Computing Gadgets

Josep Alòs
Logic & Optimization Group (LOG), University of Lleida, Spain

Carlos Ansótegui
Logic & Optimization Group (LOG), University of Lleida, Spain

Supratik Chakraborty
Department of Computer Science and Engineering, IIT Bombay, India

Eduard Torres
Logic & Optimization Group (LOG), University of Lleida, Spain

Abstract

Reductions and gadgets are key tools in computational complexity, with gadgets serving as the building blocks of reductions. Despite their importance, gadget construction has traditionally been ad hoc. We present a method to compute gadgets automatically, enabling systematic reductions and offering insights for both theory and practical solver design.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Gadgets, Satisfiability, Maximum Satisfiability, Mixed Integer Programming

1 Introduction

Reductions between constraint formalisms constitute a fundamental tool in theoretical computer science and combinatorial optimization. In particular, a *reduction* is a transformation from one problem to another, typically employed in computational complexity theory to compare their relative difficulty. If a problem A can be reduced to a problem B (for instance, reducing 3-SAT to 3-colorability problem), this means that any solution procedure for B can be used to solve A. Reductions are primarily used to establish hardness results, such as NP-completeness.

A *gadget* is a small construction used within a reduction. More precisely, it is a finite combinatorial structure that enables the translation of specific constraints from one optimization problem into corresponding constraints of another optimization problem. In this sense, gadgets are local structures designed to simulate components of the source problem inside the target problem, and they serve as the fundamental building blocks from which reductions are constructed.

Typically, gadgets have played a central role in demonstrating the hardness of optimization problems. They are the core of any reduction between combinatorial problems, and they retain this role in recent results on the non-approximability of optimization problems.

Recently, in [1], new gadgets from SAT to Max2SAT were introduced, understanding by SAT the corresponding optimization problem (MaxSAT) where all clauses have equal weight. In particular, this allowed reductions from SAT to Max2SAT in such a way that some families of SAT instances that necessarily require exponential proofs in Resolution [12] (the underlying proof system in SAT solvers) could be solved in polynomial time by state-of-the-art MaxSAT solvers. In this sense, gadgets were used to enable access to more powerful proof systems in practice. Additionally, in [2], several gadgets to reduce SAT to Max2XOR were introduced, which can be used to translate SAT instances to initial configurations of a quantum annealer.

Regrettably, in general, the gadgets presented in the literature lack systematic and efficient construction methods. As observed by Trevisan et al [14]: “Despite their importance, the construction of gadgets has always been a ‘black art’ with no general methods of construction

known." In this paper, we take a step towards mitigating this problem. Specifically, we present a tool to compute and verify gadgets automatically. While the arity of the constraints we can manage is limited from a practical point of view, our method can still provide useful insights for researchers, who may attempt to generalize these gadgets to arbitrary arities.

2 Preliminaries

We restrict our attention primarily to Boolean constraints, and use 0 and 1 to denote false and true respectively. A k-ary *constraint function* is a Boolean function $f : \{0,1\}^k \to \{0,1\}$. A *constraint family* is a set $\mathcal{F}$ of constraint functions (with possibly distinct arities). A *constraint* over variables $V = \{x_1, \ldots, x_n\}$ and constraint family $\mathcal{F}$, is a pair formed by a k-ary constraint function $f \in \mathcal{F}$ and a k-tuple $\mathcal{X} = (x_{i_1}, \ldots x_{i_k})$ of variables from V. We denote such a constraint by $f(x_{i_1}, \ldots, x_{i_k})$, or $f(\mathcal{X})$ for short. A *(weighted) constraint problem* over V and $\mathcal{F}$ is a multiset of (weight, constraint) pairs, where each constraint is over V and $\mathcal{F}$, and all weights are rational numbers. We denote such a problem by $P = \{{}_{(w_1)}\, f_1(x^1_{i_1}, \ldots, x^1_{i_{k_1}}),\ \ldots,\ {}_{(w_m)}\, f_m(x^m_{i_1},\ \ldots,\ x^m_{i_{k_m}})\}$.

An *assignment* is a function $I : \{x_1, \ldots, x_n\} \to \{0,1\}$. We extend I to constraints in the natural way: for any constraint $f(\mathcal{X})$, where $\mathcal{X} = (x_{i_1}, \ldots, x_{i_k})$, define $I\left(f(\mathcal{X})\right) = f\left(I(x_{i_1}), \ldots, I(x_{i_k})\right)$. We say that an assignment I *satisfies* the constraint $f(\mathcal{X})$ if $I\left(f(\mathcal{X})\right) = 1$. The value of an assignment I for a constraint problem $P = \{{}_{(w_i)}\, f_i(\mathcal{X}_i)\}_{i=1,\ldots,m}$, is the sum of weights of the constraints satisfied by the assignment, i.e. $I(P) = \sum_{i=1}^{m} w_i\, I(f_i(\mathcal{X}_i))$. Alternatively, the cost of assignment I is the sum of weights of the constraints falsified by the assignment. An assignment is said to be *optimal* for a constraint problem if it maximizes its value (or equivalently, minimizes its cost).

