Automaton-Based Methodology for Implementing Optimization Constraints for Quantum Annealing

Hristo Djidjev
Los Alamos National Laboratory, Los Alamos, NM 87545, USA
djidjev@lanl.gov

ABSTRACT

Quantum annealing computers are designed to produce high-quality solutions to optimization problems that can be formulated as quadratic unconstrained binary optimization (QUBO) problems. While most of the well known NP-hard problems can easily be represented as quadratic binary problems, such formulations usually contain constraints that have to be added as penalties to the objective function in order to obtain QUBOs. In this paper, we propose a method based on finite automaton representation of the constraints for generating penalty implementations for them, which uses fewer qubits than the alternatives and is general enough to be applied to a whole class of constraints.

CCS CONCEPTS

• **Theory of computation → Formal languages and automata theory**; **Quantum computation theory**; **Discrete optimization**.

KEYWORDS

quantum annealing, D-Wave, finite state automaton, QUBO, Ising problem, constrained optimization, Chimera graph

ACM Reference Format:
Hristo Djidjev. 2020. Automaton-Based Methodology for Implementing Optimization Constraints for Quantum Annealing . In *17th ACM International Conference on Computing Frontiers (CF '20), May 11–13, 2020, Catania, Italy*. ACM, New York, NY, USA, Article 0, 8 pages. https://doi.org/10.1145/3387902.3392619

1 INTRODUCTION

NP-hard optimization problems are difficult to solve within the prevailing software and hardware paradigms, motivating researches to search for different approaches. Quantum computing is currently being actively studied as a promising alternative for dealing more efficiently with such problems. Quantum annealing (QA) [2, 6] and QA computers such as the commercially available D-Wave machines are able to generate in a very short times optimal or high-quality solutions to such problems as long as they can be formulated in a way acceptable for the hardware. Specifically, an input problem of interest should be formulated as an *Ising* problem, which is a problem of minimizing a quadratic form of the type

$$Is(\boldsymbol{x}) = \boldsymbol{x}^T J \boldsymbol{x} + \boldsymbol{h}^T \boldsymbol{x} = \sum_{i,j=1}^{n} J_{ij} x_i x_j + \sum_{i=1}^{n} h_i x_i, \tag{1}$$

where $J \in \mathbb{R}^{n \times n}$ and $\boldsymbol{h} \in \mathbb{R}^n$ are coefficients, and $\boldsymbol{x} \in \{-1, 1\}^n$ is the variable. An alternative formulation, where the objective function is of the same type, but the variable $\boldsymbol{x}$ is in $\{0, 1\}^n$, is called a *Quadratic Unconstrained Binary Optimization (QUBO)* problem [7]; both formulations, Ising and QUBO, can be converted into each other by a linear transformation of the variables. Most well-known NP-hard problems can be easily formulated as problems of minimizing quadratic functions of the type (1), however such formulations almost always contain constraints in addition to the objective function [3, 8, 9], which is not allowed in an Ising/QUBO.

In order to solve constrained minimization problems, one can add the constraints to the objective as squared penalty terms. For instance, a constraint

$$\sum_{i=1}^{n} x_i = k, \; x_i \in \{0, 1\}, \tag{2}$$

which encodes the requirement that exactly k of the decision variables x_i should be set to 1 (true), and which is probably the most common one used in constrained optimization, can be included into the objective using a penalty function

$$Q(\boldsymbol{x}) = \left(\left(\sum_{i=1}^{n} x_i \right) - k \right)^2 \tag{3}$$

by adding $M \cdot Q(\boldsymbol{x})$ as a penalty term to the objective function for large enough M, where $\boldsymbol{x} = (x_1, \ldots, x_n)$. For a *feasible* solution, i.e., if the variables satisfy the constraint, the penalty term is zero and it doesn't change the value of the objective function. But in an infeasible solution, $M \cdot Q(\boldsymbol{x})$ will have a value at least M, and for a large enough M it will prevent any infeasible solution to be optimal in a minimization problem.

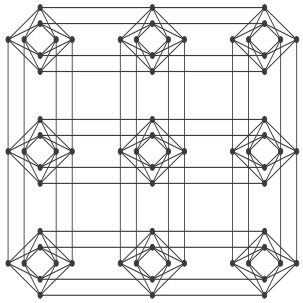

Figure 1: A Chimera graph consisting of a 3×3 array of unit cells.

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor, or affiliate of the United States government. As such, the United States government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for government purposes only.

CF '20, May 11–13, 2020, Catania, Italy
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7956-4/20/05…$15.00
https://doi.org/10.1145/3387902.3392619

One drawback in using directly penalties of the type (3) is in the small number of available qubits and couplers of existing quantum annealers. *Qubits* are the basic units that can store quantum information, analogous to the bits of classical computers, but in D-Wave machines they are also used to process information. D-Wave 2000Q machine has more than 2000 qubits interconnected using about 6000 links between them, called *couplers*. The interconnection topology is based on the *Chimera graph* (*CG*), which consists of a $k \times k$ array of *unit cells*, where each unit cell is a 4×4 complete bipartite graph (Figure 1). D-Wave 2000Q quantum processor is a 16×16 array of unit cells, but may have some qubits and couplers missing due to manufacturing defects. In order to solve an Ising problem of the type (1), each variable x_i and its corresponding coefficient h_i should be mapped to a qubit or a connected set of qubits, and each coefficient J_{ij} should be mapped to the coupler(s) joining qubits implementing variables x_i and x_j. We also refer to these coefficients as *weights* in the corresponding *CG* graph.

After an Ising or a QUBO problem is mapped onto *CG*, D-Wave's hardware goes through a process called *quantum annealing* that aims to bring the quantum system of qubits and couplers into a state of minimum energy. When measured, each qubit of that final state gives a value -1 or 1, and hence a minimum-energy output of the annealer can be converted to a solution of a minimum value for the function (1).

Qubits in current generations of D-Wave have upto 6 couplers incident to them, which means that the total number of couplers grows as $O(n)$ with the number n of qubits, while the number of quadratic terms, $2x_ix_j$, in (3) grows as $O(n^2)$. Since each quadratic term has to be mapped to a distinct coupler, and the number of couplers is fixed (to about 6000 in the newest D-Wave 2000Q machine), this means that using a penalty function of the type (3) might severely restrict the sizes of the problems that can be solved on D-Wave, even if the original objective is sparse (i.e., has linear number of terms $J_{ij}x_ix_j$.) Hence, in this paper we are proposing a new better method for generating penalty functions that have fewer quadratic coefficients than (3).

