Detecting cardinality constraints in CNF

Armin Biere[1], Daniel Le Berre[2], Emmanuel Lonca[2], and Norbert Manthey[3]

[1] Johannes Kepler University
[2] CNRS Université d'Artois
[3] Technische Universität Dresden

Abstract. We present novel approaches to detect cardinality constraints expressed in CNF. The first approach is based on a syntactic analysis of specific data structures used in SAT solvers to represent binary and ternary clauses, whereas the second approach is based on a semantic analysis by unit propagation. The syntactic approach computes an approximation of the cardinality constraints AtMost-1 and AtMost-2 constraints very fast, whereas the semantic approach has the property to be generic, i.e. it can detect cardinality constraints AtMost-k for any k, at a higher computation cost. Our experimental results suggest that both approaches are efficient at recovering AtMost-1 and AtMost-2 cardinality constraints.

1 Introduction

Current benchmarks in CNF contain various Boolean functions encoded with clauses [30,15]. Among them, cardinality constraints $\sum_{i=1}^{n} l_i \otimes k$ with $\otimes \in \{<, \leq, =, \geq, >\}$ are Boolean functions whose satisfiability is determined by counting the satisfied literals on the left hand side and compare them to the right hand side (the *threshold*). For instance, $x_1 + x_2 + \neg x_3 + \neg x_4 \leq 2$ is satisfied iff at most 2 of its literals are satisfied. A wide use case of those constraints is to encode that a domain variable v takes one value of the discrete set $\{o_1, o_2, \ldots, o_n\}$, which is represented by the n Boolean variables v_{o_i} and the cardinality constraint $\sum v_{o_i} = 1$.

Since cardinality constraints are Boolean functions, they can be expressed by an equivalent CNF. The "theoretical" approach, i.e. the one found in [12] for instance, translates a cardinality constraint $\sum_{i=1}^{n} l_i \leq k$ using $\binom{n}{k+1}$ negative clauses of size $k+1$. Such encoding is called *binomial* because of the number of generated clauses. In practice, introducing new variables to reduce the number of clauses in the CNF usually results in a better performance. Various encodings have been proposed in the last decade (see for instance [14] for a survey). We discuss commonly used encodings in next section.

Pseudo-Boolean solvers use a proof system like *generalized resolution* [21], which is a specific form of the *cutting planes* proof system [12] that *p-simulates* resolution. This way, these solvers are able to solve instances of the Pigeon Hole Principle [19] when they are given cardinality constraints but not when they are given the same problem expressed with clauses. The reason of that behavior is

that applying generalized resolution on clauses is equivalent to resolution [21], while on cardinality constraints generalized resolution is a specific form of cutting planes [12]. Retrieving cardinality constraints from clauses in the cutting planes proof system requires a very specific procedure. Take for instance the cardinality constraint

$$x_1 + x_2 + x_3 + x_4 \leq 1$$

which is equivalent to

$$\overline{x_1} + \overline{x_2} + \overline{x_3} + \overline{x_4} \geq 3$$

This cardinality constraint is represented in CNF using the following clauses:

$$\neg x_1 \vee \neg x_2, \quad \neg x_1 \vee \neg x_3, \quad \neg x_1 \vee \neg x_4, \quad \neg x_2 \vee \neg x_3, \quad \neg x_2 \vee \neg x_4, \quad \neg x_3 \vee \neg x_4$$

These clauses can be represented as binary cardinality constraints:

$$x_1 + x_2 \leq 1, \quad x_1 + x_3 \leq 1, \quad x_1 + x_4 \leq 1, \quad x_2 + x_3 \leq 1, \quad x_2 + x_4 \leq 1, \quad x_3 + x_4 \leq 1$$

Retrieving the original cardinality from the clauses represented by cardinalities ≤ 1 requires to derive intermediate constraints as shown below (from [12]):

$$\frac{\begin{array}{c} x_1 + x_2 \leq 1 \\ x_1 + x_3 \leq 1 \\ x_2 + x_3 \leq 1 \end{array}}{x_1 + x_2 + x_3 \leq 1} \qquad \frac{\begin{array}{c} x_1 + x_2 \leq 1 \\ x_1 + x_4 \leq 1 \\ x_2 + x_4 \leq 1 \end{array}}{x_1 + x_2 + x_4 \leq 1} \qquad \frac{\begin{array}{c} x_1 + x_3 \leq 1 \\ x_1 + x_4 \leq 1 \\ x_3 + x_4 \leq 1 \end{array}}{x_1 + x_3 + x_4 \leq 1} \qquad \frac{\begin{array}{c} x_2 + x_3 \leq 1 \\ x_2 + x_4 \leq 1 \\ x_3 + x_4 \leq 1 \end{array}}{x_2 + x_3 + x_4 \leq 1}$$

For the first column, summing the three cardinality constraints leads to $2x_1 + 2x_2 + 2x_3 \leq 3$, which can be reduced to $x_1 + x_2 + x_3 \leq 1$ by dividing the inequality by 2 and rounding down the threshold. The same process can be applied to derive the other cardinality constraints in the last line. Finally, summing up these four cardinality constraints of 3 literals results in a cardinality constraint of 4 literals: $3x_1 + 3x_2 + 3x_3 + 3x_4 \leq 4$. The expected cardinality constraint $x_1 + x_2 + x_3 + x_4 \leq \lfloor \frac{4}{3} \rfloor$ is obtained after division by 3 and rounding.

The described process is tedious and not easy to integrate in a solver. Thus, the idea is to find a way to detect those cardinality constraints in a preprocessing step, independent from the original proof system of the solver.

The motivation for this work is to allow solvers to take advantage of those cardinality constraints, at least for space efficiency (support of native cardinality constraints) or because of a better proof system (e.g. Generalized Resolution [21] or Cutting Planes [12]). Detecting cardinality constraints is also an interesting idea for pure SAT solvers, namely for constraints reencoding, e.g. to encode cardinality constraints back to CNF with an alternative and hopefully more efficient encoding [27,26]. This is especially useful in practice to replace the commonly used *pairwise* encoding of ≤ 1 constraints with a more efficient encoding.

2 Short Review of Known Encodings

Before we discuss how to find encoded cardinality constraints, a few common encodings for widely used constraints are introduced. For the AtMost-1 constraint

$\sum_{i=1}^{n} x_i \leq 1$ the *naïve encoding*, also known as *pairwise encoding*, is to exclude each pair of satisfied literals explicitly: $\bigwedge_{i=1}^{n} \bigwedge_{j>i}^{n} (\neg x_i \vee \neg x_j)$. This way, a constraint with n variables requires $\frac{n(n-1)}{2}$ clauses. This encoding is also referred to as *direct encoding* in the CP community [33].

