of ending in either state can be controlled [2].

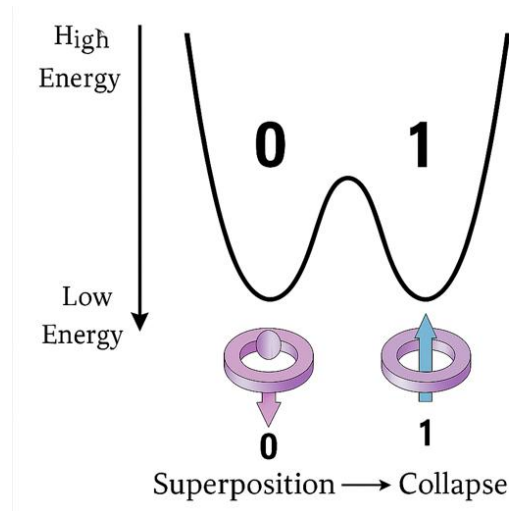


Figure 1.3.1 – Energy Landscape During Quantum Annealing

This control is achieved through the application of an external magnetic field, known as a bias [2]. The bias tilts the energy landscape in one direction, increasing the probability that the qubit collapses into a specific state. By adjusting the strength and direction of the bias, programmers can influence the final state of each qubit. This is a key mechanism in encoding and solving problems through quantum annealing [2].

The real computational power of quantum annealing arises when qubits are connected [2]. These connections are made via couplers, which define how qubits interact [2]. A coupler can encourage two connected qubits to prefer the same final state, both 0 or both 1, or to prefer the opposite state, 0 and 1, or 1 and 0 [2]. These preferences are not enforced directly, but rather by modifying the energy levels of different qubit pair configurations, making certain outcomes energetically favourable [2].

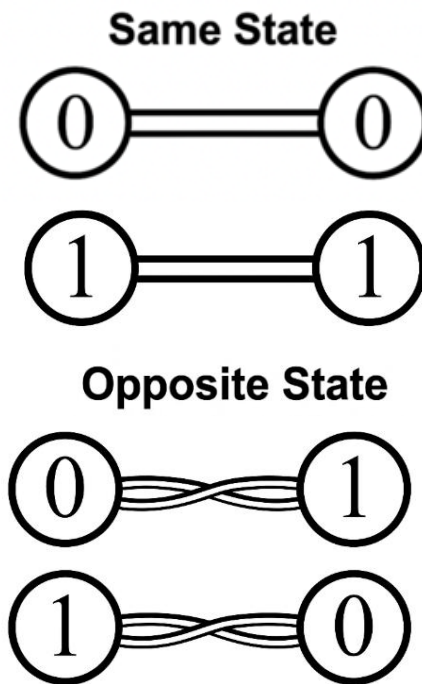


Figure 1.3.2 – Qubit Coupling Preferences

When qubits are connected and influenced by couplers, they can become entangled [1]. This is a uniquely quantum phenomenon in which the state of one qubit links to the state of another. In such cases, the qubits are considered as a single object with multiple possible states [1]. For example, two entangled qubits have four possible combined states, three qubits have eight, and so on. In general, a system of n qubits can represent 2^n states, meaning that the growth of computational capacity with each additional qubit is exponential [1].

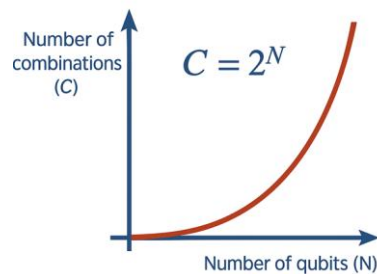


Figure 1.3.3 - Exponential Growth of Quantum State Combinations

The user programs a quantum annealer by setting the values of all qubit biases and coupler strengths [5]. This defines a custom energy landscape over the full configuration space. Then the annealer performs quantum annealing to search for the lowest energy configuration, which corresponds to the optimal solution of the problem encoded in the system [2].

Despite the complexity of this quantum process, the physical implementation is impressively fast. In D-Wave systems, a full annealing cycle, from initialization to measurement, takes approximately 20 microseconds, showing the speed and efficiency of this specialized quantum computing model [2].

1.4 The Role of Hamiltonians and Eigen Spectrum in Quantum Annealing

To fully understand how quantum annealing works, it is important to explore the physical principles at its core, particularly the concepts of the Hamiltonian and the eigen spectrum [1]. These terms represent the foundational ideas in both classical and quantum physics and are key to how quantum annealers, like those developed by D-Wave, operate [2].

A Hamiltonian is a mathematical function used to describe a physical system in terms of its energy [1]. For any given configuration of the system, the Hamiltonian calculates the energy associated with that state. In quantum annealing, the full time-dependent Hamiltonian is expressed as:

$$H_s(s) = -\frac{1}{2} \sum_i \Delta(s) \sigma_i^x + \mathcal{E}(s) \left(- \sum_i h_i \sigma_i^z + \sum_{i < j} J_{ij} \sigma_i^z \sigma_j^z \right)$$

Here:

- σ_i^x and σ_i^z are Pauli matrices,
- $\Delta(s)$ and $\mathcal{E}(s)$ are annealing schedule functions that vary with time parameter s ,
- h_i are local biases,
- J_{ij} are couplings between qubits i and j .

While Hamiltonians are often associated with quantum systems, they also apply in classical physics. For example, consider a simple classical system where an apple is resting on a table. The apple can be in two states, either sitting on the table or having fallen to the floor. The Hamiltonian would assign a higher energy to the state where the apple is on the table, since it's elevated, and a lower energy to the state where the apple is on the floor.

In the context of quantum annealing, the Hamiltonian evaluates the energy of a specific arrangement of qubits [2]. The goal is to find the lowest energy configuration of the qubits according to this Hamiltonian. That configuration represents the optimal solution to the problem.

The quantum annealing process involves two main Hamiltonians, the initial Hamiltonian and the problem Hamiltonian [2]. The initial Hamiltonian represents a system where all qubits are in a superposition of both 0 and 1. The problem Hamiltonian, also called the final Hamiltonian, contains the actual problem encoded through the biases and couplings which are specified by the user [3]. These values shape the energy landscape and define what an optimal solution looks like.

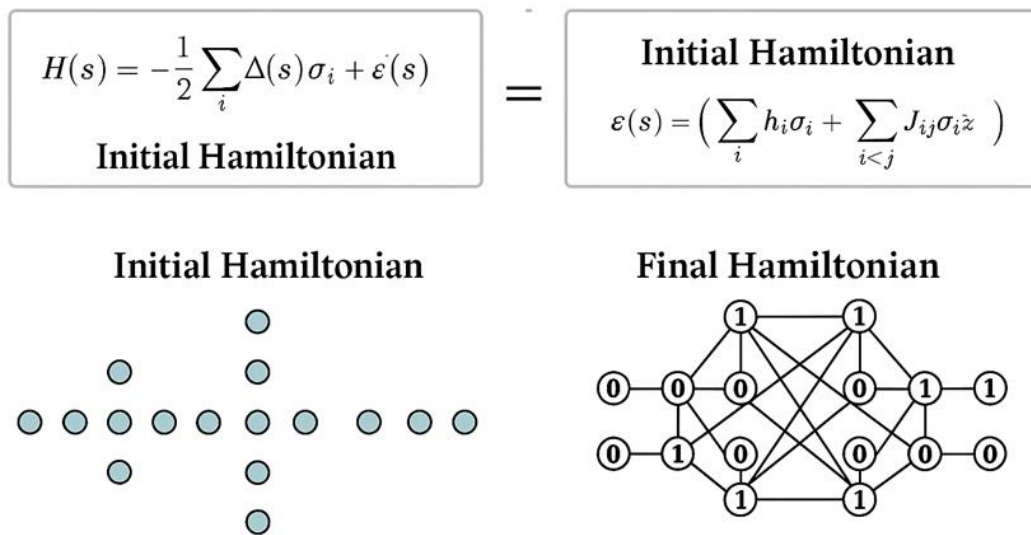


Figure 1.4.1 – Evolution from Initial to Problem Hamiltonian in Quantum Annealing

During annealing, the system is gradually transformed from the initial Hamiltonian to the problem Hamiltonian [3]. At the start, the problem Hamiltonian has no influence, and the system sits entirely in the superposition state. As the annealing progresses, the influence

of the problem Hamiltonian is increased while the initial Hamiltonian is slowly “turned down” [2]. Ideally, this smooth transition allows the system to remain in its lowest energy state throughout the process. By the end, the system has shifted fully to the problem Hamiltonian, and the qubits settle into classical 0 or 1 values that represent the solution. To visualize how energy levels change during the annealing process, physicists use a concept called the eigen spectrum [1]. This is basically a graph showing the energy of all possible quantum states over time as the system evolves. The lowest line on this graph represents the ground state, the lowest energy configuration, while higher lines represent excited states with more energy.

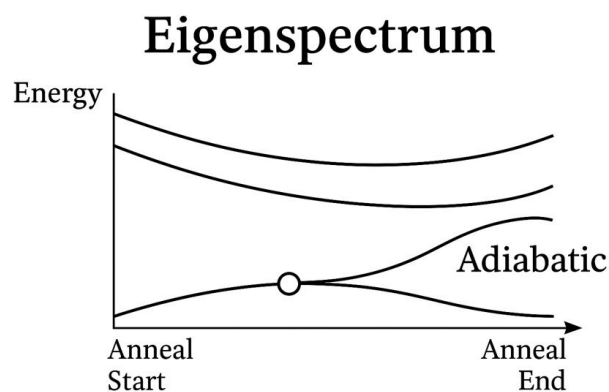


Figure 1.4.4 – Eigenspectrum During Quantum Annealing

At the beginning of the anneal, the ground state is clearly separated from all other energy levels. However, as the problem Hamiltonian is introduced, some of the excited states move closer to the ground state [3]. The narrowest point of separation between these energy levels is known as the minimum gap, or anticrossing. If the system moves too quickly through this point, or if it’s affected by external thermal noise, it may jump into a higher energy state, which results in a suboptimal solution [2].

To avoid this, the system must transition slowly enough to stay in the ground state, a process known as an adiabatic transition [1]. This principle gives its name to adiabatic quantum computing, which is closely related to quantum annealing.

In real-world scenarios, especially with large and complex problems, it is difficult for the system to stay in the absolute ground state throughout the entire annealing process [2]. However, even when the solution ends up in a slightly higher energy state, it can still be

valuable, because it can still be better than what classical solvers can find in the same time frame.

In addition, every unique problem has its own Hamiltonian and corresponding eigen spectrum, meaning that the structure of the energy landscape changes with each new task [1]. Some problems will have wide energy gaps and be easier to solve, while others will have small gaps that increase the risk of jumping to higher energy states, making them significantly harder [3].

Overall, the Hamiltonian provides the blueprint for energy in a quantum system, while the eigen spectrum helps us visualize how that energy evolves over time [1]. These concepts are crucial for understanding how and why quantum annealing works and why some problems are more difficult than others. Mastering these ideas sets the foundation for measuring quantum behaviour and assessing the success of quantum computations.

1.5 Boolean Satisfiability and Its Role in Quantum Optimization

One of the most fundamental and widely studied problems in computer science is the Boolean satisfiability problem (SAT) [9,10]. A SAT problem asks whether there exists an assignment of truth values (true or false) to a set of Boolean variables that will make a given logical formula evaluate to true. Although SAT seems quite simple, in fact, is considered to be very important in computational complexity theory and has many applications in areas such as artificial intelligence and optimization [9].

A SAT formula is usually written in Conjunctive Normal Form (CNF), where the overall formula is a conjunction (AND) of clauses, and each clause is a disjunction (OR) of literals [9]. A literal is simply a Boolean variable or its negation. For example: $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4)$.

In this formula there are two clauses. We call this satisfiable if we can assign values to the variables so that the formula evaluates to true. In case there are not then the problem would be considered unsatisfiable.

SAT also was the first problem proven to be NP-complete, as established by the Cook-Levin Theorem in the early 1970s [10]. This means that any problem that belongs in the complexity class of NP can be transformed into a SAT problem in polynomial time. Because of this, SAT is often used as a starting point for solving a wide range of computational problems [10]. A very important breakthrough that remains one of the most fundamental open questions in the field is that if we can find a general and efficient solution to SAT, then we could imply efficient solutions for all NP problems [10].

In practice, there are many classical tools for solving SAT problems. Solvers like MiniSAT, Glucose, and MapleSAT are designed to work directly with SAT formulas and use techniques like clause learning and backtracking to search for solutions efficiently [11]. In this project we use MiniZinc and Clingo, which are higher-level tools. MiniZinc is a flexible modelling language for constraint satisfaction problems [12], while Clingo is based on logic programming and answer set solving [13]. Both allow for more expressive ways to represent problems like SAT. While these tools are very effective in many cases, they can still struggle with very large or highly complex instances, where finding a solution or proving that no solution exists may require exponential time [11,12,13].

This limitation is one of the reasons for exploring quantum computation as an alternative approach. Although quantum computers do not currently offer exponential speedups for general NP-complete problems like SAT [1], they operate using different principles that may offer advantages in specific cases [2,3]. One such approach is quantum annealing, which is designed to find low-energy configurations of a quantum system [3]. If a SAT problem can be reformulated as an energy minimization problem, specifically as a Quadratic Unconstrained Binary Optimization (QUBO) model, it becomes possible to solve it using a quantum annealer [4].

It is important to clarify that current quantum annealers do not outperform classical solvers across all problem types, neither they provide a guaranteed solution to every SAT instance [2]. However, they offer a new and promising computational approach that is particularly well-suited for combinatorial optimization and constraint satisfaction [3]. By studying how SAT problems behave on quantum hardware, and how different encodings

and embeddings influence solution quality, this work aims to contribute to the growing body of research exploring the intersection of logic, optimization, and quantum computing.

Chapter 2

Foundations of Quantum Annealing for SAT

2.1 Introduction	13
2.2 From SAT to QUBO	14
2.3 Binary Quadratic Models (BQM)	16
2.4 Minor Embedding	18
2.5 Chimera and Pegasus Graph Topologies	20
2.6 D-Wave Solvers and Tools	21
2.7 Summary	22

2.1 Introduction

To fully understand the methods and experiments presented in this thesis, it is important to first explore the core concepts that establish the process of solving satisfiability problems on quantum computers. This chapter introduces the main ideas, models, and tools used throughout the pipeline, from transforming a logical SAT problem into a quantum-compatible format, to embedding it onto quantum hardware for processing.

We begin by explaining how Boolean satisfiability problems (SAT), typically expressed in Conjunctive Normal Form (CNF), are converted into Quadratic Unconstrained Binary Optimization models (QUBO) [4,14]. These QUBO problems are then translated into Binary Quadratic Models (BQM), which are the standard input for D-Wave’s quantum solvers [5]. To run such models on real quantum hardware, a step known as minor embedding is required. Minor embedding basically maps logical variables to the physical layout of qubits [6,15]. We also introduce the Chimera graph structure, which defines the architecture of D-Wave’s processors and applies specific constraints on how problems are embedded and solved [16].

Alongside these concepts, we will present the key software tools used in the project, such as the D-Wave Ocean SDK, qbsolv, minorminer and hybrid solvers, and explain how they contribute to different stages of the solving process [5,17]. This chapter introduces the key ideas needed to understand the experiments and results that follow.

2.2 From SAT to QUBO

To be able to run SAT problems on a quantum annealer, we need to translate them to a form that quantum hardware can understand and process. This form is called Quadratic Unconstrained Binary Optimization (QUBO) model. In this section we explore what a QUBO model is, why it is so central in quantum optimization and how it provides a bridge from logical problems to forms that can be executed on a quantum annealer [4,14].

A QUBO problem is a mathematical formulation used to express optimization problems where the variables are binary and there are no hard or explicit constraints. Instead, all problem requirements are embedded into the objective function. The goal is to find an assignment of variables that minimizes this quadratic function, meaning that it minimizes the overall energy or cost of the system.

Formally, a QUBO model is defined as:

$$\text{minimize } y = x^T Q x$$

Here, x is a vector of binary variables and Q is a square matrix of real-valued coefficients. The product $x^T Q x$ computes a weighted sum of variable values and interactions between them. These weights determine the “energy” or the “cost” of any given configuration [14].

In simpler terms, each binary variable can have a cost, for example “choose this” or “don’t choose this”, and pairs of variables can also have costs depending on whether they are selected together. This means that the QUBO model is asking which variables we should set to 1 to minimize the total cost.

Furthermore, SAT problems ask if there is a way to assign truth values to variables so that a given logical formula becomes true. For example, can we make the formula $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4)$ evaluate to true? This type of problems is easy to state but can be very hard to solve, especially while the number of variables and clauses increases [9].

The connection to QUBO comes from the fact that many combinatorial optimization problems, including SAT, can be rewritten in a QUBO form [4]. This is valuable because quantum annealers are designed to solve problems that are naturally expressed as QUBO or its physics equivalent, the Ising model [2].

By transforming SAT into a QUBO model, we can leverage quantum hardware to search for satisfying assignments by minimizing an energy function. A solution with low energy, ideally the lowest, corresponds to a satisfying assignment of the original logical formula.

To encode a SAT problem into a QUBO model we need to follow some steps. Each logical clause is represented as a penalty function. If the clause is satisfied, the penalty is zero. However, if it is unsatisfied, the cost increases. After that, all the clause penalties are added to form a single quadratic cost function which will be minimized when the maximum number of clauses is satisfied. The result is converted into a matrix Q , with each entry corresponding to a variable (or a pair of variables) and its contribution to the total cost [14].

Consider a simple SAT clause, such as $x_1 \vee \neg x_2$. We want this clause to be true. The only assignment that makes it not true is if $x_1=0$ and $x_2=1$. So, we define a penalty function that evaluates to 1 if this condition is met, and 0 otherwise. In other words, we want to penalize the expression $(1-x_1)*x_2$. Expanding this we have $x_2-x_1*x_2$. Now this can be expressed in QUBO form as: linear term for x_2 is +1 and quadratic term for x_1*x_2 is -1. This means that the QUBO matrix Q is the following:

$$Q = \begin{bmatrix} 0 & -0.5 \\ -0.5 & 1 \end{bmatrix}$$

Since QUBO matrices are symmetric, we divide the quadratic term between the two corresponding off-diagonal entries.

By repeating this process for each clause in the SAT problem, and aggregating the corresponding penalty terms, we construct the full QUBO objective function.

The QUBO model is widely used because is highly flexible and can express a wide range of problems, everything from SAT, to knapsack problems, to clustering, to graph colouring [14]. It acts as a "bridge model" between classical and quantum solvers and is also used in classical metaheuristics and hybrid algorithms [5].

Its “unconstrained” nature is not a limitation. The constraints can be enforced using penalty terms which are added directly into the QUBO matrix. These penalty terms are a way to turn logic or structure into energy costs.

QUBO models are the main language of quantum annealers and provide a convenient way to express logical or optimization problems as energy minimization tasks. By converting SAT problems into QUBO form, we can tap into quantum techniques for finding solutions [2,5].

2.3 Binary Quadratic Models (BQM)

Binary Quadratic Models (BQM) are a widely used mathematical representation for expressing combinatorial optimization problems where each variable can take one of two binary values, usually 0 or 1. A BQM captures the relationship between these binary variables through a cost or energy function and the goal is to find the assignment of values that minimizes this function.