We use *k-SAT* to refer to the constraint family defined by constraint functions of the form $f(x_1, \ldots, x_k) = l_1 \vee \cdots \vee l_k$, where each l_i is either x_i or $\overline{x}_i$. We use *Max-k-SAT* to refer to the union of weighted constraint families i-SAT, for $0 \leq i \leq k$, and use *MaxSAT* for $\bigcup_{i \geq 0}$*Max-i-SAT*. A *linear constraint* is a constraint of the form $\sum_{i=1}^{n} a_i x_i \diamond b$, where a_i and b are (real) constants, x_i are (real) variables, and $\diamond \in \{<, \leq, >, \geq\}$. A *pseudo-Boolean* (PB) constraint is a linear constraint, where the variables are Boolean. Finally, a *cardinality* (Card) constraint is a PB constraint where all a_i's are equal to 1.

3 Gadgets

We start with the definition of an (α, β)-gadget, as given in [1]. In the following, we abuse notation and use $\mathcal{X}$ for both a tuple of variables, and for its underlying set, depending on the context.

▶ **Definition 1.** *An* (α, β)-gadget *from* $\mathcal{F}_1$ *to* $\mathcal{F}_2$ *is a function that, for any constraint* $f(\mathcal{X})$ *over* $\mathcal{F}_1$ *returns a weighted constraint problem* $P = \{{}_{(w_i)}\, g_i(\mathcal{X} \cup \mathcal{B})\}_{i=1,\ldots,m}$ *over* $\mathcal{F}_2$ *and variables* $\mathcal{X} \cup \mathcal{B}$*, where* $\mathcal{B}$ *are fresh (extension) variables distinct from* $\mathcal{X}$*, such that* $\beta = \sum_{i=1}^{m} w_i$ *and, for any assignment* $I : \{\mathcal{X}\} \to \{0,1\}$*:*

1. *If* $I(f(\mathcal{X})) = 1$*, then for every extension* $I' : \mathcal{X} \cup \mathcal{B} \to \{0,1\}$ *of* I*, we have* $I'(P) \leq \alpha$*. Furthermore, there exists an extension with* $I'(P) = \alpha$*.*
2. *If* $I(f(\mathcal{X})) = 0$*, then for any extension* $I' : \mathcal{X} \cup \mathcal{B} \to \{0,1\}$ *of* I*, we have* $I'(P) \leq \alpha - 1$*. Additionally, if there exists an extension with* $I'(P) = \alpha - 1$*, the gadget is called* strict*.*

We can define an optimal gadget in several ways. In [14] an optimal gadget is a gadget of minimum α*.*

Interpreted as a constraint satisfaction problem, a gadget has the property that its maximum value equals α under satisfying assignments of $f(\mathcal{X})$, while for falsifying assignments, its maximum value is at most $\alpha - 1$.

▶ **Example 1** (Adapted from [6]). Given a 3-SAT clause $(x_1 \vee x_2 \vee x_3)$, the set of Max-2-SAT clauses shown in Figure 1 defines a $(7, 10)$-gadget from 3-SAT clauses to Max-2-SAT, where b_1 is a fresh variable.

$$
\begin{array}{lll}
_{(1)}\, x_1 & _{(1)}\, \overline{x}_1 \vee \overline{x}_2 & _{(1)}\, b_1 \\
_{(1)}\, x_2 & _{(1)}\, \overline{x}_1 \vee \overline{x}_3 & _{(1)}\, \overline{b}_1 \vee x_1 \\
_{(1)}\, x_3 & _{(1)}\, \overline{x}_2 \vee \overline{x}_3 & _{(1)}\, \overline{b}_1 \vee x_2 \\
 & & _{(1)}\, \overline{b}_1 \vee x_3
\end{array}
$$

Figure 1 Gadget for Example 1

Any (α, β)-gadget can be converted into an (α', β') gadget, where $\alpha' > \alpha$, by scaling every entry of the gadget's weight function by α'/α. However, this transformation may not preserve strictness. Hence, stronger gadgets correspond to smaller values of α; in the ideal case, a gadget with $\alpha = 1$ would be optimal.

