UNIVERSITÀ
DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

Elaborato finale

Improved Techniques for SAT-to-Ising encoding applied to Hybrid Quantum Annealing

Advisor

Roberto Sebastiani

Student

Giuseppe Spallitta

Co-advisor

Marco Roveri

Anno accademico 2019/2020

Ringraziamenti

...thanks to...

Contents

Summary

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of first-order formulas with respect to various theories, such as real and integer arithmetic, bit vector and floating-point arithmetic. In the last decade, multiple SMT solvers have been developed to address these problems and have been applied in real-life applications, mainly in the formal verification field.
Some problem instances require not only to obtain a valid assignment of variables satisfying the formulas, but they also need to retrieve a model which is optimum with respect to a subset of objective functions. For instance, when timed and hybrid systems are taken into consideration, studying the minimum interval of time before an event happens can be useful to ensure safety properties are always valid. In order to deal with these problems, an extension of SMT was proposed known as Optimization Modulo Theories (OMT), adding to the SMT formulation the already cited objective functions to optimize.
While SMT has been extensively studied and its state-of-the-art is quite advanced, OMT is still in its first steps; only a few OMT solvers are available (one of them, OptiMathSAT, is developed in collaboration by UniTN and FBK) and a lot of progress has still to be done. In this thesis we will concentrate on the problem of effectively encoding SAT and MaxSAT problems into Quantum Annealers. The completion of this task is achieved using OMT solvers to determine the mapping of a Boolean Formula into qubits of the annealer, thus showing a real-life application of OMT. In particular we will discuss the problem of quantum co-tunneling, how it negatively affects the encoding provided by the solvers and a proposed approach to contrast this issue.

1 Introduction

1.1 The problem

SAT is one of the eldest and most popular NP-hard problems in Computer Science. Multiple approaches have been proposed to solve SAT problems more efficiently, given its applications in real-life contexts. Among them, the usage of quantum annealers (capable to solve a specific subset of a single optimization problem) to speed up the search of a valid solution has arisen in the last years. Even though the transition from SAT instances to problem accepted by the annealers is not a trivial task, a preliminary algorithm has been developed by the University of Trento with the collaboration of the Canadian company D-Wave.
The analysis of the original code, together with the ideas suggested at the end of the paper discussing the tool, raised some issues: in particular a side-effect of quantum computing, known as co-tunneling, negatively impacts the performances of the code in the search of an optimal solution satisfying the Boolean problem. Being able to determine a more stable encoding, not heavily affected by co-tunneling, will positively affect the performances of the tool and permit further investigations on the topic.

1.2 The solution

In this thesis, a novel approach to refine the Ising encoding is discussed. The algorithm, from now on referred as **postprocessing**, focuses on updating the encoding using qubits part of chains between penalty functions. The main goals of postprocessing are the reduction of the length of these chains and, when possible, the obtainment of a more stable Ising formulation. The solution is accompanied by an analysis to the current state-of-the-art tool, which was necessary to determine the aspects requiring improvements, and an empirical evaluation to demonstrate progresses and limits of the proposed contributions.

1.3 Structure of the thesis

This thesis is divided into two main sections: Chapter 2 and 3 will provide the context necessary to understand the relevant concepts defining the backbone of the field we will analyze, while from chapter 4 to chapter 6 we will focus on the novel contributions produced and their application. The detailed organization of chapters is the following:

- Chapter 2 will provide a comprehensive background on multiple logic problems such as SAT, SMT and OMT, offering a brief sight on fundamental algorithms and languages. It will also cover quantum computing and quantum annealing.
- Chapter 3 will outline the current state-of-the-art regarding the SAT-to-Ising problem, discussing algorithms and limitations.
- Chapter 4 will analyze the state-of-the-art tool; some tests have been reported to put in evidence some major issues and improvement points.
- Chapter 5 will talk about the CP-to-OMT problem, mentioning the current state-of-the-art interface, its shortcomings and the implementation of a novel interface.

- Chapter 6 will discuss the introduced improvements and their implementation, concentrating on the definition of the postprocess algorithm.
- Chapter 7 will show performances and results regarding the novel approach.
- Chapter 8 will summarize the most relevant results and give the conclusions, additionally offering some cues for future research work.

Part I

Motivations, Background, State-of-the-art

2 Background

This chapter offers an overview on some relevant concepts essential to understand the core of this thesis. First we will provide the definition of SAT, MaxSAT and a brief overview of the state-of-the-art algorithms used to efficiently solve them. We will also discuss some extensions of the SAT framework, SMT and OMT, citing some popular solvers built in the last decade to deal with these classes of problem. Lastly, an introduction to quantum computing and the adiabatic quantum annealing is provided.

2.1 SAT

Computer science problems can be classified according to their computational complexity in returning a solution. Problems that can be solved on a deterministic sequential machine in an amount of time that is polynomial in the size of the input are known as **P** problems; on the other hand, **NP** problems do not provide a sub-exponential algorithm capable of determining the existence of a solution. The class of NP problems relevant to our discussion is known as **Propositional satisfiability problem (SAT)**.
SAT definition is quite simple: given a formula made up of Boolean variables as input, our task is to determine if each variable of the input formula can be consistently replaced by the values True ($\top$) or False ($\bot$) so that that the formula evaluates to True. If the condition above can be satisfied, then the Boolean formula is considered **satisfiable**; on the opposite case, they are defined **unsatisfiable**. Let's discuss an example: given the following formula involving 2 Boolean variables (x_1 and x_2):

$$\varphi = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \tag{2.1}$$

In this case φ is satisfiable: if we set $x_1 = \top$ and $x_2 = \bot$, the entire formula evaluates to True.
In order for a Boolean formula to be satisfiable, it is sufficient to obtain a valid assignment: as a result, multiple solutions could be acceptable for a single formula to prove its satisfiability. The problem complexity grows exponentially with the increase of Boolean variables (when N variables are involved, 2^N possible assignments have to be tested in the worst case to determine its unsatisfiability), thus its membership to NP.
SAT solving found real life applications in HW/SW synthesis and verification problems, cryptography and security issues and many other tasks. Currently SAT solvers are capable of managing up to 10^6 variables and 10^7 disjunctions for some specific cases. On the other hand, more complex problems are actually out of reach despite a low number of variables to manage, in particular related to cryptanalysis and verification of arithmetic circuits. The search of efficiency for these tasks is the ultimate goal of this research.
Boolean formulas can be converted to equivalent representations with desired properties than can help SAT solvers in determining its satisfiability. The two most popular Boolean conversions are the **conjunctive normal form (CNF)** and the **Tseitin transformation**.
A Boolean formula φ is written in conjunctive normal form is if it is structured as the conjuction of simpler disjunctive sub-formulas (each disjunction is called clause), satisfying the following formulation:

$$\bigwedge_{i=1}^{L} \bigvee_{j_i=1}^{K_i} I_{j_i} \tag{2.2}$$

Each Boolean formula can be reduced to an equivalent CNF formulation applying some transformation rules known as **De Morgan's rules**:

$$\neg(\alpha \wedge \beta) = (\neg\alpha \vee \neg\beta) \tag{2.3}$$

$$\neg(\alpha \vee \beta) = (\neg\alpha \wedge \neg\beta) \tag{2.4}$$

The main issue generated by a CNF conversion is its exponential increase in size; on the contrary the second proposed approach, Tseitin transformation, is capable of simplifying the formula maintaining a linear growth. The key idea is the introduction of auxiliary variables to represent the output of subformulas and then constrain those variables using CNF. In details, given a formula φ we need to follow these steps:

- We introduce a new variable for each subformula ψ of φ.
- We consider each subformula $\psi = \psi_1 \circ \psi_2$, where $\circ$ can be any Boolean connective, and stipulate representative of ψ is equivalent to representative of $\psi_1 \circ \psi_2$.
- Each subformula is converted into its CNF-equivalent formulation.
- We can build the Tseitin formula as:

$$\psi_\varphi \wedge \bigwedge_{\psi_1 \circ \psi_2 \in \text{ subf}(\varphi)} \text{CNF}(\psi_{\psi_1 \circ \psi_2} = \psi_{\psi_1} \circ \psi_{\psi_2}) \tag{2.5}$$

We need to point out how the formula obtained after a Tseitin transformation is not equivalent to its original form (because of the newly introduced variables); despite this, the two Boolean formulas are equisatisfiable, meaning that the first formula is satisfiable whenever the second is and vice versa and thus being suitable to be used for our tasks.

2.1.1 How to solve SAT problems: DPLL and CDCL

The first algorithm acquiring popularity to deal with SAT instances is known as **Davis–Putnam Logemann–Loveland (DPLL) algorithm** [10]. It is a complete, backtracking-based search algorithm for deciding the satisfiability of propositional logic formulae in conjunctive normal form. The solver chooses a literal and assigns a truth value between True and False, opening a new branch; this choice will simplify the formula and the value of a subset of other literals will be bounded. Once no more variable values can be deterministically assigned, we choose a second variable and repeat the procedure described above until we retrieve a satisfying assignment or a conflict is found. In the second case, we perform a chronological backtrack: we jump back to the most-recent open branching point and start the search from that point. If no path reach satisfiability, the formula is considered unsatisfiable.
The algorithm described above, albeit working, it quite inefficient because of the amount of resources wasted to perform each backtrack jump. To solve this issue, a variant of the algorithm has been proposed and it is the current state-of-the-art approach implemented in the SAT solvers: **conflict-driven clause learning (CDCL)** [8]. The algorithm behaviour is similar to DPLL until we reach a conflict: in that case information is collected to efficiently backtrack and avoid lots of redundant search. In particular CDCL relies on:

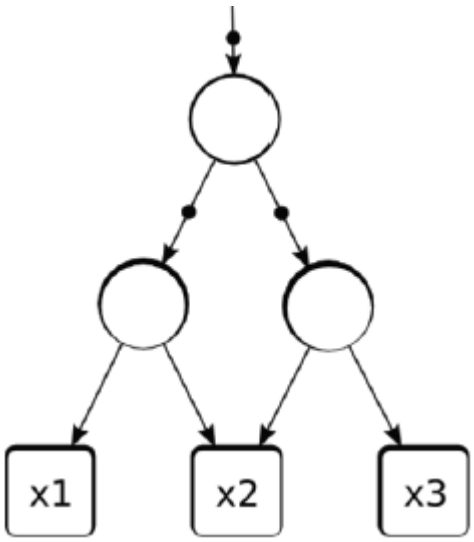


Figure 2.1: An And-Inverter Graph representing the Boolean formula $\phi = (x_1 \wedge x_2) \vee (x_2 \wedge x_3)$

- **Conflict analysis**: once a branch fails, we determine the sub-assignment η causing the conflict, determining the conflict clause.

- **Learning**: the conflict clause is stored by the solver into the conflict set, ready to be applied when required.

- **Backjumping**: we can then backtrack to the highest branching point so that the stack contains all-but-one literals in η, and then unit-propagate the unassigned literal on C.

While this approach drastically prunes the search space, we need to take into account the growth of space complexity required to store clauses. Heuristics have been proposed to determine when a learned clause is no more useful for the task in question and can be dropped, limiting the amount of space needed.

2.1.2 Representing SAT using And-Inverter Graph

Each Boolean formula can be represented using different approaches. The structural implementation relevant to the scope of this dissertation is known as **And-Inverter Graph (AIG)**. An AIG is a directed, acyclic graph characterized by:

- Terminal nodes, representing Boolean variables.

- Two-input nodes, representing logical conjunctions

- Marked edges, representing negation of a logical conjunction.

To convert a Boolean problem into an equivalent AIG representation, it is convenient to start from its CNF formulation. From CNF we can trivially construct an AIG by rewriting each OR clause as an AND function using De Morgan's Law, and then rewriting each AND function with more than 2 inputs as a sequence of 2-input AND functions. An example of AIG representation of a simple Boolean function is shown in Figure 2.1.

2.2 MaxSAT

A subclass of SAT problems is called **weighted maximum satisfiability**, or **MaxSAT**. A MaxSAT formula can be seen as optimization extension of SAT: the formula is defined as a conjunction of clauses and, for each clause, a weight (a positive real number) is assigned. Given a Boolean assignment of its variables, a score is calculated considering the weights of all satisfied clauses. This addition changes the perspective of the problem: every assignment can now be considered acceptable, since some clauses can be falsified. As a consequence MaxSat defines a new task: find a Boolean assignment so that the associated score is maximum. Similarly to SAT, multiple assignments can be the solution of a MaxSAT instance, resulting in multiple valid assignments.
To better understand it, we cover it using an example. We will consider the following Boolean formula:

$$\varphi = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \tag{2.6}$$

Each clause has also an additional weight: first clause from left has a score of 1, each subsequent clause's weight is higher by 1 with respect to the precedent. Clearly no assignment could satisfy all clauses at the same time, but this has no interest since we are solving a MaxSAT problem. Quickly considering all possible assignments, we can state that the one guaranteeing the maximum total score is $x_1 = \bot$ and $x_2 = \bot$, satisfying respectively the clauses with weights 2, 3 and 4. No other Boolean assignment could achieve the same result, so this is the only solution to our MaxSAT problem.