One approach to reduce the number of quadratic terms in a penalty is to determine the coefficients of a suitable sparser quadratic function to be used as a penalty function, using optimization. If X denotes the set of feasible and $\overline{X}$ denotes the set of infeasible solutions, then the relevant properties of the original penalty $Q(\boldsymbol{x})$ are the following :

Property 1.

(i) $Q(\boldsymbol{x}) = 0$ *for all* $\boldsymbol{x} \in X$, ;
(ii) $\min\{Q(\boldsymbol{x}) \mid \boldsymbol{x} \in \overline{X}\} = \lambda > 0$.

If we multiply all coefficients by a factor $z > 1$, that will also scale up the gap by a factor z. But D-Wave annealers have limits on the largest coefficients allowed by the hardware, which are 1 for the quadratic terms and 2 for the linear terms. Therefore, we assume that the coefficients of Q are normalized in a way that satisfy these limits. In (3), the quadratic terms are 2, so scaling all coefficients by a factor of 0.5, we find that the gap is $\lambda = 0.5$. Having larger gaps is important because it allows using a smaller value of M, which may greatly affect the accuracy of the annealer [1].

Hence, our goal is to find a quadratic function satisfying conditions analogous to (i) and (ii), but with better properties than $Q(\boldsymbol{x})$ such that sparser function $Q(\boldsymbol{x})$ and larger value for λ. Furthermore, in order to provide more degrees of freedom that will make the optimization more effective, we lift the dimension of the search space by introducing ancillary variables $\boldsymbol{t} = (t_1, \ldots, t_m)$ and replacing $Q(\boldsymbol{x})$ with the function

$$Q(\boldsymbol{x}, \boldsymbol{t}) = \boldsymbol{x}^T A\boldsymbol{x} + \boldsymbol{x}^T B\boldsymbol{t} + \boldsymbol{t}^T C\boldsymbol{t} + \boldsymbol{a}^T\boldsymbol{x} + \boldsymbol{c}^T\boldsymbol{t}, \tag{4}$$

where $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{m\times m}$, $\boldsymbol{a} \in \mathbb{R}^n$ and $\boldsymbol{c} \in \mathbb{R}^m$, and all coefficients are within the limits allowed by the hardware. Then, the following analogues of properties (i)-(ii) have been previously formulated for $Q(\boldsymbol{x}, \boldsymbol{t})$ [1, 11, 12]:

Property 2.

(i′) $\min_{\boldsymbol{t}}\{Q(\boldsymbol{x}, \boldsymbol{t}) \mid \boldsymbol{x} \in X\} = \mu$,
(ii′) $\min_{\boldsymbol{t}}\{Q(\boldsymbol{x}, \boldsymbol{t}) \mid \boldsymbol{x} \in \overline{X}\} = \mu + \gamma$, *for some* $\gamma > 0$

Note that we need a parameter μ rather than the 0 used in formulation (i)-(ii) since function (4) lacks a constant additive term that is present in (3).

In order to solve that problem, [1] formulates an optimization problem with constraints (i′)-(ii′) and an objective to maximize the parameter γ, called *gap* between feasible and infeasible solutions, plus additional constraints tied to the D-Wave hardware. The first such constraints is that the zero-nonzero structure of the coefficients of $Q(\boldsymbol{x}, \boldsymbol{t})$ is compatible with the topology of D-Wave, i.e., each variable x_i or t_j should be mapped to a distinct vertex of *CG* and, for each quadratic term of $Q(\boldsymbol{x}, \boldsymbol{t})$, there should be an edge between the vertices implementing the variables of the term. The second constraints is that the absolute values of the coefficients of $Q(\boldsymbol{x}, \boldsymbol{t})$ shouldn't exceed the ones allowed by the D-Wave hardware, which are 1 for quadratic terms (couplers) and 2 for the linear terms (qubits).

Unfortunately, the resulting optimization problem is computationally very hard. Straightforward implementation leads to a mixed-integer problem with $O(\binom{n}{k}m)$ binary variables and $O(2^{n+m})$ constraints, making it infeasible except for very small n, m, and k. For $k = 1$, Bian et al. [1] are able to solve the optimization problem for $n = 8$, $m = 8$, and find that $\gamma = 4$ is optimal, using a Satisfiability Modulo Theory (SMT), combined with dynamic programming techniques. Vyskocil et al. [11] propose a highly scalable approach for the constraint $\sum x_i = k$, which first designs an Ising program on a single cell with special properties, such that when the cell is replicated to all other cells used for the constraint, the combined Ising program satisfies the requirements (i′)-(ii′). In [12], a similar approach is used for an inequality type of constraint, e.g., $\sum x_i \le k$ or $\sum x_i \ge k$. A drawback of the approach of [11, 12] is that it is challenging to find such single-cell Ising program. The special properties of the cell Ising are tied to the specific constraint, and for each constraint a new single-cell Ising program should be designed. Moreover, there is no method or rules for designing such a program, and it is done in an ad-hoc, trial-and-error manner. It is not clear, for a given constraint, if such Ising program exists, and each new design would require a new proof of correctness.

In this paper, we propose an approach for generating penalty implementations of constraints that uses the single-cell design idea of [11, 12] discussed above, but which is general, simple, intuitive,

can be applied to a whole class of constraints, and which can produce in some cases more efficient designs than the known ones. The main contributions of this paper are:

- We describe a method for implementing any constraint that can be expressed as a regular language, i.e., constraint for which the set of the feasible tuples $(x_1, \ldots, x_n)$ defines a regular language over alphabet $\{-1, 1\}$.
- We apply the method to languages accepted by automata with 2 or 3 states.
- We show how a 2-phase version of the method can handle languages accepted by automata with any number of states.
- We construct implementations of constraints where the number of +1 variables is (a) odd, (b) even, (c) equal to 2, and (d) equal to a given k.
- We improve the number of cells needed to implement (d) from $O(nk)$ [12] to $O(n \log k)$, where n is the number of the variables.