▶ **Lemma 1** (Lemma 1 in [1]). *The concatenation of a (α_1, β_1)-gadget from $\mathcal{F}_1$ to $\mathcal{F}_2$ and a (α_2, β_2)-gadget from $\mathcal{F}_2$ to $\mathcal{F}_3$ results into a $(\beta_1 (\alpha_2 - 1) + \alpha_1,\ \beta_1\beta_2)$-gadget from $\mathcal{F}_1$ to $\mathcal{F}_3$.*

Composing simpler reductions to obtain more complex reductions is a standard technique used in complexity-theoretic arguments. However, Lemma 1 shows that concatenating gadgets corresponding to the component reductions may not always yield the optimal gadget for the overall reduction. For example, when trying to reduce a k-SAT ($k > 3$) clause to Max-2-SAT, one could first reduce a k-SAT clause to a set of 3-SAT clauses (standard reduction, see Lemma 5 in [1]), and then reduce each 3-SAT clause to Max-2-SAT. However, as shown in [1], the concatenation of the standard k-SAT to 3-SAT gadget with any (α, β)-gadget from 3-SAT to Max2SAT results in a $(\alpha(k-2), \beta(k-2))$-gadget from k-SAT to Max-2-SAT. Notice that the first parameter of the composed gadget is not necessarily small.

▶ **Lemma 2** (Adapted from Lemma 2 in [1]). *Let ϕ be a k-SAT (Max-k-SAT) instance with n clauses, and let (α, β) be a gadget for reducing a k-SAT clause to Max-2-SAT. Denote by $\mathsf{gadget}^{\alpha,\beta}(\phi)$ the Max-2-SAT instance obtained by applying the gadget to every clause in ϕ. Then, ϕ is satisfiable iff the maximum value of $\mathsf{gadget}^{\alpha,\beta}(\phi)$ is $\geq n\alpha$. Equivalently, ϕ is unsatisfiable iff the optimal cost of $\mathsf{gadget}^{\alpha,\beta}(\phi)$ is $> n(\beta - \alpha)$.*

The above lemma indicates that when reducing SAT to Max-2-SAT, one may want to use gadgets with smallest possible $\beta - \alpha$. This would allow the corresponding Max-2-SAT instances to have a lower optimum in terms of cost. In general, we may have different types of desirable constraints when designing gadgets with specific use-cases. For example, we may explicitly want (i) gadgets that are not strict, (ii) gadgets with unit clauses, since this can assist SAT-based MaxSAT solver through Boolean inference, (iii) gadgets with minimum number of extension variables, (iv) gadgets where the gap produced by extension variables for a given set of interpretations is minimal or maximal, etc. This motivates the development of a tool that can automatically search through the space of constrained gadgets, and construct ones that suit our purpose.

4 GadgetoCompiler Tool: A Glimpse

The **GadgetoCompiler** tool is designed to help users create new gadgets with custom constraints. The tool is available at `https://pypi.org/project/gadgetocompiler/` and the documentation at `https://ulog.udl.cat/static/doc/gadgetocompiler/index.html`.

GadgetoCompiler accepts the following as inputs: (i) the constraint function $f(\mathcal{X})$ for which we require a gadget, (ii) the extension Boolean variables $\mathcal{B}$ that the gadget is allowed to use, and (iii) the constraint family $\mathcal{F}$ from which the weighted constraints $_{(w_i)}\, g_i(\mathcal{X} \cup \mathcal{B})$

in the gadget must be selected. As output, we obtain the weighted constraint problem over the selected constraint family produced by the gadget.

GadgetoCompiler implements an abstract class called `Constraint` that provides a convenient way for users to specify the constraint function and the constraint family provided as input ot the gadget. The class `Constraint` implements various useful methods to get the variables in a constraint, to evaluate a constraint under a given assignment, to list the different constraints that can be built from a set of variables, etc.

▶ **Example 2.** We show below a Python snippet showing actual usage of GadgetoCompiler based on Gurobi to generate a gadget from 3-SAT to Max-2-SAT, using one auxiliary variable:

```
1 from gadgetocompiler import Clause, GurobiGadgetoCompiler
2
3 g = GurobiGadgetoCompiler.generate(
4     Clause([1,2,3]),
5     [(Clause, 2), (Clause, 1)],
6     alpha_ub=10, max_auxs=1)
7
8 g.show()
9 g.show_assignment_values()
```

