Incremental Encoding of Pseudo-Boolean Goal Functions Based on Comparator Networks

Michał Karpiński(✉) and Marek Piotrów

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

Abstract. Incremental techniques have been widely used in solving problems reducible to SAT and MaxSAT instances. When an algorithm requires making subsequent runs of a SAT-solver on a slightly changing input formula, it is usually beneficial to change the strategy, so that the algorithm only operates on a single instance of a SAT-solver. One way to do this is via a mechanism called *assumptions*, which allows to accumulate and reuse knowledge from one iteration to the next and, in consequence, the provided input formula need not to be rebuilt during computation. In this paper we propose an encoding of a Pseudo-Boolean goal function that is based on sorting networks and can be provided to a SAT-solver only once. Then, during an optimization process, different bounds on the value of the function can be given to the solver by appropriate sets of assumptions. The experimental results show that the proposed technique is sound, that is, it increases the number of solved instances and reduces the average time and memory used by the solver on solved instances.

Keywords: Incremental encoding · CNF encoding · Pseudo-Boolean constraints · Comparator networks · SAT-solvers

1 Introduction

A Pseudo-Boolean constraint (a PB-constraint, in short) is of the form $a_1x_1 + a_2x_2 + \cdots + a_nx_n \; \# \; k$, where $n, k \in \mathbb{N}$, $\{x_1, \ldots, x_n\}$ is a set of propositional literals (that is, variables or their negations), $\{a_1, \ldots, a_n\}$ is a set of integer coefficients, and $\# \in \{<, \leq, =, \geq, >\}$. PB-constraints are more expressive and more compact than clauses when representing some Boolean formulas, especially for optimization problems. PB-constraints are used in many real-life applications, for example, in cumulative scheduling [31], logic synthesis [3] or verification [9]. There have been many approaches for handling PB-constraints in the past, for example, extending existing SAT-solvers to support PB-constraints natively [14,21]. One of the most successful ideas was introduced by Eén and Sörensson [13], who show how PB-constraints can be handled through translation to SAT.

The algorithm, implemented in a tool called MiniSat+, incrementally strengthens the constraint on a goal function to find the optimum value, rebuilding partially a formula on each iteration, and making a new call to the underlying SAT-solver.

A typical SAT-solver accepts a problem instance as an input and outputs a satisfying assignment or an **Unsatisfiable** statement as a result. This can be inefficient if we want to minimize a value of a given goal function by solving many similar SAT instances (like in the aforementioned PB-solving algorithm of MiniSat+). Parsing almost the same constraint sets, and then applying the same inferences could be costly, therefore a more preservative approach is recommended.

An incremental approach for solving a series of related SAT instances was introduced, for example, in [12], as the means of checking safety properties on finite state machines. Later, the same authors implemented this technique in MiniSat [11] as a general tool, which they simply called *assumptions*. Assumptions are propositions that hold solely for one specific invocation of the solver. The goal of this paper is to propose an incremental algorithm for solving PB-constraint optimization problems by modifying an iterative SAT-based algorithm of KP-MiniSat+ [17], such that the input instance is encoded only once, and later, a set of assumptions is changed from one iteration to another, such that the encoding of the new constraint (on the goal function) is preserved, without the need to rebuild the CNF formula.

1.1 Related Work

One way to solve a PB-constraint is to transform it to a SAT instance (via Binary Decision Diagrams (BDDs), adders or sorting networks [7,13]) and process it using – increasingly improving – state-of-the-art SAT-solvers. Recent research have favored the approach that uses BDDs, which is evidenced by several new constructions and optimizations [2,30]. In our previous paper we showed that encodings based on comparator networks can still be very competitive [17]. Comparator networks have been successfully applied to construct very efficient encodings of cardinality and Pseudo-Boolean constraints. Codish and Zazon-Ivry [10] introduced pairwise selection networks. We have later improved their construction [16]. 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 $\lambda \cdot \mathit{num\_vars} + \mathit{num\_clauses}$ in the encodings, where lambda is a constant chosen empirically. The constructed encodings are small and efficient. Most encodings based on comparator networks use variations of the Batcher's Odd-Even Sorting Network [1,4,5,18].