2 PRELIMINARIES AND APPROACH OVERVIEW

2.1 Preliminaries and notation

A *finite state automaton* (FSA) is specified by a finite set of *states* Q, a finite alphabet Σ, a transition function $\delta : Q \times \Sigma \rightarrow Q$, a start state, and a set of final (accepting) states. Automata are graphically represented by using a circle for a state and an arrow between states q_0 and q_1 labeled by x for each transition $\delta(q_0, x) = q_1$ (we denote this transition also by (q_0, x, q_1)); an 'S' with an arrow points to the start state, and a double circle denotes the final state (Figure 2). Informally, automaton $\mathcal{A}$ works on a given input string s by starting in the start state (state **−1** in Figure 2), and on each next symbol σ_i of s goes to the next state based on the current state and σ_i, as determined by the transitions (arrows) between states. If after the last symbol of s is read the state is final (e.g., state **1** in Figure 2), then s is accepted by the $\mathcal{A}$, otherwise s is rejected by $\mathcal{A}$. The set of all strings s that are accepted by $\mathcal{A}$ form the language accepted by $\mathcal{A}$. For formal discussions of automata see [5]. Automata have applications in, e.g., design of compilers, circuit design, text editors, and spell checkers.

We assume that the set of the strings satisfying the constraint we want to implement, called *feasible* strings, is a regular language over the alphabet $\{-1, 1\}$. (A *regular language* is a language (set of strings) accepted by an FSA.) We will also be using *Mealy machines*, a type of FSAs with output, where each transition outputs an alphabet symbol, see [5] for additional information on FSAs.

Since our alphabet is $\{-1, 1\}$ (it is more suitable for our method than $\{0, 1\}$), we will be using constraints given in Ising format (i.e., with variables in $\{-1, 1\}$). But in some cases it will be convenient to temporarily switch back to $\{0, 1\}$ (QUBO) representation. We denote by $B(x) = (x + 1)/2$ the function that maps -1 to 0 and 1 to 1. If we replace in the binary representation of an integer l each 0 by -1, the resulting string will be called an *I-binary* (for Ising binary) representation of l. For instance, 5 in binary is 101, and in I-binary it is $1(-1)1$. Finally, for $\boldsymbol{x} \in \{-1, 1\}^n$, we denote $\sum_{i=1}^{n} B(x_i)$ by $sum_n(\boldsymbol{x})$, or simply $sum(\boldsymbol{x})$, when n is clear from the context. For instance, the condition that, for $\boldsymbol{x} \in \{-1, 1\}^n$, exactly one x_i is 1 (and all other are -1) can be written as simply $sum(\boldsymbol{x}) = 1$.

2.2 Approach overview

Our procedure for implementing a constraint as an Ising function *Is* involves the following steps, which we discuss in more detail in the next sections:

(i) Identify a small number of single-cell Ising functions we call *gadgets* (see Figure 3). These gadgets are only structurally specified (e.g., by the number and types of variables) at this stage, while the values of their coefficients will be done in step (iv).

(ii) For each gadget, divide the set of all its variable assignments (called *tiles*) into two classes: a small set of *good tiles* (see Figure 4), and a set of *bad tiles* containing all remaining assignments. Additionally, divide the set of all pairs of good tiles into *good pairs* and *bad pairs*. Defining the good tiles and pairs is the most critical step of the approach, because they should help distinguish between variable assignments that satisfy and that do not satisfy the constraint, as formalized in the following property [11, 12].

PROPERTY 3. *Given a constraint C, an Ising program Is, and a variable assignment A for Is, consider the tiles defined by the projection of Is and A to all unit cells of the Chimera graph C. Then A should satisfy C if and only if all tiles are good tiles and all pairs of adjacent tiles are good pairs.*

Defining good pairs in the approach of [11, 12] is straightforward: the adjacent "colors" of neighboring tiles should be different, see details in the next section. But defining the correct set of good tiles (or even determining if it exists) is a very challenging task, since there is no clear link between the formulation of a constraint given as a formula of the type (2) and the particular set of good tiles corresponding to that constraint. Using an FSA representation of the constraint and the method described in this paper makes this step straightforward (e.g., see Figure 4 (i)).

(iii) Determine properties of the gadgets coefficients that guarantee gap γ between optimal/suboptimal values of *Is*. While Property 3 guarantees that assignments satisfying C minimize *Is* and ones that don't satisfy C don't minimize *Is*, we also want the gap γ between the two sets to be as large as possible, as required by Property 2. Hence, in this step, we express the value of γ as a function of the coefficients of *Is*, that will allow us to compute these coefficients by solving an optimization problem in the next step (iv).

(iv) Formulate an optimization problem whose constraints define the value of γ as determined in (iii), and whose objective is to maximize γ, and solve it.

(v) Replicate, arrange and connect the gadgets, which define Ising programs for single unit cells, so that they form an Ising problem on the entire Chimera graph that implements the entire original constraint.

3 IMPLEMENTING A CONSTRAINT REPRESENTED BY A 2-STATE FSA

We illustrate the approach first on a simple case, of a 2-state FSA. Our objective is to design an algorithm that, given a constraint

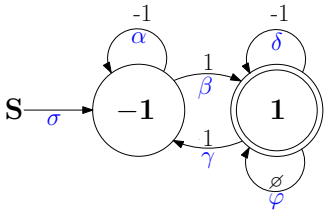

Figure 2: An FSA, $\mathcal{F}_o$, for $sum(\boldsymbol{x}) \in O$. Bold -1 and 1 are states, Greek letters denote the transitions, and regular -1 and 1 denote input characters. $\varnothing$ denotes the end-of-input character.

specified by such an FSA, can generate specifics for the implementation of most of the steps above. We only need "manually" to set the structure of the gadgets in step (i), and in step (iv) to solve the optimization problem, as they are specific for each automaton. Next we describe how each step is implemented given the automaton description. We assume that there is only one final state since if both states are final, then the FSA accepts all inputs.

In order to make it easier to follow, we describe a specific case, where the constraint is "$sum(\boldsymbol{x})$ is odd," since the generalization to any 2-state FSA is straightforward.