The *nested encoding* uses auxiliary variables to reduce the number of generated clauses from a quadratic number can to a linear number, by (recursively) splitting the constraint into two constraints:

$$\sum_{i=1}^{n} x_i \leq 1 = [y + (\sum_{i=1}^{\lfloor \frac{n}{2} \rfloor - 1} x_i) \leq 1] \wedge [\neg y + (\sum_{i=\lfloor \frac{n}{2} \rfloor - 1}^{n} x_i) \leq 1].$$

For $n = 4$, the naïve encoding requires six clauses, and the nested encoding requires six clauses as well, but has more variables. Hence, as soon as the number of variables for an AtMost-1 constraint is at most four, no more recursions are applied. This way, the nested encoding requires $3n - 6$ clauses.

The currently best known asymptotic (starting from $n > 47$ [26]) encoding for the AtMost-1 constraint is the *two product encoding* [11]. For n variables in the constraint, two integers $p = \lfloor \sqrt{n} \rfloor$ and $q = \lceil \frac{n}{p} \rceil$ are used to create two more AtMost-1 constraints: $\sum_{i=1}^{p} r_i \leq 1$ and $\sum_{i=1}^{q} c_i \leq 1$. These two constraints are used as selector for a row and a column. The variables x_i are placed in a matrix, such that each variable x_i is assigned exactly to one row selector r_s and to one column selector c_t with the clauses $(\neg x_i \vee r_s)$ and $(\neg x_i \vee c_t)$, where $s = \lfloor \frac{i-1}{q} \rfloor + 1$ and $t = ((i-1) \mod q) + 1)$. An illustration for 10 variables is given in Fig. 1.

Further proposed encodings for the AtMost-1 constraint are the log encoding [33], the ladder encoding [18,17] also defined independently in [3], the commander encoding [23], generalizations of the log encoding and the two-product encoding [14], the bimander encoding [20], as well as generalizations of the bimander encoding [7].

For cardinality constraints $\sum_{i=1}^{n} x_i \leq k$ with a higher threshold $k > 1$, many encodings have been presented. Well known and sophisticated encodings are the partial sum encoding [1], totalizer encoding [6], the sequential counter encoding [31], BDDs [13] or sorting networks [13], cardinality networks [5], as well as the perfect hashing encoding [9]. As shown in [26], these specialized encodings produce much smaller CNF formulas compared to the binomial encoding. However, it is not clear whether smaller is better in all contexts. A recent survey on practical efficiency of those encodings in the context of MaxSAT solving is available in [28].

3 Static Detection of AtMost-1 and AtMost-2 Constraints

The naïve encoding of the AtMost-1 constraint can be detected by a syntactic analysis of the formula, namely by finding cliques in the *NAND graph* (NAG) of the formula, which is the undirected graph connecting literals that occur negated in the same binary clause. In [4,2], the authors modified the solvers zChaff and Satz to recognize those constraints using unit propagation and local search. A specific data structure for binary clauses is often found in modern

SAT solvers to reduce the memory consumption of the solver. From such a graph AtMost-1 constraints can be extracted by syntactic analysis. The naïve encoding of the AtMost-2 constraint can be recognized by exploring ternary clauses. The tools 3MCard [24], Lingeling [10] and SBSAT [34] can recover cardinality constraints based on a syntactic analysis, and hence their methods are presented below additionally to the new extraction method. Both 3MCard and SBSAT do not restrict their search on clauses of special size, but consider the whole formula: SBSAT constructs BDDs based on clauses that share the same variables. By merging and analyzing these BDDs cardinality constraints can be detected [34]. The tool 3MCard builds a graph based on the binary clauses, and increases the current constraint while collecting more clauses [24]. Only Lingeling has special methods to extract AtMost-1 constraints, and AtMost-2 constraints.

3.1 Detecting the Pairwise Encoding

The structure of the pairwise encoding on the NAG is quite simple: if a clique is present in that graph, then the literals of the corresponding nodes form an AtMost-1 constraint. Since finding a clique of size k in a graph is NP-complete [22], a preprocessing step should not perform a full clique search. The algorithm for greedily finding cliques as implemented in Lingeling goes over all literals n which have not been included in an AtMost-1 constraint yet. For each n the set S of candidate literals is initialized with n. Then all literals l which occur negated in binary clauses $\bar{n} \vee \bar{l}$ together with n, e.g. l and n are connected in the NAG, are considered, in an arbitrary order, and greedily added to S after checking that for each previously added $k \in S$ a binary clause $\bar{l} \vee \bar{k}$ is also present in the formula, e.g. l and k have an edge in the NAG too. As an optimization, literals k which already occur in previously extracted AtMost-1 constraints are skipped. The final set S of nodes forms a clique in the graph. If $|S| > 2$ the clique is non-trivial and the AtMost-1 constraint $\sum_{l \in S} \leq 1$ is added [10].

3.2 Detecting the Nested Encoding

Consider the nested encoding of the AtMost-1 constraint $x_1 + x_2 + x_4 + x_5 \leq 1$, where the constraint is divided into the cardinality constraints $x_1 + x_2 + x_3 \leq 1$ and $\neg x_3 + x_4 + x_5 \leq 1$. They are represented in CNF by the six clauses $(\neg x_1 \vee \neg x_2), (\neg x_1 \vee \neg x_3), (\neg x_2 \vee \neg x_3), (x_3 \vee \neg x_4), (x_3 \vee \neg x_5), (\neg x_4 \vee \neg x_5)$. Since there is no binary clause $(\neg x_1 \vee \neg x_4)$, the above method cannot find this encoding. Here, we present another method that recognizes this encoding. The two smaller constraints can be recognized with the above method (their literals form two cliques in the NAG). Then, there is an AtMost-1 constraint for the literal x_3, as well as for the literal $\neg x_3$. By resolving the two constraints, the original constraint can be obtained. Algorithm 1 searches for exactly this encoding by combining pairs of constraints. For each variable v, all AtMost-1 constraints with different polarity are added and simplified. As a simplification it is checked, whether duplicate literals occur, or whether complementary literals

Algorithm 1: Merge AtMost-1

Input: A set of "at most 1" cardinality constraints S, the set of variables V
Output: An extended set of "at most 1" cardinality constraints

```
1 foreach v ∈ V do
2 |   foreach A ∈ S_v do
3 |   |   foreach B ∈ S_¬v do
4 |   |   |   S := S ∪ simplify(A + B)
5 |   |   end
6 |   end
7 end
8 return S ;
```

occur. In the former case, the duplicated literal has to be assigned false, because that literal has now a weight of two in that constraint, while the threshold is 1. In the latter case, all literals of the constraint $(A+B)$, except the complementary literal, has to be falsified (because $x + \overline{x} = 1$, so the threshold is reduced by one to zero). The simplified constraint is added to the set of AtMost-1 constraint, which is finally returned by the algorithm.

