# Encoding Cardinality Constraints using Multiway Merge Selection Networks

Michał Karpiński, Marek Piotrów

Institute of Computer Science, University of Wrocław
Joliot-Curie 15, 50-383 Wrocław, Poland
{karp,mpi}@cs.uni.wroc.pl

**Abstract.** Boolean cardinality constraints (CCs) state that at most (at least, or exactly) $k$ out of $n$ propositional literals can be true. We propose a new, arc-consistent, easy to implement and efficient encoding of CCs based on a new class of selection networks. Several comparator networks have been recently proposed for encoding CCs and experiments have proved their efficiency [1, 2, 8, 9]. In our construction we use the idea of the multiway merge sorting networks by Lee and Batcher [6] that generalizes the technique of odd-even sorting ones by merging simultaneously more than two subsequences. The new selection network merges 4 subsequences in that way. Based on this construction, we can encode more efficiently comparators in the combine phase of the network: instead of encoding each comparator separately by 3 clauses and 2 additional variables, we propose an encoding scheme that requires 5 clauses and 2 variables on average for each pair of comparators. We also extend the model of comparator networks so that the basic components are not only comparators (2-sorters) but more general $m$-sorters, for $m \in \{2,3,4\}$, that can also be encoded efficiently. We show that with small overhead (regarding implementation complexity) we can achieve a significant improvement in SAT-solver runtime for many test cases. We prove that the new encoding is competitive to the other state-of-the-art encodings.

## 1 Introduction

Several hard decision problems can be efficiently reduced to the Boolean satisfiability (SAT) problem and tried to be solved by recently-developed SAT-solvers. Some of them are formulated with the help of different high-level constraints, which should be either encoded into CNF formulas or solved inside a SAT-solver by a specialized extension. There has been much research on both of these approaches.

In this paper we consider encodings of Boolean cardinality constraints that take the form $x_1 + x_2 + \cdots + x_n \sim k$, where $x_1, x_2, \ldots, x_n$ are Boolean literals (that is, variables or their negations), $\sim$ is a relation from the set $\{<, \leq, =, \geq, >\}$ and $k \in \mathbb{N}$. Such cardinality constraints appear naturally in formulations of different real-world problems including cumulative scheduling [17], timetabling [3] or formal hardware verification [7].

In a direct encoding of a cardinality constraint $x_1 + x_2 + \cdots + x_n < k$ one can take all subsets of $X = \{x_1, \ldots, x_n\}$ of size $k$ and for each of them construct a CNF formula that states that at least one of the literals in the subset must be false. The direct encoding is quite efficient for very small values of $k$ and $n$, but for larger parameters another approach should be used.

### 1.1 Related work

In the last years several selection networks were proposed for encoding cardinality constraints and experiments proved their efficiency. They were based mainly on the odd-even or pairwise comparator networks. Codish and Zazon-Ivry [8] introduced pairwise selection networks that used the concept of Parberry's pairwise sorting network [15]. Their construction was later improved by Karpiński and Piotrów [12]. Abío, Asín, Nieuwenhuis, Oliveras and Rodríguez-Carbonell [1, 2] defined encodings that implemented selection networks based on the odd-even sorting networks by Batcher [5]. In [1] the authors proposed a mixed parametric approach to the encodings, where the direct encoding is chosen for small sub-problems and the splitting point is optimized when large problems are divided into two smaller ones. They proposed to minimize the function $\lambda \cdot num\_vars + num\_clauses$ in the encodings. The constructed encodings are small and efficient.

### 1.2 Our contribution

To improve the existing constructions, we started looking for selection networks that can be easily implemented and encoded with a smaller number of auxiliary variables and, if possible, not much larger number of clauses. In addition, we investigate the influence of our encodings on the execution times of SAT-solvers to be sure that the new algorithms can be used in practice. The obtained construction is presented in this paper. The main idea is to split the problem into 4 sub-problems, recursively select elements in them and then merge the selected subsequences using an idea of multi-way merging. In such a construction, we can encode more efficiently comparators in the combine phase of the merger: instead of encoding each comparator separately by 3 clauses and 2 additional variables. we propose an encoding scheme that requires 5 clauses and 2 variables on average for each pair of comparators. Moreover, in the network we can use not only comparators (2-sorters) but also $m$-sorters (for $m \leq 4$), which can be encoded directly. It should be noted here that the value of $m$ must be small, because the direct encoding of an $m$-sorter requires $2^m - 1$ clauses.

Using this generalized version of comparators we have created a novel class of networks which we call 4-**Odd-Even Selection Networks**, where the multi-way merge sorting networks by Batcher and Lee [6] are generalized in a way that we can recursively select $k$ largest elements from each of the 4 sub-problems.

Our algorithm is presented using divide-and-conquer paradigm. The key to achieve efficient algorithms lies in the construction of networks that combine the results obtained from the recursive calls. The construction of those *mergers* is one of the main results of this paper. We give a detailed construction for 4-Odd-Even Merging Network. We compare the numbers of variables and clauses of the encoding and its counterpart: the 2-Odd-Even Merging Networks [8]. The calculations show that encodings based on our network use fewer variables and clauses, when $k < n$.

The construction is parametrized by any values of $k$ and $n$, so they can be further optimized by mixing them with other constructions. For example, in our experiments we mixed them with the direct encoding for small values of parameters. We used the standard encoding of $m$-sorters, therefore arc-consistency is preserved [10]. Finally, we

present results of our experiments. We show that multi-column selection networks are superior to standard selection networks previously proposed in the literature, in context of translating cardinality constraints into propositional formulas.

We also empirically compare our encodings with other state-of-the-art encodings, not only based on comparator networks, but also on binary adders and binary decision diagrams. Those are mainly used in encodings of Pseudo-Boolean constraints, but it is informative to see how well they perform when encoding cardinality constraints.

### 1.3 Structure of the paper

The rest of the paper is organized as follows: Section 2 contains definitions and notations used in the paper. In Section 3 the construction of the 4-Odd-Even Selection Network is given. In Section 4 we compare the encoding produced by our constructions with 2-Odd-Even Selection Networks in terms of number of variables and clauses. Our experimental evaluation is presented in Section 5 followed by conclusions in Section 6.

## 2 Preliminaries

In this section we introduce definitions and notations used in the rest of the paper. Let $X$ denote a totally ordered set, for example the set of natural numbers $\mathbb{N}$ or the set of binary values $\{0,1\}$.