Incremental usage of SAT-solvers has been studied extensively in the past years, which allowed for the huge increase in the performance of SAT-based algorithms [12,25,32,33]. Recently, incremental algorithms for MaxSAT instances have appeared [24,27,34], and the experimental results show that the performance of MaxSAT-solvers can be greatly improved by maintaining the learned information and the internal state of the SAT-solver between iterations. Some incremental SAT algorithms also exist for solving PB-constraint instances. For example, Manolios and Papavasileiou [22] proposed an algorithm for PB-solving that uses a SAT-solver for the efficient exploration of the search space, but at the same time exploits the high-level structure of the PB-constraints to simplify the problem and direct the search. Some popular solvers also implement incremental methods, for example, QMaxSAT [20] or Glucose [6].

1.2 Our Contribution

Even though MaxSAT problems and Pseudo-Boolean constraint satisfaction problems have a very close relation with each other (by a simple reduction), the notion of incrementality for encoding PB-constraints has not yet been fully exploited. In this paper we show how sorter-based algorithm of KP-MiniSat+ can be extended to solve, even more efficiently, optimization problems involving PB-constraints.

MiniSat+ has served as a base for many new solvers and has been extended to test new constructions and optimizations in the field of PB-solving. Similarly, we have developed a system based on it which encodes PB-constraints using a new sorter-based algorithm [17], efficiently finds good mixed-radix bases for the encoding (see Subsect. 2.2 for a definition) and incorporates a few other optimizations. The underlying comparator network is called a 4-Way Merge Selection Network [18], and experiments showed that on many instances of popular benchmarks our technique outperformed other state-of-the-art PB-solvers. Furthermore, our solver has been recently extended to MaxSAT problems and can successfully compete with state-of-the-art MaxSAT-solvers, which is evidenced by achieving high places in MaxSAT Evaluation 2019. The new MaxSAT-solver, called UWrMaxSat [29], took second place in both Weighted Complete Track and Unweighted Complete Track of the competition.

In this paper we show how the encoding algorithm of the PB-solver can be further improved by extending the usage of assumptions in the comparator network encoding scheme. The new technique is a modification of the idea found in NaPS [30] for simplifying inequality assertions in a constraint. It is applied when a mixed-radix base is used to encode a constraint as an interconnected sequence of sorting networks. The idea is to add a certain integer constant to both sides of the constraint, such that the representation of right side constant (in the base) contains only one non-zero digit. Now, in order to enforce the inequality, one only needs to assert a single output variable of the encoding of the last network. This simplification allows for a reduction of the number of clauses in the resulting CNF encoding, as well as allows better propagation. We have successfully implemented the technique in KP-MiniSat+. In the process of minimizing the value of a goal function, the solver has to try a series of bounds on it. The main purpose of our new construction is to avoid adding new variables and clauses to the encoding after each bound change. In this paper we show how to remedy this situation by adding a certain number of fresh variables to the encoded networks and then using them as assumptions to set a value of the changing constant.

We experimentally compare our solver with other state-of-the-art general constraints solvers like PBLib [28] and NaPS [30] to prove that our techniques are good in practice. We use COMiniSatPS [26] by Chanseok Oh as the underlying SAT-solver, as it has been observed to perform better than the original MiniSat [11] for many instances.

Since more than a decade there have been organized a series of Pseudo-Boolean Evaluations [23] which aim to assess the state-of-the-art in the field of PB-solvers. We use the competition problems from the PB 2016 Competition as benchmarks for the solver proposed in this paper.

1.3 Structure of the Paper

In Sect. 2 we briefly describe our comparator network algorithm, then we explain the Mixed Radix Base technique used in MiniSat+ and we show how it is applied to encode a PB-constraint by constructing a series of comparator networks. In Sect. 3 we show how to leverage assumptions in order to build an incremental algorithm on top of KP-MiniSat+'s PB-solving algorithm. We present results of our experiments in Sect. 4, and we give concluding remarks in Sect. 5.

2 Background

The main tool in our encoding algorithms is a comparator network. 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 "lower" output is the maximum of inputs, and "upper" one is the minimum. Their standard definitions and properties can be found, for example, in [19].

