A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints

Ignasi Abío[1], Robert Nieuwenhuis[2], Albert Oliveras[2], Enric Rodríguez-Carbonell[2]

[1] Theoretical Computer Science, TU Dresden, Germany
[2] Technical University of Catalonia, Barcelona

Abstract. Adequate encodings for high-level constraints are a key ingredient for the application of SAT technology. In particular, *cardinality constraints* state that at most (at least, or exactly) k out of n propositional variables can be true. They are crucial in many applications. Although sophisticated encodings for cardinality constraints exist, it is well known that for small n and k straightforward encodings without auxiliary variables sometimes behave better, and that the choice of the right trade-off between minimizing either the number of variables or the number of clauses is highly application-dependent. Here we build upon previous work on Cardinality Networks to get the best of several worlds: we develop an arc-consistent encoding that, by recursively decomposing the constraint into smaller ones, allows one to decide which encoding to apply to each sub-constraint. This process minimizes a function $\lambda \cdot \mathit{num\_vars} + \mathit{num\_clauses}$, where λ is a parameter that can be tuned by the user. Our careful experimental evaluation shows that (e.g., for $\lambda = 5$) this new technique produces much smaller encodings in variables *and* clauses, and indeed strongly improves SAT solvers' performance.

1 Introduction

This paper presents a new encoding into SAT of *cardinality constraints*, that is, constraints of the form $x_1 + \cdots + x_n \# k$, where k is a natural number, the x_i are propositional variables, and the relation operator $\#$ belongs to $\{<, >, \leqslant, \geqslant, =\}$. Cardinality constraints are present in many practical SAT applications, such as cumulative scheduling [17] or timetabling [4]. They also arise as components of some SAT-based techniques, e.g., for MaxSAT [11].

Here we are interested in encoding a cardinality constraint C with a clause set S (possibly with auxiliary variables) that is not only equisatisfiable, but also *arc-consistent*: given a partial assignment A, if x_i is true (false) in every extension of A satisfying C, then unit propagating A on S sets x_i to true (false)[3]. Enforcing arc-consistency by unit propagation in this way has of course an important positive impact on the practical efficiency of SAT solvers.

A straightforward encoding of a cardinality constraint $x_1 + \cdots + x_n \leqslant k$ is to state, for each subset Y of $\{x_1, \ldots, x_n\}$ with $|Y| = k + 1$, that at least one variable of Y must be false. This can be done by asserting $\binom{n}{k+1}$ clauses of the form $\overline{x_{i_1}} \vee \ldots \vee \overline{x_{i_{k+1}}}$. This kind of construction frequently works well, although it is of course not reasonable for

[3] Sometimes this notion is called *generalized arc-consistency*.

large n and k, which is our aim in this work. Successively more sophisticated encodings using auxiliary variables have been defined that require fewer clauses (see Section 2). But still, for small n and k the straightforward encodings may behave better in practice. An additional issue is that, for the efficiency of the SAT solver, the choice of the right trade-off between minimizing either the number of auxiliary variables or the number of clauses is highly application-dependent.

Here we build upon and improve previous work on encoding cardinality constraints with Cardinality Networks [2,3], which use $O(n \log^2 k)$ variables and clauses (see Section 3). The idea is to get the best of several worlds: we develop a hybrid arc-consistent encoding that, by recursively decomposing the constraint into smaller ones, allows one to decide whether to apply a recursive (see Section 4) or a direct (see Section 5) encoding to each sub-constraint. This process minimizes a function $\lambda \cdot \mathit{num\_vars} + \mathit{num\_clauses}$, where λ is a parameter that can be tuned by the user (see Section 6). Our experimental evaluation shows that (e.g., for $\lambda = 5$) this new technique produces much smaller encodings in variables *and* clauses, and indeed strongly improves the performance of SAT solvers (see Section 7).

2 Related Work

Because of their practical importance, encodings of cardinality constraints into SAT have been thoroughly studied over the last few years. In this section we review some of the most important works in the literature.

In [20], Warners considered the more general pseudo-Boolean case, where constraints are of the form $a_1x_1 + \ldots + a_nx_n \leq k$, being the a_i's and the k integer coefficients and the x_i's Boolean variables. The encoding is based on using adders for numbers represented in binary. For cardinality constraints the encoding uses $O(n)$ clauses and variables, but does not preserve arc consistency.

Bailleux and Boufkhad presented in [5] an arc-consistent encoding of cardinality constraints that uses $O(n \log n)$ variables and $O(n^2)$ clauses. The encoding consists of a *totalizer* and a *comparator*. The totalizer can be seen as a binary tree, where the leaves are the x_i's variables. Each inner node is labeled with a number s and uses s auxiliary variables to represent, in unary, the sum of the leaves of the corresponding subtree. As for the comparator, it is easily encoded thanks to the unary representation, which also allows handling constraints of the form $k_1 \leq x_1 + \ldots + x_n \leq k_2$ without splitting.

A more applied work is the one of Büttner and Rintanen [19]. Although their main interest was in planning, they suggested two encodings of cardinality constraints. The first one is based on encoding an injective mapping between the true x_i's variables and k elements. It uses $O(nk)$ clauses and variables and is not arc-consistent. The other encoding is a small modification of [5]. Based on the observation that counting up to $k + 1$ suffices, they can reduce the number of variables and clauses used in each node. The resulting encoding requires $O(nk)$ variables and $O(nk^2)$ clauses, which improves on [5] if k is small enough.

In [18], Sinz proposed two different encodings, both based on counters. The first encoding uses a sequential counter where numbers are represented in unary. It needs $O(nk)$ clauses and variables and is arc-consistent. The second one is based on a parallel counter, where numbers are represented in binary. It requires $O(n)$ clauses and variables, but is not arc-consistent.

Another kind of encoding was used in [6], where a BDD-like technique was proposed for pseudo-Boolean constraints. The encoding is arc-consistent, and uses $O(n^2)$ clauses and variables when applied to cardinality constraints. The idea is as follows: given a pseudo-Boolean constraint $a_1x_1 + \ldots + a_nx_n \leq k$, the root of the BDD is labeled with variable $D_{n,k}$, expressing that the sum of the first n terms is at most k. The two corresponding children are $D_{n-1,k}$ and $D_{n-1,k-a_n}$, indicating the two cases that correspond to setting x_n to false and true, respectively. Then the necessary clauses are added to express the relationship between the variables, and trivial cases are treated accordingly.

The same authors presented in [7] a polynomial and arc-consistent encoding of pseudo-Boolean constraints. When restricted to cardinality constraints it is similar to [5], but the latter is better in terms of size.

Yet another approach for encoding cardinality constraints was suggested in [1]. The authors revisit the idea of using totalizers, and realize that totalizers require two parameters: the encoding used (unary or binary) and the way the totalizers are grouped (e.g. $(a + b) + (c + d)$ or $(((a + b) + c) + d)$). A thorough experimentation is performed, to which they add two extra aspects: (*i*) how to order the variables; and (*ii*), the use of encodings in parallel, hoping the SAT solver will focus on the most appropriate one for each problem.