A BQM is generally written as:

$$E(x) = \sum_i h_i x_i + \sum_{i < j} J_{\{ij\}} x_i x_j$$

Where:

- $x_i \in \{0, 1\}$ are the binary decision variables
- h_i represents the linear bias for each variable x_i
- $J_{\{i,j\}}$ represents the interaction (quadratic coupling) between pairs of variables x_i and x_j
- And $E(x)$ is the total energy for a given assignment of variables

This format is extremely flexible and can model a wide range of optimization problems, from constraint satisfaction to machine learning and routing problems. In the context of quantum computing, and specifically quantum annealing as implemented by D-Wave, BQMs serve as the primary format for problem input. D-Wave solvers work with BQMs, and tools provided by D-Wave (such as the Ocean SDK) which are built around this model [18].

Usually, problems are initially formulated in QUBO form due to its simplicity. Even though QUBOs and BQMs are mathematically equivalent, the BQM format has the advantage of flexibility. Meaning that is easier to work with BQMs for advanced tools like hybrid solvers, embedding engines and quantum samplers. For this reason, problems are often initially formulated as QUBOs due to their simplicity and then converted to BQMs for practical usage.

In this thesis, SAT problems are first encoded into QUBO form and then converted into BQM using D-Wave's Ocean Python tools. This is done through a simple interface:

```
from dimod import BinaryQuadraticModel  
bqm = BinaryQuadraticModel.from_qubo(Q)
```

This conversion maintains the same cost structure while adapting the problem to a format that is more solver friendly. This step is important for leveraging D-Wave's solvers and tools, such as minorminer, and it is performed before any sampling or embedding takes place.

2.4 Minor Embedding

When working with quantum annealers, one of the challenges is the process of translating a problem defined in a Binary Quadratic Model (BQM) into a form that can be executed on the quantum hardware. This process is known as minor embedding. It involves mapping a logical graph, which in our case represents the abstract structure of a problem, onto a physical graph that reflects the hardware's qubit connectivity [19].

In a BQM, variables are connected through quadratic interactions. These interactions can be visualized as edges between nodes in a graph. Each node represents a binary variable, and each edge corresponds to a quadratic interaction. This graph is commonly known as the logical graph. However, the quantum annealer has its own physical graph based on its qubit layout and connectivity, such as the Chimera or Pegasus topology. Since the hardware connectivity is limited, the logical graph often cannot be mapped directly onto the hardware. This is where minor embedding comes in use [15].

Minor embedding is the process of mapping each node of the logical graph to a connected chain of qubits in the physical graph. These chains of physical qubits are strongly coupled so that they behave as a single logical qubit. The strength of these connections is called chain strength. Choosing the right chain strength is important. If it's too weak, the chain may break during annealing, and if it's too strong, it can distort the energy landscape and lead to suboptimal solutions [6].

A coupler refers to a physical connection between two qubits on the hardware. These couplers enforce the interactions that were defined by the problem in the BQM format. During embedding, couplers are also used to connect qubits within a chain, ensuring that they represent a single logical variable.

The process of embedding is typically handled by D-Wave's tools like minorminer, which attempts to automatically find an efficient embedding of the logical graph onto the hardware topology [20]. The quality of an embedding depends on several factors, such as the number of physical qubits required, the length of chains and whether chains break during sampling.

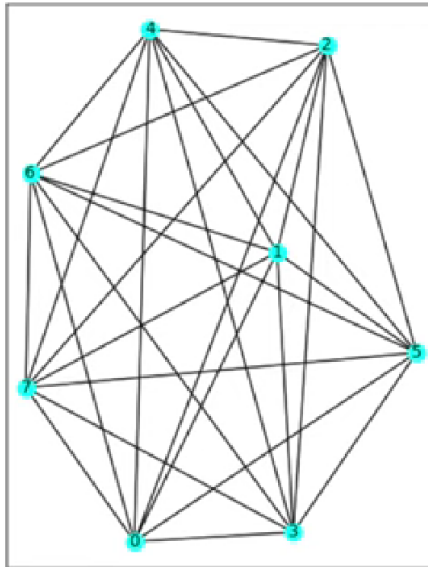


Figure 2.4.1 – Logical Graph Representation

This figure shows a fully connected logical graph with 8 nodes. Each node represents a binary variable in a Binary Quadratic Model (BQM), and the edges denote quadratic interactions between variables.

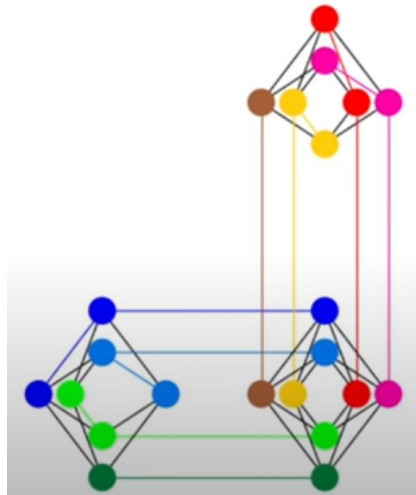


Figure 2.4.2 – Minor Embedding onto Chimera Topology

This figure illustrates how the logical graph from Figure 2.4.1 is embedded onto a Chimera graph. Each colour represents a chain of physical qubits that collectively encode a single logical variable. The connections between chains implement the quadratic terms of the original BQM.

Understanding minor embedding is essential, as it directly impacts whether a problem can be executed on the hardware. If no embedding can be found due to graph structure or qubit limitations, the problem cannot be solved on the annealer.

This embedding step connects the abstract formulation of a SAT problem (via BQM) to the concrete execution on quantum hardware. In the next section, we will look more closely at the hardware itself, specifically the Chimera and Pegasus graph topologies used by D-Wave and explore how their structures shape the embedding process.

2.5 Chimera and Pegasus Graph Topologies

In quantum annealing, the physical layout of qubits and their topology is a critical part of how problems are executed on the quantum hardware. D-Wave has developed two major topologies for their quantum processing units (QPUs), the Chimera and Pegasus topology. These topologies play a crucial role in how logical problems are embedded onto hardware, making it important to understand their structure [16].

The Chimera topology consists of a grid of unit cells. Each unit cell contains eight qubits arranged in a bipartite structure where four qubits are arranged vertically and four horizontally. Every vertical qubit in a cell is connected to all the horizontal qubits in the same cell and to neighbouring qubits in adjacent cells. This creates a sparse but regular lattice structure [16].

In the D-Wave 2000Q system, the Chimera architecture consists of a 16×16 grid of unit cells, totalling 2048 qubits. Although qubits are not all-to-all connected, this design allows efficient implementation of many practical problems, if those problems are properly embedded using minor embedding.

This is the topology that will be used in the experiments later in this thesis. All embedding and execution on the quantum hardware will be performed assuming the Chimera architecture, as it is the most accessible for our case and tools.

The Pegasus topology, used in D-Wave's newer Advantage system, significantly increases qubit connectivity. In Pegasus, each qubit is connected to up to 15 others (compared to just 6 in Chimera), making it easier to embed larger and more complex logical graphs with shorter chains [21].

The increased number of couplers allows for more efficient embeddings, reducing the likelihood of broken chains and making better use of the available hardware. This also improves solution quality for complex problems, particularly those with dense or irregular graph structures.

The transition from Chimera to Pegasus reflects to a broader push toward solving larger and harder problems using quantum annealing. However, every topology brings its own constraints and advantages, particularly when it comes to minor embedding. In Chimera, embedding often requires longer chains of physical qubits to represent a single logical variable. In Pegasus, better connectivity often reduces the chain length needed for embedding, which helps maintain solution quality and reduce computational noise [22]. The chosen topology impacts several things like, the quality of embeddings, the number of qubits required, the likelihood of chain breakage and the overall performance and feasibility of a computation.

2.6 D-Wave Solvers and Tools

D-Wave provides a suite of tools and libraries that make it easier to formulate, transform, and solve optimization problems on their quantum annealing systems. These tools support everything from high-level modelling of problems to low-level hardware interaction, and they play a central role in enabling practical use of quantum computing technologies [5].

The Ocean Software Development Kit (SDK) is D-Wave's main software platform, built in Python and offered as a collection of open-source packages [5]. It provides a set of tools for defining problems in QUBO or BQM formats, embedding them onto quantum hardware, executing anneals, and retrieving and analysing results. Ocean SDK simplifies

access to D-Wave systems and includes essential libraries like `dimod`, `dwave-system`, and `dwave-networkx`.

Minorminer is a heuristic algorithm developed by D-Wave to solve the minor embedding problem. It tries to find efficient embeddings that minimize chain lengths and reduce the likelihood of chain breaks during annealing [17]. This tool is important when working with dense graphs or problems with complex logical connectivity.

In D-Wave’s ecosystems, samplers are interfaces that return samples (possible solution) from a defined problem. Ocean supports different types of samplers. For example, quantum samplers access the quantum annealer directly to perform sampling [5]. Classical samplers, such as simulated annealing, use classical computation to approximate low energy states. Hybrid samplers combine classical and quantum resources to scale to larger problems and improve solution quality [23]. Hybrid solvers are particularly useful for large or structured problems that cannot fit entirely on the quantum processor. They manage problem decomposition and leverage classical processors for parts of the computation.

D-Wave also provides a tool called `qbsolv` which is specialized to break large QUBO problems into smaller subproblems that can either be solved classically or submitted to a quantum annealer [17]. This makes it possible to approach problems that are otherwise too large for direct quantum processing. The tool iteratively solves and reassembles parts of the problem until it converges to a global solution.

These tools will be used throughout this thesis in the transformation, embedding, and solving stages of SAT-to-QUBO pipelines. Each provides a unique set of capabilities, from direct access to quantum hardware to pre- and post-processing of complex problems.

2.7 Summary

In this chapter, we introduced the key concepts and tools that form the foundation of this thesis. We began by exploring how SAT problems, typically expressed in CNF, can be

translated into QUBO and then BQM models suitable for quantum annealing. We discussed how these models can be visualized as logical graphs and the importance of minor embedding in mapping them onto the physical architecture of quantum hardware. Special attention was given to the Chimera topology, which is used in the experiments of this thesis, and to D-Wave's solvers and software tools like Ocean SDK, minorminer, and hybrid samplers. These elements are essential for transforming, embedding, and solving problems on quantum systems. With this theoretical and technical background in place, we are now ready to move on to the practical implementation and experimentation phases presented in the next chapters.

Chapter 3

Early Experiments with D-Wave Quantum Annealer

3.1 Introduction	24
3.2 SAT Instance Generation	25
3.3 Basic QUBO Submission	25
3.4 Full SAT-to-QPU Execution	29
3.5 Observations and Summary	31

3.1 Introduction

This chapter presents a series of early experiments conducted to explore the full pipeline of solving Boolean satisfiability problems using quantum annealing. The aim of these initial implementations was to validate the transformation process from SAT to QUBO and BQM, to test the embedding onto the Chimera topology, and to execute problem instances on a quantum annealer using D-Wave’s tools [5].

Through a collection of Python scripts developed during this phase, we tested SAT generation, QUBO construction, problem embedding, and sampling using D-Wave’s Ocean SDK. These small-scale examples served as a foundational proof of concept for the rest of the thesis. While limited in size, they provided hands-on experience with the end-to-end flow of quantum SAT solving and helped shape the more advanced simulations and analysis presented in later chapters.

3.2 SAT Instance Generation

To test the transformation pipeline, we first needed SAT instances. The `3satgenerator.py` script was created to randomly generate 3-SAT problems in Conjunctive Normal Form (CNF). These instances consisted of randomly selected clauses, each with exactly three literals (see Figure 3.2.1).

```
import random

def generate_clause(variables):
    return [random.choice(variables) * random.choice([1, -1]) for _ in range(3)]

def generate_3sat(num_vars, num_clauses):
    variables = list(range(1, num_vars + 1))
    clauses = [generate_clause(variables) for _ in range(num_clauses)]
    return clauses
```

Figure 3.2.1 – Python code used to generate 3-SAT CNF instances.

This script randomly constructs 3-literal clauses and outputs formulas in DIMACS .cnf format for use in the SAT-to-QUBO transformation pipeline.

The output was written to .cnf files following the DIMACS format, which could then be loaded for transformation into QUBO models. These generated instances allowed us to test arbitrary problem sizes and clause structures, crucial for evaluating the capabilities and limitations of D-Wave's solvers.

3.3 Basic QUBO Submission

The script in Appendix A served as our first practical test of submitting a custom QUBO model to the quantum annealer. It was designed to verify that we could correctly construct a Binary Quadratic Model (BQM), submit it to D-Wave's QPU, and retrieve meaningful output. This simple experiment provided valuable insights into the interaction between logical variable representations and the hardware's physical qubit architecture.

The code defines a QUBO model that encodes the logical SAT clause: $(x_1 \vee \neg x_2 \vee x_3)$ using three binary decision variables (x_1, x_2, x_3) and one auxiliary variable (z). This clause is violated only by the assignment ($x_1=0, x_2=1, x_3=0$). The encoding penalizes this assignment using a quadratic penalty function, which is implemented via an auxiliary variable and a large penalty constant MMM . This encoding ensures that the clause is satisfied in low-energy states by assigning higher energy to invalid solutions.

The QUBO is constructed as a dictionary, where each key represents a pair of variables and the value represents the corresponding linear or quadratic coefficient in the QUBO matrix. The entries encode both logical preferences and the constraint mechanism via the auxiliary variable. The full encoding is shown in Figure 3.3.1. The values in the QUBO dictionary are derived based on the clause's penalty logic. The term $(1 - x_1) \cdot x_2 \cdot (1 - x_3)$ evaluates to 1 only when the clause $(x_1 \vee \neg x_2 \vee x_3)$ is violated. This logic is transformed into a QUBO form using the standard method of introducing an auxiliary variable z to represent the AND constraint, and penalizing its incorrect activation. The penalty terms (e.g., (0, 1): $-M$, (0, 3): $M/2$, etc.) arise from expanding this product into a quadratic polynomial and distributing its terms symmetrically in the QUBO matrix. Each value in the dictionary corresponds to either a linear term (e.g., (1,1): $1+M$ for x_2) or an interaction (e.g., (1,3): $M/2$ linking x_2 and z) that contributes to the total cost function.

```

from dimod import BinaryQuadraticModel
from dwave.system import EmbeddingComposite, DWaveSampler

# Penalty constant
M = 10

# QUBO definition (4 variables: x1, x2, x3, z)
qubo = {
    (0, 0): 0,
    (1, 1): 1 + M,
    (2, 2): 0,
    (3, 3): 2 * M,
    (0, 1): -M,
    (0, 3): M / 2,
    (1, 3): M / 2,
    (2, 3): 1 / 2,
    (0, 2): 1 / 2,
    (1, 2): -1 / 2
}

```

Figure 3.3.1 – QUBO Encoding of SAT Clause with Constraint Penalty Term

This code defines a QUBO that encodes the SAT clause $(x_1 \vee \neg x_2 \vee x_3)$ using penalty terms and an auxiliary variable. The numerical values in the dictionary reflect the quadratic formulation of the clause's violation condition.

Once the QUBO is defined, it is converted into a Binary Quadratic Model (BQM), submitted to the quantum sampler via the EmbeddingComposite wrapper, and results are printed, as illustrated in Figure 3.3.2. This part of the script ensures the logical graph is embedded onto the Chimera architecture and that meaningful results are collected.

```

# Convert to BQM
bqm = BinaryQuadraticModel.from_qubo(qubo)

# Submit to D-Wave quantum sampler
sampler = EmbeddingComposite(DWaveSampler())
response = sampler.sample(bqm, num_reads=100)

# Print results
print("Solutions:")
for sample, energy in response.data(['sample', 'energy']):
    print(f"Sample: {sample}, Energy: {energy}")

```

Figure 3.3.2 – BQM Conversion, D-Wave Sampling, and Output Display

This part of the script converts the QUBO into a Binary Quadratic Model (BQM), submits it to the quantum annealer, and prints out sampled results with energy values.

This script does a few important things:

- It builds a small QUBO that includes both logical interactions and a constraint encoded through a penalty method.
- It transforms the QUBO into a BQM, the format expected by D-Wave’s solvers.
- It uses the EmbeddingComposite wrapper to automatically embed the logical graph onto the Chimera architecture of the quantum hardware.
- It samples the BQM using the quantum annealer, collecting results from 100 reads to observe the frequency of low-energy states.

The output from this run is a set of sampled binary configurations, each accompanied by its associated energy. A representative segment of these results is shown in Figure 3.3.3.

```

Sample: [0, 0, 1, 1], Energy: -1.25
Sample: [0, 1, 1, 1], Energy: 0.25
Sample: [1, 0, 1, 0], Energy: -0.75
Sample: [1, 1, 1, 0], Energy: 0.5

```

Figure 3.3.3 – Sampled Binary Solutions with Energies Returned by D-Wave

Each sample represents a binary configuration of the variables [x1,x2,x3,z] and the corresponding energy of the solution as returned by the quantum annealer. These

configurations are evaluated against the clause $(x_1 \vee \neg x_2 \vee x_3)$, which is violated only when $x_1=0, x_2=1, x_3=0$. The auxiliary variable z is used to penalize this specific case. Low-energy samples (e.g., $[0,0,1,1]$) are preferred because they satisfy the clause and respect the penalty constraint.

Here, the lowest energy solutions (such as $[0,0,1,1]$ or $[1,0,1,0]$) are the most desirable, as they represent configurations that both satisfy the logical clause $(x_1 \vee \neg x_2 \vee x_3)$ and avoid the penalized assignment $(0,1,0)$. These results confirm that the auxiliary constraint was correctly encoded into the QUBO and that the quantum annealer identified valid, low-energy solutions.

3.4 Full SAT-to-QPU Execution

The script in Appendix B implements the full pipeline: starting from a SAT problem written in CNF format, converting it to a Binary Quadratic Model (BQM), embedding that model onto the hardware's qubit architecture, and submitting it to the quantum annealer for sampling. This code demonstrated that the conceptual SAT-to-QPU workflow could be implemented with actual tools and run on quantum hardware.

The pipeline executed follows these main steps:

1. Load a CNF file (in DIMACS format).
2. Convert the CNF to a CSP (Constraint Satisfaction Problem) using the `dwavebinarycsp` library.
3. Stitch the CSP into a BQM using penalty terms.
4. Find a minor embedding using the `minorminer` heuristic.
5. Embed and sample the problem using D-Wave's `FixedEmbeddingComposite`.