Class `Clause` inherits and implements class `Constraint`. This class allows us to define both the input constraint function to the gadget (line 4) and the constraint family involving constraint functions 2-SAT (clauses of arity 2) and 1-SAT (clauses of arity 1) in line 5. To generate the gadget, we call method `generate` in the compiler, which also takes as argument the maximum number of extension variables, and an upper bound on α. By default, the compiler minimizes α. The tool documentation explains many other parameters that control how to generate the gadget, and how to execute Gurobi. Method `show` reports that the compiler has computed an $(\alpha = 3.5, \beta = 4)$-gadget to Max-2-SAT containing 7 weighted binary clauses. Method `show_assignment_values` shows the value of every possible interpretation on the input constraint and the extension variables in the Max-2-SAT instance. The above code generates the following output:

```
Input: 1.00: 1 2 3
Gadget:
0.50: -1 -2      0.50: -2 4
0.50: 1 2        0.50: 2 -4
0.50: -1 4       1.00: 3 4
0.50: 1 -4
Sat_gadget_bounds: [2.5, 3.5]
Unsat_gadget_bounds: [0.5, 1.5]
Beta 4.0
#Assigment #Min-Loss #Max-Gain #SAT/UNSAT
(0, 0, 0, 0) 1.5 2.5 UNSAT   (1, 0, 0, 0) 1.5 2.5 SAT
(0, 0, 0, 1) 1.5 2.5 UNSAT   (1, 0, 0, 1) 0.5 3.5 SAT
(0, 0, 1, 0) 0.5 3.5 SAT     (1, 0, 1, 0) 0.5 3.5 SAT
(0, 0, 1, 1) 1.5 2.5 SAT     (1, 0, 1, 1) 0.5 3.5 SAT
(0, 1, 0, 0) 1.5 2.5 SAT     (1, 1, 0, 0) 2.5 1.5 SAT
(0, 1, 0, 1) 0.5 3.5 SAT     (1, 1, 0, 1) 0.5 3.5 SAT
(0, 1, 1, 0) 0.5 3.5 SAT     (1, 1, 1, 0) 1.5 2.5 SAT
(0, 1, 1, 1) 0.5 3.5 SAT     (1, 1, 1, 1) 0.5 3.5 SAT
```

5 Computing Gadgets Automatically

In this section, we show how we can compute (α, β)-gadgets automatically by solving a combinatorial optimization problem. We follow the approach in [14, 1] to some extent.

Given a k-ary constraint function $f(\mathcal{X})$ on Boolean variables, our goal is to automatically generate an (α, β)-gadget that minimizes some applicant-dependent objective function defined on α and β. For simplicity, we discuss the case where the objective function is α; the reasoning for other objective functions is analogous.

Let C be the set of all candidate constraints on variables $\mathcal{X} \cup \mathcal{B}$, that belong to the given constraint family $\mathcal{F}$. We formulate our optimization problem as follows:

- **Variables**: For each candidate constraint $c \in C$, we define a rational-valued variable w_c ($w_c \geq 0$) that represents the weight of the candidate in the weighted constraint problem returned by the gadget. If the value of w_c returned by the optimizing solver is > 0, constraint c with weight w_c is included in the constructed gadget; otherwise c is excluded. We also have rational variables α, β for the parameters of the gadget.
- **Objective function:** For simplicity, we consider minimizing α.
- **Constraints:** We consider the following constraints, arising from Definition 1.

(a.1) $\forall \mathcal{X} \left(f(\mathcal{X}) = 1 \implies \left(\forall \mathcal{B} \sum_{c \in C} w_c \cdot c(\mathcal{X} \cup \mathcal{B}) \leq \alpha\right)\right)$

(a.2) $\forall \mathcal{X} \left(f(\mathcal{X}) = 0 \implies \left(\forall \mathcal{B} \sum_{c \in C} w_c \cdot c(\mathcal{X} \cup \mathcal{B}) \leq \alpha - 1\right)\right)$

(b.1) $\forall \mathcal{X} \left(f(\mathcal{X}) = 1 \implies \left(\exists \mathcal{B} \sum_{c \in C} w_c \cdot c(\mathcal{X} \cup \mathcal{B}) = \alpha\right)\right)$

(b.2) $\forall \mathcal{X} \left(f(\mathcal{X}) = 0 \implies \left(\exists \mathcal{B} \sum_{c \in C} w_c \cdot c(\mathcal{X} \cup \mathcal{B}) = \alpha - 1\right)\right)$, if the gadget must be strict.

(c) $0 \leq \alpha \leq \alpha_{ub}$, and $0 \leq w_c \leq \alpha_{ub}$ for all $c \in C$, where α_{ub} is a user-provided upper bound on α.