2.1 4-Way Merge Selection Network

MiniSat+ uses Batcher's original construction [8] – the 2-Odd-Even Sorting Network. Later, it has been proposed to replace it with a *selection network*. A selection network of order (n, k) is a comparator network such that for any 0–1 input of length n it outputs its k largest elements, where k is the RHS of a constraint. Those k elements must also be sorted in order to easily assert the given constraint, by asserting only the k-th output. In this paper we use sorting networks as black-boxes, therefore we describe the algorithm in a brief manner.

The main building block of our encoding is a direct selection network, which is a certain generalization of a comparator. Encoding of the direct selection network of order (n, k) with inputs $\langle x_1, \dots, x_n \rangle$ and outputs $\langle y_1, \dots, y_k \rangle$ is the set of clauses $\{x_{i_1} \wedge \dots \wedge x_{i_p} \Rightarrow y_p \,:\, 1 \le p \le k, 1 \le i_1 < \dots < i_p \le n\}$. The direct n-sorter is a direct selector of order (n, n), therefore we need n auxiliary variables and $2^n - 1$ clauses to encode it. This shows that n should be small in order to avoid an exponential blowup in the number of clauses.

It has already been observed that using selection networks instead of sorting networks is more efficient for the encoding of constraints [10], as the resulting encodings are smaller and can achieve faster SAT-solver run-time. This fact has been successfully used to encode cardinality constraints, and we have applied this technique to PB-constraints using a construction called a 4-Way Merge Selection Network. A detailed description of the algorithm, a proof of its correctness and the corresponding analysis can be found in our previous paper [18]. We extended our construction by mixing our network with the direct encoding for small values of parameters n and k – the technique which was first described by Abío et al. [1].

2.2 Mixed Radix Base Technique

The authors of MiniSat+ devised a method to decompose a PB-constraint into a number of interconnected sorting networks, where sorters play the role of adders on unary numbers in a *mixed radix representation.*

In the classic base r radix system, positive integers are represented as finite sequences of digits $\mathbf{d} = \langle d_0, \dots, d_{m-1} \rangle$ where for each digit $0 \leq d_i < r$, and for the most significant digit, $d_{m-1} > 0$. The integer value associated with $\mathbf{d}$ is $v = d_0 + d_1 r + d_2 r^2 + \cdots + d_{m-1} r^{m-1}$. A mixed radix system is a generalization where a base $\mathbf{B}$ is a sequence of positive integers $\langle r_0, \dots, r_{m-1} \rangle$. The integer value associated with $\mathbf{d}$ is $v = d_0 w_0 + d_1 w_1 + d_2 w_2 + \cdots + d_m w_m$ where $w_0 = 1$ and for $i \geq 0$, $w_{i+1} = w_i r_i$. For example, the number $\langle 2, 4, 10 \rangle_{\mathbf{B}}$ in base $\mathbf{B} = \langle 3, 5 \rangle$ is interpreted as $2 \times \mathbf{1} + 4 \times \mathbf{3} + 10 \times \mathbf{15} = 164$ (values of w_i's in boldface).

The decomposition of a PB-constraint into sorting networks is roughly as follows: first, find a "suitable" finite base $\mathbf{B}$ for the given set of coefficients, for example, in MiniSat+ the base is chosen so that the sum of all the digits of the coefficients written in that base is as small as possible. Then for each element r_i of $\mathbf{B}$ construct a sorting network where the inputs of the i-th sorter will be those digits $\mathbf{d}$ (from the coefficients) where d_i is non-zero, plus the potential carry bits from the $(i-1)$-th sorter.

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig. 1. Coefficients of ψ in base $\mathbf{B}$

We show a construction of a sorting network system using an example. We present a step-by-step process of translating a PB-constraint $\psi = 2x_1 + 2x_2 + 2x_3 + 2x_4 + 5x_5 + 18x_6 \leq 22$. Let $\mathbf{B} = \langle 2, 3, 3 \rangle$ be the considered mixed radix base. The representation of the coefficients of ψ in base $\mathbf{B}$ may be illustrated by a 6×4 matrix (see Fig. 1). The rows of the matrix correspond to the representation of the coefficients in base $\mathbf{B}$. Weights of the digit positions of base $\mathbf{B}$ are $\bar{w} = \langle 1, 2, 6, 18 \rangle$. Thus, the decomposition of the LHS (left-hand side) of ψ is:

$$\mathbf{1} \cdot (x_5) + \mathbf{2} \cdot (x_1 + x_2 + x_3 + x_4 + 2x_5) + \mathbf{6} \cdot (0) + \mathbf{18} \cdot (x_6)$$