Figure 3.4.1 shows the first part of the code, where the SAT instance is read from a CNF file and converted to a BQM.

```

import dwavebinarycsp as dbcsp
import dimod
import minorminer
from dwave.system import DWaveSampler, FixedEmbeddingComposite
from dwave.preprocessing import ScaleComposite

# Load SAT instance from file and convert to CSP
with open('testSAT3.cnf', 'r') as fp:
    csp = dbcsp.cnf.load_cnf(fp)

print("LOADED CNF")

# Convert CSP to BQM
bqm = dbcsp.stitch(csp)
print("CONVERTED CSP to BQM")
print("BQM:", bqm)

```

Figure 3.4.1 – CNF-to-BQM Code Segment

Next, the embedding is generated and submitted to the QPU using a fixed embedding, as illustrated in Figure 3.4.2.

```

# Initialize D-Wave sampler
qpu = DWaveSampler()
embedding = minorminer.find_embedding(
    dimod.to_networkx_graph(bqm), qpu.to_networkx_graph()
)

# Sample from QPU using a fixed embedding
sampleset = FixedEmbeddingComposite(ScaleComposite(qpu), embedding=embedding).sample(
    bqm, num_reads=5000
)

print("Lowest Energy:", sampleset.first.energy)

```

Figure 3.4.2 – Embedding and Sampling from QPU

A representative snapshot of the resulting BQM is shown in Figure 3.4.3. This includes linear, quadratic, and offset terms, along with the binary var type.

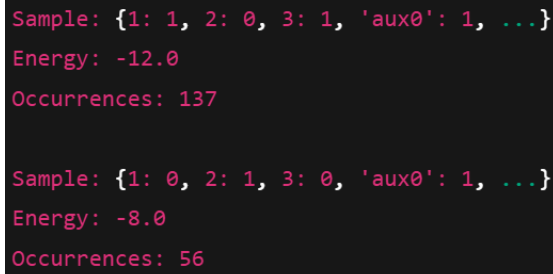
```

BinaryQuadraticModel(
  linear={1: -4.0, 2: -4.0, 3: -4.0, 'aux0': 8.0, ...},
  quadratic={(1, 2): 4.0, ('aux0', 1): -4.0, ...},
  offset=16.0,
  vartype='BINARY'
)

```

Figure 3.4.3 – Example BQM Output from the Script

Once submitted to the QPU, the annealer returned a set of sampled solutions, two of which are shown in Figure 3.4.4.



```
Sample: {1: 1, 2: 0, 3: 1, 'aux0': 1, ...}  
Energy: -12.0  
Occurrences: 137  
  
Sample: {1: 0, 2: 1, 3: 0, 'aux0': 1, ...}  
Energy: -8.0  
Occurrences: 56
```

Figure 3.4.4 – Sampled Output and Energy Results

These results reveal several key observations:

- The lowest-energy configuration (-12.0) corresponds to a fully satisfied solution, as it avoids all clause violations encoded in the QUBO.
- Multiple samples were returned, with lower-energy states appearing more frequently, as expected in quantum annealing.
- All variables, including auxiliary ones, were assigned binary values (0 or 1), each contributing to the total energy based on their role in the QUBO formulation.

3.5 Observations and Summary

The early experiments conducted in this thesis served as a foundational proof-of-concept, demonstrating that the SAT-to-QPU pipeline could be executed successfully on real quantum hardware using D-Wave’s platform. Starting from randomly generated CNF formulas, we were able to construct QUBO models, convert them into BQMs, and submit them to the quantum annealer for sampling. Using tools such as D-Wave’s Ocean SDK, StitchSolver, and EmbeddingComposite, we confirmed that the full transformation and execution flow could be implemented using accessible, open-source software.

Although the input problems were small in size, these tests verified that the various components of the pipeline, including encoding, embedding, and sampling, worked correctly together.

First, we note that automated embedding via `EmbeddingComposite` removed the need for manual qubit mapping. Second, the successful runs showed that small-scale logical structures could be embedded and solved without chain breakages, likely due to the limited size and simplicity of the logical graphs used. Finally, the experiments highlighted the importance of problem size and structure, even at this early stage, it became clear that the scalability of quantum annealing depends heavily on how well a problem's structure fits within the hardware's physical constraints.

These initial implementations laid the groundwork for the more detailed analysis presented in later chapters. Although limited by the number of variables and clauses that could be handled, they provided critical validation of the overall approach and helped shape the direction of the thesis moving forward. The next phase of this work focuses on a deeper investigation into embedding feasibility, solver behaviour, and the influence of SAT instance properties on performance.

Chapter 4

Experimental Framework and Solver Implementation

4.1 CNF Instance Generation	33
4.2 Experiments with QBSolv and Classical Solvers	36
4.3 Hybrid Solver Evaluation	40

4.1 CNF Instance Generation

To support the experimental parts of this thesis, the Python script in Appendix C was developed to efficiently create SAT problem instances in Conjunctive Normal Form (CNF). It provides three distinct modes of generation: random clause construction, CNFgen-assisted generation, and controlled variable frequency generation. Each method was included to support specific aspects of the analysis carried out in later chapters of the thesis.

The random method generates SAT formulas composed of a mix of 2-SAT, 3-SAT, and 4-SAT clauses. The user specifies the number of variables and clauses, while the script randomly allocates how many of each clause type to generate. This approach provides general-purpose CNF files with natural variability. The clause allocation logic used in this process is shown in Figure 4.1.1.


```

num_2sat = random.randint(0, num_clauses)
num_3sat = random.randint(0, num_clauses - num_2sat)
num_4sat = num_clauses - (num_2sat + num_3sat)

for _ in range(num_2sat):
    clauses.append(Clause.random(num_variables, 2))
for _ in range(num_3sat):
    clauses.append(Clause.random(num_variables, 3))
for _ in range(num_4sat):
    clauses.append(Clause.random(num_variables, 4))

```

Figure 4.1.1 – Clause allocation logic for random 2-SAT, 3-SAT, and 4-SAT clauses

Once generated, each clause is written in DIMACS format. A sample run of this generation method is illustrated in Figure 4.1.2, which shows the user input and output in the terminal.

```

$ python generate_cnf.py
Choose method (random/cnfgcn/controlled): random
Enter number of variables: 10
Enter number of clauses: 30
✅ Random CNF file 'generatedSAT.cnf' generated.

```

Figure 4.1.2 – Example of terminal output when running the random generation method

This mode was useful in the initial development of the experimental framework, allowing for quick generation of test problems to validate transformation and embedding tools.

The second method uses CNFgen [24], a powerful command-line tool that produces benchmark SAT instances based on well-known theoretical formulas such as pigeonhole principles (php), random k-CNF (randkcnf), and other structured families. This functionality is integrated into the generator via subprocess calls inside the script. CNFgen is widely used in SAT research for generating reproducible and tunable formulas.

The script supports multiple CNFgen formula types, though this thesis primarily used the randkcnf family. An example terminal interaction is shown in Figure 4.1.3, where the user generates a 3-SAT instance with 50 variables and 200 clauses.

```

$ python generate_cnf.py
Choose method (random/cnfgcn/controlled): cnfgcn
Enter number of variables: 20
Enter CNFgen formula type (e.g., php, randkcnf): randkcnf
Enter any extra arguments for CNFgen (leave empty for defaults):
3 50 200
✅ CNFgen formula 'randkcnf' generated in 'generatedSAT.cnf'.

```

Figure 4.1.3 – CNFgen-based terminal interaction producing a randkcnf formula

This CNFgen integration allowed the creation of standardized, reproducible instances for use in later embedding performance comparisons.

The third method allows the user to bias the occurrence frequency of selected variables. This is particularly important in Chapter 6, where the goal is to study, among other things, how the frequency of variable appearances affects the embedding process [6,20]. A subset of variables is defined as "high frequency," and these variables are inserted more often than others during clause construction.

Clauses are built using random literals, but with a 50% chance of including a selected high-frequency variable. Figure 4.1.6 shows a terminal run where the user selects high-frequency variables and their intended reuse frequency. The script ensures these variables appear more often during clause generation, while the output remains in DIMACS format for compatibility with the rest of the pipeline.

```

$ python generate_cnf.py
Choose method (random/cnfgcn/controlled): controlled
Enter number of variables: 15
Enter number of clauses: 60
Enter high-frequency variables (comma-separated): 3,5,7
Enter how many extra times these variables should appear: 5
✅ Controlled CNF file 'generatedSAT.cnf'
generated with high-frequency variables: [3, 5, 7]

```

Figure 4.1.4 – Terminal interaction for controlled CNF generation with specified high-frequency variables

This controlled structure is instrumental in experiments analysing minor embedding efficiency, chain length, and success rate, based on variable occurrence bias.

4.2 Experiments with QBSolv and Classical Solvers

In this section, we explore the use of classical and hybrid solvers from the D-Wave Ocean ecosystem, with a particular focus on QBSolv. These experiments aim to analyse how satisfiability problems, encoded as Binary Quadratic Models (BQMs), can be efficiently solved using classical methods when quantum execution is not available or practical. The primary tool used in this section is the script in Appendix D, which was developed to evaluate the performance of different classical samplers, Tabu Search, Simulated Annealing, and Steepest Descent, on a variety of CNF instances generated via the script from Appendix C.

QBSolv is a decomposition-based solver developed by D-Wave to handle QUBO or BQM problems that are too large to be embedded directly on the quantum annealer. Instead of submitting the whole problem at once, QBSolv breaks it into smaller subproblems, solves each one (classically or on a QPU), and then reassembles the solutions through iterative refinement. This makes it a powerful hybrid solver for large SAT instances mapped to BQMs.

Its key strength lies in its ability to work seamlessly with existing tools in the Ocean SDK and offer consistent performance on mid-sized problems.

In addition to QBSolv, we briefly explored PyQUBO, a high-level modelling framework that allows problems to be defined symbolically in Python before being compiled into QUBO or BQM format. PyQUBO supports logical constraints, mathematical expressions, and variable relationships with remarkable flexibility. A typical PyQUBO usage example is shown in Figure 4.2.1.

```

from pyqubo import Array, Constraint, solve_qubo

# Define binary variables
x = Array.create("x", shape=3, vartype="BINARY")

# Define objective function
H = Constraint((x[0] + x[1] - 1)**2, label="clause1") + \
    Constraint((x[1] + x[2] - 1)**2, label="clause2")

# Compile to QUBO
model = H.compile()
qubo, offset = model.to_qubo()

# Solve using QBSolv
response = QBSolv().sample_qubo(qubo)

```

Figure 4.2.1 – Building and solving a QUBO model using PyQUBO

While PyQUBO was not the main tool used in the experiments described here, it is a valuable part of the Ocean stack and suitable for structuring custom constraint-based models when needed.

The core experimental script loads CNF instances, transforms them into BQMs using `dwavebinarycsp`, and evaluates them using three classical samplers:

- TabuSampler (tabu search)
- SimulatedAnnealingSampler (thermal sampling)
- SteepestDescentSolver (gradient-based local optimization)

The relevant core logic for this pipeline is presented in Figure 4.2.2.

```

from dimod import TabuSampler, SimulatedAnnealingSampler, SteepestDescentSolver
from dwavebinarycsp import cnf

with open(cnf_path, 'r') as fp:
    csp = cnf.load_cnf(fp)

bqm = stitch(csp) # Convert CNF to BQM

# Sample using all solvers
for sampler_class, reads in [(TabuSampler, 3000), (SimulatedAnnealingSampler, 5000), (SteepestDescentSolver, 5000)]:
    sampler = sampler_class()
    response = sampler.sample(bqm, num_reads=reads)
    print(f"{sampler_class.__name__} Energy:", response.first.energy)

```

Figure 4.2.2 – Main sampling loop using classical solvers

This script is designed to benchmark solver performance across CNFs of increasing complexity.

Using the code in Appendix C, we created several random CNF instances of varying size and ran them through the code in Appendix D. The results are shown in Figure 4.2.3.

CNF Instance (Vars-Clauses)	TabuSampler Energy	SimulatedAnnealing Energy	SteepestDescent Energy
10v-40c	5.0	5.0	5.0
20v-80c	2.0	2.0	2.0
30v-60c	0.0	0.0	0.0
20v-100c	12.0	12.0	12.0
40v-160c	8.0	8.0	9.0

Figure 4.2.3 – Average energy values returned by each solver across CNF instances

The energy values in Figure 4.2.3 represent the lowest energies obtained by each solver when applied to the corresponding CNF-derived BQM. In this encoding, an energy of 0.0 indicates a fully satisfied assignment in which no constraints are violated, that is an optimal solution.

For instances where the energy is greater than 0.0, this typically means one of two things:

- The solver found a suboptimal solution, meaning not all constraints were satisfied;
- or
- The CNF instance may be inherently unsatisfiable, and the energy reflects the best possible assignment with minimal violations under the penalty model.

Thus, only those rows with energy 0.0 can be confirmed to correspond to optimal solutions, while others likely represent best-effort approximations by the solver.

As expected, larger or denser CNF instances often lead to higher energy configurations. All solvers returned consistent values across runs, with Steepest Descent occasionally landing in slightly worse local minima compared to Tabu or Simulated Annealing. The resulting energy trends are also visualized in Figure 4.2.4.

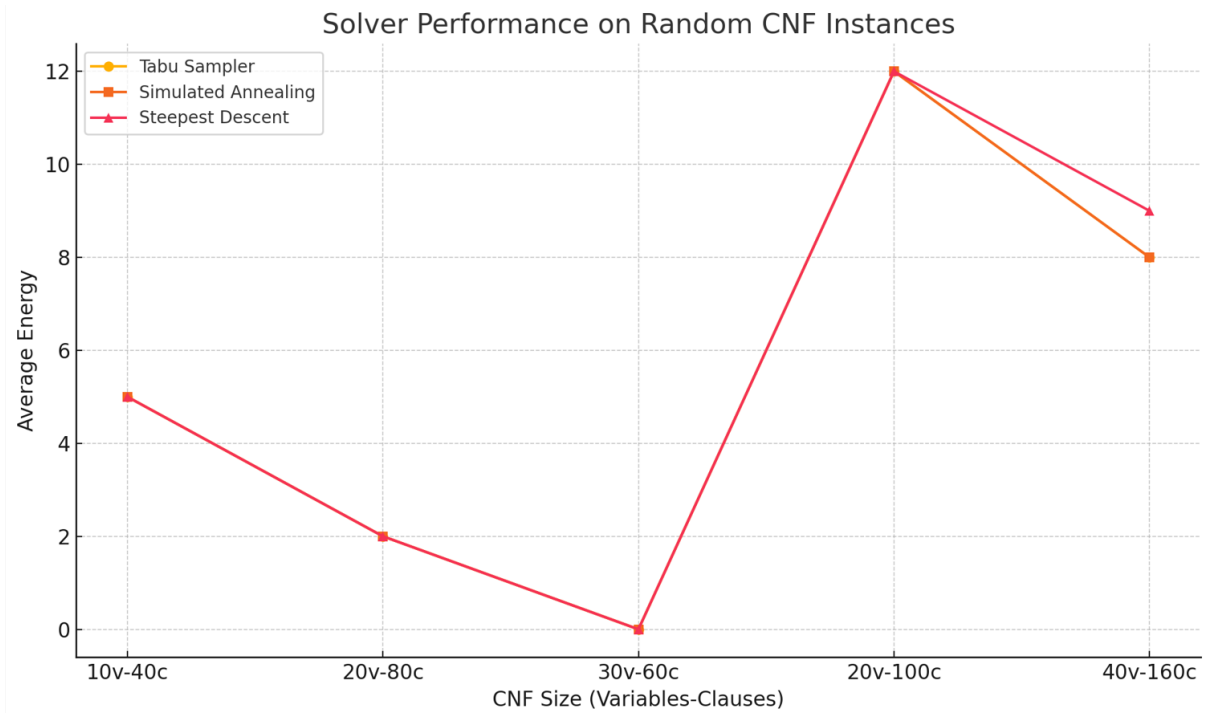


Figure 4.2.4 – Solver performance by CNF size and energy

The experiments conducted with the code in Appendix D demonstrate that:

- Classical samplers from the Ocean SDK are capable of reliably solving small to mid-sized BQMs derived from SAT problems [5].
- Tabu Search and Simulated Annealing often achieve similar energies, while Steepest Descent is slightly more prone to local optima.
- QBSolv is effective at stitching and solving problems without requiring manual tuning of solver parameters [17].
- The energy returned by each sampler provides insight into how well the original logical constraints were satisfied.

These classical methods allow the SAT-to-BQM pipeline to be studied and tested even without QPU access, making them a crucial component for evaluating solver performance and validating the transformation workflow in a fully classical environment.

4.3 Hybrid Solver Evaluation

This section provides a detailed technical analysis of three distinct hybrid approaches to solving Boolean satisfiability (SAT) problems via D-Wave's hybrid computing infrastructure. The methods under consideration are implemented in the scripts in Appendix E, Appendix F and Appendix G. Each represents a different paradigm of decomposition, sampling, and composition strategies provided by D-Wave's hybrid library [25]. The analysis covers performance across SAT instances of increasing complexity, generated using the code in Appendix C, and includes comparisons and observations.

The first implementation uses a Loop workflow to iterate over a RacingBranches structure. The branches include:

- SimulatedAnnealingProblemSampler
- TabuProblemSampler

Each branch attempts to find a solution independently, and the results are compared using ArgMin, a component that selects the state with the lowest energy among competing branches. This ensures that only the best-performing solution is passed to the next iteration. The loop runs for a maximum of 200 iterations, progressively refining the solution.

This architecture, shown in Figure 4.3.1, promotes diversity in search and leverages multiple sampling heuristics to refine the BQM solution iteratively.

```
workflow = hybrid.Loop(  
    hybrid.RacingBranches(  
        hybrid.SimulatedAnnealingProblemSampler(),  
        hybrid.TabuProblemSampler(),  
    ) | hybrid.ArgMin(),  
    max_iter=200  
)
```

Figure 4.3.1 – Loop-based hybrid workflow

The second approach relies on `ComponentDecomposer`, a hybrid framework utility that breaks a BQM into connected components, each of which can be solved independently. This helps isolate clusters of related variables and enables parallel or localized processing.

The initial state is created by sampling a random binary assignment on the BQM using `random_sample`, and wrapping it into a `State` object. Then, the `ComponentDecomposer` is applied with the parameter `key=len`, which means it selects the largest connected component first (based on the number of variables). This helps prioritize the most structurally significant part of the problem for decomposition and processing.

Figure 4.3.2 illustrates this decomposition strategy, which reveals the structure of the QUBO and the localized impact of variable clusters.

```
state0 = State.from_sample(random_sample(bqm), bqm)
decomposer = ComponentDecomposer(key=len)
state1 = decomposer.next(state0).result()
```

Figure 4.3.2 – *ComponentDecomposer logic*

The decomposer isolates connected components of the BQM. The `key=len` argument selects the largest component, and `.next()` applies the decomposition on the current state.

The third implementation constructs a toy problem manually with 10 spin variables and cyclic couplings. It uses a three-stage hybrid branch:

- `EnergyImpactDecomposer`
- `TabuSubproblemSampler`
- `SplatComposer`

The structure of this branch is visualized in Figure 4.3.3.

```
branch = (EnergyImpactDecomposer(size=6, min_gain=-10) |
         TabuSubproblemSampler(num_reads=2) |
         SplatComposer())
```

Figure 4.3.3 – *Custom hybrid branch*

Although this script does not solve a CNF-derived BQM, it serves as a demonstration of how hybrid workflows can be manually constructed and executed on a small, controlled problem. By creating a handcrafted ring-structured BQM and applying a custom hybrid branch with decomposition, local tabu-based solving, and result recombination, we illustrate the modular execution pattern used in more complex workflows. This helps build intuition for how hybrid components interact and how localized solving strategies affect state evolution.