Finally, Eén and Sörensson [10] presented three encodings for pseudo-Boolean constraints. The first encoding is BDD-based, similar to [6]. The second one, based on adder networks, improves that of [20] in that it uses less adders, but is still linear and does not preserve arc consistency. Finally, their third encoding uses *Sorting Networks* [8]. A Sorting Network takes input variables $(x_1 \ldots x_n)$ and returns as outputs $(y_1 \ldots y_n)$ the sorted input values in decreasing order. Hence, an output variable y_k will become true iff there are at least k true input variables, and false iff there are at least $n - k + 1$ false ones. Now, to express $x_1 + \cdots + x_n \geqslant k$, it suffices to add a unit clause y_k; similarly, for $x_1 + \cdots + x_n \leqslant k$ one adds $\overline{y_{k+1}}$, and both are added if the relation is =. This encoding, when restricted to cardinality constraints, preserves arc consistency and requires $O(n \log^2 n)$ clauses and variables. The Cardinality Networks of [2,3] reduce this to $O(n \log^2 k)$, which is important as often $n \gg k$.

A similar approach uses so-called Pairwise Cardinality Networks [9], which are based on Pairwise Sorting Networks [15] instead of Sorting Networks. By means of partial evaluation, this method also achieves $O(n \log^2 k)$ variables and clauses. Finally, we were recently informed that a hybrid approach based on Pairwise Cardinality Networks similar to that presented here was implemented in the BEE system [13]. However, no detailed description or experimental evaluation is available. Moreover, our proposal in this paper is more general, in the sense that it allows the user to tune the parameter λ when minimizing the objective function $\lambda \cdot num\_vars + num\_clauses$.

3 Preliminaries

In this work we describe a method for producing cardinality networks that generalizes the construction of [3]. The core idea of these approaches, which dates back to [8], consists in encoding a circuit that implements mergesort by means of a set of clauses. The most basic components of these circuits are 2-comparators.

A *2-comparator* is a sorting network of size 2, i.e., it has 2 input variables (x_1 and x_2) and 2 output variables (y_1 and y_2) such that y_1 is true if and only if at least one of the input variables is true, and y_2 is true if and only if both two input variables are true. In the following, 2-comparators are denoted by $(y_1, y_2) = \text{2-Comp}(x_1, x_2)$. As pointed out in [3], for encoding $\leqslant$-constraints, only the three clauses on the first row of Fig. 1 are needed to guarantee arc-consistency. The three clauses on the second row suffice for $\geqslant$-constraints and all six must be present when encoding $=$-constraints. Note that the usual polarity argument [16] cannot be directly applied here, as we are interested not only in preserving satisfiability, but also arc-consistency under unit propagation.

$$x_1 \rightarrow y_1,\ \ x_2 \rightarrow y_1,\ \ x_1 \wedge x_2 \rightarrow y_2,$$
$$\overline{x_1} \rightarrow \overline{y_2},\ \ \overline{x_2} \rightarrow \overline{y_2},\ \ \overline{x_1} \wedge \overline{x_2} \rightarrow \overline{y_1}$$

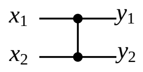


Fig. 1: A 2-comparator: clauses (left) and graphical representation (right).

4 Arbitrary-Sized Recursive Cardinality Networks

In this section we generalize the recursive construction of cardinality networks given in [3] by allowing inputs and outputs of any size, not necessarily a power of two. Not only does this avoid adding dummy variables that are not actually needed (which, as will be seen in Section 7, has an impact on performance on its own), but also becomes useful when combining with the direct (non-recursive) constructions of Section 5.

In what follows, we denote by $\lfloor r \rfloor$ and $\lceil r \rceil$ the floor and ceiling functions respectively. Moreover, for simplicity, we will assume that the constraint to be encoded is a $\leqslant$-constraint. However, similar constructions for the other constraints can be devised.

4.1 Merge Networks

A *merge network* takes as input two (decreasingly) ordered sets of sizes a and b and produces a (decreasingly) ordered set of size $a + b$. We can build a merge network with inputs $(x_1, \ldots, x_a)$ and $(x'_1, \ldots, x'_b)$ in a recursive way as follows[4]:

– If $a = b = 1$, a merge network is a 2-comparator:

$$\text{Merge}(x_1; x'_1) := \text{2-Comp}(x_1, x'_1).$$

– If $a = 0$, a merge network returns the second input:

$$\text{Merge}(; x'_1, x'_2, \ldots, x'_b) := (x'_1, x'_2, \ldots, x'_b).$$

[4] Notice we use the notation $\text{Merge}(X; X')$ instead of $\text{Merge}((X), (X'))$ for simplicity.

– If a and b are even, $a > 0$, $b > 0$ and either $a > 1$ or $b > 1$, let us define

$$\begin{aligned}
\begin{matrix}(z_1, z_3, \dots, z_{a-3}, z_{a-1},\\ z_{a+1}, z_{a+3}, \dots, z_{a+b-1})\end{matrix} &= \begin{matrix}\text{Merge}(x_1, x_3, \dots, x_{a-1};\\ x'_1, x'_3, \dots, x'_{b-1}),\end{matrix}\\
\begin{matrix}(z_2, z_4, \dots, z_{a-2}, z_a,\\ z_{a+2}, z_{a+4}, \dots, z_{a+b})\end{matrix} &= \begin{matrix}\text{Merge}(x_2, x_4, \dots, x_a;\\ x'_2, x'_4, \dots, x'_b),\end{matrix}\\
(y_2, y_3) &= \text{2-Comp}(z_2, z_3),\\
&\dots\\
(y_{a+b-2}, y_{a+b-1}) &= \text{2-Comp}(z_{a+b-2}, z_{a+b-1}).
\end{aligned}$$

Then,

$$\text{Merge}(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (z_1, y_2, y_3, \dots, y_{a+b-1}, z_{a+b}).$$

– If a is even, b is odd, $a > 0$, $b > 0$ and either $a > 1$ or $b > 1$, let us define

$$\begin{aligned}
\begin{matrix}(z_1, z_3, \dots, z_{a-1},\\ z_{a+1}, z_{a+3}, \dots, z_{a+b})\end{matrix} &= \begin{matrix}\text{Merge}(x_1, x_3, \dots, x_{a-1};\\ x'_1, x'_3, \dots, x'_b),\end{matrix}\\
\begin{matrix}(z_2, z_4, \dots, z_a, z_{a+2},\\ z_{a+4}, \dots, z_{a+b-1})\end{matrix} &= \begin{matrix}\text{Merge}(x_2, x_4, \dots, x_a;\\ x'_2, x'_4, \dots, x'_{b-1}),\end{matrix}\\
(y_2, y_3) &= \text{2-Comp}(z_2, z_3),\\
&\dots\\
(y_{a+b-1}, y_{a+b}) &= \text{2-Comp}(z_{a+b-1}, z_{a+b}).
\end{aligned}$$