Note that constraints a.1 and a.2 can be combined into $\forall \mathcal{X} \forall \mathcal{B} \left(\sum_{w \in C} w_c \cdot c(\mathcal{X} \cup \mathcal{B}) \leq \alpha - (1 - f(\mathcal{X}))\right)$. For strict gadgets, constraints b.1 and b.2 can be combined into $\forall \mathcal{X} \exists \mathcal{B} \left(\sum_{c \in C} w_c \cdot c(\mathcal{X} \cup \mathcal{B}) = \alpha - (1 - f(\mathcal{X}))\right)$. Finally, if the objective function to minimize depends on the gadget parameter β, then we must add the constraint $\beta = \sum_{c \in C} w_c$.

To solve the above optimization problem, we can expand the universal and existential quantifiers. This gives a conjunction of linear constraints on variables w_c and α arising from constraints a.1 and a.2. For each of constraints b.1 and b.2, we obtain a conjunction of disjunction of linear constraints.

▶ **Example 3.** Let $f(\mathcal{X}) = (x_1 \vee x_2 \vee x_3)$ be the input constraint function, $\mathcal{B} = \{b\}$ be a singleton set of extension variable, and Max-2-SAT be the target constraint family for our desired gadget. The set C of candidate constraints has 24 ($= \binom{4}{2} \cdot 2^2$) binary clauses and 8 ($= \binom{4}{1} \cdot 2^1$) unary clauses. Let the weight variables w_c associated with constraints $c \in C$ be as shown in Figure 2. Suppose we want to construct a strict gadget that minimizes α, where α_{ub} is given as 7.

$(w_1)\ x_1 \vee x_2$	$(w_9)\ x_1 \vee b$	$(w_{17})\ x_2 \vee b$	$(w_{25})\ x_1$
$(w_2)\ x_1 \vee \overline{x}_2$	$(w_{10})\ x_1 \vee \overline{b}$	$(w_{18})\ x_2 \vee \overline{b}$	$(w_{26})\ x_2$
$(w_3)\ \overline{x}_1 \vee x_2$	$(w_{11})\ \overline{x}_1 \vee b$	$(w_{19})\ \overline{x}_2 \vee b$	$(w_{27})\ x_3$
$(w_4)\ \overline{x}_1 \vee \overline{x}_2$	$(w_{12})\ \overline{x}_1 \vee \overline{b}$	$(w_{20})\ \overline{x}_2 \vee \overline{b}$	$(w_{28})\ b$
$(w_5)\ x_1 \vee x_3$	$(w_{13})\ x_2 \vee x_3$	$(w_{21})\ x_3 \vee b$	$(w_{29})\ \overline{x}_1$
$(w_6)\ x_1 \vee \overline{x}_3$	$(w_{14})\ x_2 \vee \overline{x}_3$	$(w_{22})\ x_3 \vee \overline{b}$	$(w_{30})\ \overline{x}_2$
$(w_7)\ \overline{x}_1 \vee x_3$	$(w_{15})\ \overline{x}_2 \vee x_3$	$(w_{23})\ \overline{x}_3 \vee b$	$(w_{31})\ \overline{x}_3$
$(w_8)\ \overline{x}_1 \vee \overline{x}_3$	$(w_{16})\ \overline{x}_2 \vee \overline{x}_3$	$(w_{24})\ \overline{x}_3 \vee \overline{b}$	$(w_{32})\ \overline{b}$

Figure 2 Candidate constraints and weights

It can be verified that every interpretation of $\{x_1, x_2, x_3, b\}$ satisfies exactly 22 ($= \binom{4}{2} \cdot 3 + 4$) of the above weighted constraints. Therefore, $\sum_{c \in C} w_c \cdot c(\mathcal{X} \cup \mathcal{B})$ is the sum of exactly 22 weights w_c in each linear constraint in our optimization problem. Expanding the quantified constraints in a.1 and a.3, we obtain a conjunction of 16 ($= 2^4$) linear constraints, of which 14 have the comparator "$\leq \alpha$" (corresponding to 7 satisfying interpretations of $f(\mathcal{X})$, extended with two interpretations of $\mathcal{B}$), and 2 have the comparator "$\leq \alpha - 1$". As an example, for $x_1 = x_2 = x_3 = b = 1$ (note $f(\mathcal{X}) = 1$), we get the following constraint from a.1: $\left(\sum_{k=0}^{6} (w_{1+4k} + w_{2+4k} + w_{3+4k})\right) + w_{28} \leq \alpha$.