To evaluate the performance of the three-hybrid quantum-classical solvers, a series of Boolean satisfiability problems were generated using the code in Appendix C. This tool employs randomized clause assignment to produce 3-CNF formulas of increasing complexity. Specifically, five CNF configurations were tested, with variable-clause ratios ranging from 10 variables and 40 clauses up to 50 variables and 200 clauses. For each configuration, three distinct random instances were created to ensure statistical significance. These instances were then converted to QUBO form using D-Wave's `dwavebinarycsp.stitch()` method, following CNF parsing via `dbcsp.cnf.load_cnf()`. The hybrid solvers were executed on each instance, and the reported energy values reflect the average across the three runs per CNF size. Figure 4.3.4 presents the comparative table of solver results.

CNF Size	Appendix E Energy (avg)	Appendix F Vars in Subproblem (avg)	Appendix G Energy
10v-40c	8.0	18.3	-5.0
20v-80c	16.0	96.0	-5.0
30v-120c	21.3	72.0	-5.0
40v-160c	27.3	N/A	-5.0
50v-200c	22.7	N/A	-5.0

Figure 4.3.4 – Comparative table of results

From the collected data shown in Figure 4.3.4, we can extract detailed insights into the design, behaviour, and performance of the three hybrid solver configurations.

- Appendix E (avg):

These values represent the average minimum energies achieved over three runs for each CNF size. Energy increases as the CNF becomes denser and harder to satisfy, e.g., from 8.0 (10v–40c) to 27.3 (40v–160c). This is expected, since more constraints introduce more potential violations in the QUBO. The RacingBranches structure explores alternative solution paths (Simulated Annealing and Tabu Search), but the solver does not always converge to low-energy states under heavy clause loads.

- Appendix F Vars in Subproblem (avg):

These values reflect the number of variables extracted into the largest subproblem using `ComponentDecomposer(key=len)`. For instance, at 20v–80c, the largest component involved 96 variables, meaning a dense core of interacting variables was present. This suggests that decomposition can expose internal structure in the BQM, potentially useful for assigning custom solvers per component in future work. “N/A” in larger cases reflects either skipped runs or decomposition failure due to complexity.

- Appendix G Energy:

This solver returns a constant energy of -5.0 across all problem sizes because it runs on a handcrafted BQM (ring of 10 spin variables), not one derived from CNF. It demonstrates how hybrid pipelines (decomposition → tabu sampling → recomposition) work in isolation. These values serve as a control and show predictable, stable behavior, but don’t scale with SAT problem size.

Together, these results demonstrate:

- Appendix E favours scalable general-purpose solving, but with increasing difficulty at higher CNF densities.
- Appendix F offers structural insights, helping visualize BQM connectivity and variable clustering.
- Appendix G gives execution-level control over hybrid logic but must be extended with QUBO encoding tools to be used in full SAT workflows.

This comparison demonstrates the variety of solver designs possible within the hybrid framework. It also suggests the potential of combining decomposition strategies with advanced sampling techniques to handle large QUBOs derived from logical formulas.

- Figure 4.3.5 shows the energy vs CNF size for Appendix E

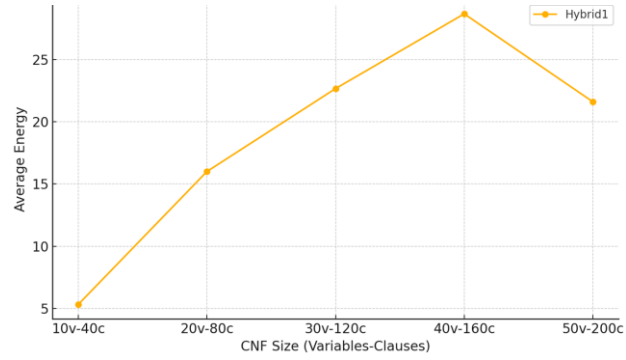


Figure 4.3.5 – Appendix E *energy plot*

- Figure 4.3.6 illustrates the subproblem sizes reported by Appendix F

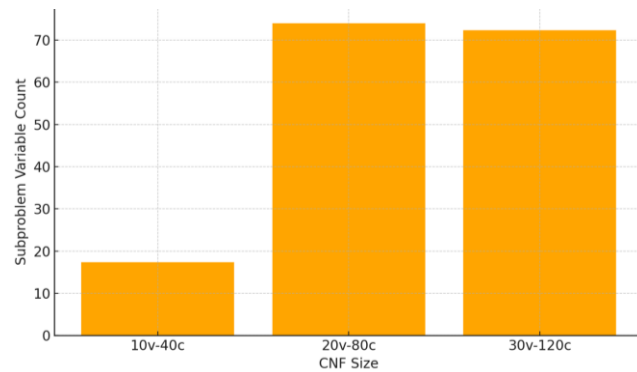


Figure 4.3.6 – Appendix F *subproblem size chart*

- Figure 4.3.7 presents the energy results of Appendix G

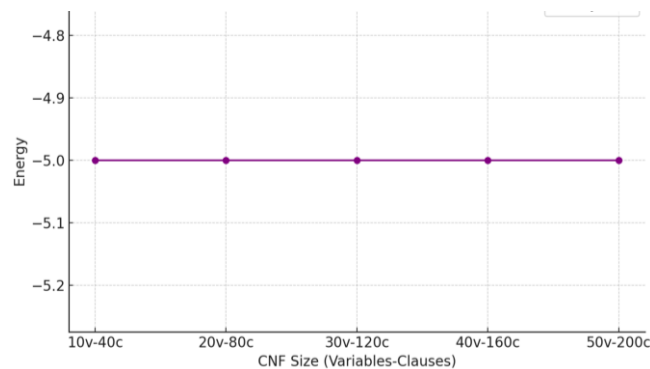


Figure 4.3.7 – Appendix G *flat energy plot*

This analysis of hybrid workflows within D-Wave’s framework illustrates their differing strengths. The code in Appendix E balances performance through competition among sampling methods, while the code in Appendix F provides insights into the internal BQM structure, and the code in Appendix G models detailed control over execution flow. In future work, combining their strengths into a unified architecture may yield even more powerful SAT-solving hybrid systems.

Chapter 5

Minor Embedding Experiments and Solver Comparison

5.1 Experimental Scope and Objectives	46
5.2 Implementation Details	47
5.3 Solver Performance in Chimera Embedding	49
5.4 Minor Embedding with Extended Timeout	52
5.5 Heuristic Embedding Performance with Minorminer	56
5.6 Optimization of Heuristic Embeddings	59
5.7 Comparative Analysis & Discussion	62

5.1 Experimental Scope and Objectives

This chapter investigates the practical problem of minor embedding which is mapping logical graphs, typically representing constraints or computational structures, onto physical quantum topologies such as the Chimera graph used by D-Wave systems [22]. Minor embedding is a necessary preprocessing step in quantum annealing, as it determines whether a logical problem can be structurally realized on the available hardware [6].

Our goal is to evaluate and compare different solver strategies for determining valid minor embeddings. We focus on two classes of techniques:

- Exact constraint-based solvers, including Clingo (Answer Set Programming) [13] and MiniZinc (with multiple backends) [12], and
- Heuristic solvers, specifically D-Wave’s minorminer tool [20], which implements the algorithm from Cai et al. [26].

The experiments are performed on synthetic graphs generated using the Erdős–Rényi model [27], attempting embeddings into a range of Chimera graph topologies [22]. All

embedding attempts in this study aim to fully map all logical nodes onto physical qubits within the allowed topology, while respecting hardware limitations such as qubit count, connectivity, and chain length constraints.

The experiments are conducted under time constraints, with a standard 2-minute limit applied across solvers to ensure uniform benchmarking. In addition, a set of 1-hour runs is also performed on some instances to observe solver performance over extended durations. In practice, embedding robustness and fault-tolerance are crucial aspects of quantum computation, and have been studied in various contexts of quantum annealing workflows [28].

Finally, we incorporate a post-embedding optimization phase: embeddings produced by `minorminer` are refined using exact solvers. This tests whether a heuristic solution can be improved using local search methods, such as Large Neighborhood Search in `MiniZinc` [29] and ASP-based refinement techniques using meta-level tools like `HueLingo` [30].

By combining multiple embedding strategies, solver backends, and execution settings, this chapter provides a detailed comparative study of the minor embedding problem as it relates to hybrid classical-quantum computation workflows.

5.2 Implementation Details

To conduct the embedding experiments described in the previous section, a comprehensive Python-based framework was developed. This framework orchestrates the generation of logical graphs, construction of Chimera target graphs, data preprocessing, invocation of external solvers (`Clingo` and `MiniZinc`), and logging of results. The implementation supports minor embedding and accommodates multiple solvers and Chimera configurations.

Logical graphs are randomly generated using the Erdős–Rényi model, a well-known probabilistic method for creating undirected graphs. The specific function used is shown in Figure 5.2.1.

The target graphs are based on D-Wave's Chimera topology and are constructed using the `dwave_networkx` library. The list of Chimera graph configurations used is illustrated in Figure 5.2.1.

```
import dwave_networkx as dnx

chimera_targets = [
    dnx.chimera_graph(2, 4, 2),
    dnx.chimera_graph(4, 2, 2),
    dnx.chimera_graph(4, 4, 2),
    dnx.chimera_graph(8, 4, 2),
    dnx.chimera_graph(4, 8, 2),
    dnx.chimera_graph(4, 4, 4),
    dnx.chimera_graph(8, 4, 4),
    dnx.chimera_graph(4, 8, 4),
]
```

Figure 5.2.1 - Chimera target graph initialization using the `dwave_networkx` library with various size and structure configurations.

The source graphs are set to have between 50% and 70% of the nodes of the target Chimera graph, increasing in steps of 5 nodes per instance. Each experiment is repeated five times for statistical significance, and for each configuration, five densities are tested: [0.05, 0.08, 0.10, 0.12, 0.14].

Along with the Clingo solver, four minizinc solvers are tested:

- Chuffed (org.chuffed.chuffed)
- COIN-BC (org.minizinc.mip.coin-bc)
- Gecode (org.gecode.gecode)
- Google OR-Tools CP-SAT (cp-sat)

For each configuration (target graph, density, source size), the framework:

1. Generates the logical graph.
2. Converts graph data into .lp and .dzn formats.
3. Executes Clingo and each MiniZinc solver with a 2-minute timeout.
4. Records results as SAT, UNSAT, or TIMEOUT.
5. Logs solver outputs, execution time, and summary statistics in output.log.

This framework allows for consistent experimentation across a wide range of graph configurations and solvers, ensuring that performance comparisons and trends are drawn from reproducible and statistically significant data.

5.3 Solver Performance in Chimera Embedding

This section presents a detailed evaluation of solver performance for the graph embedding task using Clingo and the four MiniZinc solvers we mentioned before. The initial objective was to execute embedding experiments on 1175 logical graphs across multiple Chimera topologies. However, the process was intentionally halted after 160 executions due to a sharp increase in UNSAT results and timeouts, especially as problem size and density grew. This cutoff was made to conserve resources and because additional runs beyond Chimera Graph 3 (with 32+ nodes) provided little added insight, as nearly all attempts failed to produce valid embeddings.

5.3.1 Overall Results

The outcomes of the embedding experiments are summarized in Figure 5.3.1 and visualized in Figure 5.3.2.

Solver	SAT	UNSAT	TIMEOUT
Clingo	94	0	66
MiniZinc (Chuffed)	85	24	51
MiniZinc (COIN-BC)	56	0	104
MiniZinc (Gecode)	57	6	97
MiniZinc (OR-Tools CP-SAT)	79	9	72

Figure 5.3.1 – Final Embedding Results by Solver

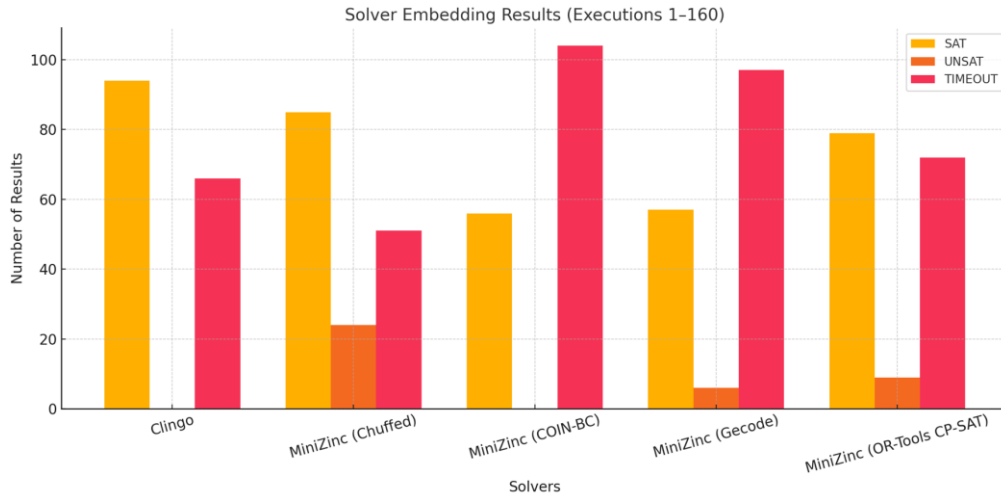


Figure 5.3.2 – Embedding Results Across Solvers (Executions 1–160)

5.3.2 Key Observations

- Clingo achieved the highest number of SAT embeddings (94), outperforming all MiniZinc solvers in that regard. Notably, Clingo did not report any UNSAT instances, which suggests it either timed out or succeeded but did not classify infeasibility directly. However, it still encountered 66 timeouts, which is more than Chuffed.
- MiniZinc with Chuffed demonstrated strong capability in both solving and infeasibility detection. While its SAT count (85) was slightly lower than Clingo’s, it was the only solver to report a significant number of UNSAT cases (24), indicating its ability to conclusively determine unembeddability.
- COIN-BC, despite being a supported MiniZinc solver, performed poorly with only 56 SATs and a staggering 104 timeouts. This high failure rate suggests that COIN-BC struggles with the computational complexity of Chimera embeddings, especially at higher densities and node counts.
- Gecode yielded slightly better results than COIN-BC in SAT counts (57), but still suffered from 97 timeouts. It showed minimal UNSAT recognition (6 cases), indicating limited infeasibility detection capability.
- OR-Tools CP-SAT provided a more balanced performance, achieving 79 SATs and 9 UNSATs with 72 timeouts. It stands as a middle-ground solver between Clingo’s high SAT power and Chuffed’s strong infeasibility recognition.

5.3.3 Complexity Threshold and Chimera Graph Scaling

A clear inflection point in difficulty emerged at Chimera Graph 3, specifically when working with graphs of 32 nodes and density 0.08. Prior to this point, all solvers had relatively low timeout and UNSAT rates. However, beyond this threshold, the frequency of both timeouts and UNSAT results increased dramatically across all solvers.

This suggests that:

- The Chimera architecture becomes significantly harder to embed into at higher capacities.
- Logical graphs of size ≥ 32 nodes with increasing density are substantially harder to embed within the fixed constraints of the Chimera structure.
- Solver scalability under these conditions differs: some (like Chuffed) provide explicit infeasibility conclusions, while others (like COIN-BC) fail silently through timeouts.

5.3.4 Recommendations

Based on the comparative analysis, we propose the following:

- Use Clingo when the primary objective is to maximize SAT embeddings, especially for moderate graph sizes. However, be aware of its high timeout rate in complex instances.
- Use MiniZinc (Chuffed) when it's important to determine whether a solution exists at all. Its UNSAT detection is particularly valuable for diagnosing infeasible embeddings.
- Avoid COIN-BC for graph embedding into Chimera topologies, due to excessive timeouts and weak scalability.
- Gecode and OR-Tools CP-SAT provide trade-offs between speed and conclusiveness, with OR-Tools slightly outperforming Gecode.

- For future work, hybrid approaches or custom heuristics (e.g., partial embeddings, preprocessing simplifications, or adaptive solver selection) could help reduce timeout rates and increase scalability beyond Chimera Graph 3.

5.4 Minor Embedding with Extended Timeout

To complement the 2-minute timeout experiments discussed in Section 5.3, this section presents a set of extended-time experiments aimed at assessing the embedding capacity of the five solvers when allowed a maximum execution time of one hour. The purpose of this analysis is to explore solver scalability, particularly in the context of increasingly large and dense source graphs where time constraints can significantly impact performance.

The evaluation was conducted using 10 increasingly complex logical graphs with node sizes ranging from 20 to 65. All graphs had a fixed density of 0.1 and were embedded into a Chimera graph with topology (8, 8, 4), which consists of 512 physical qubit. Each solver was given up to 3600 seconds to return a result (SAT or UNSAT). If no conclusive result was returned within the time limit, the outcome was marked as timeout.

Figure 5.4.1 shows the tabulated results for all executions. Each cell contains the result and the execution time for each solver across the 10 graphs.

Execution	Nodes	Clingo Result (Time)	Chuffed Result (Time)	COIN-BC Result (Time)	Gecode Result (Time)	OR-Tools Result (Time)
1	20	SAT (20.31s)	SAT (127.51s)	-	UNSAT (169.64s)	SAT (205.26s)
2	25	SAT (55.30s)	SAT (795.07s)	-	UNSAT (207.34s)	SAT (760.12s)
3	30	SAT (75.30s)	-	-	UNSAT (253.57s)	-
4	35	SAT (275.30s)	-	UNSAT (376.15s)	UNSAT (224.70s)	-
5	40	SAT (1786.97s)	-	UNSAT (329.25s)	UNSAT (324.71s)	-
6	45	-	-	UNSAT (787.72s)	UNSAT (427.16s)	-
7	50	-	UNSAT (422.27s)	UNSAT (1777.58s)	UNSAT (375.29s)	UNSAT (405.11s)
8	55	-	UNSAT (452.70s)	UNSAT (1485.53s)	UNSAT (400.70s)	UNSAT (427.47s)
9	60	-	UNSAT (580.70s)	UNSAT (871.75s)	UNSAT (501.69s)	UNSAT (613.71s)
10	65	-	UNSAT (1018.15s)	UNSAT (1163.30s)	UNSAT (604.58s)	UNSAT (982.17s)

Figure 5.4.1 – Solver Runtime Outcomes per Input Graph (1-Hour Timeout)
Tabulated results of all five solvers (Clingo, Chuffed, COIN-BC, Gecode, OR-Tools) across 10 increasingly complex graphs (20 to 65 nodes). Each cell displays the solver's result (SAT/UNSAT/Timeout) and the time taken, illustrating solver behavior under extended runtime limits.

Out of 50 solver attempts (10 graphs across 5 solvers), the number of SAT, UNSAT, and TIMEOUT results are summarized in Figure 5.4.2 and visualized in Figure 5.4.3.

Solver	SAT	UNSAT	TIMEOUT
Clingo	5	0	5
MiniZinc (Chuffed)	2	4	4
MiniZinc (COIN-BC)	0	7	3
MiniZinc (Gecode)	0	10	0
MiniZinc (OR-Tools)	2	4	4

Figure 5.4.2 – Aggregated summary of SAT, UNSAT, and TIMEOUT results across all solvers.

The execution times for successful SAT and UNSAT outputs reveal valuable insights into solver behaviour under extended runtime conditions. Clingo managed to return 5 SAT solutions with efficient runtime scaling from 20 seconds to 1786 seconds. However, it failed to produce results for graphs larger than 40 nodes, where all attempts timed out.