Then,

$$\text{Merge}(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (z_1, y_2, y_3, \dots, y_{a+b-1}, y_{a+b}).$$

– If a and b are odd, $a > 0$, $b > 0$ and either $a > 1$ or $b > 1$, let us define

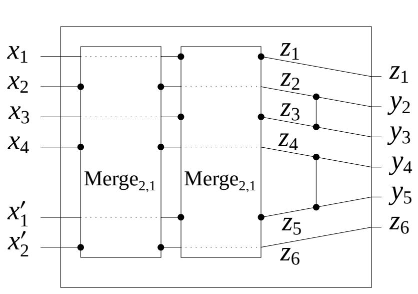
$$\begin{aligned}
\begin{matrix}(z_1, z_3, \dots, z_{a-2}, z_a,\\ z_{a+1}, z_{a+3}, \dots, z_{a+b})\end{matrix} &= \begin{matrix}\text{Merge}(x_1, x_3, \dots, x_a;\\ x'_1, x'_3, \dots, x'_b),\end{matrix}\\
\begin{matrix}(z_2, z_4, \dots, z_{a-3}, z_{a-1},\\ z_{a+2}, z_{a+4}, \dots, z_{a+b-1})\end{matrix} &= \begin{matrix}\text{Merge}(x_2, x_4, \dots, x_{a-3}, x_{a-1};\\ x'_2, x'_4, \dots, x'_{b-1}),\end{matrix}\\
(y_2, y_3) &= \text{2-Comp}(z_2, z_3),\\
&\dots\\
(y_{a+b-2}, y_{a+b-1}) &= \text{2-Comp}(z_{a+b-2}, z_{a+b-1}).
\end{aligned}$$

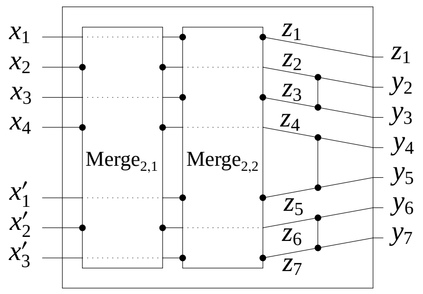
Then,

$$\text{Merge}(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (z_1, y_2, y_3, \dots, y_{a+b-1}, z_{a+b}).$$

– The remaining cases are defined thanks to the symmetry of the merge function, i.e., due to $\text{Merge}(X, X') = \text{Merge}(X', X)$.

The base cases do not require any explanation. As regards the recursive ones, first notice that the set of values $x_1, x_2, \dots, x_a, x'_1, x'_2, \dots, x'_b$ is always preserved. Further, the





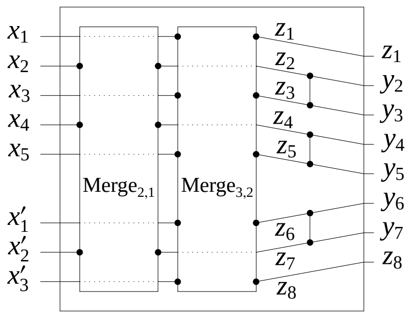


Fig. 2: Different examples of merge networks.

output bits are sorted, as $z_{2i} \geq z_{2(i+1)}$, $z_{2i} \geq z_{2(i+1)+1}$, $z_{2i+1} \geq z_{2(i+1)}$ and $z_{2i+1} \geq z_{2(i+1)+1}$ imply that $\min(z_{2i}, z_{2i+1}) \geq \max(z_{2(i+1)}, z_{2(i+1)+1})$. Figure 2 shows examples of some of these recursive cases.

The number of auxiliary variables and clauses of a merge network defined in this way can be recursively computed. A merge network with inputs of size $(1,1)$ needs 2 variables and 3 clauses. A merge network with inputs of size $(0,b)$ needs no variables and clauses. A merge network with inputs of size (a,b) with $a > 1$ or $b > 1$ needs $V_1 + V_2 + 2\left\lfloor \frac{a+b-1}{2} \right\rfloor$ variables and $C_1 + C_2 + 3\left\lfloor \frac{a+b-1}{2} \right\rfloor$ clauses, where V_1 and C_1 are the number of variables and clauses in a merge network with inputs of size $\left(\left\lceil \frac{a}{2} \right\rceil, \left\lceil \frac{b}{2} \right\rceil\right)$, and V_2, C_2 are idem in a merge network with inputs of size $\left(\left\lfloor \frac{a}{2} \right\rfloor, \left\lfloor \frac{b}{2} \right\rfloor\right)$.

In comparison to [3], in that work it was assumed that $a = b = 2^m$ for some $m \geq 0$. Thanks to this, only one base case ($a = b = 1$) and one recursive case (a, b even) were considered there. All the other cases introduced here are needed for arbitrary a and b.

4.2 Sorting Networks

A *sorting network* takes an input of size n and sorts it. It can be built in a recursive way as follows, using the same strategy as in mergesort:

– If $n = 1$, the output of the sorting network is its input:

$$\text{Sorting}(x_1) := x_1$$

– If $n = 2$, a sorting network is a single merge (i.e., a 2-comparator):

$$\text{Sorting}(x_1, x_2) := \text{Merge}(x_1; x_2).$$

– For $n > 2$, take l with $1 \leqslant l < n$: Let us define

$$\begin{aligned}(z_1, z_2, \ldots, z_l) &= \text{Sorting}(x_1, x_2, \ldots, x_l),\\ (z_{l+1}, z_{l+2}, \ldots, z_n) &= \text{Sorting}(x_{l+1}, x_{l+2}, \ldots, x_n),\\ (y_1, y_2, \ldots, y_n) &= \text{Merge}(z_1, z_2, z_l; z_{l+1}, \ldots, z_n).\end{aligned}$$

Then,

$$\text{Sorting}(x_1, x_2, \ldots, x_n) := (y_1, y_2, \ldots, y_n).$$

Again, the number of auxiliary variables and clauses needed in these networks can be recursively computed. A sorting network of input size 1 needs no variables and clauses. A sorting network of input size 2 needs 2 variables and 3 clauses. A sorting network of input size n composed by a sorting network of size l and a sorting network of size $n - l$ needs $V_1 + V_2 + V_3$ variables and $C_1 + C_2 + C_3$ clauses, where $(V_1, C_1), (V_2, C_2)$ are the number of variables and clauses used in the sorting networks of sizes l and $n - l$, and (V_3, C_3) are the number of variables and clauses needed in the merge network with inputs of sizes $(l, n - l)$.

In comparison to [3], in that work n is assumed to be a power of two. Moreover, in the recursive case l is always chosen to be $n/2$, while here we can build sorting networks of any size, and have the additional freedom of choosing the sizes of the two sorting network components.