Similarly, expanding the quantified constraints in b.1 and b.2, we obtain a conjunction of a set of 8 constraints, each of which is a disjunction of 2 linear constraints corresponding to

$b = 1$ and $b = 0$. In 7 of these 8 constraints (corresponding to satisfying interpretations of $f(\mathcal{X})$), the comparator "$= \alpha$" is used in each disjuncted linear constraint, while in the final constraint (corresponding to $x_1 = x_2 = x_3 = 0$), the comparator "$= \alpha - 1$" is used in each disjuncted linear constraint.

Finally, since $\alpha_{ub} = 7$, we add the constraints $0 \leq w_i \leq 7$ for $1 \leq i \leq 32$, and also $0 \leq \alpha \leq 7$ to complete the formulation of our optimization problem.

Mixed Integer Programming (MIP) Approach: The optimization problem described above admits a straightforward Mixed Integer Programming (MIP) formulation. We use continuous or integer variables for w_c, α and β, instead of using rational variables. To encode the disjunction of linear constraints of the form $\sum w_c = \alpha$ (constraints b.1), we first observe that $\sum w_c \leq \alpha$ is already enforced by constraint a.1. Therefore, we only need to encode disjunction of constraints of the form $\sum w_c \geq \alpha$. We use the Big-M technique [10]. For each disjuncted linear constraint of the form $\sum w_c \geq \alpha$, we introduce the constraint $\sum w_c \geq \alpha - M\,y$, where y is a fresh Boolean variable, and $M = \alpha_{ub}$. Finally, the disjunction of constraints of the form $\sum w_c \geq \alpha$ is replaced by a conjunction of the big-M encoded constraints derived above. Assuming $\mathcal{Y}$ denotes the set of all new y variables, we also add the constraint $\sum_{y \in \mathcal{Y}} y \leq |\mathcal{Y}| - 1$ to ensure that at least one y is assigned 0. To solve the MIP model, we use the Gurobi Python API [7] and OR-Tools [11] with the CP-SAT solver.

Pseudo-Boolean (PB) Approach: If we constrain the w_c and α variables to take integer values (this suffices in many cases, as in Example 3), and use a Boolean encoding of these values, the optimization problem formulated above can also be modeled as a pseudo-Boolean optimization problem. Note that the literature contains several techniques to represent bounded integer values using Boolean variables. Our tool currently implements a fairly simple approach, in which, for example, the integer variable α is replaced by $0 \cdot u_0 + 1 \cdot u_1 + \ldots + \alpha_{ub} \cdot u_{\alpha_{ub}}$, where the u_i are fresh Boolean variables. We also add the following constraint: $u_0 + u_1 + \ldots + u_{\alpha_{ub}} = 1$ to ensure that exactly one of the u_i variables is set to 1. Minimizing α can now be achieved by minimizing $\sum_{k=0}^{\alpha_{ub}} k \cdot u_k$.

6 Adding Constraints on the Gadget Structure

Since the gadget construction problem has been formulated as one of constrained optimization, we can easily incorporate additional constraints on the gadget. For example, when constructing gadgets for SAT clauses into Max-2-SAT, we can enforce that the gadget contains unit clauses, since these clauses are potentially useful when running MaxSAT solvers.

6.1 Gadget Refinement

In [2], the authors used the MaxSAT resolution rule [8] to refine gadgets and produce more compact gadgets with smaller α. We can generalize and automate this idea by disallowing inconsistent subsets of constraints in the gadget, that can be *simplified* using the MaxSAT resolution rule, without producing new clauses outside the target family. We illustrate the core idea through an example, for lack of space.

▶ **Example 4.** By applying the MaxSAT resolution rule, we can replace the constraints ${}_{(w_{25})}\, x_1$ and ${}_{(w_{29})}\, \overline{x}_1$ in Example 3 with ${}_{(\min(w_{25},w_{29}))}\, \square$, ${}_{(w_{25}-\min(w_{25},w_{29}))}\, x_1$, and ${}_{(w_{29}-\min(w_{25},w_{29}))}\, \overline{x}_1$. Notice that constraints with $\square$ do not need to be considered in the gadget; moreover, one of the remaining constraints must have weight 0. Therefore, we can state that at most one of ${}_{(w_{25})}\, x_1$ and ${}_{(w_{29})}\, \overline{x}_1$ appear in the gadget. We enforce this by adding $(w_{25} = 0 \vee w_{29} = 0)$ to

the set of constraints. Additionally, we can enforce that ${}_{(w_1)}\, x_1 \vee x_2$, ${}_{(w_2)}\, x_1 \vee \overline{x_2}$, and ${}_{(w_{29})}\, \overline{x}_1$ do not appear together in the gadget adding the constraint $(w_1 = 0 \vee w_2 = 0 \vee w_{29} = 0)$.