3.1 Constraint $sum(\boldsymbol{x})$ is odd

Denote by O and $\mathcal{E}$ the sets of all odd and even numbers, respectively. We are first looking at the constraint $sum(\boldsymbol{x}) \in O$. Our first step is to design an FSA, $\mathcal{F}_o$, for accepting I-binary strings with odd number of ones, shown on Figure 2. We have two states denoted by $-\mathbf{1}$ and $\mathbf{1}$, which we call *internal*, and we explicitly add a transition σ called *start* from S to the start state $-\mathbf{1}$ and a transition φ called *final* from the final state $F = \mathbf{1}$ to itself on an "end-of-input" symbol $\varnothing$, as the last two transitions will be used to define two of the tiles. Note that states $-\mathbf{1}$ and $\mathbf{1}$ correspond to partial sums being even or odd, respectively. As an example, on string $1(-1)(-1)(-1)11$, $\mathcal{F}_o$ uses transition σ to start at state $-\mathbf{1}$, on last bit 1 (we read the string starting from the end) uses transition β to go to state $\mathbf{1}$, on next symbol 1 uses transition γ to go back to state $-\mathbf{1}$, on the next three bits -1 uses transition α to stay in the same state, and on the last symbol 1 uses transition β to go to state $\mathbf{1}$, which is a final state, and hence the string is accepted, implying it has an odd number occurrences of symbol 1. We will next describe our implementation of steps (i)–(v) for $\mathcal{F}_o$.

(i) Identifying the gadgets. The gadgets are defined using the types of transitions (q_1, x, q_2) they represent, i.e., start, internal, or final, and by the types and positions of their variables. Accordingly, we define three types of gadgets: *start gadget*, if $q_1 = S$, *internal gadget*, if $q_1 \neq S$, $q_2 \neq F$, and *final gadget*, if $q_2 = F$.

The structure of an internal type gadget is illustrated on Figure 3. Its variables are divided into three types: *program* variables, which correspond to variables x_i of the input constraint, *interface* variables, on the endpoints of the active edges between different gadgets (active edges are ones that have nonzero weight), and *hidden* variables that would enable us, by providing more degrees of freedom, to ensure required gadget properties are satisfied. The logical structure of the internal gadget is shown on Figure 3 (a) and the physical implementation of the gadget on a cell of CG is shown on Figure 3 (b). The start and the final gadget each have

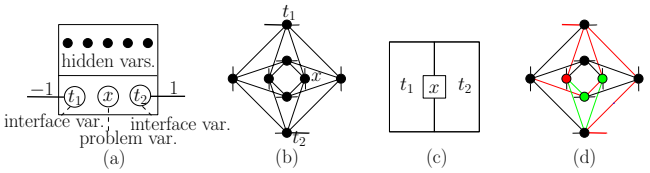

Figure 3: Illustration to the implementation of step (i) for $\mathcal{F}_o$. (a) Logical structure of an internal gadget. (b) Mapping the variables of the gadget on a unit cell of C. Short lines indicate couplers connecting to neighboring cells and longer lines indicate active couplers/nonzero coefficients. (c) Graphical representation of the gadget. (d) The coefficients of the gadget computed in Step (iv), where red, green, and black colors encode values 1, -1, and 0, respectively.

only one interface variable, corresponding to t_1 on Figure 3 (b), and no problem variables.

In order to be able to chain gadgets to one another, for each of the gadgets described so far we need another geometrically symmetrical version, where t_1 is on the bottom connecting to the previous gadget and t_2 on the top connecting to the next one. But since a gadget can be converted to its symmetrical version by just reordering the variables, we omit the symmetrical versions.

(ii) Identifying the good tiles. We define a good tile for each transition of $\mathcal{F}_o$. Hence, we have four good *internal tiles* corresponding to transitions α, β, γ, and δ. Figure 4 (i) specifies the pattern that is used to map a transition to a tile. Specifically, we use the rule:

2-state FSA transformation: *If an internal transition goes from state q_1 to state q_2 on input x, its corresponding good internal tile is defined as $[\overline{q_1}, x, q_2]$ (Figure 4(i)). A start transition from S to state q and a final transition from state q to the final state,* 1*, are transformed into a start and final tiles $[q]$ and $[\overline{q}]$, respectively.*

Here $\overline{q_1}$ denotes the complement of q_1; i.e., $\overline{1} = -1$ and $\overline{-1} = 1$. The states $\overline{q_1}, q_2$ of tile $[\overline{q_1}, x, q_2]$ are mapped to the interface variables, i.e., $t_1 = \overline{q_1}$ and $t_2 = q_2$ (see Figure 3). It is more convenient to encode states by colors, specifically, we will use green for -1 and red for 1. Figure 4 (i1)–(i4) shows the internal tiles corresponding to $\mathcal{F}_o$, i.e., to transitions α , β, γ, δ. Similarly, the patterns for start tiles and final tiles and the corresponding good tiles are shown on Figure 4 (s), (s1), (f), and (f1).

After we defined the set of good tiles, we next define the set of good pairs. For a pair $p = (\tau_1, \tau_2)$ of consecutive good tiles in left-to-right order, p is *good* if the right half of τ_1 and left half of τ_2 are in different colors, i.e., in this case, if the t_2 variable of τ_1 and the t_1 variable of τ_2 are different, and the pair is *bad* otherwise. The idea is that, when the colors are different, a variable in t_1 and a variable in t_2 joined by an edge with maximal allowed weight w will take different values, one of them +1, and the other -1, and hence the term wt_1t_2 will add $-w$ to the value of the Ising problem, i.e., decrease it by w. If the colors are the same, then the same quadratic term wt_1t_2 will increase the value of the Ising by w.

(iii) Specifying tile properties that guarantee gap γ. As mentioned above, our goal is to construct an implementation of the constraint such that the gap, or the difference between the values of the Ising problem for feasible and infeasible solutions, is as large as possible. In our construction, this gap γ will be the minimum of two values: **(a)** the gap γ_{tile} between the good and the bad tiles,

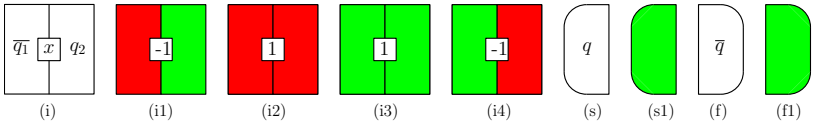

Figure 4: Defining the good tiles in Step (ii) using an FSA representation of the constraint. (i) Pattern for transforming an internal transition (q_1, x, q_2) into a good tile specification, where $\overline{q_1} = t_1$, $q_2 = t_2$. (i1), (i2), (i3), (i4) Good tiles for the FSA from Figure 2 corresponding to internal transitions $\alpha, \beta, \gamma, \delta$, respectively. (s) Pattern for a start transition (S, q). (s1) The good tile corresponding to σ. (f) Pattern for a final transition (q, F). (f1) The good tile corresponding to φ.

and **(b)** the gap γ_{pair} between good and bad pairs. Estimating the gap γ_{pair} is easy. We saw in (ii) that a good pair decreases the value of the Ising problem by w, while a bad pair increases it by w. Since the maximum value of w allowed by the hardware is 1, then the gap between a good and a bad pair is $\gamma_{pair} = 2w = 2$. For finding coefficients of the Ising program that maximize the gap γ_{tile}, we have to solve the optimization problem described next.