**Definition 1 (sequences).** *A sequence of length n, say $\bar{x} = \langle x_1, \ldots, x_n \rangle$, is an element of $X^n$. In particular, an element of $\{0,1\}^n$ is called a* binary *sequence. We say that a sequence $\bar{x} \in X^n$ is* sorted *if $x_i \geq x_{i+1}$, $1 \leq i < n$. Given two sequences $\bar{x} = \langle x_1, \ldots, x_n \rangle$ and $\bar{y} = \langle y_1, \ldots, y_m \rangle$ we define* concatenation *as $\bar{x} :: \bar{y} = \langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$.*

*We use also the following notation: $\bar{x}_{odd} = \langle x_1, x_3, \ldots \rangle$, $\bar{x}_{even} = \langle x_2, x_4, \ldots \rangle$, $\bar{x}_{a,\ldots,b} = \langle x_a, \ldots, x_b \rangle$, $1 \leq a \leq b \leq n$, and the* prefix/suffix *operators: $pref(i,\bar{x}) = \bar{x}_{1,\ldots,i}$ and $suff(i,\bar{x}) = \bar{x}_{i,\ldots,n}$, $1 \leq i \leq n$. The length of $\bar{x}$ is denoted by $|\bar{x}|$. The number of occurrences of a given value $b$ in $\bar{x}$ is denoted by $|\bar{x}|_b$.*

*A sequence $\bar{x} \in X^n$ is top $k$ sorted, for $k \leq n$, if $\langle x_1, \ldots, x_k \rangle$ is sorted and $x_k \geq x_i$, for each $i > k$.*

### 2.1 Comparator Networks

We construct and use comparator networks in this paper. Traditionally comparator networks are presented as circuits that receive $n$ inputs and permute them using comparators (2-sorters) connected by "wires". Each comparator has two inputs and two outputs. The "upper" output is the maximum of inputs, and "lower" one is the minimum. The standard definitions and properties of them can be found, for example, in [13]. The only difference is that we assume that the output of any sorting operation or comparator is in a non-increasing order.

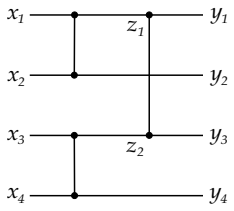



**Fig. 1.** Comparator network.

In the definitions of this section we assume that comparators are functions and comparator networks are composition of comparators. This makes the presentation clear.

*Example 1.* Figure 1 is an example of a simple comparator network consisting of 3 comparators. It outputs the maximum from 4 inputs on the top horizontal line, namely, $y_1 = \max\{x_1, x_2, x_3, x_4\}$.

**Definition 2 (selection network, m-sorter).** *A comparator network $f_k^n$ (where $k \leq n$) is a* $k$-selection network *(or* $k$-selector of order $n$*), if for each $\bar{x} \in X^n$, $f_k^n(\bar{x})$ is top $k$ sorted and is a permutation of $\bar{x}$. An $m$-selector of order $m$ is called a sorting network (or an* m-sorter*).*

The main building blocks of our networks are *mergers* – networks that output sorted sequence (or top $k$ sorted sequence) given outputs of recursive calls.

**Definition 3 (m-merger).** *A comparator network $f_k^s$ is an* m-merger of order $k$*, if for each tuple $T = \langle \bar{x}^1, \ldots, \bar{x}^m \rangle$, where each $\bar{x}^i$ is top $k$ sorted and $s = \sum_{i=1}^{m} |\bar{x}^i|$, $f_k^s(T)$ is top $k$ sorted and is a permutation of $\bar{x}^1 :: \ldots :: \bar{x}^m$.*

### 2.2 Standard Encoding of Cardinality Constraints

A clause is a disjunction of literals (Boolean variables $x$ or their negation $\neg x$). A CNF formula is a conjunction of clauses. Cardinality constraints are of the form $x_1 + \cdots + x_n \sim k$, where $k \in \mathbb{N}$ and $\sim$ belongs to $\{<, \leq, =, \geq, >\}$. We will focus on cardinality constraints with less-than relation, i.e. $x_1 + \cdots + x_n < k$. The other can be easily translated to such form (see [2]).

In [1, 2, 8, 9] authors are using sorting networks to encode cardinality constraints, where inputs and outputs of a comparator are Boolean variables and comparators are encoded as a CNF formula. In addition, the $k$-th greatest output variable $y_k$ of the network is forced to be 0 by adding $\neg y_k$ as a clause to the formula that encodes $x_1 + \cdots + x_n < k$. We use similar approach, but rather than using simple comparators (2-sorters), we also use comparators of higher order as building blocks. The $m$-selector of order $n$ can be encoded as follows: for $n$ input variables $\bar{x}$ and $m$ output variables $\bar{y}$, we add the set of clauses $\{x_{i_1} \wedge \cdots \wedge x_{i_p} \Rightarrow y_p : 1 \leq p \leq m, 1 \leq i_1 < \cdots < i_p \leq n\}$. The $m$-sorter is an $m$-selector of order $m$, therefore we need $m$ auxiliary variables and $2^m - 1$ clauses to encode it.

*Example 2.* Assume, that we would like to encode 1-selector of order 4 using the network given in Figure 1. If we name the input variables of the longer comparator as $\{z_1, z_2\}$, then the entire network can be encoded by encoding each 2-sorter separately. This produces the clause set $\{x_1 \Rightarrow z_1, x_2 \Rightarrow z_1, x_1 \wedge x_2 \Rightarrow y_2\} \cup \{x_3 \Rightarrow z_2, x_4 \Rightarrow z_2, x_3 \wedge x_4 \Rightarrow y_4\} \cup \{z_1 \Rightarrow y_1, z_2 \Rightarrow y_1, z_1 \wedge z_2 \Rightarrow y_3\}$. This approach uses 6 auxiliary variables (not counting $x_i$'s) and 9 clauses. Another way to encode the same network is to simply use a single 1-selector of order 4. This gives the clause set $\{x_1 \Rightarrow y_1, x_2 \Rightarrow y_1, x_3 \Rightarrow y_1, x_4 \Rightarrow y_1\}$, where we only need 1 additional variable and 4 clauses. Notice that to achieve $y_1 = \max\{x_1, x_2, x_3, x_4\}$ we are only interested in the value of the top output variable, therefore we do not need to assert other output variables.