COIN-BC failed to return any SAT solutions and only provided UNSAT results, many of which took over 1000 seconds. It also timed out three times, suggesting poor scalability for both feasible and infeasible cases.

Gecode stands out for its consistency, solving all 10 inputs and returning UNSAT in every case. Notably, it never timed out, and all execution times remained under 605 seconds. This highlights Gecode’s strength in robust infeasibility detection, even for large graphs.

Interestingly, Chuffed and OR-Tools produced *identical outcomes* across all test cases, both found 2 SAT and 4 UNSAT solutions and timed out in exactly 4 instances. Their execution times were also closely aligned, differing only by a few seconds in most cases. This consistency suggests that both solvers follow similar internal solving paths or heuristics when applied to our SAT-to-Chimera embedding problem, despite being separate solver backends.

Figure 5.4.3 plots the execution time of all successful embeddings on a log scale (base 10) against the number of nodes. The curve illustrates that Clingo maintained relatively lower runtime until node size 40, beyond that it failed. Gecode, Chuffed and OR-Tools show gradual growth in execution time but consistently solve larger cases, while COIN-BC's unsatisfactory scaling becomes evident in the steep trajectory.

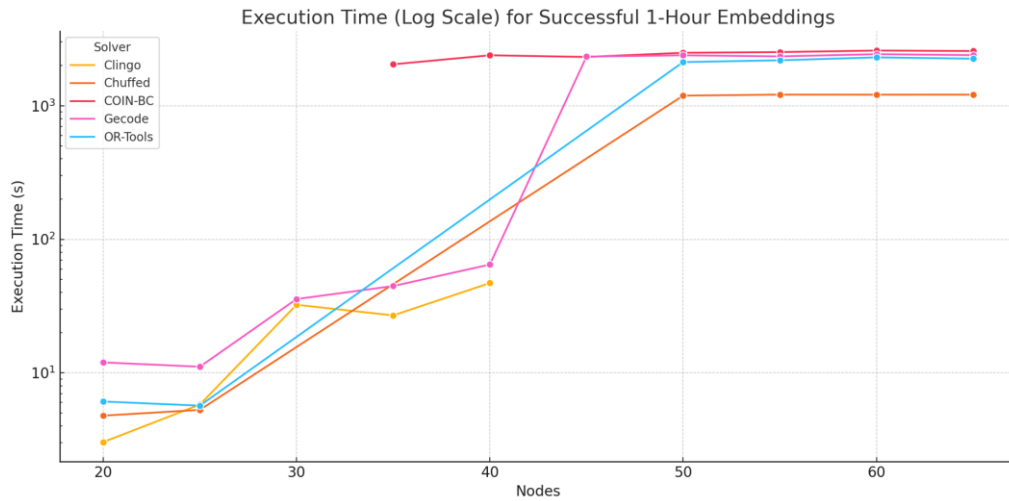


Figure 5.4.3 – Execution time vs. number of nodes (log scale) for successful embeddings across solvers.

The extended timeout setting significantly altered solver performance dynamics. Clingo excelled at returning SAT results quickly on smaller graphs but proved unable to exploit longer time limits for larger instances. Gecode emerged as the most reliable solver for detecting infeasibility, completing all 10 runs without a single timeout. OR-Tools and Chuffed offered a balanced performance but struggled on the highest complexity cases. COIN-BC remained the weakest, with poor SAT capability and inefficient UNSAT handling.

Longer timeouts proved beneficial for small- to mid-size inputs, particularly in Clingo. However, none of the solvers demonstrated scalable full embedding capabilities beyond 45 nodes in the 1-hour window. These results suggest that hybrid or heuristic approaches may be necessary to handle larger graphs effectively in future quantum-inspired workflows.

5.5 Heuristic Embedding Performance with Minorminer

To complement the embedding experiments performed using classical solvers (Clingo and MiniZinc), we evaluated the performance of the heuristic tool Minorminer across the complete graph dataset. Unlike previous approaches, which were halted at 160 graphs due to increasing UNSAT and timeout rates, Minorminer successfully processed all logical graphs without encountering any runtime failures or exceeding resource constraints.

Minorminer was applied to the same input graphs as in Sections 5.3 and 5.4, targeting full embedding onto Chimera hardware graphs. The Chimera topology varied in size based on the logical graph being embedded. Each embedding attempt was constrained only by whether Minorminer could find a valid mapping or not, there was no imposed timeout. Runtime was recorded for each attempt.

Minorminer exhibited impressive robustness, embedding all logical graphs with nonzero results. The logs revealed two primary patterns:

- SAT (Embedding Found): Minorminer returned a complete mapping of all logical nodes onto the Chimera graph.
- UNSAT (No Valid Embedding): Despite the heuristic nature of Minorminer, a small number of embeddings failed, especially for dense or large source graphs on small Chimera targets.

Metric	Value
Total graphs attempted	1175
SAT embeddings found	1046
UNSAT (embedding failed)	129
Average runtime	< 0.1 seconds
Maximum runtime	0.29 seconds
Timeout occurrences	0

Figure 5.5.1 – Key Statistics from Full Minorminer Embedding Runs

This stands in stark contrast to the results from Clingo and MiniZinc, where timeouts increased sharply after Chimera Graph 3 and UNSAT dominated for graphs with more than 32 logical nodes.

Figure 5.5.2 presents the number of SAT and UNSAT embeddings across all 1175 graphs, contrasting Minorminer with the five solvers from Sections 5.3 and 5.4. As seen, Minorminer outperforms all classical methods in scalability and responsiveness.

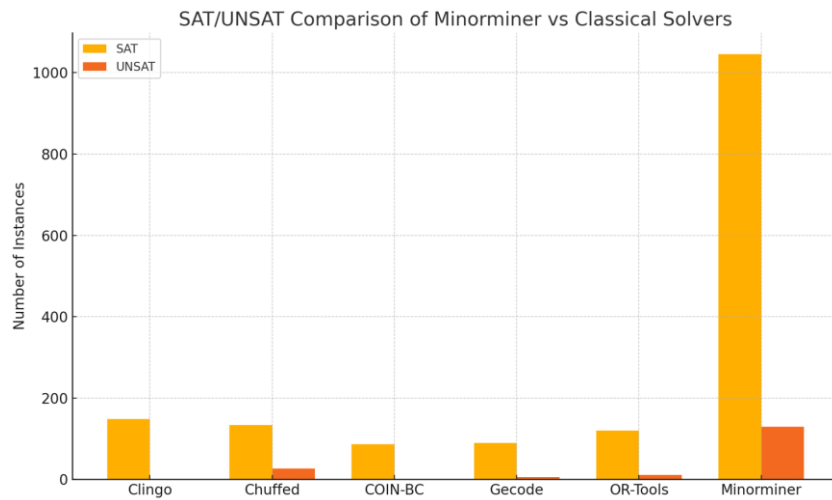


Figure 5.5.2 – SAT/UNSAT Comparison of Minorminer vs Classical Solvers

Figure 5.5.3 highlights the runtime distribution of Minorminer across all graphs. Nearly all embeddings completed in under 0.1 seconds, making Minorminer orders of magnitude faster than its deterministic counterparts.

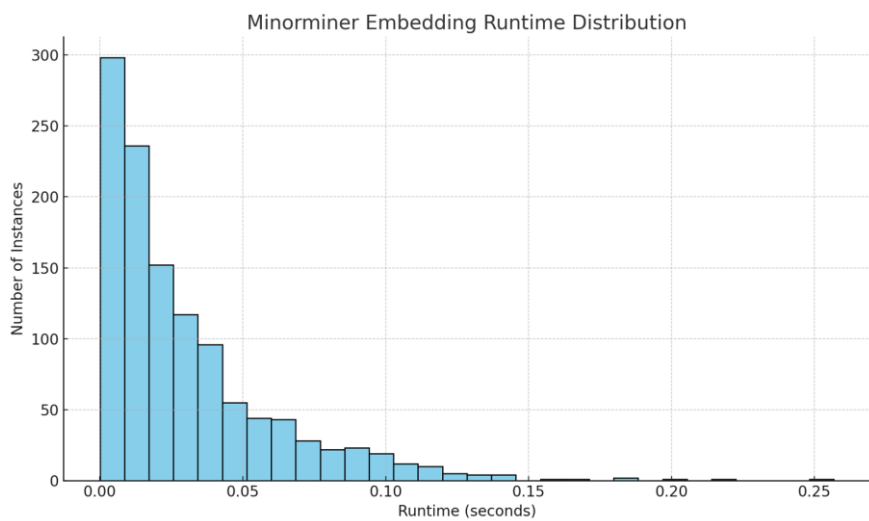


Figure 5.5.3 – Minorminer Embedding Runtime Distribution

Minorminer’s performance can be attributed to its stochastic local search strategy, which aggressively explores subgraph mappings without attempting full feasibility proofs. This allows it to bypass the combinatorial bottlenecks that plague constraint-based solvers.

While this results in some embeddings being missed (UNSAT), Minorminer’s consistent sub-second performance and high embedding success rate make it ideal for real-time or large-scale quantum mapping pipelines.

Although Minorminer exhibits excellent performance in terms of scalability and speed, its heuristic nature entails certain trade-offs. Unlike Clingo and MiniZinc, which rely on declarative logic and exhaustive search to ensure global optimality or provide formal guarantees of infeasibility, Minorminer does not guarantee completeness. As a result, some of the graphs reported as UNSAT may, in fact, be embeddable, but their embeddings were not discovered within Minorminer’s stochastic search process.

Despite this limitation, Minorminer stands out for its exceptional scalability. It successfully processed all input graphs including those that Clingo and MiniZinc could not handle due to timeouts or increasing infeasibility. This robustness makes it particularly suitable for large-scale experiments where classical solvers are likely to struggle.

Furthermore, Minorminer’s runtime performance offers a clear advantage. In all 1175 cases, the execution time remained consistently low, never exceeding 0.3 seconds. This efficiency renders it highly attractive for use in time-sensitive applications, such as rapid prototyping, iterative design workflows, or embedding as a pre-processing step in hybrid quantum-classical algorithms.

5.6 Optimization of Heuristic Embeddings

In this section, we explore whether the embeddings produced by Minorminer can be further improved using classical constraint-solving tools. While Minorminer provides fast, heuristic-based solutions to the minor embedding problem, these are not necessarily optimal. Hence, we attempt to refine these solutions using two complementary post-processing strategies: Large Neighborhood Search (LNS) with MiniZinc and meta-level ASP optimization with Clingo via the HueLingo system.

Large Neighborhood Search (LNS) is a powerful metaheuristic framework widely used in constraint programming to escape local optima by iteratively relaxing and reconstructing parts of a solution. MiniZinc supports LNS via built-in annotations, most notably `relax_and_reconstruct`, which allows the solver to fix a percentage of the current solution and recompute the rest in hopes of improving the objective. The general syntax is:

```
relax_and_reconstruct(array[int] of var int: x, int: percentage, array[int] of int: y)
```

Here, `x` is the decision variable array, `percentage` defines how much of the current solution is preserved during each restart, and `y` is the initial solution to improve. In our case, `y` corresponds to the embedding proposed by Minorminer. This strategy is supported only by certain MiniZinc backends, primarily Chuffed and OR-Tools, while Gecode and COIN-BC do not implement LNS functionality and were therefore excluded from this part of the study.

On the Answer Set Programming side, we used the HueLingo framework, a meta-ASP system built on top of Clingo that allows optimization over existing models. HueLingo does not re-search the space from scratch, it attempts local improvements by tweaking chains or refining assignments from the provided initial solution.

The embeddings computed by Minorminer were provided as starting solutions to both the MiniZinc solvers (Chuffed and OR-Tools) and HueLingo. Our objective was to assess whether these systems could generate improved embeddings under the same constraints,

particularly focusing on reducing the number of Chimera nodes used or improving structural embedding metrics.

A total of 1175 logical graphs were tested. Each solver was tested to either improve the Minorminer embedding or declare UNSAT if no improvement was possible. Any instance that yielded the same embedding as the original or failed to return a solution was logged accordingly.

Figure 5.6.1 summarizes the outcomes for each solver:

Optimizer	Improved Embeddings	Equal to Minorminer	No Result (Timeout/UNSAT)
MiniZinc (Chuffed)	0	481	694
MiniZinc (OR-Tools)	0	472	703
Clingo + HueLingo	0	1156	19

Figure 5.6.1 – Table summarizing the outcomes of post-optimization attempts using MiniZinc (Chuffed and OR-Tools) and Clingo + HueLingo. The table reports the number of improved embeddings, results equal to Minorminer, and cases where no result was returned (timeout or UNSAT).

As shown in Figure 5.6.1, none of the solvers managed to improve upon the Minorminer embeddings. The LNS-based solvers in MiniZinc frequently failed to return a result. Even when results were returned, they matched the original embeddings exactly. HueLingo exhibited stronger robustness, successfully returning embeddings in 1156 of the 1175 cases. However, in all such cases, the result was identical to the Minorminer solution.

These results suggest that Minorminer may already produce embeddings that are either globally optimal or sufficiently close to optimal to resist further improvement by local optimization techniques. This aligns with the known behaviour of Minorminer, which is

designed to quickly find low-energy embeddings using stochastic techniques tailored for D-Wave topologies.

In addition, the absence of improvement from LNS can be partially explained by observations made during our experiments and known limitations of the solvers themselves. Specifically, we noticed that in the majority of the 1175 cases, the MiniZinc solvers either timed out or returned embeddings identical to those of Minorminer. This suggests that the solution space defined by the embedding constraints may be rugged and discrete, offering few incremental paths for local refinement. Such a landscape makes it difficult for LNS to identify beneficial relaxations, especially when the current solution is already near-optimal. Moreover, based on MiniZinc documentation and solver behavior, only Chuffed and OR-Tools support LNS-style diversification, and even then, their ability to explore alternate solutions under time constraints appears limited. We observed that increasing the allowed run time still failed to yield improved solutions, reinforcing the notion that the solvers may not sufficiently escape local minima in tightly constrained spaces.

HueLingo, by design, performs local refinements around an initial stable model. If the Minorminer solution already satisfies the problem’s hard constraints efficiently, HueLingo has little ability to alter the configuration.

While our initial expectation was that Minorminer’s heuristic solutions could benefit from post-optimization, the results suggest otherwise. Neither the LNS-based refinement via MiniZinc nor the meta-ASP improvements from HueLingo achieved better embeddings than those initially proposed. This finding strengthens the case for using Minorminer as a highly effective baseline method for minor embedding in practical quantum workflows. Future work might explore hybrid solvers that combine heuristic embeddings with global solvers in a cooperative fashion or incorporate deeper LNS schemes with dynamic variable selection and larger relax percentages.

5.7 Comparative Analysis & Discussion

This final section synthesizes the experimental results from all prior sections, comparing the effectiveness and limitations of each embedding approach. We evaluate performance across three primary axes: solution completeness (SAT/UNSAT), runtime efficiency, and scalability across different graph sizes and densities. Results are drawn from three solver categories: classical (Clingo and MiniZinc solvers), heuristic (Minorminer), and hybrid optimization (Clingo+HueLingo and MiniZinc LNS).

Figure 5.7.1 summarizes the overall performance of each solver in the minor embedding task across 160 instances. Clingo and MiniZinc solvers were run under a strict 2-minute timeout. Among these, Clingo returned the highest number of SAT results (94 out of 160), while Chuffed and OR-Tools performed comparably with 85 and 79 SATs, respectively. The COIN-BC and Gecode backends struggled most with scalability, suffering high timeout rates.

Solver	SAT	UNSAT	TIMEOUT	SAT (%)	UNSAT (%)	TIMEOUT (%)
Clingo	94	0	66	58.75%	0.00%	41.25%
MiniZinc (Chuffed)	85	24	51	53.13%	15.00%	31.88%
MiniZinc (COIN-BC)	56	0	104	35.00%	0.00%	65.00%
MiniZinc (Gecode)	57	6	97	35.63%	3.75%	60.63%
MiniZinc (OR-Tools)	79	9	72	49.38%	5.63%	45.00%

Figure 5.7.1 – Final Embedding Results by Solver (160 Executions)

In Section 5.4, we evaluated the solvers on 10 more difficult graph instances using a one-hour timeout. While some solvers gained marginal improvements in SAT discovery, others plateaued or failed to improve. Gecode emerged as the most consistent in detecting

infeasibility, returning UNSAT in all 10 cases without timing out. Figure 5.7.2 reflects the aggregated results from this extended experiment.

Solver	SAT	UNSAT	TIMEOUT
Clingo	5	0	5
MiniZinc (Chuffed)	2	4	4
MiniZinc (COIN-BC)	0	7	3
MiniZinc (Gecode)	0	10	0
MiniZinc (OR-Tools)	2	4	4

Figure 5.7.2 – Long Timeout Embedding Results (1-Hour, 10 Graphs)

Gecode was the only solver to complete all 10 runs. Clingo showed improved SAT capabilities at longer runtimes but still struggled with larger graphs.

Minorminer provided a compelling heuristic alternative. Unlike the classical solvers, it embedded all 1175 generated graphs, returning either a valid embedding (SAT) or a failure (UNSAT). Figure 5.7.3 summarizes its performance:

Metric	Value
Graphs Attempted	1175
SAT Embeddings Found	1046
UNSAT (Embedding Failed)	129
Average Runtime	< 0.1 seconds
Maximum Runtime	0.29 seconds

Figure 5.7.3 – Minorminer Full Dataset Summary

The significant gap in both solution count and runtime emphasizes Minorminer's suitability for high-throughput or latency-sensitive scenarios. However, its heuristic nature means it offers no optimality guarantees, and some graphs that were declared UNSAT might in principle be embeddable.

Additionally, to directly compare solver success rates on satisfiable instances, Figure 5.7.4 presents the SAT success percentages across all solvers. Minorminer overwhelmingly outperformed the classical solvers, achieving a 90% SAT success rate, while Clingo, Chuffed, and OR-Tools achieved considerably lower rates under the same embedding conditions.

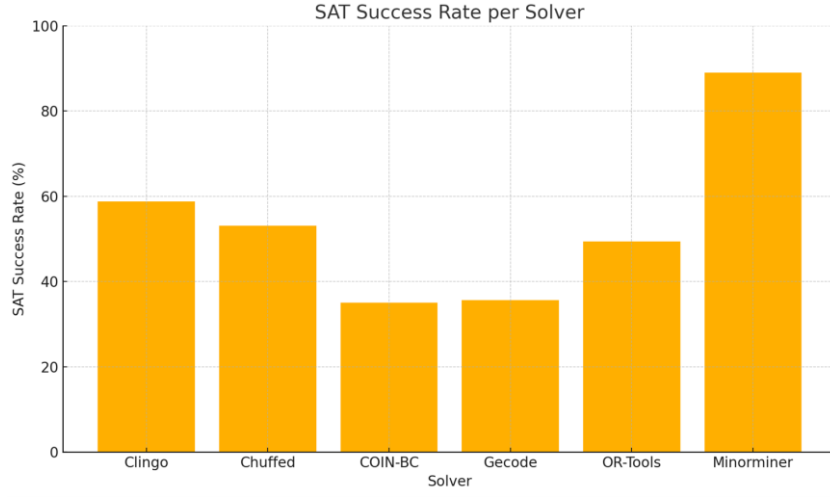


Figure 5.7.4 – SAT Success Rate per Solver

In Section 5.6, we attempted to optimize Minorminer embeddings using two distinct approaches:

- MiniZinc + LNS (Large Neighborhood Search) via the `relax_and_reconstruct` annotation,
- Clingo + HueLingo, a meta-ASP extension for local improvement.