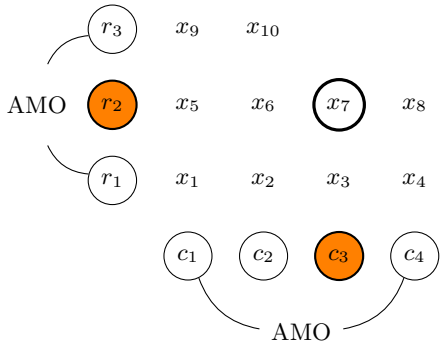
Since the nested encoding can be encoded recursively, the algorithm can be called multiple times to find these recursive encodings. To not resolve the same constraints multiple times, for each variable the already *seen* constraints can be memorized, so that in a new iteration only resolutions with new constraints are performed. In practice, our implementation loops over the variables in ascending order exactly once. This seems to be sufficient, because the recursive encoding of constraints requires that the "fresh" variable is not present yet, so that the ascending order in the variable finds this encoding nicely.

3.3 Detecting the Two-Product Encoding

The two product encoding has a similar recursive structure as the nested encoding, however, its structure is more complex. Hence, this encoding is discussed in more details. The constraint in Fig. 1 illustrates an AtMost-1 constraint that is encoded with the two-product encoding.

For all concerned literals, in the example x_1 to x_{10}, two implications are added to set the column and row selectors. For example, as x_7 is on the second row and the fourth column, the constraints $x_7 \rightarrow r_2$ and $x_7 \rightarrow c_3$ are added. In order to prevent two rows or two columns selectors to be set simultaneously, we also add AtMost-1 cardinality constraints on the c_i and on the r_i literals. Those new cardinality constraints are encoded using the pairwise encoding if their size is low, or using the two product encoding. As the product encoding of AtMost-1 constraints may generate other AtMost-1 constraints to be encoding in the same way, the algorithm may be written in a recursive way.

In the given constraint, the following implications to select a column and a row for x_7 are entailed by the encoding: $x_7 \rightarrow c_3$ and $x_7 \rightarrow r_2$. Additionally,



Selection Clauses

$(\neg x_1 \vee r_1)$ $(\neg x_1 \vee c_1)$ $(\neg x_2 \vee r_1)$
$(\neg x_2 \vee c_2)$ $(\neg x_3 \vee r_1)$ $(\neg x_3 \vee c_3)$
$(\neg x_4 \vee r_1)$ $(\neg x_4 \vee c_4)$ $(\neg x_5 \vee r_2)$
$(\neg x_5 \vee c_1)$ $(\neg x_6 \vee r_2)$ $(\neg x_6 \vee c_2)$
$(\neg x_7 \vee r_2)$ $(\neg x_7 \vee c_3)$ $(\neg x_8 \vee r_2)$
$(\neg x_8 \vee c_4)$ $(\neg x_9 \vee r_3)$ $(\neg x_9 \vee c_1)$
$(\neg x_{10} \vee r_3)$ $(\neg x_{10} \vee c_2)$

Recursive AMO

$(\neg r_1 \vee \neg r_2)$ $(\neg r_1 \vee \neg r_3)$ $(\neg r_2 \vee \neg r_3)$
$(\neg c_1 \vee \neg c_2)$ $(\neg c_1 \vee \neg c_3)$ $(\neg c_1 \vee \neg c_4)$
$(\neg c_2 \vee \neg c_3)$ $(\neg c_2 \vee \neg c_4)$ $(\neg c_3 \vee \neg c_4)$

Fig. 1. Encoding the AtMost-1 constraint $\sum_{i=1}^{i\leq 10} x_i \leq 1$ with the two product encoding, and two auxiliary AtMost-1 constraints $r_1 + r_2 + r_3 \leq 1$ and $c_1 + c_2 + c_3 + c_4 \leq 1$.

the implications $c_3 \rightarrow \neg c_2$ and $\neg c_2 \rightarrow (\neg x_2 \wedge \neg x_6)$ by transitivity show, that $x_7 \rightarrow (\neg x_6 \wedge \neg x_2)$. Since all implications are build on binary clauses, the reverse direction also holds: $x_6 \rightarrow \neg x_7$ and $x_2 \rightarrow \neg x_7$. Hence, the constraints $x_6 + x_7 \leq 1$ and $x_2 + x_7 \leq 1$ can be deduced. However, the constraint $x_2 + x_6 \leq 1$ cannot be deduced via the columns and their literals c_2 and c_3. This constraint can still be found via rows, namely with the literals r_1 and r_2. The same reasoning as for columns applies also to rows.

More generally, given an AtMost-1 constraint R, where the complement of a literal $r_i \in R$ implies some literal $\neg x_i$ ($\neg r_i \rightarrow \neg x_i$), and furthermore, this literal $\neg x_i$ implies a literal b_i, which belongs to another AtMost-1 constraint C, $\neg b_i \in C$, then by using R as row constraint, and C as column constraint, an AtMost-1 constraint that includes x_i can be constructed by searching for the remaining literals x_j. Per literal a_i in the row constraint R, literals x_i implied by $\neg r_i$ can be collected as candidates to form a row in the two-product representation. Only literals x_i that imply a different literal c_i of the column constraint C are considered, so that the literal inside each row matches exactly one column in the matrix. The literals for one row already form an AtMost-1 constraint. For the next row r_{i+1}, more literals x_i are collected in the same way, and added to the AtMost-1 constraint. This addition is sound based on the construction of the encoding: if one of the elements in the new AtMost-1 constraint is assigned to true, then this assignment implies its row and column variable to be satisfied as well. Since there is an AtMost-1 constraint enforced for both the rows and the columns, all other row and column variables are assigned false. Due to the implications in the Two-Product encoding, these falsified selector variables also falsify all variables (except the currently satisfied one) in the new AtMost-1 constraint, and hence only the initially satisfied variable remains satisfied.