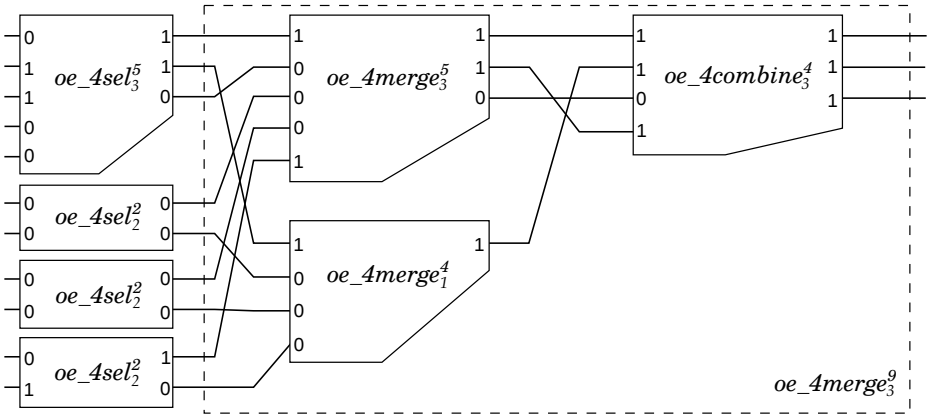



**Fig. 2.** An example of 4-Odd-Even Selection Network, with $n = 11$, $k = 3$, $n_1 = 5$, $n_2 = n_3 = n_4 = 2$.

### 2.3 Arc-consistency

Unit Propagation (UP) is a process, that for given CNF formula, clauses are sought in which all literals but one are false (say $l$) and $l$ is undefined (initially only clauses of size one satisfy this condition). This literal $l$ is set to true and the process is iterated until reaching a fix point.

In the case of cardinality constraints and SAT-solvers, arc-consistency states that: for a constraint $x_1 + \cdots + x_n < k$, as soon as $k-1$ variables among the $x_i$'s become true, unit propagation sets all other $x_i$'s to false. This has a positive impact on the practical efficiency of SAT-solvers, which is an important factor for the Constraint Programming community.

Encodings using selection networks where each $m$-sorter is encoded as described in the previous sub-section and additional clause $\neg y_k$ is added are said to be encoded in a *standard* way, and it is already known that such encodings are arc-consistent [10].

## 3 New Selection Network

Here we present a novel construction of selection network which uses sorters up to size 4 as components. We want to apply our algorithms for CNF encoding, therefore the only non-trivial operation that we are allowed to use in generalized comparator networks is $select_k^m : X^m \to X^m$, which is an $k$-selector of order $m$ (for $k \leq m \leq 4$ or $k = 1$). For the purpose of presentation we use it as a black box, but keep in mind that in the actual implementation one should encode each $k$-selector of order $m$ using the standard procedure explained in Section 2. We would also like to note that only first $k$ output for the selection network is of interest, but to stay consistent with the definitions of Section 2, we write our algorithms so that the output sequence is a permutation of the given input one. To this end we introduce the variable $\overline{out}$ in which we store all throw-away variables in an arbitrary order. The sequence $\overline{out}$ is then appended to the output of the algorithm.

**Network 1** $oe\_4sel^n_k$

**Input:** $\bar{x} \in \{0,1\}^n$; $0 \le k \le n$

1: **if** $k = 0$ **or** $n \le 1$ **then return** $\bar{x}$
2: **else if** $k = 1$ **then return** $select^n_1(\bar{x})$
3: **if** $n < 8$ **or** $k = n$ **then** $n_2 = \lfloor (n+2)/4 \rfloor$; $n_3 = \lfloor (n+1)/4 \rfloor$; $n_4 = \lfloor n/4 \rfloor$; ▷ divide evenly
4: **else if** $2^{\lceil \log(k/6) \rceil} \le \lfloor n/4 \rfloor$ **then** $n_2 = n_3 = n_4 = 2^{\lceil \log(k/6) \rceil}$ ▷ divide into powers of 2
5: **else** $n_2 = n_3 = n_4 = \lfloor k/4 \rfloor$ ▷ otherwise, if the power of 2 is too far from k/4
6: $n_1 = n - n_2 - n_3 - n_4$ ▷ $n = n_1 + \cdots + n_4$ and $n_1 \ge n_2 \ge n_3 \ge n_4$
7: $offset = 1$
8: **for all** $i \in \{1, \ldots, 4\}$ **do**
9: $\quad k_i = \min(k, n_i)$
10: $\quad \bar{y}^i \leftarrow oe\_4sel^{n_i}_{k_i}(\langle x_{offset}, \ldots, x_{offset+n_i-1} \rangle)$ ▷ recursive calls
11: $\quad offset += n_i$
12: $s = \sum_{i=1}^{4} k_i$; $\overline{out} = \mathrm{suff}(k_1 + 1, \bar{y}^1) :: \cdots :: \mathrm{suff}(k_4 + 1, \bar{y}^4)$
13: **return** $oe\_4merge^s_k(\langle \mathrm{pref}(k_1, \bar{y}^1), \ldots, \mathrm{pref}(k_4, \bar{y}^4) \rangle) :: \overline{out}$

**Ensure:** The output is top $k$ sorted and is a permutation of the inputs.

### 3.1 4-Odd-Even Selection Network

We begin with the top-level algorithm for constructing the 4-Odd-Even Selection Network (Network 1) where we use $oe\_4merge^s_k$ as a black box. It is a 4-merger of order $k$. We give detailed construction of a 4-merger called 4-Odd-Even Merger in the next sub-section.

The idea we use is the generalization of the one used in 2-Odd-Even Selection Network from [8], which is based on the Odd-Even Sorting Network by Batcher [5], but we replace the last network with Multiway Merge Sorting Network by Batcher and Lee [6]. We arrange the input sequence into 4 columns of non-increasing sizes (lines 3–6) and then recursively run the selection algorithm on each column (lines 9–11), where at most top $k$ items are selected from each column. Notice that each column is represented by ranges derived from the increasing value of variable $offset$. Notice further, that sizes of the columns are selected in such a way that in most cases all but first columns are of equal length and the length is a power of two (lines 3–5) that is close to the value of $k/4$ (observe that $[k/6, k/3)$ is the smallest symmetric interval around $k/4$ that contains a power of 2). Such a choice produces much longer propagation paths for small values of $k$ with respect to $n$. In the recursive calls selected items are sorted and form prefixes of the columns, which are then the input to the merging procedure (line 13). The base case, when $k = 1$ (line 2), is handled by the auxiliary network $select^n_1$, which outputs the maximum of $n$ elements and can be encoded with $n$ clauses.

*Example 3.* In Figure 2 we present a schema of 4-Odd-Even Selection Network, which selects 3 largest elements from the input 01100000001. In this example, $n = 11$, $k = 3$, $n_1 = 5$, $n_2 = n_3 = n_4 = 2$. First, the input is passed to the recursive calls, then the procedure $oe\_4merge^9_3$ is applied (Network 3).

**Theorem 1.** *Let $n, k \in \mathbb{N}$, such that $k \le n$. Then $oe\_4sel^n_k$ is a $k$-selection network.*

Due to the space limitations a proof will be given in the full version of the paper.