4.3 Simplified Merge Networks

A *simplified merge* is a reduced version of a merge, used when we are only interested in some of the outputs, but not all. Recall that we want to encode a constraint of the form $x_1 + \ldots + x_n \leqslant k$, and hence we are only interested in the first $k + 1$ bits of the sorted output. Thus, in a c-simplified merge network, the inputs are two sorted sequences of variables $(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b)$, and the network produces a sorted output of the desired size, c, $(y_1, y_2, \ldots, y_c)$. The network satisfies that y_r is true if there are at least r true inputs. We can build a recursive simplified merge as follows:

– If $a = b = c = 1$, let us add the clauses $x_1 \to y,\ x'_1 \to y$[5]. Then:

$$\text{SMerge}_1(x_1; x'_1) := y.$$

– If $a > c$, we can ignore the last $a - c$ bits of the first input (similarly if $b > c$):

$$\text{SMerge}_c(x_1, x_2, \ldots, x_a; x'_1, \ldots, x'_b) = \text{SMerge}_c(x_1, x_2, \ldots, x_c; x'_1, \ldots, x'_b).$$

– If $a + b \leqslant c$, the simplified merge is a merge:

$$\text{SMerge}_c(x_1, \ldots, x_a; x'_1, \ldots, x'_b) = \text{Merge}(x_1, \ldots, x_a; x'_1, \ldots, x'_b).$$

[5] Notice that these clauses correspond to the bit of the 2-comparator with lower index. Clause $\overline{x_1} \wedge \overline{x_2} \to \overline{y}$ does not need to be included here following the reasoning given in Section 3.

– If $a, b \leqslant c,\ \ a + b > c$ and c is even: Let us define

$$\begin{aligned}(z_1, z_3, \ldots, z_{c+1}) &= \text{SMerge}_{c/2+1}(x_1, x_3, \ldots; x'_1, x'_3, \ldots),\\ (z_2, z_4, \ldots, z_c) &= \text{SMerge}_{c/2}(x_2, x_4, \ldots; x'_2, x'_4, \ldots),\\ (y_2, y_3) &= \text{2-Comp}(z_2, z_3),\\ &\ldots\\ (y_{c-2}, y_{c-1}) &= \text{2-Comp}(z_{c-2}, z_{c-1}).\end{aligned}$$

and add the clauses $z_c \to y_c,\ z_{c+1} \to y_c$. Then,

$$\text{SMerge}_c(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b) := (z_1, y_2, y_3, \ldots, y_c),$$

– If $a, b \leqslant c,\ \ a + b > c$ and $c > 1$ is odd: Let us define

$$\begin{aligned}(z_1, z_3, \ldots, z_c) &= \text{SMerge}_{\frac{c+1}{2}}(x_1, x_3, \ldots; x'_1, x'_3, \ldots),\\ (z_2, z_4, \ldots, z_{c-1}) &= \text{SMerge}_{\frac{c-1}{2}}(x_2, x_4, \ldots; x'_2, x'_4, \ldots),\\ (y_2, y_3) &= \text{2-Comp}(z_2, z_3),\\ &\ldots\\ (y_{c-1}, y_c) &= \text{2-Comp}(z_{c-1}, z_c).\end{aligned}$$

Then,

$$\text{SMerge}_c(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b) := (z_1, y_2, y_3, \ldots, y_c).$$

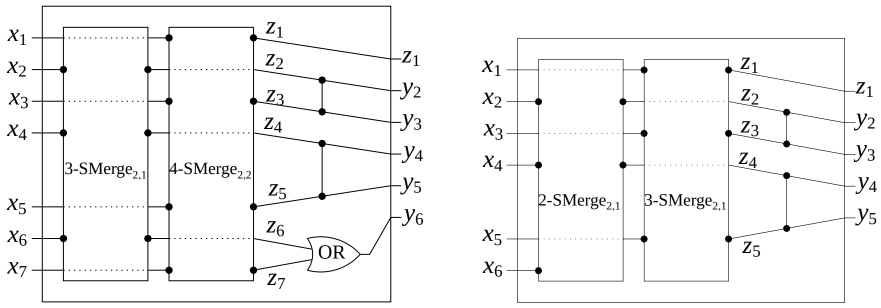


Fig. 3: Two examples of simplified merge networks.

Figure 3 shows two examples of simplified merges: The first one shows a 6-simplified merge with inputs of sizes 3 and 4. The second one corresponds to a 5-simplified merge with inputs of sizes 2 and 4.

We can recursively compute the auxiliary variables and clauses needed in simplified merge networks. In the recursive case, we need $V_1 + V_2 + c - 1$ variables and $C_1 + C_2 + C_3$ clauses, where $(V_1, C_1), (V_2, C_2)$ are the number of clauses and variables needed in simplified merge networks of sizes $\left(\left\lceil \frac{a}{2} \right\rceil, \left\lceil \frac{b}{2} \right\rceil, \left\lfloor \frac{c}{2} \right\rfloor + 1\right), \left(\left\lfloor \frac{a}{2} \right\rfloor, \left\lfloor \frac{b}{2} \right\rfloor, \left\lfloor \frac{c}{2} \right\rfloor\right)$, and

$$C_3 = \begin{cases} \frac{3c-3}{2} & \text{if } c \text{ is odd,}\\ \frac{3c-2}{2} + 2 & \text{if } c \text{ is even.}\end{cases}$$

Compared to [3], there it was assumed that $a = b = 2^m$ for some $m \geq 0$, and $c = 2^m + 1$. Similarly to merge networks, only one base case and one recursive case were considered. All the other cases introduced here are needed for arbitrary a, b and c.

4.4 *m*-Cardinality Networks

An *m-cardinality network* takes an input of size n and outputs the first m sorted bits. Recursively, an m-cardinality network with input $x_1, x_2, \ldots, x_n$ can be defined as follows:

– If $n \leqslant m$, a cardinality network is a sorting network:

$$\mathrm{Card}_m(x_1, x_2, \ldots, x_n) := \mathrm{Sorting}(x_1, x_2, \ldots, x_n).$$

– If $n > m$, take l with $1 \leqslant l < n$. Let us define

$$\begin{aligned}(z_1, z_2, \ldots, z_A) &= \mathrm{Card}_m(x_1, x_2, \ldots, x_l),\\ (z'_1, z'_2, \ldots, z'_B) &= \mathrm{Card}_m(x_{l+1}, x_{l+2}, \ldots, x_n),\\ (y_1, y_2, \ldots, y_m) &= \mathrm{SMerge}_m(z_1, z_2, \ldots, z_A; z'_1, z'_2, \ldots, z'_B),\end{aligned}$$

where $A = \min\{l, m\}$ and $B = \min\{n - l, m\}$. Then,

$$\mathrm{Card}_m(x_1, x_2, \ldots, x_n) := (y_1, y_2, \ldots, y_m).$$

Again, the number of auxiliary variables and clauses needed in these networks can be recursively computed. An m-cardinality network of size n composed by an m-cardinality network of size l and an m-cardinality network of size $n-l$ needs $V_1+V_2+V_3$ variables and $C_1+C_2+C_3$ clauses, where $(V_1, C_1), (V_2, C_2)$ are the number of variables and clauses used in the m-cardinality networks of sizes l and $n-l$, and (V_3, C_3) are idem in the m-simplified merge network with inputs of sizes $(\min\{l, m\}, \min\{n - l, m\})$.