To the best of our knowledge, no existing system is able to detect AtMost-1 constraints which are encoded in this way. We now present algorithm 2, that is able to find an approximation of the set of those constraints. Constructing new

Algorithm 2: Extract AtMost-1 Constraints Two Product Encodings

Input: A set of "at most 1" cardinality constraints S, the `NAG` of the formula
Output: An extended set of "at most 1" cardinality constraints

```
1  foreach R ∈ AMO do
2      r = min(R) ;                                   // smallest literal only
3      l = min(NAG(r)) ;                              // r is row-selector for l
4      foreach c ∈ NAG(l) ;                          // c is column-selector for l
5      do
6          foreach C ∈ AMO_b do
7              newAMO = ∅ ;          // construct a new AMO based on R and C
8              foreach k ∈ C do
9                  hitSet = R ;                       // to hit each literal once
10                 foreach hitLit ∈ NAG(k), hitLit ∉ newAMO do
11                     foreach targetLit ∈ NAG(hitLit) do
12                         if targetLit ∈ hitSet ;          // found selector pair
13                         then
14                             hitSet := hitSet \ {targetLit};     // update hit set
15                             newAMO := newAMO ∪ {hitLit} ;     // update AMO
16                     end
17                 end
18             end
19             AMO := AMO ∪ newAMO ;          // store new AMO constraint
20         end
21     end
22 end
23 return AMO
```

AtMost-1 constraints based on the idea of the two-product encoding is done by first finding two AtMost-1 constraints R and C, which contain a literal r and c, which are used by some literal l as row selector and column selector (lines 1–6). Therefore, all AtMost-1 constraints R are considered, and a literal r is considered as row-selector variable. Next, the literal l is chosen to be part in the new two product AtMost-1. To reduce the computational work, the literal r is assumed to be the smallest literal in R, and the literal l is the smallest literal, such that $\neg r \rightarrow \neg l$ holds (lines 2–3). Finally, another AtMost-1 constraint C is selected, which contains the column selector literal c.

For each pair of AtMost-1s R and C, a new AtMost-1 can be constructed (line 7), by collecting all literals x_i. The literals x_i are called `hitLit` in the algorithm, because each such literal needs to imply a unique pair of row and column selector literals. This condition can be ensured by searching for literals that are implied by the complement of the column selector literal c: $\neg c \rightarrow \neg$`hitLit`. Furthermore, a literal `hitLit` has to imply a row selector variable $r \in R$ (lines 8–10). To ensure the second condition, an auxiliary set of literals `hitSet` is used, which stores all the literals of the row selector AtMost-1 constraint R during the analysis of each column. If for the current column selector c and the current literal `hitLit` a *new*

selector `targetLit` $\in$ `hitSet` is found (line 12), then the set `hitSet` of hit literals is updated by removing the current hit literal `targetLit`, and furthermore, the current hitting literal `hitLit` is added to the currently constructed AtMost-1 constraint (lines 14–15). Finally, the new AtMost-1 constraint is added to the set of constraints (line 19).

3.4 Detecting AtMost-2 Constraints

For a small number of literals x_i, and small thresholds k, for example $k = 2$, the naïve binomial encoding is competitive. Therefore, a method for extracting this constraint is proposed as well. Similarly to the syntactic extraction of AtMost-1 constraints, the structure of ternary clauses is analyzed by a greedy algorithm. Starting with a seed literal n which does not occur in an extracted AtMost-2 constraint yet all ternary clauses with $\bar{n}$ are considered and the set of candidate literals is initialized by all literals which occur negated at least twice in these clauses. If the candidate set at one point contains less than 4 literals the algorithm moves on to the next seed literal n. Otherwise each triple of literals in S is tested to have a corresponding ternary clause in the formula. If this test fails the set of candidates is reduced by removing from S one of the literals in a triple without a matching clause. If $|S| \geq 4$ and all triples can be matched with a clause, then the AtMost-2 constraint $\sum_{l \in S} l \leq 2$ is added.

4 Semantic Detection of AtMost-k Constraints

Another approach to detect cardinality constraints is to use unit propagation in the spirit of [25]. Using a more *semantic* approach instead of a pure *syntactic* approach allows to detect some nested cardinality constraints without requiring a specific procedure at the expense of performing unit propagation in a solver instead of traversing a NAG. The main advantage of the more semantic detection is that we may detect cardinality constraints as long as the encoding preserves arc-consistency by unit propagation. This allow us to propose an algorithm for all known encodings, that is also able to detect constraints that would not have been explicitly known at problem encoding time. However, our approach may not detect all cardinality constraints, since additional variables used in some encodings may interfere with the actual constraint variables, and make our algorithm produce truncated versions of the constraints to detect.

Basically our approach starts with a cardinality constraint $\sum_{i=1}^{n} l_i \leq k$ and tries to extend it with new literals m such that $(\sum_{i=1}^{n} l_i) + m \leq k$.

Our contribution is an algorithm to detect cardinality constraints in CNF using unit propagation, such that these constraints contain as much literals as it is possible to detect using unit propagation.

4.1 Expanding a Cardinality Constraint With One Literal

The idea of the algorithm is as follows: Given a clause $cl = l_1 \vee l_2 \vee .. \vee l_n$, we want to check if it belongs to a cardinality constraint $cc = \sum_{i=1}^{n} \overline{l_i} + \sum_j m_j \leq n - 1$.

Indeed, we know that $cl = l_1 \vee l_2 \vee .. \vee l_n \equiv \sum_{i=1}^{n} l_i \geq 1 \equiv \sum_{i=1}^{n} \overline{l_i} \leq n-1 = cc'$. We are thus looking for literals m_j which extend cc'.

Going back to our nested encoding example based on a CNF $\alpha = \neg x_1 \vee \neg x_2$, $\neg x_1 \vee \neg x_3$,$\neg x_2 \vee \neg x_3$,$x_3 \vee \neg x_4$, $x_3 \vee \neg x_5$,$\neg x_4 \vee \neg x_5$. $\neg x_1 \vee \neg x_2$ does represent the cardinality constraint $x_1 + x_2 \leq 1$. If we assign both x_1 or x_2 in α, we notice that the literals $\neg x_3, \neg x_4, \neg x_5$ are derived by unit propagation in both cases. hence, we can extend $x_1 + x_2 \leq 1$ by either x_3, x_4 or x_5, i.e. that the cardinality constraints $x_1 + x_2 + x_3 \leq 1$, $x_1 + x_2 + x_4 \leq 1$,$x_1 + x_2 + x_5 \leq 1$ are derivable from α. More generally: if all valid maximal combinations of the literals in cc' imply a literal $\neg m$, then m can be added to cc'. We exploit the following property.