2.3 Satisfiability Modulo Theories

Satisfiability Modulo Theories represents an extension to the SAT problem. The input and the final goal remain the same as SAT: this means that we have to deal with first-order logic formulas and we search for a valid assignment of variables satisfying such formulas. The main difference relies on the fact that binary boolean variables can be replaced by predicates and functions belonging to non-binary theories. Some of the most relevant theories in the Formal Verification fields are:

- Linear Arithmetic over Integers, Real or both (LIRA)
- Bit vector arithmetic (BV)
- Floating-point arithmetic(FP)
- Uninterpreted functions (UF)

Given the non-binary nature of these theories, novel approaches have been studied and tested to efficiently evaluate the satisfiability of a given formula. At the current state, the most widely used algorithm is known as **Lazy SMT solving**[11], whose core idea will be briefly explained to the reader for a better comprehension. Given an SMT formula and a subset of theories, we produce a Boolean abstraction formula in which each non-binary predicate is replaced by a binary variable. This new formula is then fed to a CDCL SAT solver and used to search a satisfiable assignment. When a satisfiable assignment is obtained, the corresponding set of T-literals that make up the original problem are fed to specialized theory solvers. If each clause if T-consistent, then we proved the satisfibility of the original formula; otherwise, the conflict clauses causing unsatisfiability are extracted and learned by the CDCL algorithm , permitting the evaluation of new assignments. The algorithm goes on until satisfiability is proved or no more assignment can be returned, proving its unsatisfiability.

```
(set-info :smt-lib-version 2.6)
(set-option :print-success false)
(set-option :produce-models true)

(declare-const x Int)
(declare-const y Int)
(declare-fun f (Int) Int)

(assert (= (f x) (f y)))
(assert (not (= x y)))

(check-sat)
(get-value (x y))
```

Listing 2.1: An example of SMT-LIB encoding.

In order to be evaluated, the instance of a SMT problem has to be instantiated using a standard format. Regarding SMT encoding, SMT-LIB is the international initiative on top of which most SMT solvers are built. SMT-LIB does not only promote common input and output languages for SMT solvers; it also provides standard rigorous descriptions of background theories used in SMT systems and establishes benchmarks that can be used to test the solvers to highlight weaknesses and strengths.
The syntax of SMT-LIB-based tools is quite simple. First, we define properties of the problem and solver options (for instance determine if we want to extract unsat cores). We can also set a theory, so that efficient algorithms and heuristics are applied during execution to improve performances. After this introduction, we declare constants, variables and functions. Then assertions are expressed: they bind the values of variables, restricting the range of satisfying solutions. A simple example is provided in listing 2.1.
Among the solvers accepting SMT-LIB as input format, University of Trento and the Bruno Kessler Foundation have developed a SMT solver, called MathSAT. The code, written in C++, support all the theories mentioned above, plus other logics as Arrays arithmetic. It also supports interesting functionalities such as generation of models and proofs for satisfiable problem, extraction of unsatisfiable cores for the unsatisfiable ones and incrementality.

2.4 Optimization Modulo Theories

Optimization Modulo Theories represents an extension to the SMT framework. In addition to the already described first-order formulas involving non-binary logics, we focus on defining one or more non-binary objective function. As a result, retrieving a satisfying assignment is not sufficient anymore: the goal is now obtaining a valid model while minimizing/maximizing the objective functions, according to our choice.
The language used to encode OMT problems is an extension of the SMT-LIB standards. In addition to the syntax already discussed for SMT-LIB, some commands are available to define the cost functions we desire to optimize and the direction of optimization. The structure of this command is:

solve [maximize/minimize] <cost function>

If permitted by the solver, we can also write multi-objective optimization problems. multi-objective optimization problems require the presence of some instructions to determine the

philosophy adopted to determine the goodness of a solution. Example of valid approaches are:

- **Pareto Optimality**: given two different solutions and a set of n cost functions $x_1...x_n$, we state that the first solution is better to the second one according to this criterion if there exists a cost function x_i (where $1 \leq i \leq n$) so that x_i calculated on the values of the first solution dominates the value obtained using the second solution; in addition to that, for each cost function x_i the first solution obtain better or equal values than the second one. When using this criterion, multiple solutions can satisfy these conditions: the set of these valid assignment is called **Pareto front**.
- **Lexicographic Optimality**: first we define a total order among the various cost functions, determining a hierarchy. Given two different solutions, we will first compare the value of the first cost function on this hierarchy for both of them; if one of these solutions dominates the other it is chosen as optimal, otherwise we will compare the value associated to the second cost function and so on until reaching the last one.

The jointly work between FBK and University of Trento gave birth to an Optimization Modulo Theories solver, *OptiMathSAT* [12]*.

2.5 Constraint Programming and MiniZinc

Constraint Programming is an alternative paradigm for solving combinatorial problems. A Constraint Satisfaction Problem (CSP) is defined by a set of variables, each one with its domain of values, and a number of constraints, relations among a subset of the variables. A simple example of constraint is called ALLDIFFERENT: given some variables as input to this relation, it ensure no involved variable will assume the same value. The usual output of a CSP is a a valid assignment for each declared variables, if possible. The CSP has been extended to deal with the search of optimal solution with respect to some objective function, similarly to what happened to SMT: these problem are referred as Constraint Optimization Problems (COP). Due to the large amount of constraint available and its representational power, Constraint Programming popularity is growing in the last years and it is actually used in some formal verification related tasks.
MiniZinc is the most popular high-level declarative language used to encode both CSP and COP instances. It is developed at Monash University in collaboration with Data61 Decision Sciences and the University of Melbourne. Each MiniZinc model first declares each variable needed to define the problem; optionally a value can be assigned, obtaining a parameter whose behaviour is similar to constants. Variables which are not initialized are known as decision variable and their value is determined by the solver, trying to satisfying the conditions that will follow. The next component of the model are the constraints, Boolean expression defining bounds to the value each decision variable can assume. Lastly we have a line expressing if we are working with CSP (so we only focus on knowing if there exists an assignment satisfying each variable domain and making true all constrains) or COP (in this case the user has to define the objective function to optimize and the direction of optimization). A simple example is provided in listing 1.2.

2.5.1 Bridging CP and OMT

OMT presents some specific affinities that make it a valid candidate to deal with CSP: the availability of decision procedures for infinite-precision arithmetic, the efficient combination of Boolean and arithmetical reasoning and the ability to produce conflict explanations are shared between the two paradigms and it is not a case both are used to deal with formal verification problems. Achieving this task benefits the progression of the current state-of-the-art: the comparison between OMT solvers and CP tools on problems that do not belong to

```
int: n;
enum Man = anon_enum(n);
enum Woman = anon_enum(n);
array[Woman, Man] of int: rankWomen;
array[Man, Woman] of int: rankMen;
array[Man] of var Woman: wife;
array[Woman] of var Man: husband;

constraint forall (m in Man) (husband[wife[m]]=m);
constraint forall (w in Woman) (wife[husband[w]]=w);
constraint forall (m in Man, o in Woman) (
     rankMen[m,o] < rankMen[m,wife[m]] ->
          rankWomen[o,husband[o]] < rankWomen[o,m] );
constraint forall (w in Woman, o in Man) (
     rankWomen[w,o] < rankWomen[w,husband[w]] ->
          rankMen[o,wife[o]] < rankMen[o,w] );
solve satisfy;
```

Listing 2.2: An example of MiniZinc encoding regarding the popular Stable Marriage Problem.

their original application domain extends their application in novel fields and speed up the process of finding critical points for future research.
Despite that, transforming a CP instance into a OMT problem is not trivial; in particular there are multiple valid formulation of a CP problem into an OMT instance. Using one option instead of the others requires particular attention, since this choice could drastically affect performances (as shown in []). This situation represents the first issue hardening the bridging task. In addition to this difficulty, there are some MiniZinc features that requires a careful mapping to the SMT-LIB standard. In particular:

- FlatZinc support three basic scalar types (int, float and bool) and two compound types (set and array). Compound types do not have a direct correspondence in the SMT-LIB standard; moreover integer and float can be represented using finite precision (using respectively bit-vector and floating point arithmetic) or applying the linear arithmetic theory.

- MiniZinc introduced local and global constraints to express complex relations among variables. OMT solver currently do not introduce ad hoc decision procedures to manage them efficiently.

- MiniZinc is capable of dealing with non-linear and transcendental functions, such as the logarithm or the cosine function. On the other hand, not every solver built on SMT-LIB support them.

2.6 Quantum Annealer

In order to achieve efficiency in solving the hardest tasks, one of the proposed techniques involves the exploit of quantum technologies. In the case at issue **Quantum Annealers (QA)** have been proved to be effective for this task.
Quantum annealers are specialized chips that exploit particular quantum effects (such as superposition and tunneling) to sample or minimize energy configurations. Each configuration is composed by binary variables z_i which can assume a value -1 or 1, representing their state. The

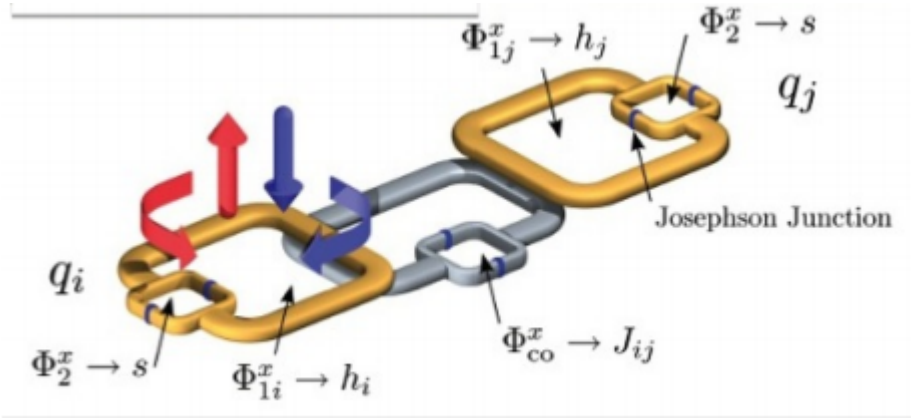


Figure 2.2: A physical representation of qubits in an Quantum Annealer.

energy configuration can be represented using the Ising Hamiltonian function, whose structure is the following:

$$H(\underline{\mathbf{z}}) = \sum_{i \in V} h_i z_i + \sum_{\langle i,j \rangle \in E} J_{ij} z_i z_j \tag{2.7}$$

Formula 2.7 shows the parameters that influence the system's behaviour:

- h_i are called **biases** and are defined as a real number in range $[-2, 2]$
- J_{ij} and lastly J_{ij} are called **couplings** to the connected pair of qubits at index i and j and is a real number in range $[-1, 1]$.

The search of the ground state of an Ising model is known as **quadratic unconstrained binary optimization (QUBO)** problem.

2.6.1 The SQUID transistor

The minimal unit composing a quantum annealear is called **qubit**: a qubit can assume a symbol value of 0, 1 or a superimposition of 0 and 1. The physical devices used to build qubits are the **Superconducting QUantum Interference Devices (SQUID)**, where the word interference refers to the electrons patterns that give birth to quantum effects. The structure of this qubit transistor is defined by two superconductor coupled by a weak link, usually a thin insulating barrier. Transistors satisfying these conditions are subject to the **Josephson effect**. The superconducting qubit structure instead encodes 2 states as tiny magnetic fields, which either point up or down. We call these states +1 and -1, and they correspond to the two states that the qubit can 'choose' between. Using the quantum mechanics that is accessible with these structures, we can control this object so that we can put the qubit into a superposition of these two states as described earlier. So by adjusting a control knob on the quantum computer, you can put all the qubits into a superposition state where it hasn't yet decided which of those +1, -1 states to be.
Each ring can be superimposed to other rings, defining more complex architectures and admitting an exchange of information among different qubits (thus the definition of couplings in the Ising formula). Figure 2.2 shows the physical structure of a qubit.