Now we construct a series of four sorting networks in order to encode the sums at each digit position of $\bar{w}$. Given values for the variables, the sorted outputs

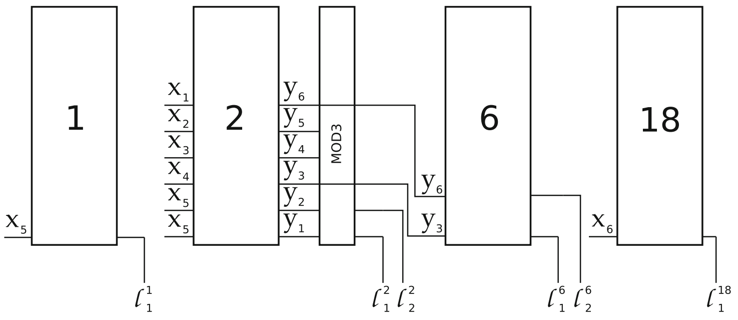

Fig. 2. Decomposition of a PB-constraint into a series of interconnected sorting networks. Outputs of sorting networks are ordered such that the bottom bit is the largest.

from these networks represent unary numbers d_1, d_2, d_3, d_4 such that the LHS of ψ takes the value $\mathbf{1} \cdot d_1 + \mathbf{2} \cdot d_2 + \mathbf{6} \cdot d_3 + \mathbf{18} \cdot d_4$.

The final step is to encode the carry operation from each digit position to the next. The first three outputs must represent valid digits (in unary) for **B**. In our example the single potential violation to this is d_2, which is represented in 6 bits. To this end we add two components to the encoding: (1) each third output of the second network is fed into the third network as carry input; and (2) a *normalizer* **MOD3** is added to encode that the output of the second network is to be considered modulo 3. The full construction is illustrated in Fig. 2.

The outputs from these four sorting networks now specify a number in base **B**, i.e., bits representing LHS of the constraint, each digit represented in unary. To enforce the constraint, we have to add clauses representing the relation ≤ 22 (in base **B**). It is done by lexicographical comparison of digits representing LHS to digits representing $22 = \langle 0, 2, 0, 1\rangle_{\mathbf{B}}$. Let l_1^1, l_1^2, l_2^2, l_1^6, l_2^6, l_1^{18} represent the outputs of the networks, like in Figure 2. Then the following set of clauses enforce the ≤ 22 constraint: $l_1^{18} \Rightarrow \neg l_1^6$ and $l_1^{18} \Rightarrow \neg(l_2^2 \wedge l_1^1)$.

Could we eliminate the clauses and the **MOD** sub-networks as well? Consider the following scheme. If we add $13 = \langle 1, 0, 2, 0\rangle_{\mathbf{B}}$ to both sides of ψ, then we get $\psi' = 2x_1 + 2x_2 + 2x_3 + 2x_4 + 5x_5 + 18x_6 + 13 < 36$. Observe that $36 = \langle 0, 0, 0, 2\rangle_{\mathbf{B}}$ and the new decomposition of the LHS is:

$$\mathbf{1} \cdot (1 + x_5) + \mathbf{2} \cdot (x_1 + x_2 + x_3 + x_4 + 2x_5) + \mathbf{6} \cdot (1 + 1) + \mathbf{18} \cdot (x_6)$$

After this change we virtually add 1s as additional outputs to the corresponding networks (one to the first network and two to the third network, as indicated by the new decomposition). This will change the number of inputs to some networks, that is, l_1^1 will be an additional input to the second network (as a carry) and l_1^6 will be a similar input to the fourth one. Thus, the fourth network will now have 2 inputs and an additional literal l_2^{18} representing its second output needs to be created.