**Network 2** $oe\_4combine^s_k$

**Input:** A pair of sorted sequences $\langle \bar{x}, \bar{y} \rangle$, where $k \leq s = |\bar{x}| + |\bar{y}|$, $|\bar{y}| \leq \lfloor k/2 \rfloor$, $|\bar{x}| \leq \lfloor k/2 \rfloor + 2$ and $|\bar{y}|_1 \leq |\bar{x}|_1 \leq |\bar{y}|_1 + 4$.

1: Let $x(i)$ denote 0 if $i > |\bar{x}|$ or else $x_i$. Let $y(i)$ denote 1 if $i < 1$ or 0 if $i > |\bar{y}|$ or $y_i$, otherwise.
2: **for all** $j \in \{1, \ldots, |\bar{x}| + |\bar{y}|\}$ **do**
3: $\quad i = \lceil j/2 \rceil$
4: $\quad$ **if** $j$ is even **then** $a_j \leftarrow \max(\max(x(i+2), y(i)), \min(x(i+1), y(i-1)))$
5: $\quad$ **else** $a_j \leftarrow \min(\max(x(i+1), y(i-1)), \min(x(i), y(i-2)))$
6: **return** $\bar{a}$

**Ensure:** The output is sorted and is a permutation of the inputs.

**Network 3** $oe\_4merge^s_k$

**Input:** A tuple of sorted sequences $\langle \bar{w}, \bar{x}, \bar{y}, \bar{z} \rangle$, where $1 \leq k \leq s = |\bar{w}| + |\bar{x}| + |\bar{y}| + |\bar{z}|$ and $k \geq |\bar{w}| \geq |\bar{x}| \geq |\bar{y}| \geq |\bar{z}|$.

1: **if** $|\bar{x}| = 0$ **then return** $\bar{w}$
2: **if** $|\bar{w}| = 1$ **then return** $select^s_k(\bar{w} :: \bar{x} :: \bar{y} :: \bar{z})$ ▷ Note that $s \leq 4$ in this case
3: $s_a = \lceil |\bar{w}|/2 \rceil + \lceil |\bar{x}|/2 \rceil + \lceil |\bar{y}|/2 \rceil + \lceil |\bar{z}|/2 \rceil$; $k_a = \min(s_a, \lfloor k/2 \rfloor + 2)$;
4: $s_b = \lfloor |\bar{w}|/2 \rfloor + \lfloor |\bar{x}|/2 \rfloor + \lfloor |\bar{y}|/2 \rfloor + \lfloor |\bar{z}|/2 \rfloor$; $k_b = \min(s_b, \lfloor k/2 \rfloor)$
5: $\bar{a} \leftarrow oe\_4merge^{s_a}_{k_a}(\bar{w}_{odd}, \bar{x}_{odd}, \bar{y}_{odd}, \bar{z}_{odd})$ ▷ Recursive calls.
6: $\bar{b} \leftarrow oe\_4merge^{s_b}_{k_b}(\bar{w}_{even}, \bar{x}_{even}, \bar{y}_{even}, \bar{z}_{even})$
7: **return** $oe\_4combine^{k_a+k_b}_k(\mathrm{pref}(k_a, \bar{a}), pref(k_b, \bar{b})) :: \mathrm{suff}(k_a + 1, \bar{a}) :: \mathrm{suff}(k_b + 1, \bar{b})$

**Ensure:** The output is top $k$ sorted and is a permutation of the inputs.

### 3.2 4-Odd-Even Merging Network

In this section we give the detailed construction of the network *oe_4merge* – the 4-Odd-Even Merger – that merges four sequences (columns) obtained from the recursive calls in Network 1. We can assume that input columns are sorted and of length at most $k$.

The network is presented in Network 3. The input to the procedure is $\langle \mathrm{pref}(k_1, \bar{y}^1), \ldots, \mathrm{pref}(k_4, \bar{y}^4) \rangle$, where each $\bar{y}^i$ is the output of the recursive call in Network 1. The goal is to return the $k$ largest (and sorted) elements. It is done by splitting each input sequence into two parts, one containing elements of odd index, the other containing elements of even index. Odd sequences and even sequences are then recursively merged (lines 5–6) into two sequences $\bar{a}$ and $\bar{b}$ that are top $k$ sorted. The sorted prefixes are then combined by *oe_4combine* into a sorted sequence to which the suffixes of $\bar{a}$ and $\bar{b}$ are appended. The result is top $k$ sorted.

Our network is the generalization of the classic Multiway Merge Sorting Network by Batcher and Lee [6], where we use 4-way mergers and each merger consists of two sub-mergers and a combine sub-network. The goal of our network is to select and sort the $k$ largest items of four sorted input sequences. The combine networks are described and analyzed in [6]. The goal of them is to correct a small, unordered part that can appear after zipping the two input sequences $\bar{x} = \mathrm{pref}(k_a, \bar{a})$ and $\bar{y} = \mathrm{pref}(k_b, \bar{b})$ (by "zipping" we mean producing a sequence $\langle x_1, y_1, x_2, y_2, \ldots \rangle$). Since $\bar{x}$ can contain up to 4 more 1's than $\bar{y}$, it is enough to apply two sets of comparators: in the first set

compare-and-exchange $y_i$ with $x_{i+2}$, $i = 1,2,\ldots$, and in the second one compare-and-exchange even with odd items in the output of the first set. For example, if $\bar{x} = 1^{m+3}0^*$ and $\bar{y} = 1^m0^*$ for some $m \in \mathbb{N}$, then after the zip operation, the resulting sequence looks like $1^{2m+1}01010^*$, therefore we need just one comparator $y_{m+1} : x_{m+3}$ to fix the order and make the sequence sorted.

Let $\bar{a} = oe\_4combine_k^{k_a+k_b}(\bar{x},\bar{y})$, that is, $a_{2i}$ ($a_{2i-1}$), $i = 1,\ldots$ is defined by the equation in line 4 (line 5, respectively) of Network 2. They correspond to the two described-above sets of comparators. In our implementation the equations are encoded into clauses, in such a way, that in average we use 2 new variables and 5 clauses for each pair of comparators: if 1's should be propagated from inputs to outputs then (1) $y_i \Rightarrow a_{2i}$, (2) $x_{i+2} \Rightarrow a_{2i}$, (3) $y_{i-1} \wedge x_{i+1} \Rightarrow a_{2i}$, (4) $y_{i-1} \wedge x_i \Rightarrow a_{2i-1}$ and (5) $y_{i-2} \wedge x_{i+1} \Rightarrow a_{2i-1}$ or, otherwise: (1) $a_{2i} \Rightarrow y_{i-1} \vee x_{i+2}$, (2) $a_{2i} \Rightarrow y_i \vee x_{i+1}$, (3) $a_{2i-1} \Rightarrow x_i$, (4) $a_{2i-1} \Rightarrow y_{i-2}$ and (5) $a_{2i-1} \Rightarrow y_{i-1} \vee x_{i+1}$. If each comparator is encoded separately, we need 4 new variables and 6 clauses for a pair of comparators. Therefore, we can save about $k$ new variables and $k/2$ clauses for each $oe\_4combine_k^{4k}$. This is the main advantage of using 4-way mergers instead of odd-even mergers.