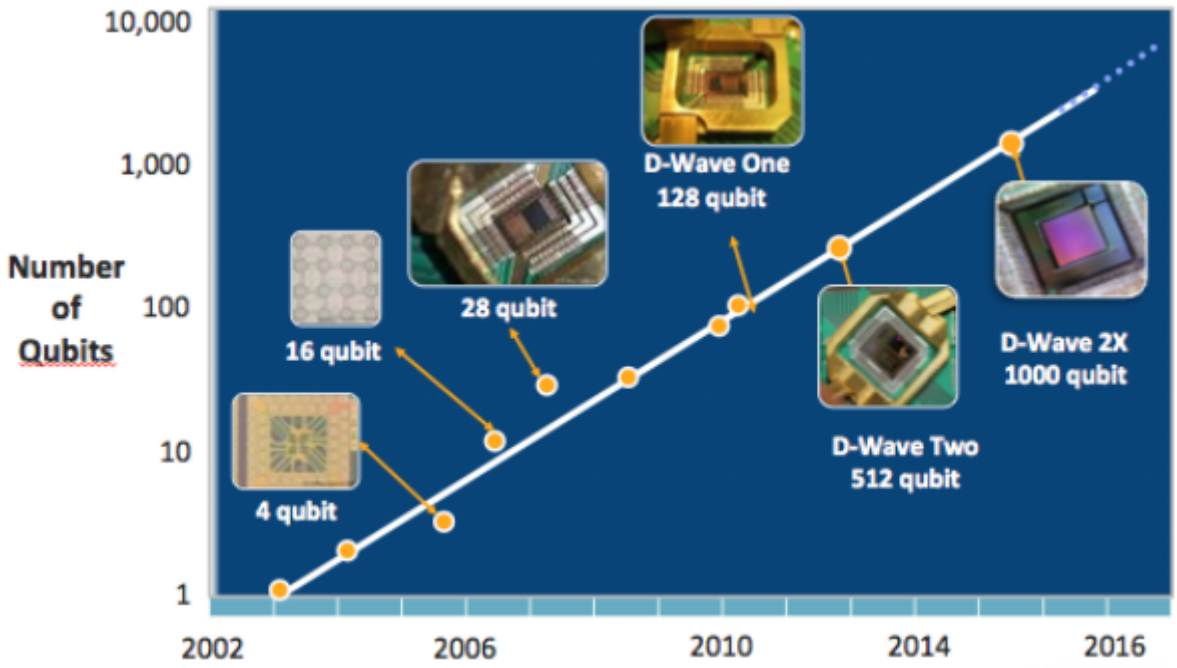


Figure 2.3: D-Wave annealers growth over the years. Courtesy of D-Wave Systems Inc.

2.6.2 D-Wave Quantum Annealers

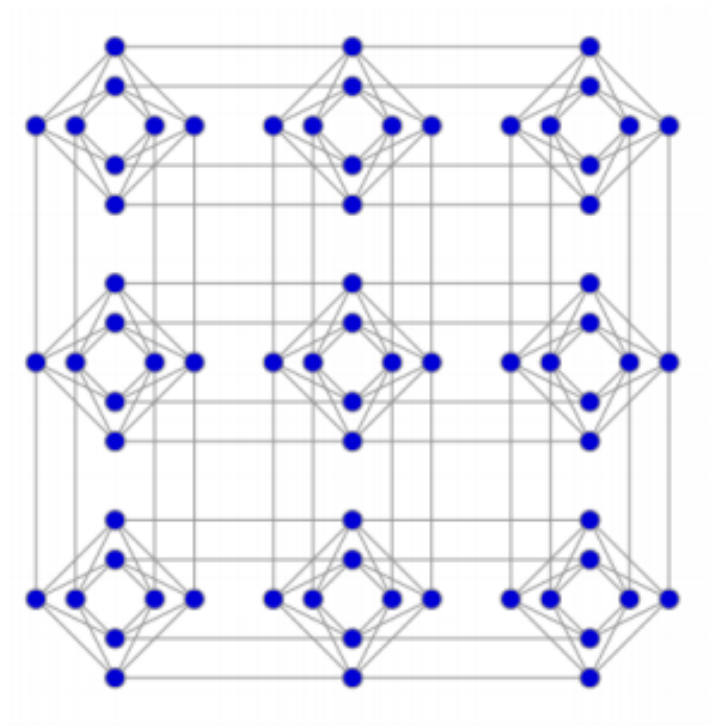
The entire energy configuration can be represented as a connected graph, where vertexes represent our qubits and edges represent connections between qubits. This configuration highlights how qubits' ground state is not considered independently, but their interaction with neighbour qubits alters the final energy state of our configuration. The choice of a specific quantum annealer influences the formulation of an Ising model: each annealer has a specific architecture and specific properties, constraining the formulation of a problem. In this thesis we will concentrate on annealers developed by the Canadian company D-Wave [3]. In the beginning, the Canadian company aimed in increasing the number of available qubits, obtaining Moore-like growth as shown in figure 2.3. In a second phase, the company put its effort in defining novel structures with better connections and fewer wasted qubits. Currently D-Wave has already developed and produced a first quantum system, known as **Chimera**. The basic features of this architecture are:

- The basic unit is the tile, which contains 8 qubits.
- Qubits of a cell unit are grouped into two groups (the horizontal and vertical qubits): qubits belonging to the same group are connected to qubits of the opposite group thanks to couplings.
- Unit cells are tiled vertically and horizontally with adjacent qubits connected, creating a 16×16 lattice of 2048 qubits.
- From previous constraints, we can see how each qubits can be connected to a maximum of 6 other variables, resulting in the sparsity of the adjacency matrix. Qubits that are not connected to each other are managed setting the related coupling value to 0.

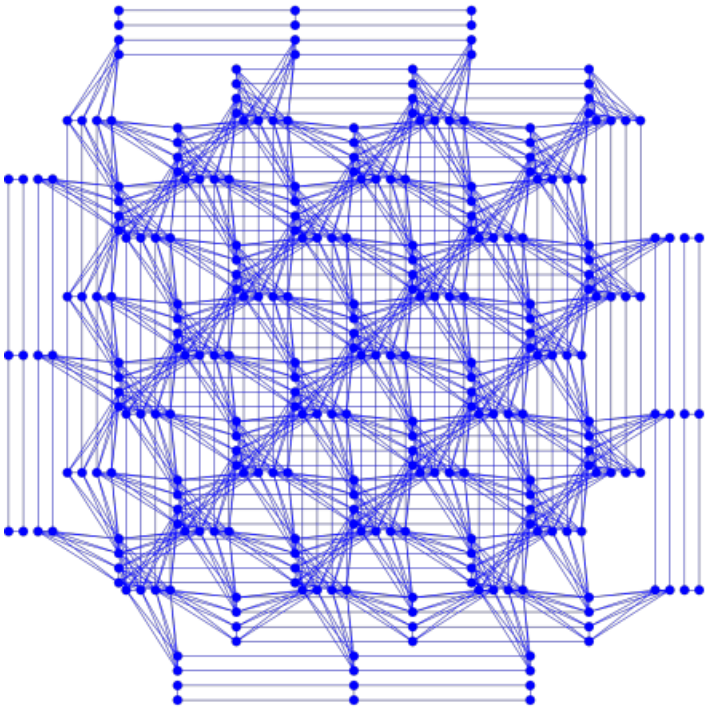
In addition to this structure, D-Wave is already studying new architectures overcoming the sparsity of the graph and the absence of cliques; in 2019 the company proposed a new architecture called **Pegasus**. The properties of this system are:

- The number of qubits is $24 * N * (N - 1)$, where N is an integer number.
- The architecture is less modular than Chimera and, in particular, it is not structured in 8-qubit tiles.
- Pegasus graph is less sparse than Chimera, presenting an higher number of interleavings among qubits (exactly 15 couplings for a single qubit).
- The increasing number of couplings have given the opportunity to obtain huge progress with respect to the old model. In particular, Pegasus provides 3 and 4-cliques and qubit duplication.

Figure 2.4 shows the graphs representing both Chimera and Pegasus topologies, rapidly displaying the main differences between the two models.



(a)



(b)

Figure 2.4: A representation of the two architecture proposed by D-Wave, respectively: (a) $3 * 3$ tiles sub-graph of Chimera (b) the graph of Pegasus6.

3 Related work

In this chapter we will deepen on the SAT-to-Ising reduction problem, which has been proved to be interesting when trying to exploit properties of quantum computing to reduce the computation time to solve SAT problems. We will discuss a preliminary algorithm that have been studied at University of Trento, distinguishing the procedure adopted to encode simple formulas from complex Boolean instances.

3.1 SAT-to-Ising

As already stated in the previous paragraphs, SAT main issue is the exponential growth of complexity, stopping computer scientists in investigating hard tasks involving a small number of variables. Quantum computing, on the other hand, exploit specific phenomena such as tunneling to search the minimum of an Ising problem, optimizing computational time. If we would be able to encode SAT/MaxSAT problem into an Ising Hamiltonian problem, we could exploit quantum calculus properties to verify satisfiability of out original problems, drastically reducing computational time. The task described is known as **SAT-to-Ising**.
Computer scientists use Quantum Annealers as black-box algorithms defining an Hamiltonian problem, whose formulation is the following:

$$H(\underline{\mathbf{z}}) = \theta + \sum_{i \in V} h_i z_i + \sum_{\langle i,j \rangle \in E} J_{ij} z_i z_j \tag{3.1}$$

As we can see qubits, biases and couplings are present in our formulation; in addition to them we consider the parameter θ_0, which is called offset and falls into the range $(+\infty, -\infty)$. Given a SAT problem, we are interested in finding a variable placement $\mathbf{x} \rightarrow \mathbf{z}$ on the quantum annealer qubits and the values of offset, biases and couplings so that:

$$P_F(\underline{\mathbf{x}}|\underline{\theta}) = \left\{ \begin{array}{ll} = 0, & if\ \underline{\mathbf{x}} \models F \\ \geq g_{min}, & if\ \underline{\mathbf{x}} \models T \end{array} \right\} \tag{3.2}$$

The gap g_{min} is essential to manage sensitivity of the quantum annealer: the optimization will be affected by noise generated by some intrinsic aspects of the annealers (more details will be provided in paragraph 3.2), so it will be rarely the case that not satisfying assignment will get 0 as final score. The choice of a valid gap (in particular trying to find the highest gap possible satisfying the problem) will help us in discriminating acceptable assignments and not acceptable ones. For instance, a formula we can easily encode into an Ising Hamiltonian function is $\varphi = x_1 \iff x_2$. A simple penalty function would be $P_F(\underline{\mathbf{x}}|\underline{\theta}) = 1 - x_1 x_2$, whose gap for satisfying assignment is 2.
The current formulation of our problem presents some issues:

- D-wave architectures present some structural limitations, reducing the number of problem we can represent.

- The problem is actually overconstrained: you need to search your solution checking $O(2^{|\underline{\mathbf{x}}|})$ models/countermodels. You have also to deal with the degrees of freedom of its formulation caused by biases and couplings.

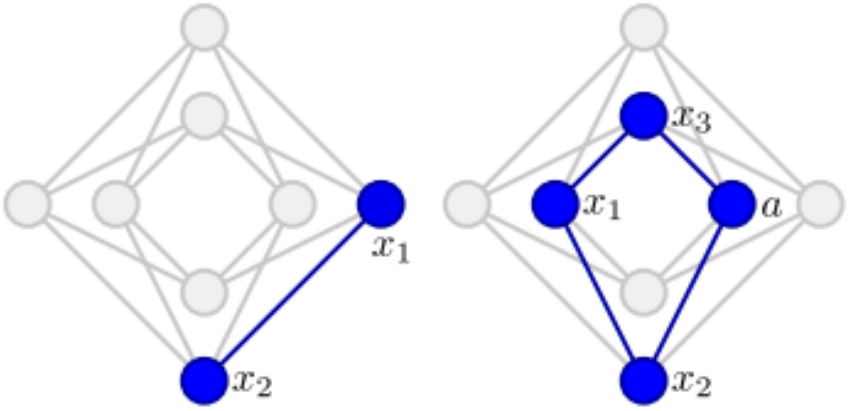


Figure 3.1: On the left, a valid positioning of qubits into a Chimera tile for the formula $\varphi = x_1 \iff x_2$. On the right, a valid positioning for the formula $\varphi' = x_3 \iff (x_1 \vee x_2)$. Both architectures are inspired by the penalty functions discussed in this paragraph.