Observe that ψ and ψ' are equivalent, but in the representation of 36 (in base **B**) only the most significant bit has a non-zero value, therefore enforcing the < 36 constraint is as easy as adding a singleton clause $\neg l_2^{18}$ (or setting it as an assumption). In consequence, the only relevant outputs of the networks (except the last one) are the ones that represent the carry bits, therefore there is no need to use normalizers. This optimization was first proposed in NaPS [30] and we have already implemented it in our previous solver [17]. Notice that after changing the RHS, we need to rebuild most of the construction in order to account for the increased number of inputs and outputs of each network and the new carry bit positions. What follows is an improvement of this strategy, such that we do not need to do the rebuilding step.

3 The Incremental Algorithm

We now show how we can better encode the goal function of a PB-constraint optimization instance by adding assumptions to the previous construction. To demonstrate each step of the algorithm, we will be using our running example, i.e., the goal function is $2x_1 + 2x_2 + 2x_3 + 2x_4 + 5x_5 + 18x_6$ and $\mathbf{B} = \langle 2, 3, 3 \rangle$ is the chosen base.

Code Notation. The pseudo-code is presented in Algorithms 1 and 2. The only non-trivial data structure used is a vector, i.e., a dynamic array, which in our case can store either numbers or literals, depending on the context. Vectors are indexed starting from 0, and x_i is the i-th element of a vector $\bar{x}$. The vector structure supports three straightforward operations:

- *pushBack* – appends a given element to the end of the vector.
- *size* – returns the number of elements currently stored in the vector.
- *clear* – removes all elements of the vector.

A special SAT-solver object *ss* is also available. It supports the following set of operations:

- *newVar* – creates a fresh variable and adds it to the solver instance.
- *addClause* – adds a clause to the solver instance (a clause is given as a sequence of literals).
- *encodeBySorter* – given a sequence of input literals of size n, it constructs a sorter with n inputs and n outputs and transforms it to a CNF formula (for example, using our 4-Way Merge Selection Network). The formula is added to the SAT-solver and the operation returns a sequence of literals representing the output of the sorter.
- *solve* – takes a set of assumptions as input and returns a model if the solver instance is satisfiable under given assumptions, otherwise returns UNSAT.

We now describe our algorithm and show how it works using our running example. We do this in a bottom-up manner, starting with the *encodeGoal* procedure (Algorithm 1).

Algorithm 1. encodeGoal

Input: A PB function $g(\bar{x}) = a_1x_1 + a_2x_2 + \cdots + a_nx_n$, where $a_1, a_2, \ldots, a_n > 0$, and a SAT-solver ss.

Output: A tuple $\langle \bar{r}, \bar{z}, \bar{y} \rangle$, where $\bar{r}$ is a mixed radix base, $\bar{z}$ are new assumption variables and $\bar{y}$ is a sequence of variables representing output of the encoding. The output is used in Algorithm 2 to force any upper bound on $g(\bar{x})$ by setting assumptions to ss.

```
 1: $\bar{r} \leftarrow findGoodBase(a_1, a_2, \ldots, a_n)$
 2: let $\bar{w}$ be the weight vector of $\bar{r}$
 3: let $\bar{z}$, $\bar{y}$, $carry$, $in$ and $out$ be empty vectors
 4: for $i = 0$ to $\bar{r}.size() - 1$ do
 5:     $in \leftarrow carry$
 6:     for $j = 1$ to $r_i - 1$ do
 7:         $z_j^{w_i} \leftarrow ss.newVar()$
 8:         $in.pushBack(z_j^{w_i})$, $\bar{z}.pushBack(z_j^{w_i})$
 9:     for $j = 2$ to $r_i - 1$ do $ss.addClause(\neg z_j^{w_i} \vee z_{j-1}^{w_i})$
10:     for $j = 1$ to $n$ do
11:         repeat $a_j \bmod r_i$ times $in.pushBack(x_j)$
12:         $a_j \leftarrow a_j / r_i$
13:     $out \leftarrow ss.encodeBySorter(in)$
14:     $carry.clear()$
15:     for $j = r_i - 1$ while $j < out.size()$ step $r_i$ do $carry.pushBack(out_j)$
16: $in \leftarrow carry$
17: for $j = 1$ to $n$ do
18:     repeat $a_j$ times $in.pushBack(x_j)$
19: $\bar{y} \leftarrow ss.encodeBySorter(in)$
20: return $\langle \bar{r}, \bar{z}, \bar{y} \rangle$
```

encodeGoal. Find a mixed radix base $\langle r_0, \ldots, r_{m-1} \rangle$ (for some $m \geq 0$) and its weight vector $\langle w_0, \ldots, w_m \rangle$ (lines 1–2). Next, decompose the goal function as shown in the previous section with the following modifications. The i-th iteration creates the i-th sorter for which inputs are stored in vector in. New assumption variables are created and passed to the sorter as additional input. A more detailed description is as follows, given we are in the i-th iteration of the main loop ($0 \leq i \leq m - 1$).