Compared to [3], in that work m is assumed to be a power of two, and n a multiple of m. Moreover, similarly to sorting networks, in the recursive case l is always chosen to be m, while here we have an additional degree of freedom.

Using the same techniques in [3] one can easily prove the arc-consistency of the encoding.

Theorem 1. *The Recursive Cardinality Network encoding is arc-consistent: consider a cardinality constraint $x_1 + \ldots + x_n \leqslant k$, its corresponding cardinality network $(y_1, y_2, ..., y_{k+1}) = \mathrm{Card}_{k+1}(x_1, x_2, \ldots, x_n)$, and the unit clause $\neg y_{k+1}$. If we now set to true k input variables, then unit propagation sets to false the remaining $n - k$ input variables.*

Proof (sketch). The proof relies on the following lemmas, which formalize the propagation properties of the building blocks of cardinality networks:

Lemma 1 (Merge Networks). *Let S be the set of clauses of*

$$(y_1, y_2, \ldots, y_{a+b}) = \mathrm{Merge}(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b).$$

Let $p, q \in \mathbb{N}$ with $0 \leq p \leq a$ and $0 \leq q \leq b$. Then:

1. $S \cup \{x_1, \ldots, x_p, x'_1, \ldots, x'_q\} \models_{\mathrm{UP}} y_1, \ldots, y_{p+q}$.
2. *If $p < a$ and $q < b$ then* $S \cup \{x_1, \ldots, x_p, x'_1, \ldots, x'_q, \overline{y_{p+q+1}}\} \models_{\mathrm{UP}} \overline{x_{p+1}}, \overline{x'_{q+1}}$.
3. *If $p = a$ and $q < b$ then* $S \cup \{x_1, \ldots, x_p, x'_1, \ldots, x'_q, \overline{y_{p+q+1}}\} \models_{\mathrm{UP}} \overline{x'_{q+1}}$.
4. *If $p < a$ and $q = b$ then* $S \cup \{x_1, \ldots, x_p, x'_1, \ldots, x'_q, \overline{y_{p+q+1}}\} \models_{\mathrm{UP}} \overline{x_{p+1}}$.

Lemma 2 (Sorting Networks). *Let $X = (x_1, x_2, \ldots, x_n)$, $X' \subseteq X$ and S be the set of clauses of $(y_1, y_2, \ldots, y_n) = \text{Sorting}(X)$. Let $p = |X'|$. Then:*

1. *$S \cup X' \models_{\text{UP}} y_1, \ldots, y_p$.*
2. *If $p = |X'| < n$, then $S \cup X' \cup \{\overline{y_{p+1}}\} \models_{\text{UP}} \overline{x_i}$ for all $x_i \notin X'$.*

Lemma 3 (Simplified Merge Networks). *Let S be the set of clauses of*

$$(y_1, y_2, \ldots, y_c) = \text{SMerge}_c(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b).$$

Let $p, q \in \mathbb{N}$ be such that $0 \le p \le a$, $0 \le q \le b$. Then:

1. *If $p + q \le c$, then $S \cup \{x_1, \ldots, x_p, x'_1, \ldots, x'_q\} \models_{\text{UP}} y_1, \ldots, y_{p+q}$.*
2. *If $p < a$, $q < b$ and $p+q < c$, then $S \cup \{x_1, \ldots, x_p, x'_1, \ldots, x'_q, \overline{y_{p+q+1}}\} \models_{\text{UP}} \overline{x_{p+1}}, \overline{x'_{q+1}}$.*
3. *If $p = a$, $q < b$ and $p + q < c$, then $S \cup \{x_1, \ldots, x_p, x'_1, \ldots, x'_q, \overline{y_{p+q+1}}\} \models_{\text{UP}} \overline{x'_{q+1}}$.*
4. *If $p < a$, $q = b$ and $p + q < c$, then $S \cup \{x_1, \ldots, x_p, x'_1, \ldots, x'_q, \overline{y_{p+q+1}}\} \models_{\text{UP}} \overline{x_{p+1}}$.*

Lemma 4 (Cardinality Networks). *Let $X = (x_1, x_2, \ldots, x_n)$, $X' \subseteq X$ and S be the set of clauses of $(y_1, y_2, \ldots, y_m) = \text{Card}_m(X)$. Let $p = |X'|$. Then:*

1. *If $p \le m$, then $S \cup X' \models_{\text{UP}} y_1, \ldots, y_p$.*
2. *If $p < m$, then $S \cup X' \cup \{\overline{y_{p+1}}\} \models_{\text{UP}} \overline{x_i}$ for all $x_i \notin X'$.*

Each lemma is proved by induction and using the corresponding lemmas of the inner building blocks. The proofs of Lemmas 1 and 3 require considering four cases according to the parities of p and q. Finally, the theorem follows as a corollary of Lemma 4.

For the sake of illustration, let us prove the case $a, b \leqslant c$, $a + b > c$, with c even, of the inductive case of property 1 in Lemma 3. So, let us consider the set of clauses of

$$(z_1, y_2, y_3, \ldots, y_c) = \text{SMerge}_c(x_1, x_2, \ldots, x_a; x'_1, x'_2, \ldots, x'_b)$$

consisting of the clauses $z_c \to y_c,\ z_{c+1} \to y_c$ and those in

$$\begin{aligned}
(z_1, z_3, \ldots, z_{c+1}) &= \text{SMerge}_{c/2+1}(x_1, x_3, \ldots; x'_1, x'_3, \ldots),\\
(z_2, z_4, \ldots, z_c) &= \text{SMerge}_{c/2}(x_2, x_4, \ldots; x'_2, x'_4, \ldots),\\
(y_2, y_3) &= \text{2-Comp}(z_2, z_3),\\
&\ldots\\
(y_{c-2}, y_{c-1}) &= \text{2-Comp}(z_{c-2}, z_{c-1}).
\end{aligned}$$

Let $p, q \in \mathbb{N}$ such that $0 \le p \le a$, $0 \le q \le b$ and $p + q \le c$. If $p = q = 0$ there is nothing to prove. Otherwise let us show $S \cup \{x_1, \ldots, x_p, x'_1, \ldots, x'_q\} \models_{\text{UP}} z_1, y_i$ for all $2 \le i \le p + q$.