(iv) Formulating and solving the optimization problem. For each gadget type, we need to determine an Ising problem defined on a subset of CG, which, in the case of a 2-state FSA, will be a single unit cell of CG (but for larger FSA we may need more than one cell). Then we need to encode the condition that there is a gap γ_{tile} between the good and the bad tiles for that gadget as an optimization problem. We have three gadgets, so we need to solve three optimization problems, but for the start and final types the solutions are trivial to find. For the start gadget, we want the interface variable t to be -1 (for green color) in an optimal solution, so we set the coefficient for the linear term containing t to the maximum allowed value of $w = 2$. All other coefficients of the Ising problem for that gadget are 0. Hence, the gap between the Ising values of bad and good start tiles is $w \cdot 1 - w \cdot (-1) = 2 + 2 = 4$. In the same way we determine the Ising problem for the final gadget.

For the internal gadget, we formulate an optimization problem as follows. We identify each internal tile by the triple (t_1, x, t_2) of its interface variables t_1, t_2 and its program variable x. We denote by $Good = \{(1,-1,-1), (1,1,1), (-1,1,-1), (-1,-1,1)\}$ the set of good tiles from Figure 4, and let $Bad = \{-1,1\}^3 \setminus Good$.

Then the optimization problem is to determine an Ising program Is_σ defined on the variables of a unit cell σ such that

(a) $\forall\, (t_1, x, t_2) \in Good\ \exists \boldsymbol{t} = (t_3, \ldots, t_7)$ s.t. $Is_\sigma(x, t_1, t_2, \boldsymbol{t}) = \mu$,

(b) $\forall\, (t_1, x, t_2) \in Good\ \forall \boldsymbol{t} = (t_3, \ldots, t_7)\ Is_\sigma(x, t_1, t_2, \boldsymbol{t}) \geq \mu$,

(c) $\forall\, (t_1, x, t_2) \in Bad\ \forall \boldsymbol{t} = (t_3, \ldots, t_7)\ Is_\sigma(x, t_1, t_2, \boldsymbol{t}) \geq \mu + \gamma_{tile}$,

and γ_{tile} is maximized. Here μ is a real variable accounting for the lack of constant term in the definition of an Ising form.

The resulting problem can be reformulated as a mixed-integer programming (MIP) problem. Each $t_i \in \{-1, 1\}$ is replaced by its binary equivalent $t'_i = B(t_i) \in \{0, 1\}$. Since Is_σ is a linear function with respects to its coefficient, conditions (b) and (c) can be encoded as linear constraints. For condition (a), we introduce new binary variables $t_i(g)$, $i = 3, \ldots, 7$ for each tile $g = (t_1, x, t_2) \in Good$, such that $Is_\sigma(x, t_1, t_2, \boldsymbol{t}) = \mu$ for $\boldsymbol{t} = (t_3(g), \ldots, t_7(g))$.

We note that the corresponding mixed integer program is not linear yet but cubic, due to terms of the type $\alpha t' t''$, where α is a coefficient of Is_σ (a variable for the optimization) and t' and t'' are among the binary variables $t_i(g)$ defined above. Such terms can be linearized following standard techniques, e.g., see [10].

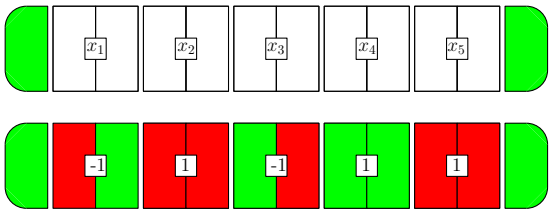

Figure 5: Top: arrangement of gadgets for the constraint $sum(\boldsymbol{x}) \in O$. Bottom: illustration of a good tiling for the constraint.

Because of its small size, the resulting optimization problem (as well the optimization problems discussed later in this paper) can be solved in a fraction of a second using modern MIP solvers such as Gurobi [4]. From the solution, we get the coefficients of Is_σ.

The solution is illustrated on Figure 3 with coefficients' values (weights on qubits and couplers) encoded by colors. Although the nonzero coefficients in this case are in $\{-1, 1\}$, this is just a coincidence. In general, they can be any numbers in the interval $[-1, 1]$ for the couplers (quadratic terms) and in $[-2, 2]$ for the qubits (linear terms).

(v) Putting the gadgets together to form an Ising problem Is for the constraint. The gadgets are arranged in a row with a start gadget first, followed by n internal gadgets, and ending with a final gadget, with the corresponding interface variables in neighboring gadgets connected by an edge with weight 1. Figure 5 illustrates an arrangement for $n = 5$ and a good tiling for $\boldsymbol{x} = \{-1, 1, -1, 1, 1\}$.

One can prove the following statement for any 2-state FSA.

Lemma 3.1. *If Is is constructed from a 2-state FSA following steps (i)-(v) (Section 2.2) and the 2-state FSA Transformation rule, then it implements its corresponding constraint with gap $\gamma = \min\{2, \gamma_{tile}\}$.*

The idea of the proof is that, if a string $\boldsymbol{x}$ accepted by the FSA generates a sequence of states $q_0, \ldots, q_{n+1}$, then the corresponding sequence of tiles $[q_0], [\overline{q_0}, x_1, q_1], [\overline{q_1}, x_2, q_2], \ldots, [\overline{q_{n+1}}]$ has all tiles and pairs good (by the definition of good tiles and since the adjacent colors for the first pair are q_0 and $\overline{q_0}$, for the second pair are q_1 and $\overline{q_1}$, etc.

3.2 Constraint $sum(\boldsymbol{x})$ is even

The above procedure is generalized to any 2-state FSA in a straightforward way. For the $sum(n) \in \mathcal{E}$ constraint, the final state of $\mathcal{F}_o$ should be changed to -1 and, therefore, the color of the right boundary gadget on Figure 5 should be changed to red.

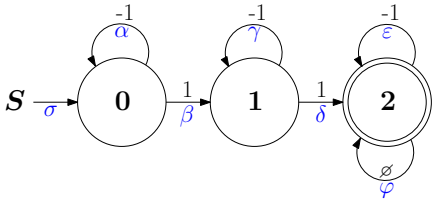

Figure 6: An automaton accepting all binary strings with exactly two 1s. The meaning of the symbols is the same as in Figure 2.