*Example 4.* In Figure 2, in dashed lines, a schema of 4-Odd-Even merger is presented with $s = 9$, $k = 3$, $k_1 = 3$ and $k_2 = k_3 = k_4 = 2$. First, the input columns are split into two by odd and even indexes, and the recursive calls are made. After that, a combine operation fixes the order of elements, to output the 3 largest ones. For more detailed example of Network 3, assume that $k = 6$ and $\bar{w} = 100000$, $\bar{x} = 111000$, $\bar{y} = 100000$, $\bar{z} = 100000$. Then $\bar{a} = oe\_4merge_5^{12}(100,110,100,100) = 111110000000$ and $\bar{b} = oe\_4merge_3^{12}(000,100,000,000) = 100000000000$. The combine operation gets $\bar{x} = \text{pref}(5,\bar{a}) = 11111$ and $\bar{y} = \text{pref}(3,\bar{b}) = 100$. Notice that $|\bar{x}|_1 - |\bar{y}|_1 = 4$ and after zipping we get 11101011. Thus, two comparators from the first set are needed to fix the order.

**Theorem 2.** *The output of Network 3 is top k sorted.*

Due to the space limitations a proof will be given in the full version of the paper.

## 4 Comparison of Odd-Even Selection Networks

In this section we would like to estimate and compare the number of variables and clauses in encodings based on our algorithm to other encoding based on odd-even selection. Such encoding – which we call 2-Odd-Even Selection Network – was already analyzed by Codish and Zazon-Ivry [8]. We start by counting how many variables and clauses are needed in order to merge 4 sorted sequences returned by recursive calls of 4-Odd-Even Selection Network and 2-Odd-Even Selection Network. Then, based on those values we prove that overall number of variables and clauses is almost always smaller when using 4-column encoding rather than 2-column encoding. In the next section we show that the new encoding is not just smaller, but also have better solving times in many benchmark instances.

To simplify the presentation we assume that $k \leq n/4$ and both $k$ and $n$ are the powers of 4. We also omit the ceiling and floor function in the calculations, when it is convenient for us.

**Definition 4.** *Let $n,k \in \mathbb{N}$. For given (selection) network $f_k^n$ let $V(f_k^n)$ and $C(f_k^n)$ denote the number of variables and clauses used in the standard CNF encoding of $f_k^n$.*

We remind the reader that a single 2-comparator uses 2 auxiliary variables and 3 clauses. In case of a 4-comparator the numbers are 4 and 15.

We count how many variables and clauses are needed in order to merge 4 sorted sequences returned by recursive calls of 2-Odd-Even Selection Network and 4-Odd-Even Selection Network, respectively. Two-column selection network using odd-even approach is presented in [8]. We briefly introduce this network with the following three-step recursive procedure (omitting the base case):

1. Split the input $\bar{x} \in \{0,1\}^n$ into two sequences $\bar{x}^1 = \bar{x}_{odd}$ and $\bar{x}^2 = \bar{x}_{even}$.
2. Recursively select top $k$ sorted elements from $\bar{x}^1$ and top $k$ sorted elements from $\bar{x}^2$.
3. Merge the outputs of the previous step using an 2-Odd-Even Merging Network of order $(2k,k)$ and output the top $k$ from $2k$ elements.

If we treat the merging step as a network $oe\_2merge_k^{2k}$, then the number of 2-comparators used in the 2-Odd-Even Selection Network of order $(n,k)$ can be written as:

$$|oe\_2sel_k^n| = \begin{cases} 2|oe\_2sel_k^{n/2}| + |oe\_2merge_k^{2k}| & \text{if } k < n \\ |oe\_sort^k| & \text{if } k = n \\ |max^n| & \text{if } k = 1 \end{cases} \tag{1}$$

One can check that Step 3 requires $|oe\_2merge_k^{2k}| = k\log k + 1$ 2-comparators (see [8]), which leads to the simple lemma.

**Lemma 1.** $V(oe\_2merge_k^{2k}) = 2k\log k + 2$, $C(oe\_2merge_k^{2k}) = 3k\log k + 3$.

The schema of this network is presented in Figure 3. In order to count the number of comparators used in merging 4 sorted sequences we need to expand the recursive step by one level (see Figure 3b).

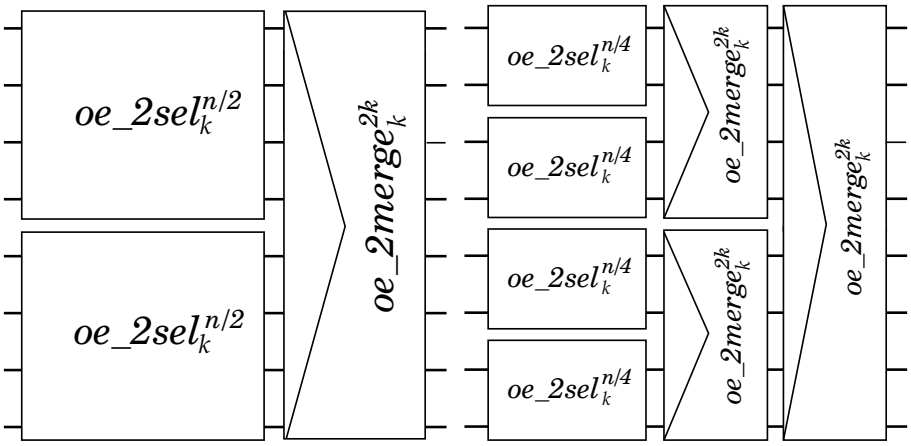
Now we do the counting for our 4-way merging network based on Network 3.

**Lemma 2.** *Let $k \in \mathbb{N}$, then: $V(oe\_4merge_k^{4k}) \leq (k-2)\log k + 5k - 1$; $C(oe\_4merge_k^{4k})$ $\leq (\frac{5}{2}k - 5)\log k + 21k - 6$.*

*Proof.* We separately count the number variables and clauses used.

In the base case (line 2) we can assume – for the sake of the upper bound – that we always use 4-comparators. Notice, that the number of 4-comparators is only dependent on the variable $s$. The solution to the following recurrence gives the sought number: $\{A(4) = 1; A(s) = 2A(s/2), \text{for } s > 4\}$, which is equal to $s/4$. Therefore we use $s$ auxiliary variables and $(15/4)s$ clauses. We treat the recursive case separately below.