Here we focus on the subcase p and q even, being the other three cases analogous. Hence, let $p = 2p'$ and $q = 2q'$. In $x_1, x_2, \ldots, x_p$ there are p' odd indices and p' even indices. Similarly, in $x'_1, x'_2, \ldots, x'_q$ there are q' odd indices and q' even indices. Thus, using the IH (note $p' + q' \le c/2 < c/2 + 1$), we have that the clauses of the subnetwork $(z_1, z_3, \ldots, z_{c+1}) = \text{SMerge}_{c/2+1}(x_1, x_3, \ldots; x'_1, x'_3, \ldots)$ propagate by unit propagation the literals z_1, ..., $z_{2(p'+q')-1}$; and that the clauses of $(z_2, z_4, \ldots, z_c) = \text{SMerge}_{c/2}(x_2, x_4, \ldots; x'_2, x'_4, \ldots)$ propagate by unit propagation the literals z_2, ..., $z_{2(p'+q')}$. Altogether, all literals z_j with $1 \le j \le p + q$ can be propagated by unit propagation.

Let us take $2 \le i \le p + q$. If i is odd then, thanks to literals z_{i-1} and z_i and clause $z_{i-1} \wedge z_i \to y_i$ of the 2-comparator $(y_{i-1}, y_i) = \text{2-Comp}(z_{i-1}, z_i)$, literal y_i is propagated. If i is even, then thanks to literal z_i and clause $z_i \to y_i$, literal y_i is propagated too.

5 Direct Cardinality Networks

In this section we introduce an alternative technique for building cardinality networks which we call *direct*, as it is non-recursive. This method uses many fewer auxiliary variables than the recursive approach explained in Section 4. On the other hand, the number of clauses of this construction makes it competitive only for small sizes. However, this is not a problem as we will see in Section 6, as the two techniques can be combined.

As in the recursive construction described in Section 4, the building blocks of direct cardinality networks are merge, sorting and simplified merge networks:

– **Merge Networks**. They are defined as follows[6]:

$$\text{Merge}(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (y_1, y_2, y_3, \dots, y_{a+b-1}, y_{a+b}),$$

with clauses $\{x_i \to y_i,\ x'_j \to y_j,\ x_i \wedge x'_j \to y_{i+j}\ :\ 1 \leqslant i \leqslant a, 1 \leqslant j \leqslant b\}$. Notice we need $a + b$ variables and $ab + a + b$ clauses.

– **Sorting Networks**. A sorting network can be built as follows:

$$\text{Sorting}(x_1, x_2, \dots, x_n) := (y_1, y_2, \dots, y_n),$$

with clauses $\{x_{i_1} \wedge x_{i_2} \wedge \cdots \wedge x_{i_k} \to y_k\ :\ 1 \leqslant k \leqslant n, 1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n\}$. Therefore, we need n auxiliary variables and $2^n - 1$ clauses.

– **Simplified Merge Networks**. The definition of c-simplified merge is the same as in Section 4, except for the cases in which $a, b \leqslant c$ and $a + b > c$, where:

$$\text{SMerge}_c(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (y_1, y_2, \dots, y_c),$$

with clauses $\{x_i \to y_i,\ x'_j \to y_j,\ x_i \wedge x'_j \to y_{i+j}\ :\ 1 \leqslant i \leqslant a, 1 \leqslant j \leqslant b, i + j \leqslant c\}$. This approach needs c variables and $(a + b)c - \frac{c(c-1)}{2} - \frac{a(a-1)}{2} - \frac{b(b-1)}{2}$ clauses.

– m-**Cardinality Networks**. As in Section 4, except for the case $n > m$, where:

$$\text{Card}_m(x_1, x_2, \dots, x_n) := (y_1, y_2, \dots, y_m)$$

with clauses $\{x_{i_1} \wedge x_{i_2} \wedge \cdots \wedge x_{i_k} \to y_k\ :\ 1 \leqslant k \leqslant m, 1 \leqslant i_1 < i_2 < \cdots < i_k \leqslant n\}$. This approach needs m variables and $\binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{m}$ clauses.

As regards the arc-consistency of the encoding, the following can be easily proved:

Theorem 2. *The Direct Cardinality Network encoding is arc-consistent.*

Proof (sketch). The proof uses lemmas analogous to Lemmas 1, 2, 3 and 4. For illustration purposes, let us show property 1 in Lemma 3. Let us consider the clause set of $(y_1, y_2, \dots, y_c) = \text{SMerge}_c(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b)$, i.e.,

$$\{x_i \to y_i,\ x'_j \to y_j,\ x_i \wedge x'_j \to y_{i+j}\ :\ 1 \leqslant i \leqslant a, 1 \leqslant j \leqslant b, i + j \leqslant c\}.$$

Let $p, q \in \mathbb{N}$ be such that $0 \leq p \leq a$, $0 \leq q \leq b$ and $p + q \leq c$. If $p = q = 0$ there is nothing to prove. Otherwise let us consider $1 \leq k \leq p + q$. Let $0 \leq i \leq p$ and $0 \leq j \leq q$ be such that $i + j = k$. If $i = 0$ then $j = k$ and the clause $x'_j \to y_j$ propagates y_k. Similarly, if $j = 0$ then $i = k$ and the clause $x_i \to y_i$ propagates y_k. Finally, if $i \geq 1$ and $j \geq 1$ the clause $x_i \wedge x'_j \to y_{i+j}$ propagates y_k.

[6] Direct merge networks are similar to the totalizers of [7].

6 Combining Recursive and Direct Cardinality Networks

The recursive approach produces shorter networks than the direct approach when the input is middle-sized. Still, the recursive method for building a network needs to inductively produce networks for smaller and smaller input sizes. At some point, the networks we need have a sufficiently small number of inputs such that the direct method can build them using fewer clauses and variables than the recursive approach. Here a *mixed encoding* is presented: large cardinality networks are build with the recursive approach but their components are produced with the direct approach if their size is small enough.

In more detail, assume a merge of input sizes a and b is needed. We can use the direct approach, which needs $V_D = a + b$ auxiliary variables and $C_D = ab + a + b$ clauses; or we could use the recursive approach. With the recursive approach, we have to built two merge networks of sizes $\left(\left\lceil \frac{a}{2} \right\rceil, \left\lceil \frac{b}{2} \right\rceil\right)$ and $\left(\left\lfloor \frac{a}{2} \right\rfloor, \left\lfloor \frac{b}{2} \right\rfloor\right)$. These networks are also built with this mixed approach. Then, we compute the clauses and variables needed in the recursive approach, V_R and C_R, with the formula of Section 4.1: $V_R = V_1 + V_2 + 2\left\lfloor \frac{a+b-1}{2} \right\rfloor$, $C_R = C_1 + C_2 + 3\left\lfloor \frac{a+b-1}{2} \right\rfloor$, where (V_1, C_1) and (V_2, C_2) are, respectively, the number of variables and clauses needed in the recursive merge networks.