4 IMPLEMENTING CONSTRAINTS REPRESENTED BY LARGER FSAS

When the number of states of the FSA is greater than two, we can apply the same procedure described above, but will need more than two interface variables and may need more than one unit cell per gadget. Next we look at two specific examples.

4.1 Implementing a constraint represented by a 3-state FSA

Applying the method of the previous section to an FSA with more than two states is straightforward as long as a unit cell has enough many qubits and couplers (or, equivalently, enough hidden variables) to allow the optimization problem to have a solution with gap $\gamma_{tile} > 0$. Unfortunately, the Chimera graph unit cells do not allow much larger FSAs to be implemented, but, as we show next, one can still do it for FSAs with 3 states. As an example, we will apply our methodology to the constraint $sum(\boldsymbol{x}) = 2$.

An FSA with 3 states corresponding to that constraint is illustrated on Figure 6. First we need to define the gadgets. As in the previous case, we have start, internal, and final gadgets, and we next need to specify the precise positions for the interface and program variables. Figure 4 (i) shows the structure of an internal gadget. Since the states of the FSA are three, one bit on a side is not enough to encode all three possible states (by the values of the interface variables) as in the previous section, so we will need a *pair* of numbers in $\{-1, 1\}^2$ to encode the states. Specifically, we encode the states **0**, **1**, and **2** by the pairs $(-1, 1)$, $(-1, -1)$, and $(1, -1)$, hence we need to use two interface variables on each side of an internal gadget, see Figure 7 (g1), (g2). The complements of the states will be encoded as $(1, -1)$, $(1, 1)$, and $(-1, 1)$ for $\overline{\mathbf{0}}$, $\overline{\mathbf{1}}$, and $\overline{\mathbf{2}}$, respectively.

For a tile τ, we will denote by $t_1(\tau), \ldots, t_4(\tau)$ the corresponding interface variables. We use colors in the illustration of a tile, green for -1 and red for $+1$, for its four corners, corresponding to $t_1(\tau), \ldots, t_4(\tau)$.

In order to determine the set of good tiles, we use the patterns given in Figure 4 (i), (s), and (f), and get the set of all good tiles as illustrated on Figure 7 (i1)–(i5), (s1), and (f1). A pair of adjacent tiles is good, again, if the touching sides are in different colors. More formally, a pair (τ_1, τ_2) of adjacent tiles is good if $t_3(\tau_1) = -t_1(\tau_2)$ and $t_4(\tau_1) = -t_2(\tau_2)$. In such a case, and assuming the weights on the edges joining τ_1 and τ_2 are set to the maximum admissible value of 1, the contribution $val(\tau_1, \tau_2)$ of the edges between τ_1 and τ_2 (i.e., of the terms containing products $t_3(\tau_1)t_1(\tau_2)$ and $t_4(\tau_1)t_2(\tau_2)$) to the final Ising function is -2. If the pair is not good, then either

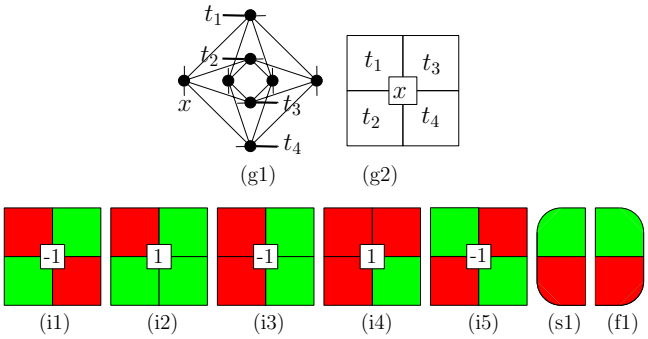

Figure 7: The placement of the program and interface variables of the internal gadget are shown in (g1) and their graphical representation are shown in (g2). The set of all good internal tiles for the FSA on Figure 6 corresponding to transitions $\alpha, \ldots, \varepsilon$ are shown in (i1)–(i5). In (s1) and (f1) we show the good start and final tiles.

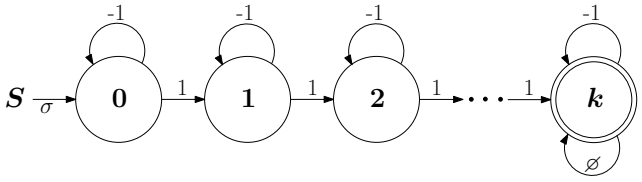

Figure 8: An FSA for constraint $sum(\boldsymbol{x}) = k$.

$t_3(\tau_1) = t_1(\tau_2)$ or $t_4(\tau_1) = t_2(\tau_2)$, and thus $val(\tau_1, \tau_2)$ will either 0 or 2. Hence the pair gap is $\gamma_{pair} = \min\{0, 2\} - (-2) = 2$.

For finding the gap γ_{tile}, we solve the corresponding optimization problem, which has solution $\gamma_{tile} = 2$. Hence, the constraint $sum(\boldsymbol{x}) = 2$ can be implemented with gap $\min\{\gamma_{tile}, \gamma_{pair}\} = 2$ using $n + 2$ tiles. This is an improvement over the previous best solution from [11], which uses $4(n + 2)$ tiles.

4.2 Implementing the constraint $sum(\boldsymbol{x}) = k$

For FSAs with more than four states, we need more than four interface variables and hence more than one unit cell per gadget. We will show how our approach can be applied to a specific constraint, $sum(\boldsymbol{x}) = k$ for an arbitrary k. An FSA for that constraint is shown on Figure 8.

As in the previous sections, first we need to define the gadgets. The automaton has k states, which require $lk = \lceil \log(k+1) \rceil$ bits (and hence lk interface variables) for its representation. We are going to encode, for $i = 0, \ldots, k$, the state $\boldsymbol{i}$ by its lk-bit I-binary (with bits in $\{-1, 1\}$) representation, and the state $\overline{\boldsymbol{i}}$ by the lk-bit I-binary representation of $2^{lk} - i - 1$ (i.e, flipping all bits of $\boldsymbol{i}$). We are going to use 2 interface variables per cell, one of them to connect to the previous and another to connect to the next cell in the arrangement. Hence, we design the internal gadget to consist of a row of $lk + 1$ unit cells of CG as illustrated on Figure 9 (i). The gadget contains $2lk$ interface variables, lk on top and lk on bottom, and one problem variable in a separate cell. (For efficiency, one can fit that variable into some of the other unit cells, but we prefer to keep it separate to simplify presentation.)