In lines 6–8, create a new variable $z_j^{w_i}$ for each $0 < j < r_i$ (line 7). Add the new variables as input to the current sorter (line 8). Next, for $1 < j < r_i$ add the clause $z_j^{w_i} \Rightarrow z_{j-1}^{w_i}$ to the instance (line 9).

The purpose for this step is as follows. The new variables allow to represent any number between 0 and $r_i - 1$ (for a given $0 \leq i \leq m - 1$) in unary, and the new clauses enforce the order of the bits. Now, if we would like to set the variables $\langle z_1^{w_i}, \ldots, z_{r_i-1}^{w_i} \rangle$ such that they represent a number $0 < j < r_i - 1$, we need to only set $z_j^{w_i} = 1$ and $z_{j+1}^{w_i} = 0$, and the unit propagation will set all other $z_{j'}^{w_i}$'s such that exactly j of them will be set to true. If $j = 0$, then we only

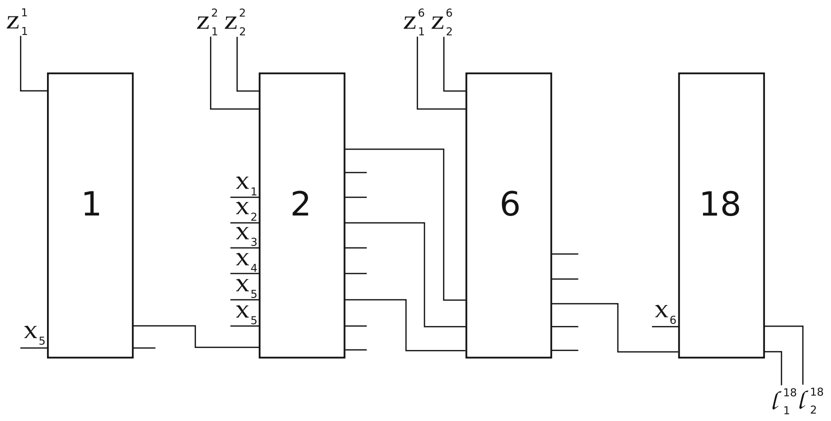

Fig. 3. An example of a novel PB-constraint decomposition. The top variables are stored as assumptions and their values are adjusted after each iteration of the algorithm.

need to set $z_1^{w_i} = 0$, and if $j = r_i - 1$ we set $z_{r_i-1}^{w_i} = 1$, and similarly the unit propagation correctly sets the rest of the variables.

Let a_j^i, $1 \le j \le n$, denote the value of a_j in line 11 in the i-th iteration of the loop 4–15. In lines 10–12, add multiple copies of the variables $x_1, \ldots, x_n$ to *in*, in such a way that they represent (in unary) terms of the sum $\sum_{j=1}^{n}(a_j^i \mod r_i)x_j$. Since each sorter acts as an adder, the sequence *out* in line 13 represents the value of the sum (plus *carry* and $\langle z_1^{w_i}, \ldots, z_{r_i-1}^{w_i}\rangle$) in unary.

In our running example we create five new variables: z_1^1, z_1^2, z_2^2, z_1^6, z_2^6, and the set of clauses consists of $z_2^2 \Rightarrow z_1^2$ and $z_2^6 \Rightarrow z_1^6$.