To overcome this difficulties, we can add a non-fixed numbers of ancillary variables to provide the missing links. The problem will slightly change into the task:

$$min_{[\underline{\mathbf{a}} \in \{-1,1\}^k]} P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}}|\underline{\theta}) = \left\{ \begin{array}{ll} = 0, & if\ \underline{\mathbf{x}} \models F \\ \geq g_{min}, & if\ \underline{\mathbf{x}} \models T \end{array} \right\} \tag{3.3}$$

The more ancillary variables we add to our formulation, the more complex the resulting task will be, so we will always try to limit their use only when necessary. Ancillary variable are essential for basic encoding formulas: an example could be $\varphi = x_3 \iff (x_1 \vee x_2)$. If the Quantum Annealer admitted cliques the encoding would be trivial; since this is not the case, an ancilla is added to obtain a valid penalty function:

$$P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}}|\underline{\theta}) = \frac{5}{2} - \frac{1}{2}x_1 - \frac{1}{2}x_2 + x_3 + \frac{1}{2}x_1x_2 - x_1x_3 - x_2a - x_3a \tag{3.4}$$

Figure 3.1 shows the positioning of the obtained Ising Hamiltonian functions into a Chimera tile.

3.1.1 Determining the penalty function

The task of retrieving the parameters characterizing the penalty function of a Boolean formula can be espressed by a SMT problem:

$$\forall \underline{\mathbf{x}} \left[\begin{array}{l} (F(\underline{\mathbf{x}}) \rightarrow \forall \underline{\mathbf{a}}.(P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}}|\underline{\theta}) \geq 0)) \wedge \\ (F(\underline{\mathbf{x}}) \rightarrow \exists \underline{\mathbf{a}}.(P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}}|\underline{\theta}) = 0)) \wedge \\ (\neg F(\underline{\mathbf{x}}) \rightarrow \forall \underline{\mathbf{a}}.(P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}}|\underline{\theta}) \geq g_m in)) \wedge \\ (\neg F(\underline{\mathbf{x}}) \rightarrow \exists \underline{\mathbf{a}}.(P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}}|\underline{\theta}) = g_m in)) \wedge \end{array} \right] \tag{3.5}$$

The last row of equation (3.5) can be omitted if we are not searching an exact penalty function. Since this omission drastically reduce the computational time to search a valid solution, we will usually consider it omitted.
Classic satisfiability theories do not deal with quantifiers, so it is essential to successfully remove them, obtaining and equivalent formula. The most popular approach adopted is **Shannon expansion**. Given an universal quantifier with respect to a set of variables, we can build an equivalent formula as the union of set of clauses whose structure is identical to the original structure, additionally setting the variables inside the considered set to one of their possible

value. On the other hand, if we are presented an existential quantifier, we can recycle the previous algorithm but we will connect all the clauses using the AND operator instead of OR. The procedure is exponential: the more variables we consider, the more combinations of true/false value we can obtain. Once the expansion takes place, the original problem is reduced to the following SMT problem on linear real algebric theory:

$$\begin{aligned}\Phi(\theta) = \; & \textstyle\bigwedge_{Z_i \in \underline{\mathbf{x}}, \underline{\mathbf{a}}} (-2 \leq \theta_i) \wedge (\theta_i \leq 2) \\ & \wedge \textstyle\bigwedge_{Z_i, Z_j \in \underline{\mathbf{x}}, \underline{\mathbf{a}}, i<j} (-1 \leq \theta_{ij}) \wedge (\theta_{ij} \leq 1) \\ & \wedge \textstyle\bigwedge_{\{\underline{\mathbf{x}} \in \{-1,1\}^n | F(\underline{\mathbf{x}}=\top\}} \bigwedge_{\underline{\mathbf{a}} \in \{-1,1\}^h} P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}} | \underline{\theta}) \geq 0 \\ & \wedge \textstyle\bigwedge_{\{\underline{\mathbf{x}} \in \{-1,1\}^n | F(\underline{\mathbf{x}}=\top\}} \bigvee_{\underline{\mathbf{a}} \in \{-1,1\}^h} P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}} | \underline{\theta}) = 0 \\ & \wedge \textstyle\bigwedge_{\{\underline{\mathbf{x}} \in \{-1,1\}^n | F(\underline{\mathbf{x}}=\bot\}} \bigwedge_{\underline{\mathbf{a}} \in \{-1,1\}^h} P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}} | \underline{\theta}) \geq g_{min} \\ & \wedge \textstyle\bigwedge_{\{\underline{\mathbf{x}} \in \{-1,1\}^n | F(\underline{\mathbf{x}}=\top\}} \bigvee_{\underline{\mathbf{a}} \in \{-1,1\}^h} P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}} | \underline{\theta}) = g_{min}\end{aligned} \tag{3.6}$$

While the SMT formulation will help us in determining if there exists a valid assignment of parameters to apply the SAT-to-Ising conversion, we recall from paragraph 3.1 that determining the highest value that g_{min} can assume is a fundamental task; as a consequence, we can use equation 3.6 to define an OMT problem, where the cost function is simply g_min and the direction of optimization is the search of the maximal value.

3.2 Issues in Encoding for Quantum Annealers

Converting a Boolean circuit into an Ising formulation would apparently seem an easy task to achieve: actually the nature and the architecture of the annealers drastically reduce the number of SAT problems we can embed. The main issues causing the inefficiency are:

- **The number of qubits is not unlimited**: even though the recent architectures provides more than 2000 qubits, they are usually not enough to encode the majority of circuits. This is due by the nature of the two encoding: the number of qubits representing the input width and the circuit size are two different metrics for complexity, where the former is definitively bigger than the latter.

- **The number of couplings is not unlimited**: as already mentioned in chapter 2, each architecture can be interconnected to a limited number of other qubits. Consequently, the encoding process has to take into account this upper limit and, if more connections are required, map a Boolean variable into multiple qubits.

- **Noise impacts the system performance**: the ideal condition of a quantum annealer would require a temperature of 0 K and to be heavily shielded by electro-magnetic rays, preserving the properties of the superconductor rings. In reality these conditions are never met and D-Wave system are subject to performance degradation.

Because of the problems described above, advanced algorithms have been studied and tested in order to reduce the amount of qubits required during the reduction of the SAT circuit.

3.3 Encoding complex Boolean formulas

Determine an efficient encoding is decisive, given the limited of number of qubits of Chimera. Penalty functions have some properties we can exploit to iteratively build complex formula from easier ones. The first property is **NPN-Equivalence**: given a boolean formula $F(\underline{\mathbf{x}})$ with its associated penalty function $P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}} | \underline{\theta})$ and a new Boolean function $F^*(\underline{\mathbf{x}})$ identical to the previous one but a single variable at a generic index i (so that it appears with inverse cardinality in the new formulation), we can recycle $P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}} | \underline{\theta})$ to obtain the penalty function

for the new problem. In particular, $P_{F^*}(\underline{\mathbf{x}}, \underline{\mathbf{a}}|\underline{\theta}) = P_F(\underline{\mathbf{x}}, \underline{\mathbf{a}}|\underline{\theta^*})$ where θ^* is a new vector of biases and couplings so that for each $z, z' \in \underline{\mathbf{x}}, \underline{\mathbf{a}}$:

$$\theta_z^* = \left\{ \begin{array}{ll} -\theta_z, & if\ z = x_i \\ \theta_z, & otherwise \end{array} \right\} \tag{3.7}$$

$$\theta_{zz'} = \left\{ \begin{array}{ll} -\theta_{zz'}, & if\ z = x_i \vee z' = x_i \\ \theta_{zz'}, & otherwise \end{array} \right\} \tag{3.8}$$

In simple words, once we extract the penalty function for a Boolean Formula, its variants can be easily computed switching signs of biases and couplings. For instance, if we had the Boolean formula $\varphi = x_1 \iff -x_2$, we could easily invert signs of couplings and biases associated with x_2, trivially obtaining $P_F(\underline{\mathbf{x}}|\underline{\theta}) = 1 + x_1 x_2$.
The second property fundamental to simplify the task of retrieving penalty functions is the **AND-decomposition**. Given a Boolean function that can be rewritten as a combination of simpler functions so that $F(x) = \wedge_k F_k(\underline{x^k})$, where each $F_k(x)$ is associated to a penalty function $P_{F_k}(\underline{x^k}, \underline{a^k}|\underline{\theta^k})$ with minimum gap g^k_{min}, $\underline{x} = \cup_k \underline{x^k}$ and $\underline{a} = \cup_k \underline{a^k}$, then we can define a penalty function for the original formula in the following way:

$$\begin{aligned} P_F(\underline{x}, \underline{a}|\underline{\theta}) &= \sum_k P_{F_k}(\underline{x^k}, \underline{a^k}|\underline{\theta^k}) \ \text{ with } g_{min} = \min_k(g^k_{min}) \\ \theta_i &= \sum_k \theta_i^k \\ \theta_{ij} &= \sum_k \theta_{ij}^k \end{aligned} \tag{3.9}$$

Equation 3.9 works in the case each bias and coupling value satisfies its range. The subformulas making up the decomposition could share some common variables and, in some cases, this could lead to the attainment of out-of-range parameters. This issue can be easily fixed: we can scale up or down the impact of each penalty function adding a weight w_k greater than 0. This addition slightly modifies equation 3.9, determining the general formulation:

$$\begin{aligned} P_F(\underline{x}, \underline{a}|\underline{\theta}) &= \sum_k P_{F_k}(\underline{x^k}, \underline{a^k}|\underline{\theta^k}) * w_k \ \text{ with } g_{min} = \min_k(g^k_{min}) * w_k \\ \theta_i &= \sum_k \theta_i^k * w_k \\ \theta_{ij} &= \sum_k \theta_{ij}^k * w_k \end{aligned} \tag{3.10}$$

Adding these weights weakens the minimum gap, so it is not used in practice. An alternative to equation 3.10 relies on the renaming of the shared variable, so that each $\underline{x^k}$ is disjoint with respect to the others. To achieve this goal, when two conjuncts F_k, F'_k share a Boolean variable x_i we rename the second occurrence with a fresh variable x'_i and conjoin the two variable using the simple formula $(x_i \iff x'_i)$. Now we can re-define the original formula and its associated penalty function as:

$$\begin{aligned} F^*(\underline{x}^*) &= \bigwedge_k F_k(\underline{x}^{k*}) \wedge \bigwedge_{\{x_i \text{ shared}\}} (x_i \iff x'_i) \\ P_{F^*}(\underline{x}^*, \underline{a}|\underline{\theta}) &= \sum_k P_{F_k}(\underline{x^k}, \underline{a^k}|\underline{\theta^k}) + \sum_{\{x_i \text{ shared}\}} (1 - x_i x'_i) \end{aligned} \tag{3.11}$$

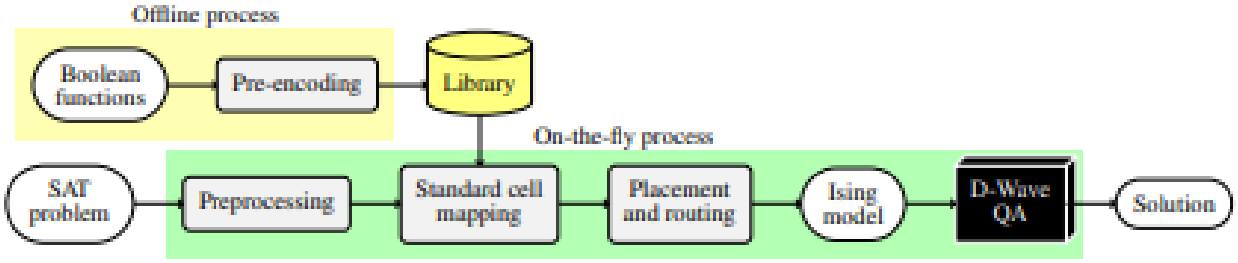


Figure 3.2: A graphical representation of the encoding process for larger Boolean functions.

$$g_{min} = min_k(g^k_{min}, 2)$$

Given the nature of the penalty functions for $(x_i \iff x'_i)$, whose gap is 2, we ensure no issue can emerge and no parameter range error can be observed. Combining the two properties above we can define the procedure to apply in order to retrieve penalty functions decomposing Boolean formulas into smaller chunks, easier to convert:

- First we Tseitin-style decompose $F(x)$ into an equi-satisfiable formula so that:

$$F * (\underline{x}, \underline{y}) = \bigwedge_{i=1}^{m-1} (y_i \iff F(\underline{x}^i, \underline{y}^i)) \wedge F_m(\underline{x}^m, \underline{y}^m) \tag{3.12}$$

- When two conjuncts F_1 and F_2 share one variable y_j, rename the second with a fresh new variable y'_i, conjoining $(y'_i \iff y_j)$.
- Compute the penalty function for each conjunct separately.
- Sum the penalty functions obtained in the previous step to obtain a single function.