Next step is to determine the set of good tiles and we use the method from the previous section. Figure 9 (i1) shows the pattern for good tiles (the same as Figure 4 (i) except for the orientation),

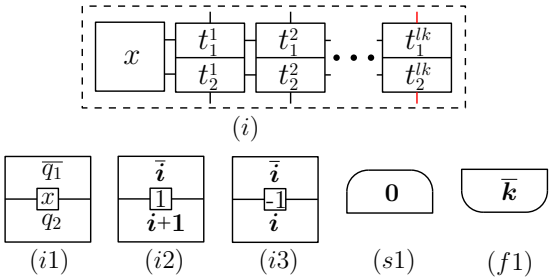

Figure 9: (i) Structure of an internal gadget consisting of $lk+1$ unit cells for the automaton from Figure 8; (i1) Pattern for constructing good internal tiles ; (i2), (i3) The set of good internal tiles for the FSA from Figure 8; (s1), (f1) The good start and final tiles.

and (i2), (i3) show all good internal tiles, based on the FSA from Figure 8, where $i \in \{0, \ldots, k-1\}$ for (i2) and $i \in \{0, \ldots, k\}$ for (i3). The start and final gadgets have structure similar to the internal gadget; the only differences are that they have no x variable (and no x cell) and they interface/connect with only one other cell. An illustration of the start and final good tiles are shown on Figure 9 (s1) and (f1).

Estimating the gap and designing an Ising function for the start and final gadgets is easy. For the start gadget we set all weights of the downward-facing interface variables to +2 (the maximum allowed weight) thereby forcing them to favor values -1 in an optimal solution. For the final gadget, if the i-th bit of $\boldsymbol{k}$ is b_i, then the i-th bit of $\overline{\boldsymbol{k}}$ is $-b_i$, and we set the weight of the i-th upward facing interface variable to +2, if $b_i = 1$ and to -2, if $b_i = -1$. All other weights in the gadget are set to 0.

However, designing directly an Ising function for the internal gadget is more complex than in the previous cases (where we were dealing with a single unit cell) because of its size. The objective is to ensure that there is as large as possible gap $\gamma_{tile} > 0$ between the Ising values for good and bad tiles, and, as we discussed in the introduction, this is a computationally hard optimization problem for $lk > 2$. On the other hand, this task, finding an Ising problem with a big gap between good and bad tiles, is of the same type as the original one, i.e., to find an Ising problem with a big gap between feasible and infeasible solutions. The only difference is that, before, the feasible set was defined as a set of strings $\boldsymbol{x}$, while now it is defined as a set of triplets $[\overline{q_1}, x, q_2]$, where q_1 and q_2 are integers and $x \in \{-1, 1\}$. For the latter type of feasible set, we need to use a representation as an FSA of Mealy type (an FSA with output). These FSAs work in a similar way as the standard ones, but don't have a final state, and, instead, at each transition they generate an output symbol specific for the transition. We want to construct an automaton that on input x and q_1 outputs q_2, for all good tiles $[\overline{q_1}, x, q_2]$. We observe that a tile $[\overline{q_1}, x, q_2]$ is good if and only if $q_2 = q_1 + B(x)$, $q_1 \in \{0, \ldots, k - B(x)\}$ (recall that $B(x)$ maps $\{-1, 1\}$ to $\{0, 1\}$).

Hence, we design an FSA with output that, starting at state $x \in \{-1, 1\}$ and given input h in I-binary, outputs $h + B(x)$ in I-binary. Such an FSA is shown on Figure 10. State $\mathbf{-1}$ encodes that there is no carry and state **1** encode a carry at the corresponding position. For example, for $x = 1$ and $q_1 = 5$, q_1 in I-binary will be

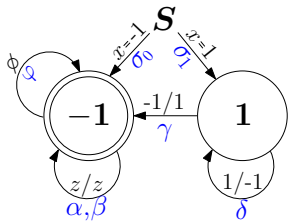

Figure 10: A Mealy FSA for binary addition. Labels on top of an internal transition denote a pair of input/output symbols. z/z is short for the two transitions -1/-1 (α) and 1/1 (β).

1(−1)1. The automaton from Figure 10 will use transition σ_1 on x to go to state **1**, then on the last bit 1 of q_1 will use transition δ to stay in the same state and output symbol -1, on the next bit -1 use transition γ to go to state $\mathbf{-1}$ and output symbol 1, and finally on the next bit 1 use transition β to stay in the same state and output 1. The output string is 11(−1), which is in I-binary $q_2 = 6 = q_1 + B(x)$.

We construct the set of the good internal tiles using the pattern shown on Figure 11 (i), which uses only a single unit cell of *CG*. The set of the good internal tiles generated using the pattern and the states of the FSA from Figure 8 are shown on Figure 11 (i1)-(i4). The patterns and the good start and final tiles are shown on Figure 11 (s), (s1), (s2), (f), (f1).

As an example, Figure 12 (i) shows the arrangement of gadgets for the constraint $sum_4(\boldsymbol{x}) = 2$. The internal gadgets form an $n \times lk = 4 \times 2$ array of unit cells. The left-boundary tiles encode the vector $\boldsymbol{x}$, the bottom-boundary tiles encode the value $k = 2$ (2 is 10 in binary; its complement is 01, but our most significant bit is on the right, so the final is 10, or red-green), and the top and right boundaries are in green and red, respectively.

Figure 12 (ii) shows a good tiling for that constraint. The vector $\boldsymbol{x}$, represented on the left-boundary tiles, is 1(−1)1(−1), hence $sum(\boldsymbol{x}) = 2$ and $\boldsymbol{x}$ satisfies the constraint. Each row corresponds to a transition of the FSA on Figure 8. The top row (green-green) encodes the transition σ from the start state S to state **0**, or the start tile [0], where 0 is encoded in I-binary as (−1)(−1), or green-green; the second row corresponds to the transition from state **0** to state **1** on symbol 1, i.e., the good tile $[0, 1, \overline{1}]$; the next row corresponds to the transition from state **1** to state **1** on symbol 0, i.e., the good tile $[1, 0, \overline{1}]$, etc. The bottom row (red-green) encode the good final tile $\overline{2}$ (or the value of parameter $k = 2$).