Remember that we represent the value of the LHS as an expression $w_0 \cdot d_0 + \cdots + w_m \cdot d_m$, as explained in the previous section (each d_i is a sum of some input variables). For each $0 \le i \le m-1$ and $0 < j < r_i$ we add $z_j^{w_i}$ to d_i. In our running example the decomposition will look like this:

$$\mathbf{1} \cdot (z_1^1 + x_5) + \mathbf{2} \cdot (z_1^2 + z_2^2 + x_1 + x_2 + x_3 + x_4 + 2x_5) + \mathbf{6} \cdot (z_1^6 + z_2^6) + \mathbf{18} \cdot (x_6)$$

In lines 10–15 a single sorter is created and the carry bits are set for the next one. Notice that the output of the current sorter (stored in the *out* vector) is only needed for calculating the carry bits passed to the next sorter (line 15). This is because the only necessary output variable which enforces constraints belongs to the last sorter (created in lines 16–19). Therefore no additional normalizers are required, which is another advantage of using our construction.

We show in Fig. 3 how such a construction looks for our running example. The new assumption variables are shown on the top. Compared to the example from Fig. 2 we added some new inputs, therefore we needed to also create additional outputs for each network. Notice that this changed the carry bit positions but no normalizers were constructed.

Algorithm 2. optimizeGoal

Input: A PB function $g(\bar{x}) = a_1x_1 + a_2x_2 + \cdots + a_nx_n$, where $a_1, a_2, \ldots, a_n > 0$, and a SAT-solver ss that is filled already with encodings of other constraints.

Output: A model that minimizes the value of $g(\bar{x})$ and satisfies other constraints or $UNSAT$.

```
 1: $\langle res, model\rangle \leftarrow ss.solve(\varnothing)$
 2: if $res = UNSAT$ then return $UNSAT$
 3: $\langle \bar{r}, \bar{z}, \bar{y}\rangle \leftarrow encodeGoal(g, ss)$                    # see Algorithm 1
 4: $UB \leftarrow g(model)$, $LB \leftarrow 0$, $optModel \leftarrow model$
 5: let $assump$ be an empty vector and $\bar{w}$ be the weight vector of $\bar{r}$
 6: while $UB > LB$ do
 7:     $bound \leftarrow \lceil 0.65 \cdot UB + 0.35 \cdot LB\rceil$                # $LB < bound \leq UB$
 8:     $assump.clear()$, $b \leftarrow bound$
 9:     for $i = 0$ to $r.size() - 1$ do
10:         $j \leftarrow b \mod r_i$, $b \leftarrow b/r_i$
11:         if $j \neq 0$ then $j \leftarrow r_i - j$, $b \leftarrow b + 1$
12:         if $j = 0$ then $assump.pushBack(\neg z_1^{w_i})$
13:         else if $j = r_i - 1$ then $assump.pushBack(z_{r_i-1}^{w_i})$
14:         else $assump.pushBack(z_j^{w_i})$, $assump.pushBack(\neg z_{j+1}^{w_i})$
15:     $assump.pushBack(\neg y_b)$
16:     $\langle res, model\rangle \leftarrow ss.solve(assump)$       # $g(\bar{x}) < bound$ is enforced by $assump$
17:     if $res = SAT$ then
18:         $UB \leftarrow g(model)$                                   # $g(model) < bound$
19:         $optModel \leftarrow model$
20:         $ss.addClause(\neg y_b)$
21:     else
22:         $LB \leftarrow bound$
23: return $optModel$
```

optimizeGoal. The optimization procedure is presented in Algorithm 2. Notice that we assume $a_1, a_2, \ldots, a_n > 0$. The goal function can be easily normalized to satisfy this condition (see [13]). After encoding every constraint into CNF formulas we first check if the given set of constraints is satisfiable (lines 1–2). If it is, then we can optimize the goal function given the constraints. We encode the goal function using the *encodeGoal* procedure (line 3). The optimization strategy used is the binary search with the 65/35 split ratio. The detailed description follows.

For the current bound on the constraint (stored in the *bound* variable) compute how many 1s need to be added to both sides of the inequality, such that the RHS has only the most significant position set in the base $\langle r_0, \ldots, r_{m-1}\rangle$. Let c be that number, that is, if *bound* is divisible by w_m then c is zero, otherwise c is set to $w_m - bound \mod w_m$, and let $\langle c_0, \ldots, c_{m-1}\rangle$ be the representation of c in base $\langle r_0, \ldots, r_{m-1}\rangle$. Notice that c_m is omitted since it is equal to 0. For each $0 \leq i \leq m - 1$, let $j = c_i$ and do:

– if $j = 0$, set $z_1^{w_i} = 0$,
– if $j = r_i - 1$, set $z_{r_i-1}^{w_i} = 1$,
– otherwise set $z_j^{w_i} = 1$ and $z_{j+1}^{w_i} = 0$.