Finally, we compare the values of V_R, V_D, C_R and C_D, and decide which method is better for building the merge network. Notice that we cannot minimize both the number of variables and clauses; therefore, here we try to minimize the function $\lambda \cdot V + C$, for some fixed value $\lambda > 0$.[7] The parameter λ allows us to adjust the relative importance of the number of variables with respect to the number of clauses of the encoding. Notice that this algorithm for building merge networks (and similarly, sorting, simplified merge and cardinality networks) can easily be implemented with dynamic programming. See Section 7 for an experimental evaluation of the numbers of variables and clauses in cardinality networks built with this mixed approach.

The arc-consistency of the mixed encoding easily follows from the arc-consistency of the two encodings it is based on.

Theorem 3. *The Mixed Cardinality Network encoding is arc-consistent.*

Proof (sketch). The proof uses lemmas analogous to Lemmas 1, 2, 3 and 4. In turn, these lemmas are proved by combining the proofs outlined in Theorems 1 and 2.

7 Experimental Evaluation

In previous work [3], it was shown that power-of-two (Recursive) Cardinality Networks have overall better performance than other well-known methods such as Sorting Networks [10], Adders [10] and the BDD-based encoding of [6]. In what follows we will show that the generalization of Cardinality Networks to arbitrary size and their combination with Direct Encodings, yielding what we have called the **Mixed** approach, makes them significantly better, both in the size of the encoding and the SAT solver runtime.

We start the evaluation focusing on the size of the resulting encoding. In Figure 4 we show a representative graph, which indicates the size, in terms of variables and clauses, of the encoding of a cardinality network with input size 100 and varying output size m.

[7] This function can be replaced by any other monotone function that can be efficiently evaluated.

It can be seen that, since we minimize the function $\lambda \cdot V + C$, where V is the number of variables and C the number of clauses, the bigger λ is, the fewer variables we obtain, at the expense of a slight increase in the number of clauses. Also, it can be seen that using power-of-two Cardinality Networks as in [3] is particularly harmful when m is slightly larger than a power of two.

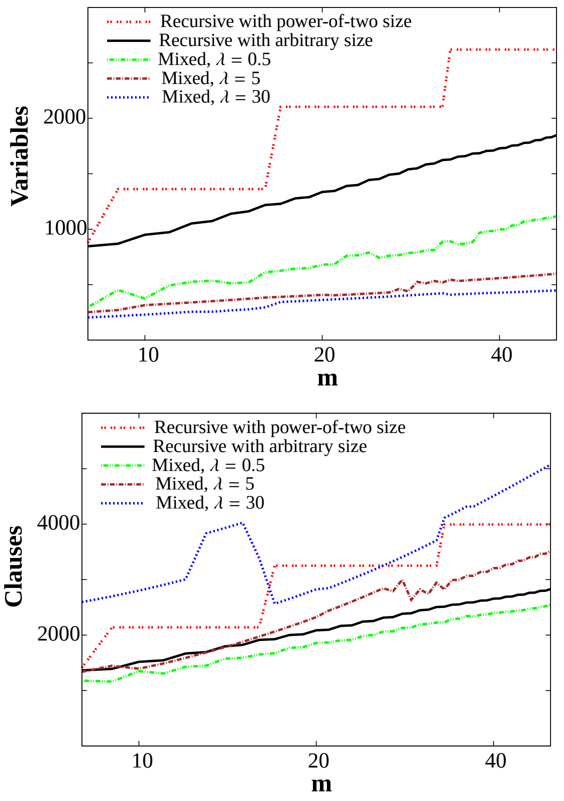


Fig. 4: Number of variables and clauses generated by **Mixed** and the Recursive Cardinality Networks approaches with input size 100 and different output sizes m.

Although having a smaller encoding is beneficial, this should be accompanied with a reduction in SAT solver runtime. Hence, let us now move to assess how our new encoding affects the performance of SAT solvers. In this evaluation, in addition to considering the power-of-two Recursive Cardinality Networks in [3] (**Power-of-two CN**), the (arbitrary-size) Recursive Cardinality Networks presented in Section 4 (**Arbitrary-sized CN**) and the **Mixed** approach of Section 6, we have also included other well-known encodings in the literature: the adder-based encoding (**Adder**) of [10] and the BDD-based encoding (**BDD**) of [6]. We believe these encodings are representative of all different approaches that have been used to deal with cardinality constraints. Other works, like the adder-based encoding of [20], the BDD-based one of [10] or the work by Anbulagan and Grastien [1], are small variations or combinations of the encodings we have chosen. Moreover, we have implemented an SMT-based approach (**SMT**) to Cardinality Constraints. In a nutshell, we have coupled a SAT solver with a theory solver that handles all cardinality constraints. As soon as a cardinality constraint is violated by the current partial assignment, the SAT solver is forced to backtrack and, when the value of a variable can be propagated thanks to a cardinality constraint, this information is passed to the SAT solver. In other words, cardinality constraints are not translated into SAT, but rather tackled by a dedicated algorithm, similar in nature to what some pseudo-Boolean solvers do. See [14] for more information about SMT.

The SAT solver we have used in this evaluation is Lingeling version *ala*, a state-of-the-art CDCL (Conflict-Driven Clause Learning) SAT solver that implements several in/preprocessing techniques. All experiments were conducted on a 2Ghz Linux Quad-Core AMD with the three following sets of benchmarks:

1.-MSU4 suite. These benchmarks are intermediate problems generated by an implementation of the *msu4* algorithm [12], which reduces a Max-SAT problem to a series of SAT problems with cardinality constraints. The *msu4* implementation was run of a variety of problems (filter design, logic synthesis, minimum-size test pattern generation, haplotype inference and maximum-quartet consistency) from the Partial Max-SAT division of the Third Max-SAT evaluation[8]. The suite consists of about 14000 benchmarks, each of which contains multiple $\leqslant$-cardinality constraints.

2.-Discrete-event system diagnosis suite. The second set of benchmarks we have used is the one introduced in [1]. These problems come from discrete-event system (DES) diagnosis. As it happened with the Max-SAT problems, a single DES problem produced a family of “SAT + cardinality constraints” problems. This way, out of the roughly 600 DES problems, we obtained a set of around 6000 benchmarks, each of which contained a single very large $\leqslant$-cardinality constraint.

3.-Tomography suite. The last set of benchmarks we have used is the one introduced in [5]. The idea is to first generate an $N \times N$ grid in which some cells are filled and some others are not. The problem consists in finding out which are the filled cells using only the information of how many filled cells there are in each row, column and diagonal. For that purpose, variables x_{ij} are used to indicate whether cell (i, j) is filled and several $=$-cardinality constraints impose how many filled cells there are in each row, column and diagonal. We generated 2600 benchmarks (100 instances for each size $N = 15 \ldots 40$).

Results are summarized[9] in Table 1, which compares the **Mixed** (with $\lambda = 5$) encoding with the aforementioned encodings. The time limit was set to 600 seconds per benchmark and we only considered benchmarks for which at least one of the methods took more than 5 seconds. There are three tables, one for each benchmark suite. In each table, columns indicate in how many benchmarks the **Mixed** encoding exhibits the corresponding speed-up or slow-down factor with respect to the method indicated in each row. For example, in the table for the **MSU4 suite**, the first row indicates that in 43 benchmarks, Power-of-two Cardinality Networks timed out (**TO**) whereas our new encoding did not. The columns next to it indicate that in 732 benchmarks the novel encoding was at least 4 times faster, in 2957 between 2 and 4 times faster, etc.