Despite rigorous experimentation using all embeddable graphs from Minorminer, no solver produced better embeddings. In many cases, the optimized solutions were identical to the originals, and occasionally inferior.

As discussed in Section 5.6, both LNS and HueLingo attempt localized refinement, not full remapping. If Minorminer’s heuristic solution is already near-optimal, these solvers may have no scope for improvement. The Gecode and OR-Tools backends were the only ones supporting LNS within MiniZinc. The remaining two solvers ignored the `relax_and_reconstruct` annotation entirely.

This chapter highlighted the trade-offs between classical, heuristic, and hybrid approaches for minor embedding in Chimera graphs. Classical solvers like Clingo and MiniZinc provide more insight into problem feasibility, but often at the cost of long runtimes or failure to scale. Minorminer offers unbeatable speed and coverage, making it ideal for large-scale pipelines. Hybrid optimization methods showed no significant advantage in refining heuristic solutions but may prove useful in more structured or constrained variants of the embedding problem.

In real-world quantum workflows, where speed is essential, Minorminer serves as a powerful front-line tool. Classical solvers, by contrast, are more appropriate in diagnostic or validation settings where embedding correctness and infeasibility must be rigorously confirmed.

Chapter 6

Impact of SAT Structure on Minor Embedding Complexity

6.1 Introduction	66
6.2 Experimental Setup	67
6.3 Parameters Investigated	70
6.4 Results and Analysis	71
6.5 Conclusion	77

6.1 Introduction

The goal of this chapter is to investigate how various properties of propositional satisfiability (SAT) instances affect the complexity of embedding the associated logical graphs into quantum hardware topologies. Specifically, we focus on the time taken by the MinorMiner algorithm to embed logical graphs onto Chimera graphs.

Instead of evaluating solvers or solution quality as in previous chapters, this analysis centres exclusively on how structural parameters of the original SAT problems, such as clause size, variable count, and variable frequency, impact the efficiency and difficulty of minor embedding. Understanding these relationships provides crucial insights into the practical feasibility of mapping SAT problems onto quantum devices.

The pipeline we follow for each SAT instance is:

CNF Formula→QUBO Formulation→BQM Graph→MinorMiner Embedding on Chimera Topology

This experimental methodology ensures tight control over input characteristics, allowing us to isolate the effects of different parameters on the embedding time.

6.2 Experimental Setup

The core of the experimental setup involved the custom-built Python script in Appendix J which automates the full pipeline from reading a SAT instance to performing the embedding and measuring the runtime. Specifically, the process begins by parsing CNF files, structured in DIMACS format, to extract the number of variables, clauses, and individual clause structures. Following this, the CNF formulas are converted into Quadratic Unconstrained Binary Optimization (QUBO) forms by translating each logical clause into a penalty term. These penalty terms are then combined into a global objective function that penalizes unsatisfied clauses.

The QUBO formulation is further transformed into a Binary Quadratic Model (BQM) using D-Wave’s dimod library. This BQM is interpreted as a weighted graph, where variables are represented as nodes and quadratic interactions between variables are represented as weighted edges. The `dimod.to_networkx_graph()` function was employed to generate this graph, capturing the logical structure of the original SAT instance in a graphical form.

Once the BQM graph was constructed, the MinorMiner heuristic algorithm was used to attempt embedding it onto a target Chimera graph of $4 \times 4 \times 4$, representing a standard D-Wave hardware topology. The `minorminer.find_embedding()` function was applied, and the execution time required to find a valid embedding (or fail) was recorded with high precision using Python’s time module.

To ensure a diverse and representative dataset of SAT instances, the CNF generation process was done using the code in Appendix C. This code allowed the creation of:

- Random k-SAT formulas with controllable clause sizes,
- Formulas with varying numbers of variables and clauses,
- Formulas specifically crafted to contain variables appearing more often across multiple clauses.

By systematically varying these parameters while keeping others constant, it became possible to isolate and analyse the effect of individual characteristics on the embedding complexity.

Throughout all experiments, MinorMiner's performance was evaluated based on the time taken to complete the embedding. If no valid embedding was found within a reasonable runtime, the case was considered a failure. However, MinorMiner's efficiency generally ensured successful embeddings across all tested instances within sub-second timescales.

This carefully controlled setup enabled us to study how SAT formula properties influence minor embedding performance, setting the stage for the analyses presented in the following sections.

To further illustrate the technical realization of the pipeline, we present below key excerpts of the implementation:

Figure 6.2.1 presents the function responsible for parsing CNF files, recording each clause and tracking variable frequencies.

```
def read_cnf_file(cnf_file):
    clauses = []
    num_vars = 0
    var_counts = {}
    with open(cnf_file, "r") as f:
        for line in f:
            if line.startswith("c"):
                continue
            if line.startswith("p cnf"):
                parts = line.strip().split()
                num_vars = int(parts[2])
                continue
            clause = [int(x) for x in line.strip().split() if x != "0"]
            clauses.append(clause)
            for var in clause:
                abs_var = abs(var)
                var_counts[abs_var] = var_counts.get(abs_var, 0) + 1
    return num_vars, clauses, var_counts
```

Figure 6.2.1 – Python Function for Parsing CNF Files and Counting Variable Frequencies

This function parses DIMACS-formatted CNF files, recording the clause structure and tracking the frequency of variable appearances, a crucial feature later used for experimental analysis.

Figure 6.2.2 shows the transformation of a parsed CNF formula into a QUBO representation by computing clause-based penalties.

```
x = {var: Binary(f"x{var}") for var in range(1, num_vars + 1)}
penalty_weight = 3
H = 0
for clause in cnf_formula:
    clause_penalty = 1 - sum(x[abs(var)] if var > 0 else (1 - x[abs(var)])) for var in clause)
    H += penalty_weight * clause_penalty**2
```

Figure 6.2.2 – Construction of the QUBO Model from CNF Clauses

Each clause was mapped to a quadratic penalty term. The global Hamiltonian H penalizes unsatisfied clauses, producing an optimization landscape suitable for quantum annealing.

Finally, Figure 6.2.3 illustrates the final step of the pipeline: compiling the QUBO model into a BQM and calling the embedding function using MinorMiner.

```
model = H.compile()
qubo, _ = model.to_qubo()
bqm = dimod.BinaryQuadraticModel.from_qubo(qubo)
qubo_graph = dimod.to_networkx_graph(bqm)

embedding = minorminer.find_embedding(list(qubo_graph.edges), chimera_graph.edges)
```

Figure 6.2.3 – BQM Compilation and Embedding Using MinorMiner

After compiling the QUBO into a BQM graph, MinorMiner was used to find minor embeddings into a 4x4x4 Chimera topology. The runtime of this step was the primary performance metric studied in this chapter.

6.3 Parameters Investigated

To better understand how the structure of SAT instances affects the embedding process, we systematically varied a number of key parameters during our experiments.

First, we explored the clause size by generating 2-SAT, 3-SAT, and 4-SAT formulas. Larger clauses tend to create denser connections between variables, which can influence how difficult the resulting graph is to embed onto a Chimera topology.

Next, we varied the number of variables while keeping the number of clauses fixed. This allowed us to study how the size of the logical graph affects embedding time, without changing the problem’s overall constraint density.

We also experimented with changing the number of clauses. By increasing the number of clauses for a fixed set of variables, we could observe how added constraint density impacts the graph structure and the complexity of finding a valid embedding.

Finally, we introduced high-frequency variable reuse into some formulas. This means that certain variables were forced to appear more frequently across multiple clauses. We expected that such variables would become highly connected nodes in the graph, potentially making embedding harder and increasing execution time.

In all cases, care was taken to adjust one parameter at a time, so that we could clearly measure its individual effect on MinorMiner’s embedding performance. The next section presents and analyzes the results from these experiments.

6.4 Results and Analysis

6.4.1 Effect of Clause Size

The first set of experiments examined how increasing the clause size affects the difficulty of embedding logical graphs. In these tests, all instances used 10 variables and 10 clauses, varying only the clause size.

Clause Size	Embedding Time(ms)
2-SAT	1.00
3-SAT	4.06
4-SAT	5.47

Figure 6.4.1 – Embedding Time vs Clause Size

The relationship is illustrated in Figure 6.4.2.

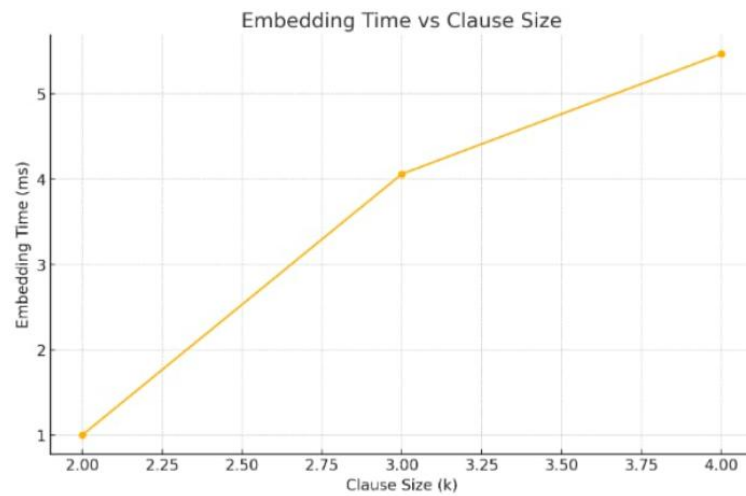


Figure 6.4.2 – Embedding Time vs Clause Size

Analysis:

As clause size grows, the resulting QUBO graphs become denser. Larger clauses connect more variables simultaneously, increasing the number of quadratic terms. This added connectivity complicates the embedding, requiring MinorMiner to find more intricate mappings, thus leading to longer execution times.

6.4.2 Effect of Variable Count

In the second set of experiments, the number of clauses was fixed at 10, and the number of variables was varied from 10 to 100.

Variables	Embedding Time(ms)
10	1.00
20	1.04
30	2.55
40	3.20
50	2.51
60	2.63
70	2.99
80	3.32
90	2.74
100	2.67

Figure 6.4.3 – Embedding Time vs Number of Variables

This trend is shown in Figure 6.4.4.

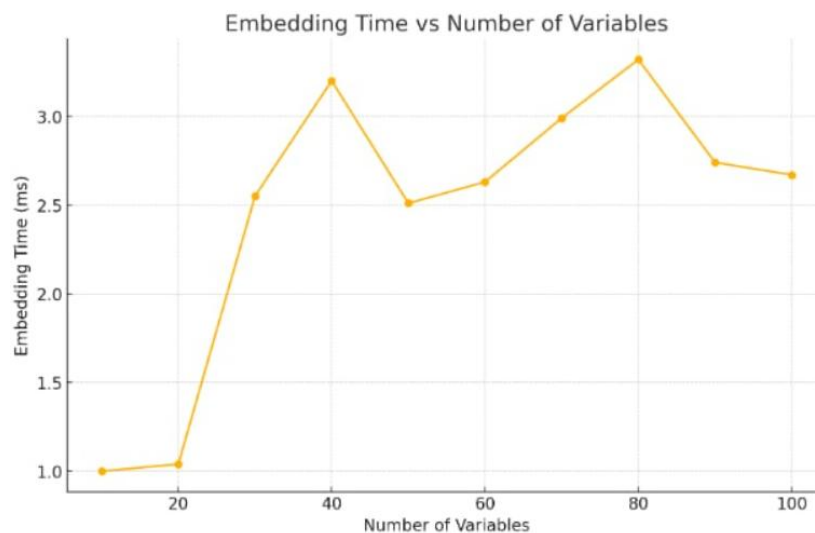


Figure 6.4.4 – Embedding Time vs Number of Variables

Analysis:

The embedding time increases slightly as more variables are introduced, but the impact is relatively mild. Increasing the number of variables does not immediately lead to a denser or more connected graph unless the number of clauses also grows proportionally.

6.4.3 Effect of Clause Count

The third experiments fixed the number of variables at 10 and varied the number of clauses from 10 to 100.

Clauses	Embedding Time(ms)
10	1.00
20	4.99
30	7.89
40	15.26
50	14.80
60	16.23
70	13.83
80	16.76
90	18.67
100	17.12

Figure 6.4.5 – Embedding Time vs Clause Count

The result is visualized in Figure 6.4.6.

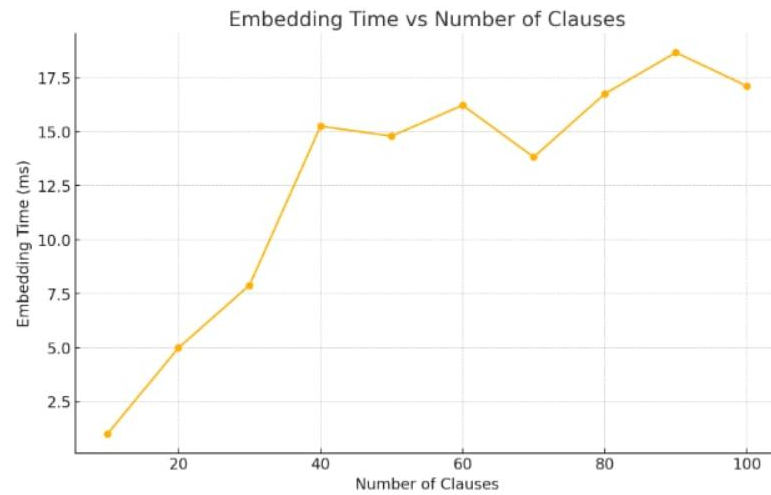


Figure 6.4.6 – Embedding Time vs Number of Clauses

Analysis:

Adding clauses greatly increases graph density because each new constraint introduces additional connections between variables. This increases the overall complexity of the QUBO graph, resulting in longer chains during minor embedding and consequently higher computational effort.

6.4.4 Effect of High-Frequency Variables

Finally, the impact of variable reuse was studied by intentionally making some variables appear more frequently across clauses.

High-Frequency Variables	Embedding Time(ms)
0	2.55
1	8.98
2	11.93
3	9.88
4	28.30
5	23.36

Table 6.4.7 – Embedding Time vs High-Frequency Variables

The behaviour is plotted in Figure 6.4.8.

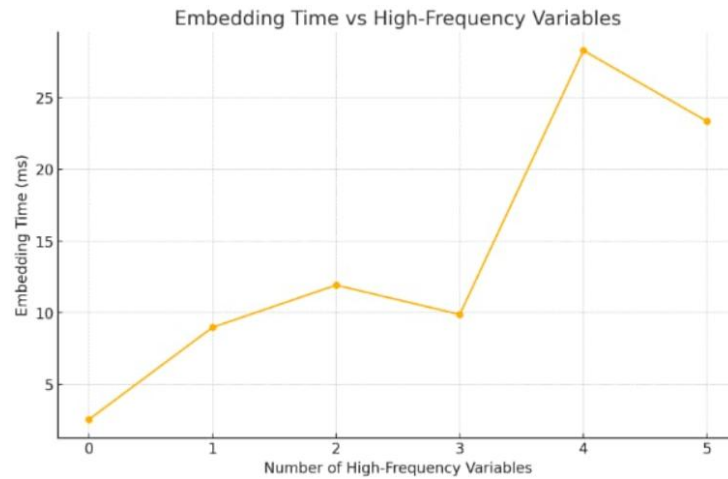


Figure 6.4.8 – Embedding Time vs High-Frequency Variables

Analysis:

Embedding time rises significantly when variables are reused heavily. Variables appearing in multiple clauses create "hotspots" that must be represented by longer chains of physical qubits during embedding. This places a substantial burden on MinorMiner's search, increasing the time and difficulty of finding a feasible mapping.

6.4.5 Overall Comparison

To provide a comprehensive perspective on the relative influence of different SAT instance characteristics on minor embedding complexity, all experimental results were consolidated into a single comparative graph (Figure 6.4.9). This unified visualization enables a direct comparison of how clause size, variable count, clause count, and high-frequency variable reuse impact embedding time.

The comparative analysis reveals several important trends. Firstly, increasing clause size from 2-SAT to 4-SAT moderately raises embedding time, reflecting the denser connectivity introduced by larger clauses. Secondly, variations in variable count, while keeping the number of clauses fixed, result in relatively minor changes in embedding time. This indicates that the mere addition of variables does not substantially increase graph complexity unless accompanied by a corresponding increase in constraint density.

In contrast, increasing the number of clauses significantly affects embedding performance. As more clauses are introduced, the logical graphs become denser, leading to longer embedding times due to the more complex interactions that MinorMiner must handle.

Finally, the reuse of high-frequency variables exhibits the most pronounced effect. When certain variables appear repeatedly across multiple clauses, the resulting graph contains highly connected nodes, or “hotspots,” which impose a substantial burden on the embedding process. Embedding times rise sharply in such cases, indicating a strong correlation between localized graph density and embedding difficulty.

Overall, the results demonstrate that constraint density, whether caused by an excess of clauses or by high-frequency variable reuse, is the primary factor that exacerbates embedding complexity. These findings highlight the importance of maintaining sparsity in logical graphs when targeting quantum hardware platforms for satisfiability problems.

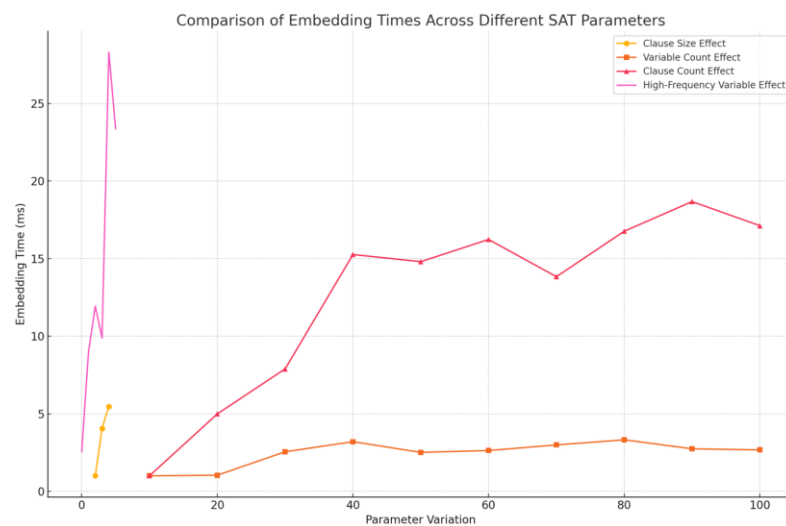


Figure 6.4.9 – Comparison of Embedding Times Across Different SAT Parameters

6.5 Conclusion

This chapter analyzed how structural properties of SAT instances affect the complexity of embedding their logical graphs into quantum hardware topologies. Through systematic experiments, we observed that larger clause sizes and variable counts moderately increased embedding times, while higher clause counts and frequent variable reuse had a much stronger impact.