From the algorithm described above, we can define a universal approach to deal with larger Boolean functions, avoiding the application of the SMT formulation which results too much time and resource demanding.
Before starting the encoding, we compute in advance a collection of valid encoding for simple SAT formulas so that they can be mapped using a low number of qubits (in particular when using the Chimera architecture we aim to encode the formula using a single tile). The original SAT formula we want to encode is pre-processed, in order to reduce its size or complexity in terms of its graphical representation. As an example, a good pre-processing approach is using the and-inverted-graph representation, which transform a formula in a combination of AND and NOT functions. Using the simplified structure and the previously computed library we then define a mapping between Boolean variables and the annealer's qubits. This phase represents the most complex and resource-demanding procedure of the algorithm. The fundamental steps of this procedure are:

- Use the library of pre-computed penalty functions to map each function part of the decomposed and simplified SAT formula into a valid Ising encoding. This phase is known as standard cell mapping.
- Now it is necessary to embed the entire formula onto the QA hardware. To achieve this task, we first assign a disjoint subgraph of the QA hardware graph to each penalty function chosen in the previous step.

- Lastly we need to ensure that qubits representing the same variable are chained so that they can assume the same value when given as input to the annealer: we can accomplish it using penalty functions in the form $1 - x_1 x_2$, where x_1 and x_2 are qubits representing the same variable. This step is a direct consequence to formula 3.11.

The algorithm is heavily discussed in [5] and more details are provided about the choice of and the heuristic adopted to obtain a more stable encoding; for the sake of brevity we will not discuss it further. Figure 3.2 summarizes the entire process.

Part II

Contributions

4 Tool analysis

In this chapter we will discuss the state-of-the-art tool developed by the jointly work of University of Trento and D-Wave. More specifically we will emphasise critical points of the current version, suggesting possible direction to improve the performance of the algorithm.

4.1 Studying the Ising encoding

To determine the behaviour of the placement algorithm, it was required to graphically represent the annealer architecture when applied to a SAT-to-Ising encoding task.
A new script has been developed to accomplish this task, called *graph_to_dot*. The first step was the building of a graph reflecting the architecture of the quantum annealer chosen at execution. Information about the topology of the network has been retrieved using **D-Wave NetworkX**, an extension of NetworkX providing tools and data for working with the D-Wave systems [2]. Data about the roles of each qubits has been collected from the *place_and_route* algorithm: for each AIG-related subformula the algorithm returns a subset of qubits to the graph and the values of weights extracted from the pre-computed libraries. Using this information, we choose a color for each penalty function and use it to determine the related qubits and the involved couplings in the graphical representation. The color of edges representing chains of a Boolean variable have been set to black. An example of Boolean formula placement resulting from the execution of *place_and_route* is shown in figure 4.1.

The script was integrated to the original code as a debug option, permitting other users to test their encoding and better understanding the results behind the implemented procedures.

4.1.1 The issue of co-tunnelling

The major issue visible from figure 4.1 is the presence of long chains of qubits necessary to connect subformulas sharing a Boolean variable. Higher size of chains are the principal reason behind the phenomenon known as **co-tunneling**.
Co-tunneling is a side-effect issue caused by the presence of long chains of qubits in the annealer's placement [7]. Among its negative effects, the major one is the worsening of the stability of the annealers in the search of the Hamiltonian final state.
This side effect can be easily overcome cleverly rearranging the retrieved encoding. Each qubit belonging to the chain can be potentially more useful: unused arcs adjacent to them and set to 0 could be used to determine novel penalty functions. The results would be more complex structures that could help us in obtaining better penalty function, less prone to be subject to co-tunelling and with higher gap_{min}.

4.1.2 The Pegasus genlib

A second relevant issue emerged from the analysis of the Ising encoding. The number of penalty functions used to map the Boolean formulas into inside the Pegasus architecture is very small with respect to the one provided for Chimera. This is due to the novelty of the Pegasus architecture during the development at that time: as a result, the focus of the previous work was concentrated on the extension of the Chimera genlib file. While the two libraries are quite similar, there are some structural differences that prevent the user to interchangeably use one library for both architectures. Listing 4.1 provides the definition of a Boolean gate from each library, showing the most obvious differences the two.

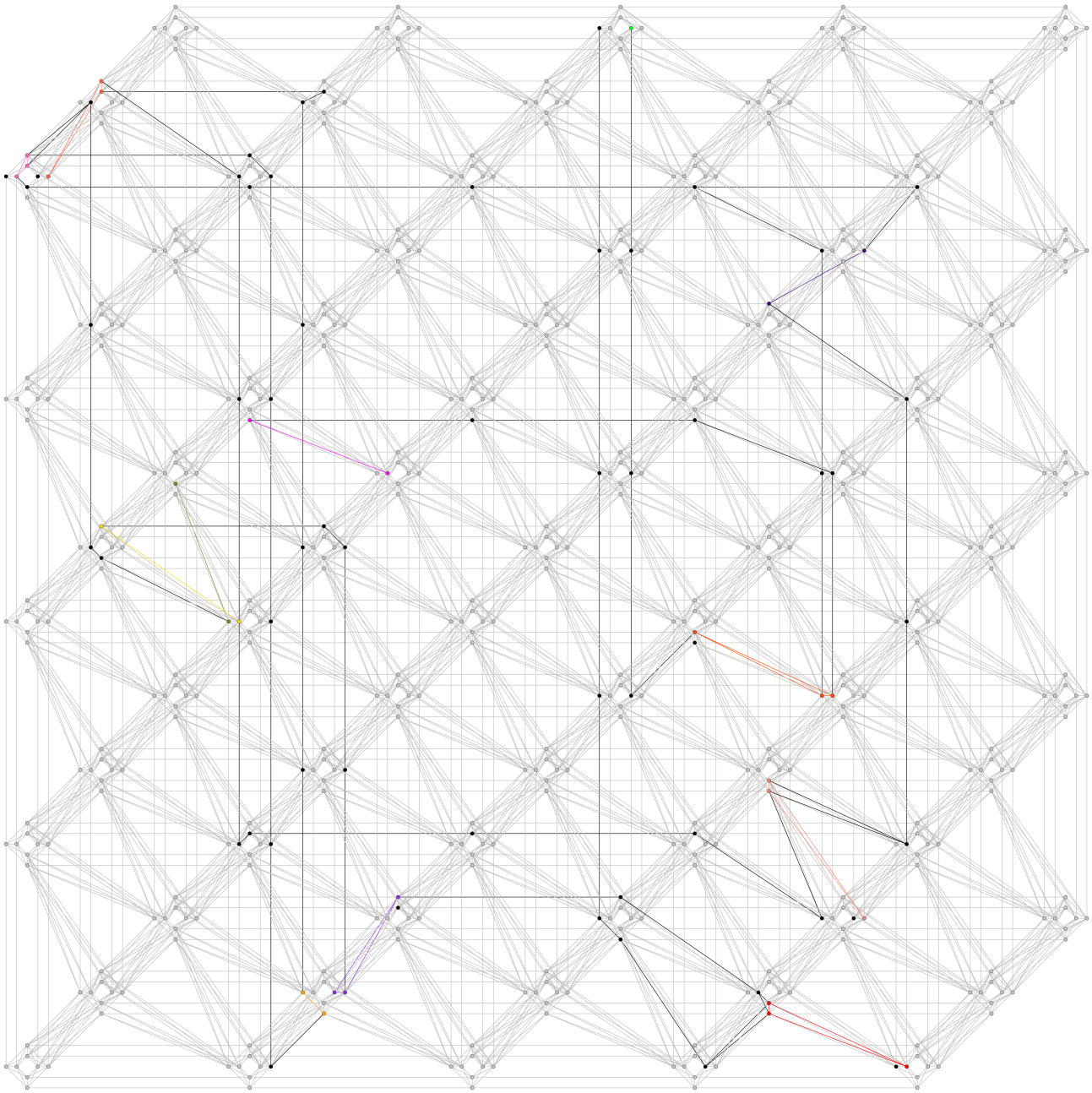


Figure 4.1: A graphical representation of the encoding of a benchmark problem, C17, into the Pegasus architecture.

```
# Chimera genlib
GATE gate4 4.000 A = (C) * (B);#{
"R": [[0,0,0],[0,1,0],[0,0,1],[1,1,1]],
"h": [[0.25],[-0.25],[-0.25],[0.25]],
"J": [[0,0,-0.5,-0.75],[0,0,0.25,-0.5],[0,0,0,0],[0,0,0,0]],
"en0": 1.5,"g": 1,"chimera": 1,"vP": [[[1],[3],[2]]],"dims": [1,1,2]}
 PIN B INV 0 0 0 0 0 0
 PIN C INV 0 0 0 0 0 0

# Pegasus genlib
GATE and 3.00 O = (I0 * I1);#{
"off": 1.5, "(bias pos1)": 1.0, "(bias pos2)": -0.5,
"(bias pos3)": -0.5, "(bias pos4)": 0.0, "(coupl pos1 pos2)": -1.0,
"(coupl pos1 pos3)": -1.0, "(coupl pos1 pos4)": 0.0,
"(coupl pos2 pos3)": 0.5, "(coupl pos2 pos4)": 0.0,
"(coupl pos3 pos4)": 0.0, "pos1": 3.0, "pos2": 2.0,
"pos3": 1.0, "pos4": 0.0, "ancilla_used": 0.0}
            PIN * INV 0 0 0 0 0 0
```

Listing 4.1: The definition of the penalty function $A = B \wedge C$ according to the Chimera and Pegasus libraries. While the conversion of most data would be trivial (in particular weights of biases and couplings) it would be difficult to map Chimera encodings (which are mapped using Chimera tiles as basic unit) into Pegasus (which does not provide the concept of tile).

Encoding the Ising problem using very small penalty functions is not convenient for our purpose: the probability of determining complex subgraphs using the wasted qubits from chains drastically decreases and so we obtain no increase in the definition of functions with higher gap_{min}. In order to solve this issue, an extension of the Pegasus library has been proposed and it will be further discussed in the following chapters.

4.2 The searchPF function

The previous section suggests the idea to recompute penalties function reducing the length of the chains. These nodes, used as ancillas and considering their unused couplings, could provide more complex encoding, reducing the co-tunneling effect and guaranteeing more energy stability. The tool already provide a class of algorithms to compute offset, biases and couplings of an Ising problem. Each function provided some specific constraints in order to satisfying properties such as getting the maximum g_{min} The function relevant to the scope of this thesis, called **searchPF**, was the most suitable to our purpose, given its simplicity and its ease to be extended.
SearchPf requires as input the following objects:

- A graph representing the topology of the involved qubits. Graphs are created using the *GraphViz* library, setting the nodes name as the integer in the range [1,N] with N the number of qubits.
- A function representing the Booolean formula we desire to map into the given graph.
- Two integers nx ans na, determining respectively the number of qubits representing the variables and the number of qubits working as ancillas. These parameters determine the

N. of qubits	Running time
7	1.49 s
8	1.93 s
9	5.01 s
10	33,4 s
11	$>$ 180 s

Table 4.1: A table showing the time required to retrieve the penalty function of $A = B \wedge C$ with an increasing number of variables, most of them used as ancillas.

role of each node of the graph: nodes whose label is in the range $[1, nx]$ represent Boolean variables, while the remaining nodes are used as ancillas.

In order to get the weights defining the penalty function, an Optimization Satisfiability problem is instantiated using the previously mentioned parameters. The problem is fed to an OMT solver (in this case OptiMathSAT was chosen), which returns in response an empty dictionary if no valid penalty functions has been found; otherwise, a dictionary containing the parameters value and the minimum gap is returned. Multiple constraints are set to map the problem into the SMT-LIB language:

- **Architecture constraints**: we ensure the resulting penalty function respects the topology of the given graph. As a consequence, some couplings need to be set to 0 a.
- **Range constraints**: As stated in section 1.5, values of biases and couplings need to be constrained so that they cannot violate their ranges. For each coupling and each bias an assert condition setting lower an upper bounds are added to the OMT encoding.
- **Expansion gap constraints**: in order to retrieve the parameters of the Ising function we have to solve the quantified OMT problem described in equation 3.6. The role of these constraints is the definition of multiple assertions to bind the values of biases and couplings so that they respect the Boolean formula used as reference.