The number of variables used in the combine network is at most $k-1$, because a new variable is not needed for $a_i$, where $i > k$, because such $a_i$ can be replaced by a zero in clauses containing it, and not for $a_1 = x_1$. Therefore, the total number of variables is bounded by solution to the following recurrence:




(a) one step (b) two steps

**Fig. 3.** 2-Odd-Even Selection Network

$$B(s,k) = \begin{cases} 0 & \text{if } s \leq 4 \\ B(s_a,k_a) + B(s_b,k_b) + k - 1 & \text{otherwise} \end{cases}$$

where $k \leq s = s_a + s_b \leq 4k$, $s_b \leq s_a \leq s_b + 4$ and $k_a = \min(s_a, \lfloor k/2 \rfloor + 2)$ and $k_b = \min(s_b, \lfloor k/2 \rfloor)$. Therefore $s/2 \leq s_a \leq s/2 + 2$, $s/2 - 2 \leq s_b \leq s/2$, $k_a \leq k/2 + 2$ and $k_b \leq k/2$. We claim that $B(s,k) \leq (k-2)(\log s - 2) + \frac{1}{4}s - 1$. This can be easily verified by induction.

The upper bound of the number of clauses can now be easily computed noticing that in the combine we require either 2 or 3 clauses for each new variable (depending on the parity of the index), therefore the number of clauses in the combiner is bounded by $2.5 \times \#vars + 3.5$. Constant factor 3 is added because additional clauses can be added for values $a_{k+1}$ and $a_{k+2}$ (see equations in Section 3.2). The overall number of clauses in the merger (omitting base cases) is then at most $2.5 \cdot B(s,k) + 3.5(k-1)$, where factor $(k-1)$ is the upper bound on the number of combines used in the recursive tree of the merger. Elementary calculations give the desired result. □

Combining Lemmas 1 and 2 gives the following corollary.

**Corollary 1.** *Let $k \in \mathbb{N}$. Then $3V(oe\_2merge_k^{2k}) - V(oe\_4merge_k^{4k}) \geq (5k+2)\log(\frac{k}{2}) + 9 \geq 0$, and for $k \geq 8$, $3C(oe\_2merge_k^{2k}) - C(oe\_4merge_k^{4k}) \geq (\frac{13}{2}k + 5)\log(\frac{k}{8}) - \frac{3}{2}k + 30 \geq 0$.*

This shows that using our merging procedure gives a smaller encoding than its 2-column counterpart and the differences in the number variables and clauses used is significant.

The main result of this section is as follows.

**Theorem 3.** *Let $n,k \in \mathbb{N}$ such that $1 \leq k \leq n/4$ and n and k are both powers of 4. Then:*

$$dsV_k(n) = V(oe\_2sel_k^n) - V(oe\_4sel_k^n) \geq \frac{(n-k)(5k+2)}{3k} \log\left(\frac{k}{2}\right) + 3\left(\frac{n}{k} - 1\right).$$

*Proof.* Let $dV_k = 3V(oe\_2merge_k^{2k}) - V(oe\_4merge_k^{4k})$ (from Corollary 1), then:

$$\begin{aligned} dsV_k(n) &= V(oe\_2sel_k^n) - V(oe\_4sel_k^n) \\ &= 2V(oe\_2sel_k^{n/2}) + V(oe\_2merge_k^{2k}) - 4V(oe\_4sel_k^{n/4}) - V(oe\_4merge_k^{4k}) \\ &= 4V(oe\_2sel_k^{n/4}) + 3V(oe\_2merge_k^{2k}) - 4V(oe\_4sel_k^{n/4}) - V(oe\_4merge_k^{4k}) \\ &= 4dsV_k(n/4) + dV_k \end{aligned}$$

The solution to the above recurrence is $dsV_k(n) \geq \frac{1}{3}(\frac{n}{k} - 1)dV_k$. Therefore:

$$\begin{aligned} dsV_k(n) &\geq \frac{1}{3}\left(\frac{n}{k} - 1\right)\left((5k+2)\log\left(\frac{k}{2}\right) + 9\right) \\ &= \frac{(n-k)(5k+2)}{3k}\log\left(\frac{k}{2}\right) + 3\left(\frac{n}{k} - 1\right). \end{aligned}$$

□

Similar theorem can be proved for the number of clauses (when $k \geq 8$).

## 5 Experimental Evaluation

As it was observed in [1], having a smaller encoding in terms of number of variables or clauses is not always beneficial in practice, as it should also be accompanied with a reduction of SAT-solver runtime. In this section we assess how our encoding based on the new family of selection networks affect the performance of a SAT-solver.

### 5.1 Methodology

Our algorithms that encode CNF instances with cardinality constraints into CNFs were implemented as an extension of MINICARD ver. 1.1, created by Mark Liffiton and Jordyn Maglalang[1]. MINICARD uses three types of solvers:

- *minicard* - the core MINICARD solver with native AtMost constraints,
- *minicard_encodings* - a cardinality solver using CNF encodings for AtMost constraints,
- *minicard_simp_encodings* - the above solver with simplification / pre-processing.

[1] See https://github.com/liffiton/minicard

The main program in *minicard_encodings* has an option to generate a CNF formula, given a CNFP instance (CNF with the set of cardinality constraints) and to select a type of encoding applied to cardinality constraints. Program run with this option outputs a CNF instance that consists of collection of the original clauses with the conjunction of CNFs generated by given method for each cardinality constraint. No additional preprocessing and/or simplifications are made. Authors of *minicard_encodings* have implemented six methods to encode cardinality constraints and arranged them in one library called *Encodings.h*. Our modification of MINICARD is that we added implementation of the encoding presented in this paper and put it in the library *Encodings_MW.h*. Then, for each CNFP instance and each encoding method, we used MINICARD to generate CNF instances. After preparing, the benchmarks were run on a different SAT-solver. Our extension of MINICARD, which we call KP-MINICARD, is available online[2].

In our evaluation we use the state-of-the-art SAT-solver COMINISATPS by Chanseok Oh[3] [14], which have collectively won six medals in SAT Competition 2014 and Configurable SAT Solver Challenge 2014. Moreover, the modification of this solver called MAPLECOMSPS won the Main Track category of SAT Competition 2016[4]. All experiments were carried out on the machines with Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz.

Detailed results are available online[5]. We publish spreadsheets showing running time for each instance, speed-up/slow-down tables for our encodings, number of timeouts met and total running time.