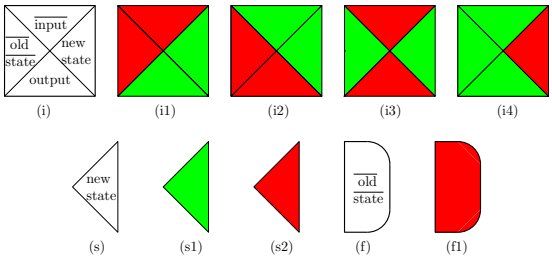

Figure 11: (i) Pattern for generating tiles for a Mealy FSA. (i1), (i2), (i3), (i4) Good internal tiles corresponding to transitions α , β, γ, δ of the FSA from Figure 10. (s),(s1), (s2) Pattern and good start tiles corresponding to transitions σ_0 and σ_1, respectively. (f), (f1) Pattern and good final tile corresponding to transition φ.

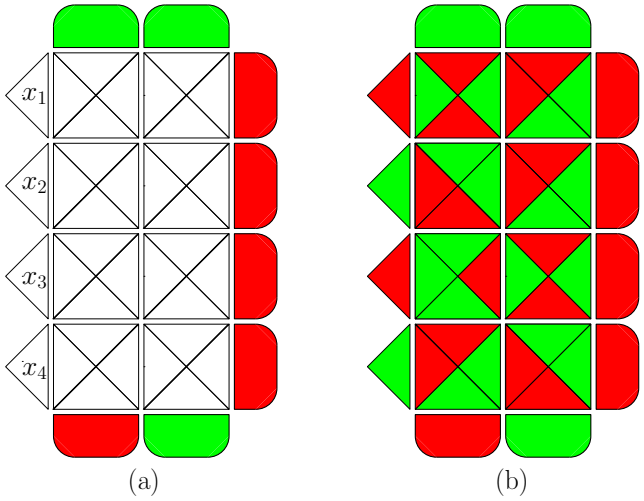

Figure 12: (i) Arrangement of gadgets for the constraint $sum_4(\boldsymbol{x}) = 2$. (ii) Example of a good tiling for that constraint.

It only remains to determine the gap for the presented implementation. As discussed above, the gap between a good and a bad tile pair is $\gamma_{pair} = 2$. In order to determine the gap γ_{tile} between good and bad tiles, we solve the corresponding optimization problem for the good internal tiles from Figure 11, which is computationally easy as the corresponding gadget is on a single unit cell, and find $\gamma_{tile} = 2$. Hence, the gap between feasible and infeasible solutions is $\gamma = 2$.

5 CONCLUSION

We develop an approach for using an FSA representation of a constraint of an optimization problem to implement it as a penalty using a smaller number of quadratic terms and a larger gap, thereby increasing the size of the problems that can be solved on quantum annealers such as D-Wave and in creasing the accuracy of the solutions. We applied it to several common constraints and to the Chimera graph topology used by the current D-Wave 2000Q systems. Our approach is very general and more efficient than the existing ones. Specifically, it reduces the number of cells for implementing the constraint $\sum x_i = 2$ from the best previously known $4(n+2)$ to $n+2$, and for the general constraint $\sum x_i = k$ from the best previously known $O(nk)$ [11] to $O(n \log k)$. Our results suggests, in particular, that any language accepted by a 2-state FSA could be implemented as a penalty constraint using n unit cells as in Section 3, if the optimization problem that determines the coefficients has a solution for a $\gamma > 0$.

ACKNOWLEDGMENTS

Research presented in this article was supported by the Laboratory Directed Research and Development program of Los Alamos National Laboratory under project numbers 20180267ER and 20190065DR.

REFERENCES

[1] Z. Bian, F. Chudak, R. Israel, B. Lackey, W. Macready, and A. Roy. 2014. Discrete optimization using quantum annealing on sparse Ising models. *Frontiers in Physics* 2 (2014), 56. DOI: http://dx.doi.org/10.3389/fphy.2014.00056

[2] P.I. Bunyk, E.M. Hoskinson, M.W. Johnson, E. Tolkacheva, F. Altomare, A.J. Berkley, R. Harris, J.P. Hilton, T. Lanting, A.J. Przybysz, and J. Whittaker. 2014. Architectural Considerations in the Design of a Superconducting Quantum Annealing Processor. *IEEE Trans on Appl Superconductivity* 24, 4 (2014), 1–10.

[3] Guillaume Chapuis, Hristo Djidjev, Georg Hahn, and Guillaume Rizk. 2018. Finding Maximum Cliques on the D-Wave Quantum Annealer. *Journal of Signal Processing Systems* 91 (5 2018), 363–377. DOI: http://dx.doi.org/10.1007/s11265-018-1357-8

[4] Gurobi Optimization, Inc. 2015. Gurobi Optimizer Reference Manual. (2015). http://www.gurobi.com

[5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[6] M. Johnson, M. Amin, S. Gildert, and *et al.* 2011. Quantum annealing with manufactured spins. *Nature* 473 (2011), 194–198.

[7] Gary Kochenberger, Jin-Kao Hao, Fred Glover, Mark Lewis, Zhipeng Lü, Haibo Wang, and Yang Wang. 2014. The Unconstrained Binary Quadratic Programming Problem: A Survey. *Journal of Combinatorial Optimization* 28, 1 (July 2014), 58–81. DOI: http://dx.doi.org/10.1007/s10878-014-9734-0

[8] A. Lucas. 2014. Ising formulations of many NP problems. *Frontiers in Physics* 2, 5 (2014), 1–27.

[9] Elijah Pelofske, Georg Hahn, and Hristo Djidjev. 2019. Solving large minimum vertex cover problems on a quantum annealer. In *Proceedings of the 16th ACM International Conference on Computing Frontiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019*. ACM, 76–84. DOI: http://dx.doi.org/10.1145/3310273.3321562

[10] Ramteen Sioshansi and Antonio Conejo. 2017. *Optimization in Engineering: Models and Algorithms*. Springer.

[11] Tomas Vyskocil and Hristo Djidjev. 2019. Embedding Equality Constraints of Optimization Problems into a Quantum Annealer. *Algorithms* 12, 4 (Apr 2019), 77. DOI: http://dx.doi.org/10.3390/a12040077

[12] Tomas Vyskocil, Scott Pakin, and Hristo Djidjev. 2019. Embedding inequality constraints for quantum annealing optimization. In *Quantum Technology and Optimization Problems (QTOP)*.