The function has been heavily tested to learn its behaviour, its complexity and thus offering cues for improvements. First we studied the relation between the required time to obtain a solution to the OMT problem and the number involved variables, with the goal of determining an upper bound for the number of nodes so that the OMT problems can return their solution in a feasible amount of time, to be applied in real-life applications. Table 4.1 shows the results of this analysis; we can see how we are bound to use a very low number of qubits. This is not surprising: equation 3.6 is subject to an exponential growth with respect to the number of involved variables. Currently there are no alternative to obtain an equivalent form of equation more efficiently. More details will be provided in the conclusive chapter, suggesting future approaches to overcome this limit.
We additionally implemented a small section of code to write the output of the procedure on a TXT file, giving us the opportunity to easily understand how *searchPf* generates the problem and its inefficiencies. Some aspect have jumped to our eyes:

- The number of coupling variables generated by the script were higher than the required ones. The architecture constraints tends to create a weight of each pair of qubits, then when the connection is not reflected in the topology of the graph its value is set to 0. While the correctness of the solution is achieved by this approach, an higher number of decision variables is not desirable since it lengthens the time to get the solution.

- When an instance is fed to the OMT solver, it tends to return solutions where the connections among ancillas get -1 as value. This result is in contradiction with our goal; most of these parameters should have a value different from -1 not to be part of the chain.

5 Improving SAT-to-Ising

This chapter will suggest a novel algorithm to refine the encoding and retrieve a more stable solution, starting from the weaknesses put in evidence in chapter 4.

5.1 Algorithm

The goal of postprocessing is the re-definition of the Ising encoding, modifying offset, biases and couplings between qubits so that the longer chains are reduced in size. Since all nodes that make up the chain are linked by couplings whose value is -1, we will ensure to modify these weights; in addition to that, we will force these nodes in using their unused couplings when possible, .
The only way to achieve this task while not altering the original SAT formulation is to re-compute the scores of each chunk of the simplified formula, defining new OMT problems that involve the additional qubits. In more details, given two penalty functions encoded in a quantum annealer architecture linked by a chain:

- We split the chains into two disjoint sets of nodes. The first set will be associated to the first penalty function, the second to the other.
- We use nodes composing the original penalty functions and the newly assigned qubits of the chain to calculate again the weights of. In particular, we use the nodes belonging to the chain as ancillas for the new function, forcing each coupling between chain nodes and penalty function qubits to be different from zero. We need also to set the most external node of the chain as the actual shared variable, so that we ensure that there is a way to link qubits representing the same Boolean variable and enforcing their equality.
- At this point, we have obtained a subgraph of the original architecture referring to the same. To compute new weights for, we will again rely on formula, but the presence of new ancillas will help us in reducing the number of coupling with score -1. These weights are then used to modify the Ising encoding before being passed as input to the annealer.

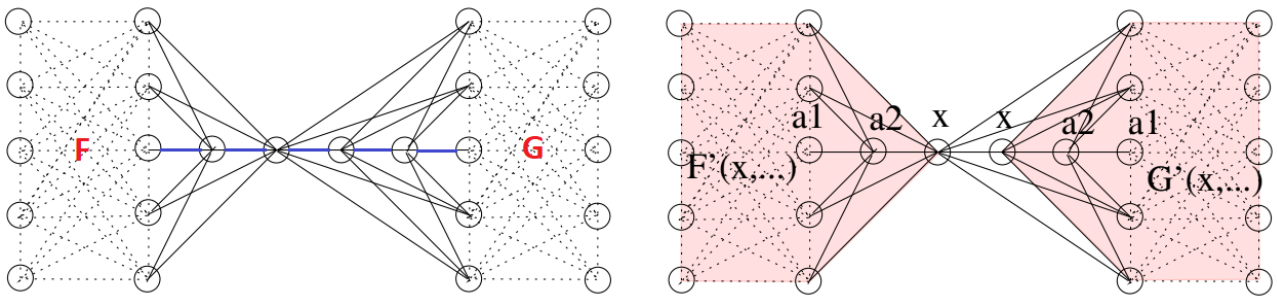


Figure 5.1: A graphical representation of the postprocessing algorithm involving two general penalty functions (F and G) linked by a chain (blue edges). It is important to notice the swap of role between the original node representing the shared variable to ensure consistency of the variable's value between functions.

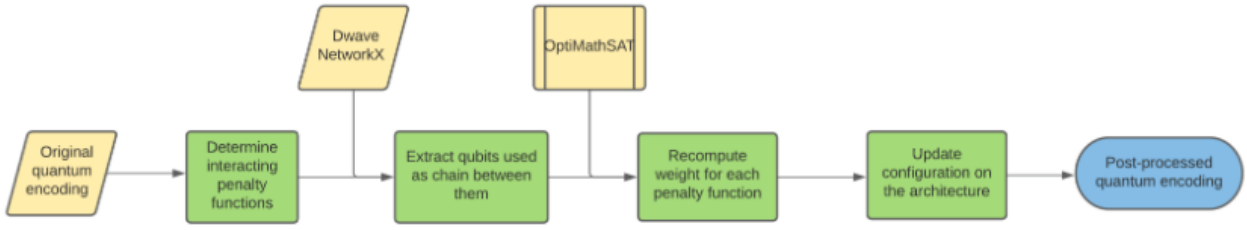


Figure 5.2: A graphical representation of the workflow necessary to implement the postprocessing algorithm. Yellow nodes represent data and state-of-the-art tools; green nodes represent tasks implemented in this thesis.

A graphical representation of the algorithm is shown in figure 5.1. Post-processing cannot be applied to every quantum architecture. In order to be successful, it is necessary that qubits presents an high numbers of connections with other qubits. If this condition is not satisfied, we will not obtain great results from the re-computation of scores for penalty functions: nodes belonging to the chain will not link with external qubits and no new couplings to consider will emerge. D-Wave currently proposes two main architectures: the older one, Chimera, is clearly affected by this issue. The graph of its nodes is sparse and a qubits has at most 6 neighbour nodes, resulting in a non-valid candidate to test the novel algorithm. On the other hand the most recent architecture, Pegasus, is less sparse and it usually does not fall into the issue described above. As a consequence, from now on we will assume that the algorithm will be executed only on the Pegasus architecture.

5.2 Implementation

While the definition of the approach is linear, in practice we have to deal with numerous issues, mainly caused by computational constraints. As a consequence, the implementation relies on some heuristic that avoid apparent deadlock points during execution. The workflow of the implementation is shown in figure 5.2. More details on each step will be provided in the next paragraphs.

5.2.1 Extracting encoding information

The original version of the code returned the list of values assigned to each parameter, without any additional information about the mapping between Boolean sub-formulas and qubits. While it was not important in the precedent version, obtaining this information is now essential to determine how to rearrange qubits in new penalty functions.
To solve the issue, it was required to modify the *place_and_route* library and collect data from the intermediate steps of the procedure. The routines used to determine the chosen qubits and are left unaltered; we simply added some intermediate steps which collected the missing data. In particular we first create a dictionary, called *core_penalty*, where each functions is assigned the set of qubits necessary to map it into the quantum architecture, not considering chains. We also build a second dictionary, *inverted_chains*, where the key-value items of *chains* are swapped; its main role will be to quickly determine if a qubits does not represent represent a logic node and thus not considered by the procedures described in the next paragraphs.

5.2.2 Re-assigning qubits

Once the needed data are stored and available, we can start to post-process the original encoding determining what functions might be extended and what chains are involved. For each pair of penalty functions we have to determine if there is a chain connecting them which can be split

in two disjoint sub-sets of qubits without altering the correctness of the encoding of the original formula. These operations have been implemented in the *postprocessing* function.
Given the set of all penalty functions, we take each possible pair and verify if there are Boolean variables shared by both of them using the field *field* of the data structure *penalties*. If the resulting set is empty, we can discard the couple of functions and move on the next one. In the opposite case, we call the function *get_chain* that will try to identify the chains suitable to be split. We identify the first node of the chain, represented by the qubit associated to the specified Boolean variable and contained into the *core_qubits* of the first function. From that point we recall Pegasus arcitecture from the **DwaveNetworkX** library and check if there are adjacent nodes belonging to the same chain. The new node is then added to the possible final answer and used as new middle point to search adjacent nodes, until we reach a qubits which is also part of the *core_qubits* of the second penalty function. There are some specific cases requiring exceptional handling, including:

- If multiple paths arise from a single node, then we stop the search and return an empty list. The reason behind this choice is the subsequent role swap among the removed variables, which would lead to qubits on the other path to match ancillas and thus return wrong results.

- Similarly to the previous case, we discard also candidate chains that do not use the entire set of available qubits. In this case we know a third function shares the same variable and splitting the chain would case at least one equality with a future ancilla node. The number of times the event occurs is quite low, but it as to be taken into account to avoid unexpected behaviours.

- Lastly we have also to check the case in which the starting and the final node overlap. In this case we do not start the search and immediately have to return the empty list, otherwise we will reach the error case.

In the case a non-empty list is returned by *get_chain*, we can proceed to assign these qubits to the two penalty functions. In order to better organize the collection of the next data, a novel dictionary has been generated, called *penalties_post*. It will contain three fields for each penalty function:

- *nodes*, containing the list of additional qubits that we will assign to that function.

- *edges*, containing the list of additional edges of qubits to consider in the successive steps.

- *swap*, containing pairs of qubits whose role (ancilla/Boolean variable) need to be swapped in order to maintain the exactness of the resulting encoding, as discussed in figure 5.1.

Starting from the non-empty list of qubits, the middle point is chosen from the chain, so that two disjoint sets of nodes are generated. Qubits from the first set are progressively assigned to the first penalty function; moreover we check if there are connections among other qubits from the same penalty function not contemplated by the original problem and, when discovered, added to the *edges* field. Not every node can be added: we recall from table 4.1 that at the current state we have an upper bound on the number of managed qubits to avoid deadlock points. Consequently, an additional check has been considered while scanning the list of nodes so that the upper bound is never outdated.

5.2.3 Recomputing penalty functions

In order to generate the OMT instance necessary to recompute the parameters on the enhanced penalty functions, we need to build the quadruplets of data necessary and call the *searchPF* function using them as argument.
The easiest arguments to obtain are n_x and n_a: the data structures *penalties* and *penalties_post* contain the involved nodes for each function. In details, n_x can be quickly retrieved counting the number of Boolean variables from *penalties*, while n_a is calculated subtracting n_x from the total number of qubits associated to a specific function.
To build the graph representing the interconnection among the nodes, we mainly rely on *penalties_post*. We instantiate a number of nodes equal to the number of needed qubits. Since the common identifiers of Pegasus nodes are not acceptable by **NetworkX** (each nome is represent by a quadruplet of integers), we converted them so that they can be mapped into integer number within the range $[0, n_a + n_x[$. This useful mapping is also stored into a dictionary, *pegasus_to_basic*, simplifying the role swap process. Using the novel nomenclature, we define the essential edges from both *penalties* and *penalties_post*. Lastly, we apply the role swap between the most external ancillas and the variable nodes, adapting *pegasus_to_basic* so that it can reflect the update.
The most difficult task is the definition of the correct relation to use as reference when determining constraints on OMT decision variables. The goal is achieved thanks to the introduction of a procedure, *create_relation*. The algorithm uses the field *form* coming from the genlib file as initial reference, then adapt it so that it satisfies the SMT-LIB standard and correctly refers to the right qubits. Some of the decisive operations involve:

- As a preliminary step, if the genlib chosen at execution is the one accepted by Pegasus, we translate the name of the involved variables in order to use the nomenclature accepted by the Chimera genlib. This task gives us the opportunity to generalize the algorithm and not interfere with the execution of the code when Chimera is chosen as quantum architecture.

- Change each Boolean operator of the initial reference so that it is represented using Python symbols (for instance we rewrite each + into its equivalent representation *or*).

- Combining *pegasus_to_basic* and the field *variables* of *penalties*, we translate each generic variable into its equivalent node of the graph, making sure the role swap is taken into consideration.

The output of the procedure, which was a string, is lastly fed to the Python *eval* function with the aim that a Boolean object is returned.
We also tried to enhance the actual encoding, modifying the definition of the SMT file. Two fundamental directions have been followed. First, we avoided the creation of useless decision variables, such as couplings whose value is set to zero because of architecture constraints. Instead of generating these parameters and waste resources, we extended the class of architectural constraints so that if the edges was not in the original graph, then we can immediately skip to the following one.
The second direction tried to exploit the formulation of a subset of decision variables defined as the weighted sum of some parameters. Instead of setting them using a combination of declaration of variable and assertion to set its value, we define each variable using *let expressions*. Let expressions provide the ability to make expressions more compact by abbreviating common sub-expressions and could impact on the time complexity, albeit slightly.