Proposition 1 *Let α be a CNF. Let $\alpha(S)$ be the conjunction of the literals propagated in α under the set of assumptions S. Let $cc = \sum_{i=1}^{n} l_i \leq k$. Let $L = \{l_i \mid 1 \leq i \leq n\}$ and $L_k = \{S | [S \subseteq L] \wedge [|S| = k]\}$. If $\alpha \models cc$ and $\forall S \in L_k, \alpha(S) \models \neg m$ then $\alpha \models (\sum_{i=1}^{n} l_i) + m \leq k$.*

Proof. Let us suppose that ω is a model of α, $\alpha \models \sum_{i=1}^{n} l_i \leq k$ and $\forall S \in L_k, \alpha(S) \models \neg m$. Let us suppose that ω is not a model of $\alpha \wedge (\sum_{i=1}^{n} l_i) + m \leq k$. This implies that at least $k+1$ literals in $\{l_1, ..., l_n\}$ are set to true. As $\alpha \models \sum_{i=1}^{n} l_i \leq k$, m must be set to true, which is inconsistent with the fact that $\forall S \in L_k, \alpha(S) \models \neg m$. □

If several of those literals exist, it is not valid to add them at once to cc'. In our running example, x_3, x_4 and x_5 are candidates to extend $x_1 + x_2 \leq 1$ but extending cc' with all literals leads to the cardinality constraint $x_1 + x_2 + x_3 + x_4 + x_5 \leq 1$, which is not derivable from α. We also need to pay attention to unit clauses in the original formula and literals implied by unit propagation. Those literals are by definition candidates to the cardinality constraint expansion. Adding those literals may results in a case where the only literals that will be able to expand the constraint through our algorithm are known to be falsified.

Consider the formula $\neg x_1 \vee \neg x_2, \neg x_1 \vee \neg x_3, \neg x_3 \vee \neg x_2, \neg x_4$, and suppose you treat $\neg x_1 \vee \neg x_2$ as the cardinality constraint $x_1 + x_2 \leq 1$. Two literals are candidates to the clause expansion: x_3 and x_4. If we choose x_4, then the generated cardinality constraint is not tight because we know that x_4 must be falsified. Note that if unit propagation leads to unsatisfiability ($\bot$ is detected), we do not have to filter out the candidates, because all literals are implied by a falsified formula. In this case, we must pay attention that the candidate we choose for the expansion is not the complement of a literal that is already present in the initial cardinality constraint. This check is done in line 1 of Algorithm 3.

Algorithm 3 exploits Proposition 1 to find the complement of a literal that may expand a cardinality constraint. The set `candidates` keeps all the remaining candidates (the literals whose negation may expand the constraint). For each unit propagation phase, only literals that are propagated are kept, that is eliminating the ones for which there exists a subset L_k that does not propagate them (this is, in fact, preserve arc-consistency). This procedure implies that for each literal $\neg m$ in the set `candidates`, the literal m may expand the current cardinality constraint.

Algorithm 3: findExpandingLiterals

Input: a CNF formula α, a cardinality constraint $\sum_{i=1}^{n} l_i \leq k$
Output: a set of literals m such that $(\sum_{i=1}^{n} l_i) + \overline{m} \leq k$

```
1 candidates ← {v_i | v_i ∈ VARS(α)} ∪ {¬v_i | v_i ∈ VARS(α)} \ {¬l_i};
2 foreach S ⊆ {l_i} such that |S| = k do
3 |   propagated ← unitProp(α, S) ;
4 |   if ⊥ ∉ propagated then
5 |   |   candidates ← candidates ∩ propagated ;
6 |   |   if candidates = ∅ then
7 |   |   |   return ∅;
8 end
9 return candidates ;
```

Lemma 1. *Let α be a CNF. Let $cc = \sum_{i=1}^{n} l_i \leq k$ such that $\alpha \models cc$. $\forall m \in$ findExpandingLiterals(α,cc), $\alpha \models \sum_{i=1}^{n} l_i + \overline{m} \leq k$.*

4.2 Maximal Cardinality Constraint Expansion

In practice, we are not going to learn any arbitrary cardinality constraint, but only the ones which cannot be extended further. Moreover, if a cardinality constraint corresponding to a clause cannot be extended at all, we will keep it in its clausal form. Algorithm 3 computes all the literals that are propagated through unit propagation by all sets L_k. Once this set is empty, we are not able to find a literal that extends this constraint using the unit propagation. As long as there exists such literals, they may be added as proved by Proposition 1, as written in the following lemma.

Lemma 2. *Let α be a CNF. $\forall c \in \alpha$, $\alpha \models$ expandCardFromClause(α, c).*

We iteratively find a new expanding literal, add the literal to the constraint, then search a new literal, and repeat these steps until there are no more expansion candidates. It is not necessary to compute all sets L_k when the second iteration is reached. In fact, to find the n^{th} literal of a constraint, we have computed $\binom{n-1}{k}$ of the $\binom{n}{k}$ propagations that are required by the current call to Algorithm 3. The only sets L_k that are not analyzed yet are the sets containing the literal that was added to the constraint in the most recent step. This procedure is shown in Algorithm 4. With this insight, we build an efficient algorithm to compute maximum cardinality constraints in Algorithm 5.

The computation of the $\binom{n}{k}$ unit propagations is the costly part of the algorithm. We assume that the unit propagation cost is bounded by the number of literals to produce the following lemma.

Lemma 3. *Let α be a CNF with n variables and l literals. Let c be a clause of α of size $|c| = k + 1$. expandCardFromClause(α, c) has a complexity in $O(\binom{n}{k} \times l)$.*

Algorithm 4: refineExpandingLiterals

Input: a CNF formula α, a cardinality constraint $\sum_{i=1}^{n} l_i + l_{new} \leq k$, a set of literals L

Output: a set of literals m such that $(\sum_{i=1}^{n} l_i) + l_{new} + \overline{m} \leq k$

```
1 candidates ← L;
2 foreach S' = S ∪ {l_new} such that S ⊆ {l_i} and |S| = k − 1 do
3 |   propagated ← unitProp(α, S') ;
4 |   if ⊥ ∉ propagated then
5 |   |   candidates ← candidates ∩ propagated ;
6 |   |   if candidates = ∅ then
7 |   |   |   return ∅ ;
8 end
9 return candidates ;
```

4.3 Replacing Clauses by Cardinality Constraints

The last step in our approach is to detect clauses that are entailed by the cardinality constraints found so far. This step is important, because it allows to avoid considering clauses that would lead to already revealed cardinality constraints. Furthermore, we need to keep the clauses not covered by any cardinality constraint to build a mixed formula of cardinality constraints and clauses, which is logically equivalent to the original formula.