### 5.2 Encodings

We use our multi-column selection network for evaluation – the 4-Odd-Even Selection Network (**4OE**) based on Networks 1, 2 and 3. We compare our encoding to some others found in the literature. We consider the Pairwise Cardinality Networks [8]. We also consider a solver called MINISAT+[6] which implements techniques to encode Pseudo-Boolean constraints to propositional formulas [9]. Since cardinality constraints are a subclass of Pseudo-Boolean constraints, we can measure how well the encodings used in MINISAT+ perform, compared with our methods. The solver chooses between three techniques to generate SAT encodings for Pseudo-Boolean constraints. These convert the constraint to: a BDD structure, a network of binary adders, a network of sorters. The network of adders is the most concise encoding, but it can have poor propagation properties and often leads to longer computations than the BDD based encoding. The network of sorters is the implementation of classic odd-even (2-column) sorting network by Batcher [5]. Calling the solver we can choose the encoding with one of the parameters: *-ca*, *-cb*, *-cs*. By default, MINISAT+ uses the so called **Mixed** strategy, where program chooses which method (adders, BDDs or sorters) to use in the encodings. We don't include **Mixed** strategy in the results, as the evaluation showed that it performs

[2] See https://github.com/karpiu/kp-minicard
[3] See http://cs.nyu.edu/%7echanseok/cominisatps/
[4] See http://baldur.iti.kit.edu/sat-competition-2016/
[5] See http://www.ii.uni.wroc.pl/%7ekarp/sat/2018.html
[6] See https://github.com/niklasso/minisatp

almost the same as *-cb* option. The generated CNFs were written to files with the option *-cnf=<file>*. Solver MINISAT+ have been slightly modified, namely, we fixed a pair of bugs such as the one reported in the experiments section of [4].

To sum up, here are the competitors' encodings used in this evaluation:

- **PCN** - the Pairwise Cardinality Networks (our implementation),
- **CA** - encodings based on Binary Adders (from MINISAT+),
- **CB** - encodings based on Binary Decision Diagrams (from MINISAT+),
- **CS** - the 2-Odd-Even Sorting Networks (from MINISAT+).

Encodings **4OE** and **PCN** were extended, following the idea presented in [1], where authors use Direct Cardinality Networks in their encodings for sufficiently small values of $n$ and $k$. Values of $n$ and $k$ for which we substitute the recursive calls with Direct Cardinality Network were selected based on the optimization idea in [1]. We minimize the function $\lambda \cdot V + C$, where $V$ is the number of variables and $C$ the number of clauses to determine when to switch to direct networks, and following authors' experimental findings, we set $\lambda = 5$.

Additionally, we compare our encodings with two state-of-the-art general purpose constraint solvers. First is the PBLIB ver. 1.2.1, by Tobias Philipp and Peter Steinke [16]. This solver implements a plethora of encodings for three types of constraints: at-most-one, at-most-k (cardinality constraints) and Pseudo-Boolean constraints. The PBLIB automatically normalizes the input constraints and decides which encoder provides the most effective translation. One of the implemented encodings for at-most-k constraints is based on the sorting network from the paper by Abío et al. [1]. One part of the PBLIB library is the program called *PBEncoder* which takes an input file and translate it into CNF using the PBLIB. We have generated CNF formulas from all benchmark instances using this program, then we have run COMINISATPS on those CNFs. Results for this method are labeled **PBE** in our evaluation.

The second solver is the NPSOLVER by Norbert Manthey and Peter Steinke[7], which is a Pseudo-Boolean solver that translates Pseudo-Boolean constraints to SAT similar to MINISAT+, but which incorporates novel techniques. We have exchanged the SAT-solver used by default in NPSOLVER to COMINISATPS because the results were better with this one. Results for this method are labeled **NPS** in our evaluation.

### 5.3 Benchmarks

The set of benchmarks we used is **PB15 suite**, which is a set of instances from the Pseudo-Boolean Evaluation 2015[8]. One of the categories of the competition was *DEC-LIN-32-CARD*, which contains 2289 instances – we use these in our evaluation. Every instance is a collection of cardinality constraints.

[7] See http://tools.computational-logic.org/content/npSolver.php
[8] See http://pbeva.computational-logic.org/

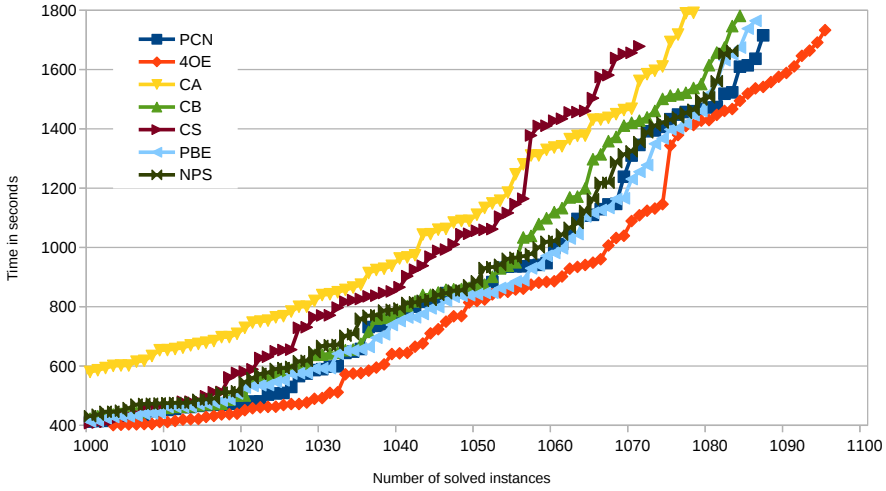



**Fig. 4.** The number of solved instances of PB15 suite in given time.

### 5.4 Results

The time-out limit in the SAT-solver was set to 1800 seconds. When comparing two encodings we only considered instances for which at least one achieved the SAT-solver runtime of at least 10% of the time-out limit. All other instances were considered trivial, and therefore were not included in the speed-up/slow-down results. We also filtered out instances for which relative percentage deviation of the running time of encoding **A** w.r.t. the running time of encoding **B** was less than 10% (and vice-versa).

In Figure 4 we present a cactus plot, where x-axis gives the number of solved instances of PB15 suite and the y-axis the time needed to solve them (in seconds) using given encoding. From the plot we can see that the **4OE** encoding outperforms all other encodings.