6.2 Incremental Gadgets

For gadgets that depend on an arity parameter, such as the arity of SAT clauses, we can consider building a gadget incrementally by adding appropriate constraints. This is particularly useful in practice when looking for patterns in gadgets of small arity that can potentially be generalized to obtain gadgets of larger arity. For example, suppose we want to build a gadget for a clause of arity k. We can first compute a gadget for a clause of arity $k-1$. Then, we can enforce that a certain percentage of constraints appearing (resp. not appearing) in the gadget for arity $k-1$ also appear (resp. do not appear) in the gadget for arity k. A solution of the optimization problem with such added constraints provides a candidate gadget for generalization to higher arities.

7 Verifying Gadgets

In addition to generating gadgets, GadgetoCompiler also provides methods to verify whether a weighted constraint problem corresponds to an (α, β)-gadget for a given input constraint.

The first method (`show_assignment_values` in the tool) performs an exhaustive enumeration and checks that all interpretations of $\mathcal{X} \cup \mathcal{B}$ satisfy constraints a.1, a.2, b.1, and b.2 of the gadget. Owing to its significant complexity, this method should only be used for small problem instances. The second method (function `verify`)[1] proceeds as follows. To verify constraints a.1 and a.2, we can avoid expanding universal variables by checking whether negations of a.1 and a.2 (involving only existentially quantified variables) are both satisfiable. If not, then the gadget satisfies constraints a.1 and a.2; otherwise, the gadget is invalid. To verify constraints b.1 and b.2, we need to reason about universal quantified variables, regardless of whether we negate the constraints or not. In general, to handle universal quantifiers, we can either use a formalism (and solver) that supports them directly, or expand the universal quantifier explicitly. We can also restrict our analysis to a sample of the universal interpretations. While this approach is incomplete, it can be useful during the early stages of gadget development, or as part of a meta-algorithm that guarantees completeness.

In GadgetoCompiler, we sample the universal variables using Covering Arrays (CAs) of strength t [9]. A $CA(N, t, S)$ is a set of N test cases for a set S of variables such that all t-tuples (partial interpretations of length t) are covered in at least one test case. In our context, N is the number of interpretations of quantified Boolean variables that we sample, and S is the set of universally quantified Boolean variables. If constraint b.1 or b.2 evaluates to false with such a sample, the gadget is invalid. Otherwise, we extend the sample by increasing t, or by generating a new sample of strength t that does not contain any of the previous samples (using Covering Arrays with Constraints [3, 4]).

8 Experimental Analysis

Preliminary versions of the GadgetoCompiler tool have been used in some earlier works (computing gadgets from SAT to Max-2-SAT [1], and from SAT to XOR2SAT [2, 13]).

[1] CA tool incorporation not fully automated yet. Ready at rebuttal.

Here, we present additional experimental analysis using the release version of the tool. Our execution environment consists of a cluster with nodes having two AMD 7402 processors, each with 24 cores at 2.8 GHz and 21 GB of RAM per core.

We used Gurobi version 12.0.0. Based on our preliminary experimentation, we used the encoding that fully expands the (a) and (b) constraints, as explained in Section 5. Moreover, we found that running Gurobi using more than one thread does not significantly improve the results. Instead, we found that setting different random seeds had more beneficial impact. Therefore, we ran a parallel portfolio with 10 seeds. We set a time limit of 24 hours ('-' in tables) and memory limit of 20GB per seed.

Method	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
[1]	0.1s	5s	1200s	-	-
GadgetoCompiler (GB)	0.1s	**0.9s**	**6s**	**334s**	-

Table 1 Run-time comparison: GadgetoCompiler vs [1]

We first compare our approach with [1], which computes optimal gadgets from k-SAT to Max-2-SAT minimizing α. Table 1 shows these results. For each k, we set $|\mathcal{B}| = k-2$ and $\alpha_{ub} = 2.5(k-2)+1$, as described in [1]. It is clear from Table 1 that GadgetoCompiler improves the results obtained in [1].