We use the rule described by Barth in [8] and used in 3MCard [24] to determine if a clause (written as an at-most-k constraint) is dominated by a revealed cardinality constraint. This rule states that $L \geq d$ dominates $L' \geq d'$ iff $|L \setminus L'| \leq d - d'$. So, before considering a clause for cardinality constraint expansion, we check using this rule if the clause is dominated by one of our new constraints. In this case, we do not search any expansion, and remove this clause from the problem. We also remove the clauses that have been expanded to cardinality constraints, as they are trivially dominated by the new constraint.

Algorithm 5: expandCardFromClause

Input: a CNF formula α, a clause c

Output: a cardinality constraint cc or c

```
1 cc ← Σ_{l∈c} l̄ ≤ |c| − 1 ;
2 candidates ← findExpandingLiterals(α, cc) ;
3 while candidates ≠ ∅ do
4 |   select m in candidates;
5 |   cc ← Σ_{l_i∈cc} l_i + m̄ ≤ |c| − 1 ;
6 |   candidates ← refineExpandingLiteral(α, cc, candidates \ {m}) ;
7 end
8 if |cc| > |c| then return cc;
9 else return c ;
```

Algorithm 6: revealCardsInCNF

Input: a CNF formula α and a bound k
Output: a formula $\phi \equiv \alpha$ containing cardinality constraints with threshold $\leq k$ and clauses

```
1 φ ← ∅ ;
2 foreach clause c ∈ α of increasing size —c— such that |c| ≤ k + 1 do
3 |   if there is no cardinality constraint cc ∈ φ which dominates c then
4 |   |   φ ← φ ∪ expandCardFromClause(α, c) ;
5 |   end
6 end
7 return φ ;
```

To avoid redundancy it is important to first consider the smallest clauses as candidates for the expansion. In fact, while considering the smallest clauses first, we find the cardinality constraints with the lowest threshold first. Consider that $l_1 + ... + l_n \leq k$ has been discovered, and that we take a look at a constraint where the sum part sums a subset of $\{l_1, ..., l_n\}$ and where the threshold is $k + d$ ($d > 0$). In this case, the latter constraint is always dominated by the former cardinality constraint, so there is no need to expand it.

As the new cardinality constraints are consequences of the formula and the removed clauses are consequences of the cardinality constraints, we ensure that the new formula is equivalent to the original one, as written in the following theorem.

Theorem 1. *Let α be a CNF. Let k an arbitrary integer.*

$$\alpha \equiv \mathit{revealCardsInCNF}(\alpha, k)$$

In our nested encoding example, our approach will work as follows. We first try to extend $x_1 + x_2 \leq 1$. Our approach will find either $x_1 + x_2 + x_3 \leq 1$ or $x_1 + x_2 + x_4 + x_5 \leq 1$. Suppose it finds the longest one. The CNF is reduced from clauses dominated by that cardinality constraint: $\neg x_1 \vee \neg x_2, \neg x_4 \vee \neg x_5$. The next clause to consider is $\neg x_1 \vee \neg x_3$. We try to extend $x_1 + \neg x_3 \leq 1$. We can extend it to $x_1 + x_2 + x_3 \leq 1$. The clause $\neg x_2 \vee \neg x_3$, is removed from the CNF, because this clause is dominated by that new cardinality constraint. The next cardinality to extend is $\neg x_3 + x_4 \leq 1$. The cardinality constraint $\neg x_3 + x_4 + x_5 \leq 1$ is found. The remaining clauses are dominated by the cardinality constraints, so they are removed from the CNF. The procedure stops, since no more clauses have to be considered. Note that if the first cardinality constraint found is $x_1 + x_2 + x_3 \leq 1$, the procedure will be unable to reveal $x_1 + x_2 + x_4 + x_5 \leq 1$, because all clauses containing $\neg x_1$ would be removed from the CNF.

In terms of complexity, the worst case will be reached if we try to expand all clauses; implying the following complexity bound.

Lemma 4. *Let α be a CNF with n variables, m clauses and l literals. Let k an integer such that $0 < k \leq n$ and $m_k \leq m$ the number of clauses of size $\leq k + 1$ in α. revealCardsInCNF(α, k) has a complexity in $O(m_k \times \binom{n}{k} \times l)$.*

Preprocessor Solver	#inst.	Lingeling Lingeling	Synt.(Riss) Sat4jCP	Sem.(Riss) Sat4jCP	no SBSAT	no Sat4jCP
Pairwise	14	**14 (3s)**	13 (244s)	14 (583s)	6 (0s)	1 (196s)
Binary	14	3 (398s)	2 (554s)	**7 (6s)**	6 (7s)	2 (645s)
Sequential	14	0 (0s)	**14 (50s)**	**14 (40s)**	10 (6s)	1 (37s)
Product	14	0 (0s)	**14 (544s)**	11 (69s)	6 (25s)	2 (346s)
Commander	14	1 (3s)	7 (0s)	**14 (40s)**	9 (187s)	1 (684s)
Ladder	14	0 (0s)	11 (505s)	11 (1229s)	**12 (26s)**	1 (36s)

Fig. 2. Six encodings of the pigeon hole instances: number of solved instances and sum of run time for solved instances per solver configuration per encoding.

5 Experimental Results

The experimental results show that the proposed methods detect a significant amount of cardinality constraints in CNFs. For this analysis we use academic benchmarks, like Sudoku puzzles and the pigeon hole problem, for which we know how many cardinality constraints are present in the CNF, and which are easy to solve using Generalized Resolution when the constraints are expressed using cardinality constraints. All the benchmarks were launched on Intel Xeon X5550 processors (@2.66GHz) with 32GB RAM and a 900s timeout.

The static approach is implemented in the latest release of Lingeling. The static approach plus the specific handling of the two product encoding are implemented on Riss (so called Syntactic). The semantic detection of cardinality constraints is implemented on Riss. As Riss does not take advantage of those cardinality constraints for solving the benchmarks, we use it as a fast preprocessor to feed Sat4j which uses Generalized Resolution to solve the new benchmark with a mix of clauses and cardinality constraints. It allows us to check if the cardinality constraints found by the incomplete approaches are sufficient to solve those benchmarks. We compare the proposed approaches against SBSAT[1].