5.2.4 Updating the configuration

Once we retrieve the parameters for each recomputed penalty function, it is necessary that the quantum encoding reflects these modifies. We decided to rewrite the entire formulation of the problem from scratches, discarding the values of parameters obtained by the old version of the algorithm. Similarly to the original algorithm, we consider the updated chains setting the respective couplings as -1 and progressively increasing the offset by 1 for each edge. After this, for each function we scan the output of the OMT problem and update each parameter accordingly. The OMT solver considers the nomenclature adopted by **NetworkX**, so results are not ready to be used: the manipulation of *pegasus_to_basic* helps us in determining what qubits are involved and what parameters need to be modified.

5.2.5 Validating the final result

The final output provided by the quantum annealer is a dictionary containing the truth assignment for each qubit and the resulting energy as a float number. From equation 3.5 we know that if the final state reach an energy equal to zero we should obtain a satisfiable assignment; on the other hand, an energy higher than 2 indicates an unsatisfiable solution. When using the postprocessing algorithm in its early stages we could not ensure to satisfy this fundamental property; the presence of bugs could lead to the zeroing of the Hamiltonian state with a wrong assignment or the exact opposite. For this reason, additional validation has been provided thanks to the development of an algorithm which test both the old and the post-processed encoding and determining the truth value of the reported solutions. The main interest was the generation of an automatized approach, instead of manually verify the validity or
To accomplish this task, it was required to define a SAT problem: we decided to use OptiMathSAT as SAT solver and, consequently, we needed to write this file under the SMT-LIB standard. Each file is characterized by two main sections:

- The definition of each Boolean variable and its truth value assignments.
- The definition of the Boolean formula we decided to test.

Regarding the first task, we simply needed to scan the returned dictionary and, for each qubits, determine if it was used as an ancilla (and thus its value was not relevant to verify the correctness) or it was yet part of the chains. In the previous steps a data structure, *removed_ancillas*, collects the information, so we only have to check if the qubits belong to this list. In the case of a positive outcome, we discard it and go on: on the other end we determine the physical Boolean variable associated to that node using the *chains* dictionary and declare it using the structure:

define-fun <variable name> () Bool <true/false>

Regarding the second task, we recall how *penalties* collects information about each encoded penalty function, in particular storing the reference Boolean formula represented inside the field *form*. We used this field as the basis for the assertions, then we adapt it replacing the general variables with the one involved in a specific function and modifying the Boolean operators so that they can match the SMT-LIB standard.
The two outputs have been merged into a single string variable and then fed to OptiMathSAT, which provided in response the satisfiability of the generated assignment.

6 Results

In this chapter multiple tests will be discussed to prove the correctness of the newly implemented approach, considering not only ordinary cases but also relevant border cases which quite often tend to appear. We will also cover the application on a small set of benchmark files, showing how improvements are brought by the suggested procedure. For each example the graphical representation of the encoding into the Pegasus architecture is shown. The figures will respect the following conventions:

- Gray nodes and gray edges represent respectively unused qubits and unused couplings.
- Qubits and edges belonging to a specific penalty functions have the same color.
- Black nodes and black edges represent chains connecting shared variables among sub-formulas.
- Yellow edges represent couplings activated from the postprocessing algorithm. Both unused coupling and couplings from chains can be considered from the procedure and thus be marked with yellow.

6.1 Evaluation metrics

Before starting the discussion of the experimental evaluation, it is fundamental to dwell on the elements we will use to evaluate the effectiveness of the novel approach. While the correctness of the postprocessing procedure surely represents the focal point of our tests, we need to define some parameters to evaluate the quality of the obtained results. In particular, our interest will be aimed at the following properties:

- **Length of chains**: as stated previously, long chains negatively impact the stability of encoding while searching the final state. Reporting the average of chains' length can offer an idea about the expected stability we have reached.
- **Number of "wasted" qubits**: a qubit is considered "wasted" if it is linked exclusively with qubits belonging to the same chain. The unused couplings could give benefit to the definition of the novel instance, generating more complex sub-graphs that is useful when calling the *searchPF* function. In particular we will take into account the percentage of "wasted" qubits with respect to the overall number of used qubits for a SAT problem.
- **Maximization of** min_{gap}: every time we recompute the value of offset, biases and couplings for each sub-formula we generate on OMT problem, whose optimization section tries to maximize the value of min_{gap}. Obtaining, when possible, values higher than their original formulation will result in an higher stability, since it will be easier for the annealer to determine satisfying assignments.

6.2 First cases

At first we decided to test the postprocessing algorithm converting simpler formulas, which would clearly prove the correctness of the translation in a variety of cases of different nature. Since a limited number of available samples are available (mainly due to the accepted format

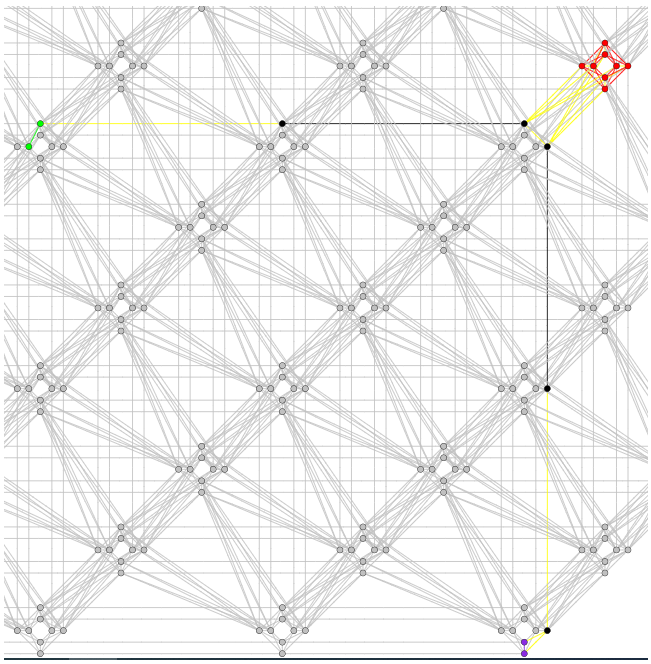


Figure 6.1: Graphical representation of the fulladder test.

of input file, binary AIG which is rarely used in the generation of benchmarks), the generation of these input files had to be considered as part of the project.
The algorithm uses the open source library *pyAiger* [9] to convert a Boolean formula into its AIG-equivalent representation, producing an AGG file. The AGG format is the ASCII format of an And-Inverter Graph: while almost identical to AIG, presents some differences that prevent *ABC* to accept it. In particular, AIG is semantically a subset of the ASCII format. The binary format may need to reencode literals, but translating a file in binary format into ASCII format and then back in to binary format will result in the same file. Luckily the computer scientists behind the birth of the format developed a tool written in C, called *aigtoaig*, enable a simple interface to convert the AGG file into AIG. Using the two tools we were able to generated problems with specific properties, giving us the opportunity to test it in its entirety.

6.2.1 Testing multiple chains from a single penalty function

In this batch of experimental tests we tested the correctness of postprocessing in the case of a penalty function with two or more shared variables. First we tested the case *fulladder* with three sub-formulas:

- F1: $FULLADDER(i1, i2, i3, o1, o2)$
- F2: $i1 \neq e1$
- F3: $i2 \neq e2$

Penalty function F1 shares two variables with the other sub-formulas, ensuring two chains will start from it. Moreover, it represented the first occasion to test the newly generated penalty functions inserted into the Pegasus genlib file. Figure 6.1 shows the placement of the algorithm and the effects of the postprocessing algorithm. Metrics about the two encodings are shown in table 6.1. We can see how the postprocessing managed to correctly re-assign unused qubits guaranteeing, when possible, the generation of complex structures, in particular the one associated with F1. We can also observe a reduction to the chains' length and the presence of wasted qubits.

	Classic encoding	Postprocessing encoding
Average chain length	3.5	1
N. wasted qubits	5	2
Max min_gap	3	3
Correct solution	Yes	Yes

Table 6.1: Evaluation of postprocessing in the *fulladder* test

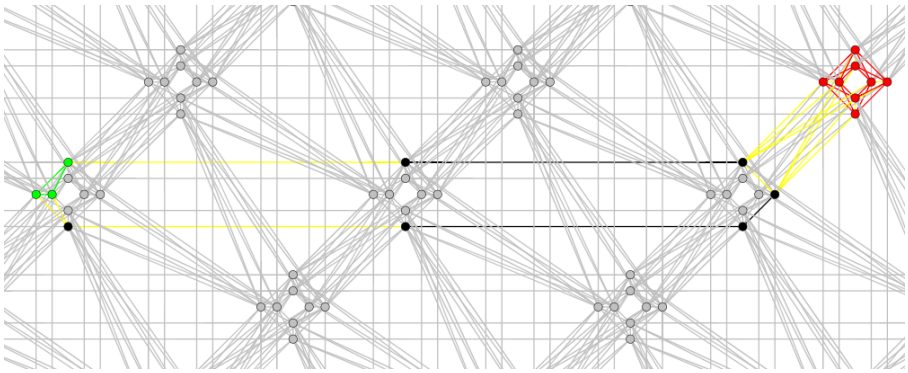


Figure 6.2: Graphical representation of the *doublesharing* test.

A second test, named *doublesharing*, has been performed with similar conditions, to cover another frequent event. This time we considered two subformulas sharing two common variables:

- F1: $FULLADDER(i1, i2, i3, o1, o2)$
- F2: $o3 \iff (i1 \wedge i2)$

This test took place in order to verify if the code section dedicated to the management of multiple chains between two functions worked correctly. The placement and the evaluation of the encodings are shown respectively in figure 6.2 and table 6.2.

6.2.2 Testing variation of min_{gap}

This batch of experimental trials will focus on testing the postprocessing procedure using various gates from different complexity, in order to determine if it is actually possible to obtain alternative encodings with higher gaps for a subset of penalty functions. In particular we will test Boolean formulas requiring a number of qubits in a range between 4 to 8 variables, all of them extracted from the old Chimera genlib (opportunely converted into a valid Pegasus formulation).

	Classic encoding	Postprocessing encoding
Average chain length	4	1.5
N. wasted qubits	6	2
Max min_gap	3	3
Correct solution	Yes	Yes

Table 6.2: Evaluation of postprocessing in the *doublesharing* test

Gate22

The two sub-formulas involved into the test case are:

- F1 ($gate22$): $o1 = (i1 \vee i2) \wedge (i1 \vee i3) \wedge (i2 \vee i3)$
- F2: $i1 \neq e1$

Figure 6.3 and table 6.3 report the result of the SAT-To-Ising transformation.

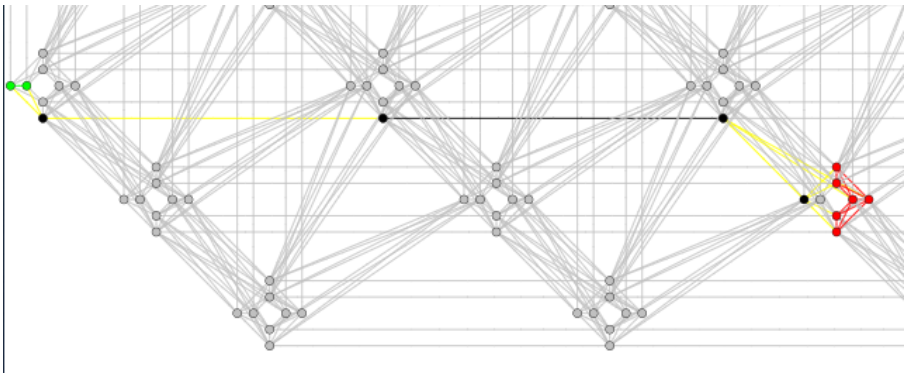


Figure 6.3: Graphical representation of the *gate22* test.

	Classic encoding	Postprocessing encoding
Average chain length	4	1
N. wasted qubits	4	2
Max min_gap	2	4
Correct solution	Yes	Yes

Table 6.3: Evaluation of postprocessing in the *gate22* test

7 From CP to OMT

This chapter will discuss the CP-to-OMT problem and the implementation of an interface achieving this task. This work represents an additional step with respect to the SAT-to-Ising problem, but it could provide benefit to it: a new class of problems could be fed to quantum annealers and pave the way for new directions.