For the next experiments, we set the gadget weights to be integers. For each k, we set $|\mathcal{B}| = k-2$ and $\alpha_{ub} = 3(k-2)+2$, as described in [1]. With this setup, we considered the following use cases that a gadget developer may explore: **MIP-Gurobi**, GadgetoCompiler using Gurobi; **PB-RoundingSAT**, GadgetoCompiler using the PB solver RoundingSAT [5]; **MIP-OR-Tools**, GadgetoCompiler using the ORtools solver; **Refined**, GadgetoCompiler using Gurobi with refinement constraints as explained in Section 6.1; **Incremental**, GadgetoCompiler using Gurobi with an incremental strategy, as described in Section 6.2, keeping 90% of constraints that appear in the gadget for preceding arity, and enforcing that 100% of constraints that do not appear in the gadget for the preceding arity remain absent; **Units**, GadgetoCompiler using Gurobi while forcing unit clauses on the input variables, as described in Section 6; and **Units and** $\beta-\alpha$, which minimizes $\beta-\alpha$ while forcing unit clauses on the input variables. Table 2 shows the runtimes for all these cases. We separate the last two cases (Incremental and Units) because forcing units does not yield optimal gadgets, although they can still be useful in practice depending on the target application. Overall, we notice that **MIP-Gurobi** is the best basic solving approach. We also see that **Refined** scales a bit better. Finally, although we do not get optimal α's both **Incremental** and **Units** allow us to generate a gadget for $k=7$.

Method	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
α	4	6	9	11	-
β	5	7	11	13	-
MIP-Gurobi	0.1s	0.6s	18s	6521s	-
PB-RoundingSAT	0.12s	21s	-	-	-
MIP-OR-Tools	0.7s	188s	-	-	-
Refined	0.1s	0.7s	81s	2181s	-
α (Incremental)	4	7	10	13	15
β (Incremental)	5	8	12	16	28
Incremental	0.1s	0.1s	6s	1491s	45317s
α (Units)	5	8	11	14	17
β (Units)	7	11	15	19	23
Units	0.1	0.5	5s	1166s	-
Units and $\beta-\alpha$	0.1s	0.2s	4s	50s	21663s

Table 2 Runtimes for different gadget computations.

9 Conclusions

We have presented a versatile tool for generating and analyzing weighted constraint gadgets. The tool has been successfully applied in previous works [1, 2, 13] to produce preliminary gadgets, inspiring generalizations for input constraints of any arity. Its flexibility allows the combination of different constraint functions in the weighted problem generated by the gadget and enables its use as a component in meta-algorithms for constructing more complex gadgets. These capabilities make it a valuable resource for both practical gadget design and further research in constraint optimization.

References

1 Carlos Ansótegui and Jordi Levy. Reducing SAT to max2sat. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 1367–1373. ijcai.org, 2021.

2 Carlos Ansótegui and Jordi Levy. Sat, gadgets, max2xor, and quantum annealers. *Quantum Inf. Process.*, 24(10):338, 2025.

3 Carlos Ansótegui, Felip Manyà, Jesus Ojeda, Josep M. Salvia, and Eduard Torres. Incomplete maxsat approaches for combinatorial testing. *J. Heuristics*, 28(4):377–431, 2022. URL: `https://doi.org/10.1007/s10732-022-09495-3`, `doi:10.1007/S10732-022-09495-3`.

4 Carlos Ansótegui and Eduard Torres. Effectively computing high strength mixed covering arrays with constraints. *J. Parallel Distributed Comput.*, 185:104791, 2024. URL: `https://doi.org/10.1016/j.jpdc.2023.104791`, `doi:10.1016/J.JPDC.2023.104791`.

5 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-boolean solving. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1291–1299. ijcai.org, 2018.

6 M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete graph problems. *Theor. Comput. Sci.*, 1(3):237–267, 1976.

7 Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026. URL: `https://www.gurobi.com`.

8 Javier Larrosa and Federico Heras. Resolution in max-sat and its relation to local consistency in weighted csps. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, pages 193–198. Professional Book Center, 2005.

9 Elizabeth Maltais and Lucia Moura. Finding the best cafe is np-hard. In *LATIN 2010: Theoretical Informatics*, pages 356–371, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

10 George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization*. Wiley, New York, 1988.

11 Laurent Perron and Frédéric Didier. Cp-sat. URL: `https://developers.google.com/optimization/cp/cp_solver/`.

12 John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

13 Pol Rodríguez-Farrés, Rocco Ballester, Carlos Ansótegui, Jordi Levy, and Jesús Cerquides. Implementing 3-sat gadgets for quantum annealers with random instances. In Leonardo Franco, Clélia de Mulatier, Maciej Paszynski, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Computational Science - ICCS 2024 - 24th International Conference, Malaga, Spain, July 2-4, 2024, Proceedings, Part VI*, volume 14837 of *Lecture Notes in Computer Science*, pages 277–291. Springer, 2024.

14 Luca Trevisan, Gregory B. Sorkin, Madhu Sudan, and David P. Williamson. Gadgets, approximation, and linear programming. *SIAM J. Comput.*, 29(6):2074–2097, 2000.