5.1 Pigeon hole principle

These famous benchmark are known to be extremely hard for resolution based solvers [19]. For $n+1$ pigeons and n holes, the problem is to assign each pigeon in a hole while not having more that one pigeon per hole. Each Boolean variable $x_{i,j}$ represents pigeon i is assigned hole j. The problem is expressed by $n+1$ clauses $\bigvee_{j=1}^{n} x_{i,j}$ and n cardinality constraints $\sum_{j=1}^{n+1} x_{i,j} \leq 1$. Those benchmarks are generated for n from 10 to 15, and n from 25 to 200 by steps of 25, using six different encodings: binomial, product, binary, ladder, commander and sequential. The results are presented in Table 2.

As expected, without revealing the cardinality constraint, Sat4j can only solve one or two problems. The semantic approach can detect many cardinality

[1] We tried to run 3MCard on those benchmarks but the solver was not able to read or solve most of the benchmarks.

constraints and let Sat4j solve more instances than the other solvers — it is particularly efficient for the pairwise, sequential and commander encodings. Note that the instance using the pairwise encoding for $n = 200$ has 402000 variables and 4020201 clauses, which shows that our approach scales. The static analysis (with specific reasoning for the two product encoding) is also very efficient on most of the encodings, and is the best on the product encoding, as intended. The commander and binary encodings are most difficult to reveal using our techniques. SBSAT is not as efficient as our approaches, even if it got the best results for the ladder encoding. Lingeling is very efficient for the pairwise encoding, but not for the others, as intended too.

5.2 Small hard combinatorial benchmarks

Benchmarks of unsatisfiable balanced block designs are described in [32,16]. They contain the cardinality constraints AtMost-2. We use the benchmarks submitted to the SAT09 competition (called sgen) which were the smallest hard unsatisfiable formulas as well as benchmarks provided by Jakob Nordström and Mladen Miksa from KTH [29]. Both the syntactic and the semantic detection are able to recover quickly all cardinality constraints of these benchmarks and let the solver use them to prove unsatisfiability, as presented in Table 3. The solvers Lingeling and Sat4jCP, which do not use cardinality detection or generalized resolution, emphasize the fact these benchmarks are too difficult for existing solver.

5.3 Sudoku benchmarks

Sudoku puzzles contain only = 1 cardinality constraints represented by a clause and an AtMost-1 constraint. The puzzles are trivial to solve but interesting from a preprocessing point of view because the cardinality constraints share a lot of variables which may mislead our approximation methods. We use two instances representing empty $n \times n$ for $n = 9$ and $n = 16$. The grids contain $n^2 \times 4$ AtMost-1 constraints, for $n = 9(16)$ there are 324(1024) constraints. All constraints are encoded with the pairwise encoding. The AtMost-1 constraints that can be found by the syntactic approach all contain 9 (resp. 16) literals, as expected. However, some constraints are missing: 300 constraints revealed out

Preprocessor Solver	#inst.	Lingeling Lingeling	Synt.(Riss) Sat4jCP	Sem.(Riss) Sat4jCP	no SBSAT	no Sat4jCP
Sgen unsat	13	0 (0s)	**13 (0s)**	**13 (0s)**	9 (614s)	4 (126s)
Fixed bandwidth	23	2 (341s)	**23 (0s)**	**23 (0s)**	**23 (1s)**	13 (1800s)
Rand. orderings	168	16 (897s)	**168 (7s)**	**168 (8s)**	99 (2798s)	69 (3541s)
Rand. 4-reg.	126	6 (1626s)	**126 (4s)**	**126 (5s)**	84 (2172s)	49 (3754s)

Fig. 3. Various hard combinatorial benchmarks families: number of solved instances and sum of run time for solved instances per solver configuration per family

of 324 and 980 revealed out of 1024. The semantic preprocessor finds all the cardinality constraints for those benchmarks because each cardinality constraint has a binary clause which belongs to only this constraint, and hence is not discarded when another constraint is found. This specific clause is used to retrieve the cardinality constraint by addition of literals.

6 Conclusion

We presented two approaches to derive cardinality constraints from CNF. The first approach is based on the analysis of a NAND graph, the data structure used in some SAT solvers to handle binary clauses efficiently, to retrieve AtMost-1 or AtMost-2 constraints. The second approach, based on using unit propagation on the original CNF, is a generic way to derive AtMost-k constraints. We show that both approaches are able to retrieve cardinality constraints in known hard combinatorial problems in CNF. Our experiments suggest that the syntactic approach is particularly useful to derive AtMost-1 and AtMost-2 constraints while the semantic is more robust because this method is also able to detect cardinality constraints with an arbitrary threshold. Our approaches are useful for tackling the smallest unsolved UNSAT problems of the SAT 2009 competition (sgen) which are all solved by Sat4j using generalized resolution within seconds once the AtMost-2 constraints have been revealed (this fact was already observed in [34]). The difference between the syntactic and semantic approaches is visible with the challenge benchmark from [16] that no solver could solve in one day. It is solved in a second after deriving 22 AtMost-2 and 20 AtMost-3 constraints when revealing the constraints using the semantic approach. The syntactic approach is not able to reveal those AtMost-3 constraints.

We have been able to reveal cardinality constraints on large application benchmarks. However, using that information to improve the run time of the solvers is future work. Checking if those constraints result from the original problem specification or are "hidden" constraints, i.e. constraints not explicitly known in the specification, is an open problem. An interesting research question, out of the scope of this paper, is to study the proof system of the combination of our semantic preprocessing step (extension rule and domination rule) plus generalized resolution.

Acknowledgement

The authors would like to thank Jakob Nordström and his group from KTH for providing us the hard combinatorial benchmarks, pointing out the challenge benchmark, and the fruitful discussions about Sat4j proof system which motivated this work. The authors would like to thank the Banff International Research Station which hosted the Theoretical Foundations of Applied SAT Solving workshop (14w5101) which essentially contributed to this joint work. The authors would like to thank the anonymous reviewers for their helpful comments to improve this paper. This work has been partially supported by ANR TUPLES.