Overall, the results highlight that constraint density, whether caused by an excess of clauses or by highly reused variables, is the primary factor making minor embedding more challenging. These findings emphasize the importance of designing sparse logical graphs for efficient quantum embedding.

Chapter 7

Conclusions and Future Work

7.1 General Conclusions	78
7.2 Future Work	79

7.1 General Conclusions

This thesis investigated the complete pipeline for solving Boolean satisfiability (SAT) problems through quantum annealing, starting from logical formulation in Conjunctive Normal Form (CNF), through conversion into Quadratic Unconstrained Binary Optimization (QUBO) models, Binary Quadratic Models (BQM), and finally embedding onto Chimera quantum hardware graphs. Through systematic experiments and analysis, the work demonstrated how structural properties of SAT instances, particularly clause count and variable frequency, critically affect the difficulty and time needed for minor embedding. By carefully controlling the generation of SAT instances and isolating parameters such as clause size, number of variables, number of clauses, and variable reuse frequency, it was possible to observe and quantify their individual influence on embedding complexity.

The experimental results revealed that increased constraint density, either through more clauses or the reuse of high-frequency variables, leads to substantially higher embedding times. In contrast, variations in clause size and the number of variables had a milder effect, unless they indirectly caused denser connectivity in the logical graph. It was also shown that heuristic solvers, such as D-Wave's Minorminer, greatly outperform classical constraint solvers like Clingo and MiniZinc when scalability and speed are required.

Minorminer achieved sub-second embeddings across over a thousand instances, while exact solvers struggled with increasing problem sizes, often encountering timeouts or returning infeasibility results. Attempts to refine Minorminer embeddings through classical optimization methods such as Large Neighborhood Search and meta-Answer Set Programming did not yield better embeddings, suggesting that heuristic solutions are already close to optimal under current hardware and constraint models.

Beyond the practical achievements, this work highlights important strategic insights for quantum optimization workflows. When preparing problems for quantum annealing, attention to graph sparsity and constraint distribution becomes crucial for ensuring feasible and efficient embeddings. Furthermore, the results underscore that heuristic methods are not merely approximate but are sometimes the only practical means to achieve embedding in reasonable time, especially for large and densely constrained instances. Altogether, this thesis provides a solid foundation for understanding and improving the process of adapting SAT problems to quantum annealing systems and offers valuable guidance for the design and preparation of quantum-amenable optimization tasks.

7.2 Future Work

Several avenues for future research arise naturally from the findings of this thesis. A clear extension would be to replicate the experimental setup on more advanced hardware topologies, particularly the Pegasus graph used in D-Wave’s newer quantum processors. The higher connectivity of Pegasus could potentially reduce the difficulty of embedding dense graphs and alter the scaling behaviour observed in this work. Furthermore, exploring other types of combinatorial optimization problems beyond SAT, including Max-SAT, constraint satisfaction problems, or graph colouring, would broaden the understanding of embedding challenges across domains.

Another promising direction would be the development of adaptive or hybrid embedding strategies that combine heuristic and constraint-based approaches, allowing the system to switch between fast heuristics and exact methods depending on problem features.

Machine learning models that predict embedding difficulty based on graph metrics could also be integrated into preprocessing workflows, providing smart heuristics for solver and strategy selection before actual embedding attempts are made.

Finally, a deeper study connecting embedding characteristics to actual quantum annealing performance could be conducted, closing the loop from SAT formulation all the way to solution sampling and postprocessing. Such an integrated view would help in assessing how embedding quality impacts final solution quality on quantum devices and provide critical feedback for further refinement of SAT-to-quantum workflows.

References

- [1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 2010.
- [2] C. C. McGeoch, *Adiabatic Quantum Computation and Quantum Annealing: Theory and Practice*, Morgan & Claypool, 2014.
- [3] T. Kadowaki and H. Nishimori, “Quantum annealing in the transverse Ising model,” *Physical Review E*, vol. 58, no. 5, pp. 5355, 1998.
- [4] A. Lucas, “Ising formulations of many NP problems,” *Frontiers in Physics*, vol. 2, p. 5, 2014.
- [5] D-Wave Systems Inc., *Ocean SDK Documentation*, 2023. [Online]. Available: <https://docs.ocean.dwavesys.com>
- [6] V. Choi, “Minor-embedding in adiabatic quantum computation: I. The parameter setting problem,” *Quantum Information Processing*, vol. 7, no. 5, pp. 193–209, 2008.
- [7] T. Hadzic and J. N. Hooker, “Postsolution analysis for integer programming,” *INFORMS Journal on Computing*, vol. 16, no. 3, pp. 259–276, 2004.
- [8] S. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge University Press, 2004.
- [9] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009.
- [10] M. Sipser, *Introduction to the Theory of Computation*, Cengage Learning, 2012.
- [11] A. Biere, *MiniSAT User Guide*, 2008.
- [12] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, “MiniZinc: Towards a standard CP modelling language,” in *Proc. Principles and Practice of Constraint Programming (CP)*, 2007.
- [13] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, “Clingo = ASP + Control: Preliminary Report,” *arXiv preprint arXiv:1405.3694*, 2014.
- [14] E. Boros and P. L. Hammer, “Pseudo-Boolean optimization,” *Discrete Applied Mathematics*, vol. 123, no. 1–3, pp. 155–225, 2002.
- [15] D-Wave Systems, *Understanding Minor Embedding* [YouTube Video], 2020. [Online]. Available: <https://youtu.be/S6fz0s8zI3Y>

- [16] D-Wave Systems, *QPU Topology – Ocean Documentation*, [Online]. Available: https://docs.ocean.dwavesys.com/en/stable/docs_qpu.html
- [17] D-Wave Systems, *qbsolv Documentation*, 2023. [Online]. Available: <https://docs.ocean.dwavesys.com/projects/qbsolv>
- [18] D-Wave Systems, *dimod: A Shared API for Binary Quadratic Models*, [Online]. Available: https://docs.ocean.dwavesys.com/en/stable/docs_dimod/intro/using_dimod.html
- [19] D-Wave Systems, *Minor-Embedding – Ocean Documentation*, [Online]. Available: <https://docs.ocean.dwavesys.com>
- [20] D-Wave Systems, *minorminer Documentation*, 2023. [Online]. Available: <https://github.com/dwavesystems/minorminer>
- [21] D-Wave Systems, *Next-Generation Topology of D-Wave Quantum Processors*, Whitepaper, 2020.
- [22] T. Boothby, J. King, and A. Roy, “Fast clique minor generation in Chimera qubit connectivity graphs,” *Quantum Information Processing*, vol. 15, pp. 495–508, 2020. [Online]. Available: <https://arxiv.org/abs/1507.04774>
- [23] D-Wave Systems, *Understanding Samplers in Ocean SDK*, 2023. [Online]. Available: <https://docs.ocean.dwavesys.com/en/stable/concepts/samplers.html>
- [24] M. Heule, *CNFgen GitHub Repository*. [Online]. Available: <https://github.com/marijnheule/CNFgen>
- [25] D-Wave Systems Inc., *Hybrid Solver Examples – dwave-hybrid documentation*, 2023. [Online]. Available: <https://docs.ocean.dwavesys.com/projects/hybrid/en/latest/>
- [26] C. Cai, W. G. Macready, and A. Roy, “A practical heuristic for finding graph minors,” *arXiv preprint arXiv:1406.2741*, 2014.
- [27] P. Erdős and A. Rényi, “On random graphs I,” *Publicationes Mathematicae*, vol. 6, pp. 290–297, 1959.
- [28] B. O’Gorman, R. Babbush, and A. Aspuru-Guzik, “An error suppression technique for adiabatic quantum optimization,” *Scientific Reports*, vol. 5, 11182, 2015.
- [29] MiniZinc, *Gecode Solver Library Reference – relax_and_reconstruct*, 2023. [Online]. Available: <https://docs.minizinc.dev/en/2.4.3/lib-gecode.html>
- [30] M. Sugimori, M. Takemura, and K. Inoue, “On Improving Answer Set Programs via Meta-Level Search,” in *Proc. 20th Int. Conf. on Principles of Knowledge*

Representation and Reasoning (KR 2024). [Online]. Available:
<https://proceedings.kr.org/2024/72/kr2024-0072-sugimori-et-al.pdf>

Appendix A

qubo_to_bqm_dwave_submission.py

```
1  from dimod import BinaryQuadraticModel
2  from dwave.system import EmbeddingComposite, DWaveSampler
3
4  # Penalty constant for auxiliary variable constraint
5  M = 10
6
7  # Define the QUBO matrix (4 variables: x1, x2, x3, z)
8  qubo = {
9      # Diagonal terms (linear coefficients)
10     (0, 0): 0,          # x1
11     (1, 1): 1 + M,      # x2
12     (2, 2): 0,          # x3
13     (3, 3): 2 * M,      # z (auxiliary variable)
14
15     # Off-diagonal terms (interaction coefficients)
16     (0, 1): -M,         # x1 * x2
17     (0, 3): M / 2,      # x1 * z
18     (1, 3): M / 2,      # x2 * z
19     (2, 3): 1 / 2,      # x3 * z
20     (0, 2): 1 / 2,      # x1 * x3
21     (1, 2): -1 / 2      # x2 * x3
22 }
23
24 # Create the Binary Quadratic Model
25 bqm = BinaryQuadraticModel.from_qubo(qubo)
26
27 # Use D-Wave's quantum annealer
28 sampler = EmbeddingComposite(DWaveSampler())
29 response = sampler.sample(bqm, num_reads=100)
30
31 # Print the solutions and their energies
32 print("Solutions:")
33 for sample, energy in response.data(['sample', 'energy']):
34     print(f"Sample: {sample}, Energy: {energy}")
```

Appendix B

submit_stitched_bqm.py

```
import dwavebinarycsp as dbcsp
import dimod
import minorminer
from dwave.system import DWaveSampler, FixedEmbeddingComposite
from dwave.preprocessing import ScaleComposite

with open('testSAT3.cnf', 'r') as fp:
    csp = dbcsp.cnf.load_cnf(fp)

print("LOADED")

bqm = dbcsp.stitch(csp)
print("CONVERTED\n\n")

from dwave.system import DWaveSampler, EmbeddingComposite
print(bqm)

qpu_adv = DWaveSampler(token="DEV-ecf9c571a4cce7d480d8febd7c6388843e9e9fd",
                        solver = {'topology__type': 'pegasus'})
sampler=qpu_adv
embedding = minorminer.find_embedding(
    dimod.to_networkx_graph(bqm), sampler.to_networkx_graph()
)

print(embedding)

print("CALLING SOLVER")

sampleset = FixedEmbeddingComposite(
    ScaleComposite(sampler),
    embedding=embedding,
).sample(
    bqm,
    num_reads=5000,
)

print(sampleset.first.energy)

print("SOLVER OK")
```

Appendix C

generate_cnf.py

```
import random
import subprocess
import shutil
from sat import Clause, Formula

def generate_random_cnf(output_file, num_variables, num_clauses):
    """
    Generate a CNF file with mixed 2-SAT, 3-SAT, and 4-SAT clauses.
    """
    num_2sat = random.randint(0, num_clauses)
    num_3sat = random.randint(0, num_clauses - num_2sat)
    num_4sat = num_clauses - (num_2sat + num_3sat)

    clauses = []
    for _ in range(num_2sat):
        clauses.append(Clause.random(num_variables, 2))
    for _ in range(num_3sat):
        clauses.append(Clause.random(num_variables, 3))
    for _ in range(num_4sat):
        clauses.append(Clause.random(num_variables, 4))

    with open(output_file, "w") as f:
        f.write(f"p cnf {num_variables} {num_clauses} {num_2sat} {num_3sat} {num_4sat}\n")
        for clause in clauses:
            clause_list = []
            for i in range(num_variables):
                if clause.vars_pos & (1 << i):
                    clause_list.append(i + 1)
                if clause.vars_neg & (1 << i):
                    clause_list.append(-(i + 1))
            f.write(" ".join(map(str, clause_list)) + " 0\n")

    print(f"✅ Random CNF file '{output_file}' generated.")

def generate_cnfgen_cnf(output_file, formula_type, num_variables, extra_args=""):
    """
    Generate a CNF file using CNFgen.
    """
    cnfgen_path = shutil.which("cnfgen")
    if cnfgen_path is None:
        print(f"❌ Error: CNFgen is not installed or not in PATH.")
        return

    if not extra_args:
        if formula_type == "randkcnf":
            extra_args = f"3 {num_variables} 100" # Default: 3-SAT, 100 clauses
        elif formula_type == "php":
            extra_args = "5" # Default: PHP(5)
        else:
            print(f"❌ Error: Missing arguments for CNFgen formula type '{formula_type}'.")
            return

    command = f"{cnfgen_path} {formula_type} {extra_args} > {output_file}"
    try:
        subprocess.run(command, shell=True, check=True)
        print(f"✅ CNFgen formula '{formula_type}' generated in '{output_file}'.")
    except subprocess.CalledProcessError:
        print(f"❌ Error: CNFgen execution failed.")
```

```

if __name__ == "__main__":
    output_file = "generatedSAT.cnf"
    method = input("Choose method (random/cnfgen/controlled): ").strip().lower()
    num_variables = int(input("Enter number of variables: "))

    if method == "random":
        num_clauses = int(input("Enter number of clauses: "))
        generate_random_cnf(output_file, num_variables, num_clauses)

    elif method == "cnfgen":
        formula_type = input("Enter CNFgen formula type (e.g., php, randknf): ").strip()
        extra_args = input("Enter any extra arguments for CNFgen (leave empty for defaults): ").strip()

        if not extra_args:
            if formula_type == "randknf":
                extra_args = f"3 {num_variables} 100"
            elif formula_type == "php":
                extra_args = "5"

        generate_cnfgen_cnf(output_file, formula_type, num_variables, extra_args)

    elif method == "controlled":
        num_clauses = int(input("Enter number of clauses: "))
        high_freq_vars = list(map(int, input("Enter high-frequency variables (comma-separated): ").split(",")))
        high_freq_count = int(input("Enter how many extra times these variables should appear: "))

        generate_controlled_cnf(output_file, num_variables, num_clauses, high_freq_vars, high_freq_count)

    else:
        print("❌ Invalid method selected.")

def generate_controlled_cnf(output_file, num_variables, num_clauses, high_freq_vars, high_freq_count):
    """
    Generate a CNF file where certain variables appear more frequently.

    Args:
        output_file (str): Output file name.
        num_variables (int): Number of variables.
        num_clauses (int): Number of clauses.
        high_freq_vars (list): List of variables to appear more often.
        high_freq_count (int): Additional occurrences for high-frequency variables.
    """
    clauses = []
    all_vars = list(range(1, num_variables + 1))

    for _ in range(num_clauses):
        clause_size = random.randint(2, 4) # Clause size (2-SAT to 4-SAT)
        clause = random.sample(all_vars, clause_size) # Select variables randomly

        # Ensure high-frequency variables appear more often
        if random.random() < 0.5: # 50% probability to insert high-frequency vars
            clause.append(random.choice(high_freq_vars))

        clause = [var if random.choice([True, False]) else -var for var in clause] # Random negation
        clauses.append(clause)

    # Write CNF file
    with open(output_file, "w") as f:
        f.write(f"p cnf {num_variables} {num_clauses}\n")
        for clause in clauses:
            f.write(" ".join(map(str, clause)) + " 0\n")

    print(f"✅ Controlled CNF file '{output_file}' generated with high-frequency variables: {high_freq_vars}")

```

Appendix D

benchmark_classical_solvers.py

```
import dwavebinarycsp as dbcsp
import dimod
from dwave.samplers import SimulatedAnnealingSampler, SteepestDescentSolver, TabuSampler
import sys

def main():
    if len(sys.argv) != 2:
        print("Usage: python solve_sat.py <cnf_file>")
        sys.exit(1)

    cnf_file = sys.argv[1]

    with open(cnf_file, 'r') as fp:
        csp = dbcsp.cnf.load_cnf(fp)

    print("LOADED")

    bqm = dbcsp.stitch(csp)
    print("CONVERTED\n\n")

    solvers = [
        ("TabuSampler", TabuSampler(), 3000),
        ("SimulatedAnnealingSampler", SimulatedAnnealingSampler(), 5000),
        ("SteepestDescentSolver", SteepestDescentSolver(), 5000)
    ]

    for solver_name, sampler, num_reads in solvers:
        print(f"Running {solver_name} with num_reads={num_reads}\n")
        response = sampler.sample(bqm, num_reads=num_reads)
        print(f"{solver_name} Energy: {response.first.energy}\n")
        print("SOLVER OK\n")

if __name__ == "__main__":
    main()
```

Appendix E

hybrid_racing_solver.py

```
import dimod
import dwave.system
import hybrid
import dwavebinarycsp as dbcsp

with open('generatedSAT.cnf', 'r') as fp:
    csp = dbcsp.cnf.load_cnf(fp)

bqm = dbcsp.stitch(csp)
print("CONVERTED\n\n")

print("Problem OK")

# Define the hybrid workflow
workflow = hybrid.Loop(
    hybrid.RacingBranches(
        hybrid.SimulatedAnnealingProblemSampler(),
        hybrid.TabuProblemSampler(),
    ) | hybrid.ArgMin(),
    max_iter=200 # Stops after iter iterations
)

print("Workflow OK")

# Run the workflow
init_state = hybrid.State.from_problem(bqm)
print("initial state OK")
final_state = workflow.run(init_state).result()
print("Final state OK")

# Get the best solution found
best_solution = final_state.samples.first
print("Best solution:", best_solution)
print("Energy:", best_solution.energy)
```


Appendix F

component_decomposer_test.py

```
import dimod
from hybrid.decomposers import ComponentDecomposer
from hybrid.core import State
from hybrid.utils import random_sample
import dwavebinarycsp as dbcsp

with open('generatedSAT.cnf', 'r') as fp:
    csp = dbcsp.cnf.load_cnf(fp)

bqm = dbcsp.stitch(csp)
print("CONVERTED\n\n")

state0 = State.from_sample(random_sample(bqm), bqm)

decomposer = ComponentDecomposer(key=len)
state1 = decomposer.next(state0).result()
state2 = decomposer.next(state1).result()

print(list(state1.subproblem.variables))
print(list(state2.subproblem.variables))
```

Appendix G

hybrid_custom_branch_demo.py

```
import dimod
from hybrid.decomposers import EnergyImpactDecomposer
from hybrid.composers import SplatComposer
from hybrid.core import State
from hybrid.utils import random_sample
from hybrid.utils import min_sample
import dwavebinarycsp as dbcsp
from hybrid.samplers import TabuSubproblemSampler
import hybrid

bqm = dimod.BQM({t: 0 for t in range(10)},
                {(t, (t+1) % 10): 1 for t in range(10)},
                0, 'SPIN')
# Run one iteration on a branch
branch = (EnergyImpactDecomposer(size=6, min_gain=-10) |
          TabuSubproblemSampler(num_reads=2) |
          SplatComposer())
new_state = branch.next(State.from_sample(min_sample(bqm), bqm))
print(new_state.subsamples)
```