7.1 The first interface: FZN2OMT

The first tentative to implement a CP-to-OMT converter is represented by FZN2OMT, developed by Patrick Trentin at the University of Trento and heavily discussed in [6]. The interface, written in Python, managed the majority of issues raised in chapter 2.5.1 and is capable of translating a good variety of MiniZinc/FlatZinc problems. The compiler was developed in order to test the performance of OptiMathSAT in solving CP problems, which was an unexplored field at that time. Results showed how OMT solvers are usually penalized by the absence of procedures for global constraints and the management of Pseudo-Boolean constraints with respect to MiniZinc, offering cues for further studies and improvements.
One of the main issues behind this solver is its dependence from MathSAT: as already stated, OptiMathSAT is built on top of the SMT solver and, since it is proprietary code, the code cannot be distributed as open-source code. This condition prevents other computer scientists to easily test the bridge between the two paradigms and slows down the search of OptiMathSAT critical points to fix. As a consequence, the first goal we decided to pose was the independence from the SMT solver and the release of the code under the GNU General Public License v. 3.0 [4].
Once accomplished this first essential task, it was interesting to extend this primitive interface introducing new updates, most of them generated by MathSAT and OptiMathSAT evolution in the last years. Some arithmetics are currently not considered in the translation, in particular the Bit-Vector algebra (when expressly stated, Bit Vector variable are simply translated into SMT Integer variables); enabling it would improve the bridging between the two paradigms, speeding up the development of OMT.

7.2 Open-source FZN2OMT

In the next paragraphs we will discuss the workflow and the design choices adopted to build the open-source version of FZN2OMT. The first part will cover the development of an equivalent version of the previous FZN2OMT free from MathSAT dependencies, while improvements and extensions to the original tool will be discussed in the second section.

7.2.1 Designing the skeleton

In order to develop a standalone version of the interface satisfying the requirements underlined in paragraph 3.1, it was necessary to modify the majority of data structures and algorithms. The goal was to remove, when possible, procedures and structures which were useless for the tasks we wanted to integrate; the remaining cases required huge modifies to simplify their behaviour and step away from their original encoding. Lastly, all dependencies from the SMT solver had to be removed to avoid legal issues.
As a first step, we took the OptiMathSAT code and manually analyzed its structure to define the MathSAT dependencies that were useless for the paradigm conversion task. In particular,

all references to algorithms dealing with the search of a valid solution were outside the scope of the project and promptly removed. Not all MathSAT libraries could be entirely removed from the project without impacting the execution of the code: the OMT and the SMT solvers share some data structures essential to efficiently build an SMT-LIB encoding. To ensure the possibility to license the new tool under GPL, all these structures and the methods calling them had to be re-written from scratches. In the beginning we were simply interested in making sure no link with the SMT solver were present in the final release: as a result, we built dummy classes with attributes and empty methods inspired by MathSAT files.
At the end of this process, we obtained an executable interface with no dependency from the SMT solver developed by UniTN and capable of scanning the input file, even though no translation is returned. It is important to point out how the FlatZinc language parser was part of OptiMathSAT source code and it was not required to rewrite it nor modify its code.

7.2.2 Storing variables and constants

The skeleton built in chapter 7.2.1 represents the starting point to define a working interface, satisfying the requirements previously described and accepting all the FlatZinc problems managed by the old compiler.
The first aspect we concentrated on was the definition of classes for variables declared in the FlatZinc Language file. Since each variable can be associated to different data types, we first added a class to efficiently store information about the type, **DataType**; this information will help us in further manipulations and simplifications, as we will deepen later. A DataType is simply defined by a unique identifier and the name we assign to the represented type, such as Int or Real. Since the solvers do not admit user-defined types, we added an initialization procedure to create the relevant data types before we parse the CP file.
The next step was the definition of a class to store information about frequently used symbols, such as equality and addition; we named this class **Symbol**. This class does not differ too much the previous class: in addition to the name and the identifier we also associate its type using an instance of the DataType class.
Each variable is embedded in a class representing the basic unit of a formula, called **Term**. For the sake of simplicity, we designed this class reducing to the bone its body:

- To identify and distinguish variables, we use the already mentioned combination of ID + name.
- To store its type we declare a Symbol with name identical to the one provided for the variable, that will also store the type thanks to its DataType attribute.
- Lastly, we introduced a vector of Term called *children*. When creating variables, this attribute remains empty. Its importance will be cleared when discussing the creation of constraints.

The definition of constants is identical to the procedure for decision variables, with an additional step to link the symbol with its value. This association was managed creating an unordered map using the name as key and the assigned parameter as value.
All these newly created instances were collected defining a container class, **TermManager**. A brief graph summing up the hierarchy of classes and their role in the program is shown in figure xx.

7.2.3 Introducing basic behaviour

The fundamental task that the interface is required to satisfy is the encoding of FlatZinc constraints into SMT-LIB equivalent assertions. Some of these constraints (the MiniZinc manual calls them **FlatZinc builtins**) define simple relations among variables; the other constraints,

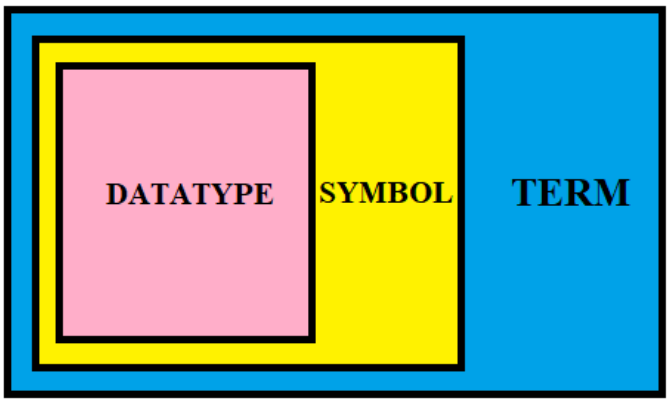


Figure 7.1: An abstract representation of the class hierarchy in the open-source FZN2OMT interface.

global constraints, represent high-level modelling abstractions, for which many solvers implement special, efficient inference algorithms and require more complex encoding. Each constraint is managed with the

7.2.4 Extending FZN2OMT

Once the essential behaviours were implemented, we decided to extend the project adding that would increase its popularity and its usage.
First of all, OptiMathSAT is only one of the list of the OMT solvers available to the final users. Other popular solvers are z3 [13] and Barcelogic [1], each one with their strengths and weaknesses. All these solvers support the SMT-LIB standard, but the format accepted by each tool presents some minor differences with respect to the others. Even though the community is currently working in removing these differences and provide a unified format to simply test a single instance of a problem in multiple tools, at the moment we need to adapt the SMT-LIB encoding to satisfy the semantic of each solver. Some of the most evident differences involve:

- The setting of the background theory to exploit for a problem: since the SMT-LIB admits, for each tool we have to make sure a supported logic is passed, otherwise an error will be returned an the execution aborted.

- The definition of the objective functions: each solver has different format to define the direction of the optimization and the cost function we are interested in optimizing.

Once read the manual guide for all these solvers and uniquely determined the differences that would cause an error, the best strategy that emerged to accomplish the task was the update of the *Environment* class. Every time the declaration of a variable or an assertion is needed, the class now checks if the output language is not the default one and, in case of a positive answer, the string is modified before being written in the output file. The principal modifies concerns changes of keywords and the definition of the cost functions. The choice of the output solver is done when executing FZN2OMT: a command-line parameter has been added where the user can change the default output solver (OptiMathSAT) with the other tools named at the beginning of this section. The *TermManager* class read this parameters and set a flag,

enabling the translation in run-time when required.
The second extension proposed to the interface involves the Bit-Vector arithmetic. At the current state, FZN2OMT does not provide support for Bit-Vector encoding and Integer value cannot be mapped to an equivalent finite-precision encoding. On the other hand, huge progresses has been made in the definition of algorithms to efficiently search solutions for this theory, so testing this arithmetic would be of interest for the field Optimization Modulo Theories. SMT-LIB defines a different syntax for Bit-Vector variables and operations: as a result, the original structure of the *Term* class has been extended. A new field has been added, *class_term*, indicating the size of the variable required to store the specific Integer value. If the variable is a constant, then the parameter is set in order to use the minimum number of bits to represent the number, otherwise a default value is set (the default value is originally set to 25, but it can be manually modified at execution time). This field is essential, since SMT-LIB requires to add the size of the variable when declared.
Once renovated the class, it was necessary to extend the arithmetic operations to avoid run-time error. In particular, we had to be careful while adding or multiplying BitVectors, since the size of the resulting variable dynamically depends from the starting terms. As a result, a set of new operator has been implemented starting from the relevant operators from the Integer and Float domain. When two constant are passed as input, then the interface simply compute the result of the operation and create a new variable storing it. On the opposite, if at least one of the input is a decision variable, then we have to create an empty variable whose size is capable of store values in the worst possible case, avoiding any Integer overflow issues. The heuristic adopted was the following:

- In the case of additions or subtractions of two variables, we get the higher size between the two BitVectors and increase this value to 1.
- In the case of multiplications, we get the higher size between the two BitVectors and double it.
- The remaining operations, such as two's or one's complement, simply set the size of the resulting answer as equal to the input size.

The last step was the update of local and global constraints, so that they could handle BitVectors when asked. For each constraint in which the BitVector arithmetic actually impact the definition of the assertion, a map between Integer and their equivalent BitVector operations is generated. A flag in the *Environment* class tracks the choice of the user about the type of encoding to apply, quickly swapping between the two typologies of functions.
For each operation, both the unsigned and the signed version are coded, covering the full extent of the arithmetic. To enable the BitVector conversion we rely on a command-line parameter, in a fashion similar to the multi-language option.

8 Conclusions

From the analysis discussed in the second part of this thesis, we see how the development of a postprocessing procedure can benefit the quantum encoding and provides more stable encoding. The most interesting success is represented by the capability of obtaining new penalty functions w simply re-arranging some qubits

8.1 Future work

8.1.1 Enhancing CP-to-OMT

A second future direction to take into account would focus on bridging the gap between Constraint Programming and Optimization Modulo Theories. The open-source interface described in the previous chapters can be extended to efficiently manage each MiniZinc structure and procedure. To cover the full extent of CP standards some additions are required, such as defining an encoding based on Floating-Point Numbers or dealing with non-linear constraints and objectives (in this case the first future direction will help in completing the task). This may require a general review of the current implementation of the FlatZinc interface, so as to become modular and easy to extend. Moreover, we can improve the situation by developing a T-solver for the theory of sets, which is currently managed using a quite inefficient mix of Boolean and arithmetic constraints. To conclude, another interesting issue concerns CP global constraints, constraints that capture a relation between a non-fixed number of variables. In particular, it would be necessary to study approaches to integrate dedicated procedures to the SMT and OMT framework emulating these constraints.

Bibliography

[1] Barcelogic. https://barcelogic.com/en/.

[2] D-wave networkx.

[3] D-wave: The quantum computing company. *https://www.dwavesys.com*.

[4] Gnu general public license. https://www.gnu.org/licenses/gpl-3.0.html.

[5] Zhengbing Bian, Fabian Chudak, William Macready, Aidan Roy, Roberto Sebastiani, and Stefano Varotti. Solving sat and maxsat with a quantum annealer: Foundations and a preliminary report. In Clare Dixon and Marcelo Finger, editors, *Frontiers of Combining Systems*, pages 153–171, Cham, 2017. Springer International Publishing.

[6] Francesco Contaldo, Patrick Trentin, and Roberto Sebastiani. From minizinc to optimization modulo theories, and back (extended version), 2019.

[7] T. Lanting, R. Harris, J. Johansson, M. H. S. Amin, A. J. Berkley, S. Gildert, M. W. Johnson, P. Bunyk, E. Tolkacheva, E. Ladizinsky, N. Ladizinsky, T. Oh, I. Perminov, E. M. Chapple, C. Enderud, C. Rich, B. Wilson, M. C. Thom, S. Uchaikin, and G. Rose. Cotunneling in pairs of coupled flux qubits. *Phys. Rev. B*, 82:060512, Aug 2010.

[8] Joao Marques-Silva, Ines Lynce, and Sharad Malik. *Conflict-driven clause learning SAT solvers*, pages 131–153. Number 1 in Frontiers in Artificial Intelligence and Applications. IOS Press, Netherlands, 1 edition, January 2009.

[9] Mvcisback. mvcisback/py-aiger.

[10] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll (t). *Journal of the ACM (JACM)*, 53(6):937–977, 2006.

[11] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3(3-4):141–224, 2007.

[12] Roberto Sebastiani and Patrick Trentin. Optimathsat: A tool for optimization modulo theories. pages 447–454, 07 2015.

[13] Z3Prover. Z3prover/z3. https://github.com/Z3Prover/z3.