References

1. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Generic ilp versus specialized 0-1 ilp: an update. In: Pileggi, L.T., Kuehlmann, A. (eds.) ICCAD. pp. 450–457. ACM (2002)
2. Ansótegui, C., Larrubia, J., Li, C.M., Manyà, F.: Exploiting multivalued knowledge in variable selection heuristics for sat solvers. Ann. Math. Artif. Intell. 49(1-4), 191–205 (2007)
3. Ansótegui, C., Manyà, F.: Mapping problems with finite-domain variables to problems with boolean variables. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT (Selected Papers. Lecture Notes in Computer Science, vol. 3542, pp. 1–15. Springer (2004)
4. Ansótegui Gil, C.J.: Complete SAT solvers for Many-Valued CNF Formulas. Ph.D. thesis, University of Lleida (2004)
5. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks and their applications. In: Kullmann, O. (ed.) SAT. Lecture Notes in Computer Science, vol. 5584, pp. 167–180. Springer (2009)
6. Bailleux, O., Boufkhad, Y.: Efficient cnf encoding of boolean cardinality constraints. In: Rossi, F. (ed.) CP. Lecture Notes in Computer Science, vol. 2833, pp. 108–122. Springer (2003)
7. Barahona, P., Hölldobler, S., Nguyen, V.H.: Representative Encodings to Translate Finite CSPs into SAT. In: 11th International International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming (CP) – CPAIOR 2014, Cork, Ireland, May 19-23, 2014, to appear (2014)
8. Barth, P.: Linear 0-1 inequalities and extended clauses. In: Voronkov, A. (ed.) LPAR. Lecture Notes in Computer Science, vol. 698, pp. 40–51. Springer (1993)
9. Ben-Haim, Y., Ivrii, A., Margalit, O., Matsliah, A.: Perfect hashing and cnf encodings of cardinality constraints. In: Cimatti, A., Sebastiani, R. (eds.) SAT. Lecture Notes in Computer Science, vol. 7317, pp. 397–409. Springer (2012)
10. Biere, A.: Lingeling, plingeling and treengeling entering the sat competition 2013. In: Balint, A., Belov, A., Heule, M., Järvisalo, M. (eds.) Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions. vol. B-2013-1, pp. 51–52. University of Helsinki, Department of Computer Science Series of Publications B (2013)
11. Chen, J.C.: A new sat encoding of the at-most-one constraint. In: In Proc. of the Tenth Int. Workshop of Constraint Modelling and Reformulation (2010)
12. Cook, W., Coullard, C., Turán, G.: On the complexity of cutting-plane proofs. Discrete Applied Mathematics 18(1), 25 – 38 (1987)
13. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into sat. JSAT 2(1-4), 1–26 (2006)
14. Frisch, A., Giannaros, P.: Sat encodings of the at-most-k constraint: Some old, some new, some fast, some slow. In: Proceedings of the The 9th International Workshop on Constraint Modelling and Reformulation (ModRef 2010) (2010)
15. Fu, Z., Malik, S.: Extracting logic circuit structure from conjunctive normal form descriptions. In: VLSI Design. pp. 37–42. IEEE Computer Society (2007)
16. Gelder, A.V., Spence, I.: Zero-one designs produce small hard sat instances. In: Strichman, O., Szeider, S. (eds.) SAT. Lecture Notes in Computer Science, vol. 6175, pp. 388–397. Springer (2010)
17. Gent, I.P., Nightingale, P.: A new encoding of alldifferent into sat. Proc. 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems pp. 95–110 (2004)

18. Gent, I., Prosser, P. andSmith, B.: A 0/1 encoding of the gaclex constraint for pairs of vectors. In: ECAI 2002 workshop W9: Modelling and Solving Problems with Constraints. University of Glasgow (2002)
19. Haken, A.: The intractability of resolution. Theoretical Computer Science 39(0), 297 – 308 (1985)
20. Hölldobler, S., Nguyen, V.H.: On SAT-Encodings of the At-Most-One Constraint. In: Katsirelos, G., Quimper, C.G. (eds.) Proc. The Twelfth International Workshop on Constraint Modelling and Reformulation, Uppsala, Sweden, September 16-20. pp. 1–17 (2013)
21. Hooker, J.N.: Generalized resolution and cutting planes. Ann. Oper. Res. 12(1-4), 217–239 (1988)
22. Karp, R.: Reducibility among combinatorial problems. In: Miller, R., Thatcher, J., Bohlinger, J. (eds.) Complexity of Computer Computations, pp. 85–103. The IBM Research Symposia Series, Springer US (1972)
23. Klieber, W., Kwon, G.: Efficient cnf encoding for selecting 1 from n objects. In: International Workshop on Constraints in Formal Verification (2007)
24. van Lambalgen, M.: 3MCard 3MCard A Lookahead Cardinality Solver. Master's thesis, Delft University of Technology (2006)
25. Le Berre, D.: Exploiting the real power of unit propagation lookahead. Electronic Notes in Discrete Mathematics 9, 59–80 (2001)
26. Manthey, N., Heule, M., Biere, A.: Automated reencoding of boolean formulas. In: Biere, A., Nahir, A., Vos, T.E.J. (eds.) Haifa Verification Conference. Lecture Notes in Computer Science, vol. 7857, pp. 102–117. Springer (2012)
27. Manthey, N., Steinke, P.: Quadratic direct encoding vs. linear order encoding, a one-out-of-n transformation on cnf. In: Proceedings of the First International Workshop on the Cross-Fertilization Between CSP and SAT (CSPSAT11) (2011)
28. Martins, R., Manquinho, V.M., Lynce, I.: Parallel search for maximum satisfiability. AI Commun. 25(2), 75–95 (2012)
29. Miksa, M., Nordstrom, J.: Long proofs of (seemingly) simple formulas. In: this issue (2014)
30. Ostrowski, R., Grégoire, É., Mazure, B., Sais, L.: Recovering and exploiting structural knowledge from cnf formulas. In: Hentenryck, P.V. (ed.) CP. Lecture Notes in Computer Science, vol. 2470, pp. 185–199. Springer (2002)
31. Sinz, C.: Towards an optimal cnf encoding of boolean cardinality constraints. In: van Beek, P. (ed.) CP. Lecture Notes in Computer Science, vol. 3709, pp. 827–831. Springer (2005)
32. Spence, I.: sgen1: A generator of small but difficult satisfiability benchmarks. ACM Journal of Experimental Algorithmics 15 (2010)
33. Walsh, T.: Sat v csp. In: Dechter, R. (ed.) CP. Lecture Notes in Computer Science, vol. 1894, pp. 441–456. Springer (2000)
34. Weaver, S.: Satisfiability Advancements Enabled by State Machines. Ph.D. thesis, University of Cincinnati (2012)