Table 1 presents speed-up and slow-down factors for encoding **4OE** w.r.t. all other encodings. From the evaluation we can conclude that the best performing encoding is **4OE**. From the data presented in Table 1 our encoding achieve better speed-up factor w.r.t. all other encodings. Total running time for **4OE** is 629.78 hours on all 2289 instances. All other encodings required more time to finish the computation. Also, **4OE** solved the most number of instances – 1095. The second to last column of Table 1 shows the difference in total running time of all encodings w.r.t. **4OE** (in HH:MM format – hours and minutes). The last column indicates the difference in the number of solved instances of all encodings w.r.t. **4OE** (here all instances are counted, even the trivial ones). We can see, for example, that for **4OE** computations finished about 7 hours sooner for **4OE** than **CS**. This shows that using 4-column selection networks is more desirable than using 2-column selection/sorting networks for encoding cardinality constraints. Encodings **CA** and **CS** had the worst performance on PB15 suite. We can also see that even the state-of-the-art constraint solvers have larger running times and solved

| | 4OE speed-up | | | | | | 4OE slow-down | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TO | 4.0 | 2.0 | 1.5 | 1.1 | Total | TO | 4.0 | 2.0 | 1.5 | 1.1 | Total | Time dif. | #s dif. |
| PCN | 9 | 11 | 5 | 5 | 5 | 35 | 1 | 2 | 1 | 3 | 3 | 10 | +02:55 | -8 |
| CA | 22 | 15 | 28 | 21 | 21 | 107 | 5 | 3 | 5 | 4 | 11 | 28 | +10:54 | -17 |
| CB | 18 | 11 | 15 | 8 | 24 | 76 | 7 | 2 | 6 | 3 | 28 | 46 | +04:54 | -11 |
| CS | 27 | 13 | 14 | 13 | 18 | 85 | 3 | 0 | 18 | 14 | 13 | 48 | +06:55 | -24 |
| PBE | 15 | 13 | 10 | 10 | 20 | 68 | 6 | 16 | 5 | 6 | 27 | 60 | +02:48 | -9 |
| NPS | 17 | 15 | 7 | 11 | 29 | 67 | 5 | 16 | 6 | 5 | 26 | 58 | +03:51 | -12 |

**Table 1.** Comparison of encodings in terms of SAT-solver runtime on PB15 suite. We count number of benchmarks for which **4OE** showed speed-up or slow-down factor with respect to different encodings, the difference in total running time of each encoding w.r.t. **4OE** and the difference in the number of solved instances of each encoding w.r.t. **4OE**.

less instances on this set of benchmarks, as **PBE** and **NPS** finished computations more than about 3–4 hours later than **4OE**.

## 6 Conclusions

In this paper we presented a multi-column selection network based on odd-even approach, that can be used to encode cardinality constraints. We showed that its CNF encoding is smaller than the 2-column version. We extended the encoding by applying the Direct Cardinality Networks of [1] for sufficiently small input. The new encoding was compared with the selected state-of-the-art encodings based on comparator networks, adders and binary decision diagrams as well as with two popular general constraints solvers. The experimental evaluation shows that the new encoding yields better speed-up and overall runtime in the SAT-solver performance.

Developing new methods to encode cardinality constraints based on comparator networks is important from the practical point of view. Using such encodings gives an extra edge in solving optimization problems for which we need to solve a series of problems that differ only in that a bound on cardinality constraint $x_1 + \cdots + x_n \leq k$ becomes tighter, i.e., by decreasing $k$ to $k'$. In this setting we only need to add one more clause $\neg y_{k'}$, and the computation can be resumed while keeping all the previous clauses untouched. This operation is allowed because if a comparator network is a $k$-selection network, then it is also a $k'$-selection network, for any $k' < k$. This property is called *incremental strengthening* and most state-of-art SAT-solvers provide an interface for doing this.

On a final note, we would like to point out that the construction can be used to encode *Pseudo-Boolean constraints*, which are more expressive than cardinality constraints. Application of sorting networks in this setup was already reported by Eén and Sörensson [9] and others. Applying our algorithm instead of standard Odd-Even or Pairwise Sorting Networks can lead to increase in the number of solved instances. In fact, we have developed a PB-solver which we call KP-MINISAT+ [11], which uses the algorithm presented in this paper. The code is available online[9] as well as results of the experimental evaluation[10].

[9] See https://github.com/karpiu/kp-minisatp

[10] See http://www.ii.uni.wroc.pl/%7ekarp/pos/2018.html

## References


1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints. In: C. Schulte, (ed), Principles and Practice of Constraint Programming - CP 2013, LNCS, vol. 8124, pp. 80–96. Springer Heidelberg, (2013).
2. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks: a theoretical and empirical study. Constraints, 16(2):195–221, (2011).
3. Asín, R., Nieuwenhuis, R.: Curriculum-based course timetabling with SAT and MaxSAT. Annals of Operations Research, 218(1):71–91, (2014).
4. Aavani, A., Mitchell, D.G., Ternovska, E.: New encoding for translating pseudo-Boolean constraints into SAT. In: Frisch, A.M., Gregory, P. (eds.) SARA, AAAI (2013).
5. K. E. Batcher.: Sorting networks and their applications. In: Proc. of the April 30–May 2, 1968, Spring Joint Computer Conference, AFIPS '68 (Spring), pp. 307–314, ACM, New York, NY, USA, (1968).
6. K. E. Batcher, Lee, D.: A Multiway Merge Sorting Network. In: IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 2, February 1995, pp. 211–215.
7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Proc. of 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), LNCS vol. 1579, pp. 193–207, Springer Heidelberg (1999).
8. Codish, M., Zazon-Ivry, M.: Pairwise cardinality networks. In: E. Clarke and A. Voronkov, (eds), Logic for Programming, Artificial Intelligence, and Reasoning, LNCS vol. 6355, pp. 154–172. Springer Heidelberg (2010).
9. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation, 2:1–26, (2006).
10. Karpiński, M.: Encoding cardinality constraints using standard encoding of generalized selection networks preserves arc-consistency. Theoretical Computer Science, vol. 707, pp. 77–81, (2018).
11. Karpiński, M., Piotrów, M.: Competitive Sorter-based Encoding of PB-Constraints into SAT. To appear in the proceedings of Pragmatics of SAT 2018 Workshop, (2018).
12. Karpiński, M., Piotrów, M.: Smaller Selection Networks for Cardinality Constraints Encoding. In: G. Pesant, (ed), Principles and Practice of Constraint Programming - CP 2015, LNCS, vol. 9255, pp. 210-225. Springer International Publishing, (2015).
13. Knuth, D. E.: The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, (1998).
14. Oh, Ch.: Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL. PHD thesis, New York University, (2016).
15. Parberry, I.: The pairwise sorting network. Parallel Processing Letters, 2:205–211, (1992).
16. Philipp, T., Steinke, P.: PBLib – A Library for Encoding Pseudo-Boolean Constraints into CNF. Theory and Applications of Satisfiability Testing – SAT 2015, LNCS, vol. 9340, pp. 9–16. Springer International Publishing, (2015).
17. Schutt, A., Feydy, T., Stuckey, P., Wallace, M.: Why cumulative decomposition is not as bad as it sounds. In: I. Gent, (ed), Principles and Practice of Constraint Programming - CP 2009, LNCS, vol. 5732, pp. 746–761. Springer Heidelberg, (2009).