Appendix H

benchmark_embedding_solvers.py

```
import networkx as nx
import dwave_networkx as dnx
import subprocess
import os
import time

# Paths for Clingo & MiniZinc
clingo_path = "/home/students/cs/2021/ckonst04/.local/bin/clingo" # Update with your Clingo path
minizinc_path = "/opt/minizinc_2.8.7/bin/minizinc" # Update with your MiniZinc path
embed_file = os.path.abspath("embed444.pl") # Clingo model
minizinc_model = "graph.mzn" # MiniZinc model

# Output log file
log_file = "output.log"

# Time limit for execution (in seconds)
timeout_limit = 120 # 2 minutes

# List of Chimera graphs to process
chimera_graphs = [
    dnx.chimera_graph(2, 4, 2),
    dnx.chimera_graph(4, 2, 2),
    dnx.chimera_graph(4, 4, 2),
    dnx.chimera_graph(8, 4, 2),
    dnx.chimera_graph(4, 8, 2),
    dnx.chimera_graph(4, 4, 4),
    dnx.chimera_graph(8, 4, 4),
    dnx.chimera_graph(4, 8, 4)
]

# Density values to iterate over
densities = [0.05, 0.08, 0.1, 0.12, 0.14]

# MiniZinc solvers to use
minizinc_solvers = {
    "Chuffed": "org.chuffed.chuffed",
    "COIN-BC": "org.minizinc.mip.coin-bc",
    "Gecode": "org.gecode.gecode",
    "OR-Tools CP-SAT": "cp-sat"
}
```

```

# Function to generate a random source graph
def generate_random_graph(num_nodes, density):
    return nx.erdos_renyi_graph(n=num_nodes, p=density)

# Function to write graphs to an ASP file "graph.lp" (for Clingo)
def write_graph_to_lp(source_graph, chimera_graph, filename="graph.lp"):
    abs_filename = os.path.abspath(filename)

    with open(abs_filename, "w") as f:
        for node in source_graph.nodes:
            f.write(f"node({node}).\n")
        for u, v in source_graph.edges:
            f.write(f"adjs({u},{v}).\n")
            f.write(f"adjs({v},{u}).\n")
        for node in chimera_graph.nodes:
            f.write(f"chimera({node}).\n")
        for u, v in chimera_graph.edges:
            f.write(f"adjt({u},{v}).\n")
            f.write(f"adjt({v},{u}).\n")

# Function to write graphs to a MiniZinc data file "graph.dzn"
def write_graph_to_dzn(source_graph, chimera_graph, filename="graph.dzn"):
    abs_filename = os.path.abspath(filename)

    num_snodes = len(source_graph.nodes)
    num_tnodes = len(chimera_graph.nodes)

    with open(abs_filename, "w") as f:
        f.write(f"num_snodes = {num_snodes};\n")
        f.write(f"num_tnodes = {num_tnodes};\n")

        # Writing adjacency matrix for logical nodes (source graph)
        f.write(f"adjs = array2d(0..{0}, 0..{0}, [\n".format(num_snodes - 1))
        for i in range(num_snodes):
            row = ["true" if (i, j) in source_graph.edges or (j, i) in source_graph.edges else "false" for j in range(num_snodes)]
            f.write(", ".join(row) + ",\n" if i < num_snodes - 1 else ", ".join(row) + "\n")
        f.write("]);\n")

        # Writing adjacency matrix for Chimera nodes (target graph)
        f.write(f"adjt = array2d(0..{0}, 0..{0}, [\n".format(num_tnodes - 1))
        for i in range(num_tnodes):
            row = ["true" if (i, j) in chimera_graph.edges or (j, i) in chimera_graph.edges else "false" for j in range(num_tnodes)]
            f.write(", ".join(row) + ",\n" if i < num_tnodes - 1 else ", ".join(row) + "\n")
        f.write("]);\n")

# Function to run Clingo with a timeout
def run_clingo(graph_file, execution_count, chimera_idx, num_nodes, density):
    start_time = time.time()
    abs_graph_file = os.path.abspath(graph_file)
    command = [clingo_path, embed_file, abs_graph_file]

    try:
        result = subprocess.run(command, capture_output=True, text=True, timeout=timeout_limit)
        end_time = time.time()
        execution_time = end_time - start_time

        sat_status = "SAT" if "assign" in result.stdout else "UNSAT"

        with open(log_file, "a") as log:
            log.write(f"\n{' '*50}\n")
            log.write(f"Execution {execution_count}: Clingo\n")
            log.write(f"Graph: Chimera Graph {chimera_idx + 1}\n")
            log.write(f"Density: {density}\n")
            log.write(f"Nodes: {num_nodes}\n")
            log.write(f"Execution Time: {execution_time:.4f} seconds\n")
            log.write(f"Result: {sat_status}\n")
            log.write(f"{' '*50}\nClingo Output:\n{result.stdout}\n")

        print(f"Execution {execution_count}: Clingo | Time: {execution_time:.2f}s | {sat_status}")

    except subprocess.TimeoutExpired:
        end_time = time.time()
        execution_time = end_time - start_time
        with open(log_file, "a") as log:
            log.write(f"\nExecution {execution_count}: Clingo Timeout ({execution_time:.2f}s) | UNSAT\n")
        print(f"Execution {execution_count}: Clingo | Timeout ({execution_time:.2f}s) | UNSAT")

```

```

# Function to run MiniZinc with multiple solvers
def run_minizinc(execution_count, chimera_idx, num_nodes, density):
    for solver_name, solver_id in minizinc_solvers.items():
        start_time = time.time()
        command = [minizinc_path, "--solver", solver_id, minizinc_model, "graph.dzn"]

        try:
            result = subprocess.run(command, capture_output=True, text=True, timeout=timeout_limit)
            end_time = time.time()
            execution_time = end_time - start_time

            sat_status = "SAT" if "assign" in result.stdout else "UNSAT"

            with open(log_file, "a") as log:
                log.write(f'\n{' '*50}\n')
                log.write(f"Execution {execution_count}: MiniZinc ({solver_name})\n")
                log.write(f"Graph: Chimera Graph {chimera_idx + 1}\n")
                log.write(f"Density: {density}\n")
                log.write(f"Nodes: {num_nodes}\n")
                log.write(f"Execution Time: {execution_time:.4f} seconds\n")
                log.write(f"Result: {sat_status}\n")
                log.write(f"{' '*50}\nMiniZinc Output ({solver_name}): \n{result.stdout}\n")

            print(f"Execution {execution_count}: MiniZinc ({solver_name}) | Time: {execution_time:.2f}s | {sat_status}")

        except subprocess.TimeoutExpired:
            end_time = time.time()
            execution_time = end_time - start_time
            with open(log_file, "a") as log:
                log.write(f"\nExecution {execution_count}: MiniZinc ({solver_name}) Timeout ({execution_time:.2f}s) | UNSAT\n")
            print(f"Execution {execution_count}: MiniZinc ({solver_name}) | Timeout ({execution_time:.2f}s) | UNSAT")

# Main execution loop
execution_count = 1

for idx, chimera in enumerate(chimera_graphs):
    num_chimera_nodes = len(chimera.nodes)

    for density in densities:
        num_nodes = int(0.5 * num_chimera_nodes)
        max_nodes = int(0.7 * num_chimera_nodes)

        while num_nodes <= max_nodes:
            for i in range(5):
                print(f"Executing {execution_count}: Chimera Graph {idx + 1} | Density {density} | Nodes {num_nodes}")

                source_graph = generate_random_graph(num_nodes, density)

                write_graph_to_lp(source_graph, chimera, filename="graph.lp")
                write_graph_to_dzn(source_graph, chimera, filename="graph.dzn")

                run_clingo("graph.lp", execution_count, idx, num_nodes, density)
                run_minizinc(execution_count, idx, num_nodes, density)

                execution_count += 1

            num_nodes += 5

print(f"\n🟢 Output saved to '{log_file}'")

```

Appendix I

extended_embedding_test.py

```
import networkx as nx
import dwave_networkx as dnx
import subprocess
import os
import time

clingo_path = "/home/students/cs/2021/ckonst04/.local/bin/clingo"
minizinc_path = "/home/students/cs/2021/ckonst04/.local/minizinc/bin/minizinc"
embed_file = os.path.abspath("embed444.pl")
minizinc_model = "graph.mzn"
log_file = "output.log"

minizinc_solvers = {
    "Chuffed": "org.chuffed.chuffed",
    "COIN-BC": "org.minizinc.mip.coin-bc",
    "Gecode": "org.gecode.gecode",
    "OR-Tools CP-SAT": "cp-sat"
}

# Large Chimera graph
chimera = dnx.chimera_graph(8, 8, 4)
selected_densities = [0.1]
selected_num_nodes = [20, 25, 30, 35, 40, 45, 50, 55, 60, 65]

def generate_random_graph(num_nodes, density):
    return nx.erdos_renyi_graph(n=num_nodes, p=density)

def write_graph_to_lp(source_graph, chimera_graph, filename="graph.lp"):
    with open(filename, "w") as f:
        for node in source_graph.nodes:
            f.write(f"node({node}).\n")
        for u, v in source_graph.edges:
            f.write(f"ads({u},{v}).\n")
            f.write(f"ads({v},{u}).\n")
        for node in chimera_graph.nodes:
            f.write(f"chimera({node}).\n")
        for u, v in chimera_graph.edges:
            f.write(f"adjt({u},{v}).\n")
            f.write(f"adjt({v},{u}).\n")
```

```

def write_graph_to_dzn(source_graph, chimera_graph, filename="graph.dzn"):
    num_snodes = len(source_graph.nodes)
    num_tnodes = len(chimera_graph.nodes)
    with open(filename, "w") as f:
        f.write(f"num_snodes = {num_snodes};\n")
        f.write(f"num_tnodes = {num_tnodes};\n")

        f.write(f"adjs = array2d(0..{0}, 0..{0}, [\n".format(num_snodes - 1))
        for i in range(num_snodes):
            row = ["true" if (i, j) in source_graph.edges or (j, i) in source_graph.edges else "false" for j in range(num_snodes)]
            f.write(", ".join(row) + (",\n" if i < num_snodes - 1 else "\n"))
        f.write("]);\n")

        f.write(f"adjt = array2d(0..{0}, 0..{0}, [\n".format(num_tnodes - 1))
        for i in range(num_tnodes):
            row = ["true" if (i, j) in chimera_graph.edges or (j, i) in chimera_graph.edges else "false" for j in range(num_tnodes)]
            f.write(", ".join(row) + (",\n" if i < num_tnodes - 1 else "\n"))
        f.write("]);\n")

def run_clingo(graph_file, execution_count, num_nodes, density):
    start_time = time.time()
    command = [clingo_path, embed_file, os.path.abspath(graph_file)]

    try:
        result = subprocess.run(command, capture_output=True, text=True, timeout=3600)
        execution_time = time.time() - start_time
        sat_status = "SAT" if "assign(" in result.stdout or "embedded(" in result.stdout else "UNSAT"

        with open(log_file, "a") as log:
            log.write(f"\n{' '*50}\nExecution {execution_count}: Clingo\n")
            log.write(f"Density: {density} | Nodes: {num_nodes}\n")
            log.write(f"Time: {execution_time:.4f}s | Result: {sat_status}\n")
            log.write(f"\n{' '*50}\nClingo Output:\n{result.stdout}\n")

        print(f"Execution {execution_count}: Clingo | Time: {execution_time:.2f}s | {sat_status}")

    except Exception as e:
        print(f"Clingo Error: {e}")

def run_minizinc(execution_count, num_nodes, density):
    for solver_name, solver_id in minizinc_solvers.items():
        start_time = time.time()
        command = [minizinc_path, "--solver", solver_id, minizinc_model, "graph.dzn"]

        try:
            result = subprocess.run(command, capture_output=True, text=True, timeout=3600)
            execution_time = time.time() - start_time
            sat_status = "SAT" if "assign(" in result.stdout else "UNSAT"

            with open(log_file, "a") as log:
                log.write(f"\n{' '*50}\nExecution {execution_count}: MiniZinc ({solver_name})\n")
                log.write(f"Density: {density} | Nodes: {num_nodes}\n")
                log.write(f"Time: {execution_time:.4f}s | Result: {sat_status}\n")
                log.write(f"\n{' '*50}\nMiniZinc Output:\n{result.stdout}\n")

            print(f"Execution {execution_count}: MiniZinc ({solver_name}) | Time: {execution_time:.2f}s | {sat_status}")

        except Exception as e:
            print(f"MiniZinc ({solver_name}) Error: {e}")

execution_count = 1
for density in selected_densities:
    for num_nodes in selected_num_nodes:
        print(f"\n=== Execution {execution_count} ===")
        print(f"Density {density} | Nodes {num_nodes}")

        source_graph = generate_random_graph(num_nodes, density)
        write_graph_to_lp(source_graph, chimera, filename="graph.lp")
        write_graph_to_dzn(source_graph, chimera, filename="graph.dzn")

        run_clingo("graph.lp", execution_count, num_nodes, density)
        run_minizinc(execution_count, num_nodes, density)

        execution_count += 1

print("\n🟢 Output saved to 'output.log'")

```

Appendix J

embedding_runtime_analysis.py

```
import networkx as nx
import matplotlib.pyplot as plt
import minorminer
import dwave_networkx as dnx
import dimod
from pyqubo import Binary
import time
import os

# -----
# Step 1: Read CNF from File
# -----

def read_cnf_file(cnf_file):
    """ Reads a CNF file in DIMACS format and extracts variables and clauses. """
    clauses = []
    num_vars = 0
    var_counts = {}

    with open(cnf_file, "r") as f:
        for line in f:
            if line.startswith("c"): # Ignore comments
                continue

            if line.startswith("p cnf"): # Read problem line (e.g., "p cnf 5 6")
                parts = line.strip().split()
                num_vars = int(parts[2])
                continue

            clause = [int(x) for x in line.strip().split() if x != "0"]
            clauses.append(clause)

            # Count variable occurrences
            for var in clause:
                abs_var = abs(var)
                var_counts[abs_var] = var_counts.get(abs_var, 0) + 1

    return num_vars, clauses, var_counts
```



```

# Load CNF File
cnf_file = "generatedSAT.cnf"
start_time = time.time()
num_vars, cnf_formula, var_counts = read_cnf_file(cnf_file)
read_cnf_time = time.time() - start_time

print(f"\n🔍 Read CNF File: {cnf_file}")
print(f"Variables: {num_vars}, Clauses: {len(cnf_formula)}")
print(f"🕒 CNF Read Time: {read_cnf_time:.4f} seconds")

# -----
# Step 2: Convert CNF to QUBO
# -----

x = {var: Binary(f"x{var}") for var in range(1, num_vars + 1)}

penalty_weight = 3
H = 0

for clause in cnf_formula:
    clause_penalty = 1 - sum(x[abs(var)] if var > 0 else (1 - x[abs(var)]) for var in clause)
    H += penalty_weight * clause_penalty**2

start_time = time.time()
model = H.compile()
qubo, _ = model.to_qubo()
qubo_time = time.time() - start_time

bqm = dimod.BinaryQuadraticModel.from_qubo(qubo)
qubo_graph = dimod.to_networkx_graph(bqm)

print(f"🕒 QUBO Conversion Time: {qubo_time:.4f} seconds")

# -----
# Step 3: Save & Visualize QUBO Graph
# -----

def save_graph(graph, filename, title, bqm=None):
    """ Saves a NetworkX graph as an image file with proper edge weights. """
    plt.figure(figsize=(6, 6))
    pos = nx.spring_layout(graph)
    nx.draw(graph, pos, with_labels=True, node_color="lightblue", edge_color="gray", node_size=2000, font_size=16)

    edge_labels = {}
    for i, j in graph.edges():
        if bqm and ((i, j) in bqm.quadratic or (j, i) in bqm.quadratic):
            edge_labels[(i, j)] = f"{bqm.quadratic.get((i, j), bqm.quadratic.get((j, i), 0)):.1f}"

    nx.draw_networkx_edge_labels(graph, pos, edge_labels=edge_labels, font_size=12, font_color="red")
    plt.title(title)
    plt.savefig(filename, dpi=300)
    plt.close()

output_folder = "output_graphs"
os.makedirs(output_folder, exist_ok=True)

qubo_graph_path = os.path.join(output_folder, "qubo_graph.png")
save_graph(qubo_graph, qubo_graph_path, "QUBO Graph (Before Embedding)")

print(f"📁 Saved QUBO Graph to {qubo_graph_path}")

# Debugging QUBO Graph Info
print(f"\n🔍 QUBO Graph: {qubo_graph.number_of_nodes()} nodes, {qubo_graph.number_of_edges()} edges")

```

```

# -----
# Step 4: Find Minor-Miner Embedding
# -----

chimera_graph = dnx.chimera_graph(4, 4, 4)

start_time = time.time()
embedding = minorminer.find_embedding(list(qubo_graph.edges), chimera_graph.edges)
embedding_time = time.time() - start_time

print(f"🕒 Minor-Miner Embedding Time: {embedding_time:.4f} seconds")

# Handle failed embeddings
if not embedding:
    print("❌ Minorminer failed to find an embedding. Try increasing Chimera size.")
else:
    print(f"✅ Embedding Successful: {len(embedding)} logical qubits mapped.")

# -----
# Step 5: Save & Visualize Chimera Embedding
# -----

def save_chimera_embedding(chimera_graph, embedding, filename, title):
    """ Saves an improved visualization of the Chimera embedding. """
    plt.figure(figsize=(8, 8))
    dnx.draw_chimera_embedding(
        chimera_graph,
        embedding,
        show_labels=True,
        node_size=600
    )
    plt.title(title)
    plt.savefig(filename, dpi=300)
    plt.close()

chimera_embedding_path = os.path.join(output_folder, "chimera_embedding.png")
save_chimera_embedding(chimera_graph, embedding, chimera_embedding_path, "Embedded Graph on Chimera")

print(f"📁 Saved Improved Chimera Embedding to {chimera_embedding_path}")

# -----
# Step 6: Detailed Embedding Analysis
# -----

def analyze_embedding(var_counts, embedding):
    """ Analyzes the embedding difficulty by comparing variable occurrences and chain length. """

    print("\n🔍 Full Minorminer Embedding Mapping:")
    for logical_var, physical_qubits in embedding.items():
        print(f"Logical Qubit {logical_var} ➡ Physical Qubits {physical_qubits}")

    print("\n🔍 Embedding Analysis (Detailed View)")
    print("-" * 65)
    print(f"{'Logical Qubit':<15} {'Occurrences':<12} {'Chain Length':<12} {'Physical Qubits}'")
    print("-" * 65)

    for var in sorted(var_counts.keys()):
        occurrences = var_counts[var]
        physical_qubits = embedding.get(var, [])
        chain_length = len(physical_qubits) - 1 if len(physical_qubits) > 1 else 0
        print(f"{'var':<15} {'occurrences':<12} {'chain_length':<12} {'physical_qubits}'")

# Run the embedding analysis
analyze_embedding(var_counts, embedding)

# -----
# Step 7: Log Execution Times
# -----

print("\n🕒 Execution Times Summary:")
print(f"    - Read CNF Time: {read_cnf_time:.4f} sec")
print(f"    - QUBO Conversion Time: {qubo_time:.4f} sec")
print(f"    - Embedding Time: {embedding_time:.4f} sec")

```