[8] See `http://www.maxsat.udl.cat/08/index.php?disp=submitted-benchmarks`.

[9] See `http://www.lsi.upc.edu/~oliveras/espai/CP13.ods` for detailed data.

	Speed-up factor of Mixed					Slow-down factor of Mixed				
	TO	**4**	**2**	**1.5**	**TOT.**	**1.5**	**2**	**4**	**TO**	**TOT.**
MSU4 suite										
Power-of-two CN	43	732	2957	1278	*5010*	1	23	13	11	*48*
Arbitrary-sized CN	10	149	544	726	*1429*	3	106	43	80	*232*
Adder	985	1207	1038	1250	*4480*	0	13	36	40	*89*
BDD	187	1139	1795	1292	*4413*	4	10	31	36	*81*
SMT	1143	323	102	53	*1621*	0	1417	211	63	*1691*
DES suite										
Power-of-two CN	13	21	265	638	*937*	6	12	7	46	*71*
Arbitrary-sized CN	19	21	75	404	*519*	5	12	11	45	*73*
Adder	218	235	611	1283	*2347*	0	5	3	42	*50*
BDD	705	3944	759	51	*5459*	0	0	0	0	*0*
SMT	3003	1134	262	73	*4472*	0	15	19	15	*49*
Tomography suite										
Power-of-two CN	118	388	408	175	*1089*	64	82	159	121	*426*
Arbitrary-sized CN	104	430	432	169	*1135*	67	81	158	11	*417*
Adder	492	591	371	143	*1597*	14	20	39	35	*108*
BDD	0	0	0	0	*0*	112	1367	184	51	*1714*
SMT	0	10	25	11	*46*	112	1250	155	68	*1585*

Table 1: Comparison of SAT solver runtime. Figures show number of benchmarks in which **Mixed** shows the corresponding speed-up/slow-down factor w.r.t. other methods.

We can see from the table that in the **MSU4** and **DES** suites, which contain benchmarks coming from real-world applications, our new encoding in general outperforms the other methods (except for some instances in which **Mixed** times out and the other cardinality network-based encodings do not; also, in **MSU4**, SMT and **Mixed** obtain comparable results). We want to remark that the gain comes both from using arbitrary-sized networks as well as from combining them with direct encodings, as can be seen from the second row of each table. In particular, this shows the negative impact of the dummy variables of [3], which hinder the performance in spite of the unit propagation of the SAT solver. Finally, in the **Tomography** suite, the BDD-based encoding and the SMT system outperform all other methods, but among the rest of the approaches the **Mixed** encoding exhibits the best performance. Altogether, the **Mixed** encoding is the most robust technique according to the results of this evaluation.

8 Conclusion and Future Work

The contributions of this paper are: (*i*) an extension of the recursive cardinality networks of [3] to arbitrary input and output sizes; (*ii*) a non-recursive construction of cardinality networks that is competitive for small sizes; (*iii*) a parametric combination of these two approaches for producing cardinality networks that not only improves on the size of the encoding, but also yields significant speedups in SAT solver performance.

As regards future work, we plan to develop encoding techniques for cardinality constraints that do not process constraints one-at-a-time but simultaneously, in order to exploit their similarities. We foresee that the flexibility of the approach presented here with respect to the original construction in [3], will open the door to sharing the internal networks among the cardinality constraints present in a SAT problem.

Acknowledgments. Abío is supported by DFG Graduiertenkolleg 1763 (QuantLA). All other authors are partially supported by Spanish MEC/MICINN under SweetLogics project (TIN 2010-21062-C02-01). We also thank the reviewers for their comments.

References

1. Anbulagan and Alban Grastien. Importance of Variables Semantic in CNF Encoding of Cardinality Constraints. In V. Bulitko and J. C. Beck, editors, *Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA '09*. AAAI, 2009.
2. Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks and their applications. In *Int. Conf. Theory and Applications of Satisfiability Testing (SAT), LNCS 4501*, pages 167–180, 2009.
3. Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality Networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.
4. Roberto Asín Achá and Robert Nieuwenhuis. Curriculum-based course timetabling with SAT and MaxSAT. *Annals of Operations Research*, pages 1–21, February 2012.
5. Olivier Bailleux and Yacine Boufkhad. Efficient CNF Encoding of Boolean Cardinality Constraints. In F. Rossi, editor, *Principles and Practice of Constraint Programming, 9th International Conference, CP '03*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003.
6. Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. A translation of pseudo boolean constraints to sat. *JSAT*, 2(1-4):191–200, 2006.
7. Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New Encodings of Pseudo-Boolean Constraints into CNF. In O. Kullmann, editor, *12th International Conference on Theory and Applications of Satisfiability Testing, SAT '09*, volume 5584 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2009.
8. K. E. Batcher. Sorting Networks and their Applications. In *AFIPS Spring Joint Computing Conference*, pages 307–314, 1968.
9. Michael Codish and Moshe Zazon-Ivry. Pairwise cardinality networks. In Edmund M. Clarke and Andrei Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *Lecture Notes in Computer Science*, pages 154–172. Springer, 2010.
10. Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.
11. Zhaohui Fu and Sharad Malik. Solving the minimum-cost satisfiability problem using SAT based branch-and-bound search. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ICCAD '06, pages 852–859, New York, NY, USA, 2006. ACM.
12. J. Marques-Silva and J. Planes. Algorithms for Maximum Satisfiability using Unsatisfiable Cores. In *2008 Conference on Design, Automation and Test in Europe Conference, DATE '08*, pages 408–413. IEEE Computer Society, 2008.
13. Amit Metodi, Michael Codish, and Peter J. Stuckey. Boolean equi-propagation for concise and efficient sat encodings of combinatorial problems. *J. Artif. Intell. Res. (JAIR)*, 46:303–341, 2013.
14. 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.
15. Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 2:205–211, 1992.
16. David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.
17. Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Why cumulative decomposition is not as bad as it sounds. In *Proceedings of the 15th international conference on Principles and practice of constraint programming*, CP'09, pages 746–761, Berlin, Heidelberg, 2009. Springer-Verlag.

18. C. Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In P. v. Beek, editor, *Principles and Practice of Constraint Programming, 11th International Conference, CP '05*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005.
19. M. Büttner and J. Rintanen. Satisfiability planning with constraints on the number of actions. In S. Biundo, K. L. Myers, and K. Rajan, editors, *15th International Conference on Automated Planning and Scheduling, ICAPS '05*, pages 292–299. AAAI, 2005.
20. Joost P. Warners. A Linear-Time Transformation of Linear Inequalities into Conjunctive Normal Form. *Information Processing Letters*, 68(2):63–69, 1998.