This is done in lines 9–14, where variable b in the i-th iteration is set to the value of $\lceil bound/w_{i+1} \rceil$. Thus, j is the i-th digit (in base $\bar{r}$) of $bw_{i+1} - bound$. Next, add a singleton clause enforcing the constraint (line 15) to the set of assumptions.

In our running example let us assume that the current bound is 23, therefore $c = 13$, so the assumptions are $z_1^1 = 1$, $z_1^2 = 0$, $z_2^6 = 1$ (z_2^2 and z_1^6 will be set by unit propagation), which means that in order to enforce a constraint <36, we only need to add $\neg y_2$ as another assumption. Note that $\bar{y}$ is the output of the last sorter created by the *encodeGoal* procedure, so y_2 is equivalent to the l_2^{18} in Fig. 3.

Finally, we run the underlying SAT-solver under the current set of assumptions (line 16) and based on the answer we strengthen the bounds on the goal function (lines 17–22). The binary search continues until the optimum is found. For example, if the algorithm determines that the next bound to check for our running example is 19, then we revert the assignment of the assumptions and now we set $z_1^1 = 1$, $z_2^2 = 1$ and $z_2^6 = 1$, since now we need to add 17 to both sides of the inequality so that the encoding is still equisatisfiable with the < 36 constraint. Notice that no other operation is necessary. As we will see in the next section, the fact that we are building the sorting networks structure only once for the goal function leads to a performance increase in both running time and memory use, compared to other state-of-the-art methods. Let us now prove the correctness of our algorithm, for the sake of completeness.

Theorem 1. *Let $g(\bar{x}) = a_1x_1 + a_2x_2 + \cdots + a_nx_n$, where $a_1, a_2, \ldots, a_n > 0$ are integer coefficients and $x_1, x_2, \ldots, x_n$ are propositional literals. Let ϕ be a CNF formula. Algorithm 2 returns a model of ϕ which minimizes the value of $g(\bar{x})$ or UNSAT, if ϕ is unsatisfiable.*

Proof (sketch). If ϕ is unsatisfiable, then the algorithm terminates on line 2. Assume that ϕ is satisfiable. The binary search of *optimizeGoal* will find the optimal model of ϕ with respect to the goal function $g(\bar{x})$, if the distance between upper and lower bounds decreases in each iteration of the algorithm. It is obviously true if $g(\bar{x}) < bound$ is enforced on SAT solver by *assump* set in lines 8–15. To prove this, let $\langle \bar{r}, \bar{z}, \bar{y} \rangle$ be the result of line 3, m be the size of $\bar{r}$ and let $\bar{w}$ be the weight vector of base $\bar{r}$ (see Subsect. 2.2). Fix the value of *bound* and let $b_0 = bound$ and define b_{i+1} and j_i to be the values of variables b and j after line 11 in the i-th iteration of the loop in lines 9–14. Notice that $b_{i+1} = \left\lceil \frac{b_i}{r_i} \right\rceil$ and $j_i = r_ib_{i+1} - b_i$. By induction one can prove the following invariants of the loop: $b_i = \left\lceil \frac{bound}{w_i} \right\rceil$ and $\sum_{s=0}^{i-1} j_sw_s = b_iw_i - bound$.

Therefore, after the loop, we have $bound = b_mw_m - \sum_{s=0}^{m-1} j_sw_s$. It follows that the inequality $g(\bar{x}) < bound$ is equivalent to $g(\bar{x}) + \sum_{s=0}^{m-1} j_sw_s < b_mw_m$ (1). Each value j_s is set (in unary) on variables $z_1^{w_s}, \ldots, z_{r_i-1}^{w_s}$ in lines 12–14 (see also line 9 in Algorithm 1). In this way the LHS of (1) is set in the encoding generated by *encodeGoal*. The sequence $\bar{y} = (y_1, y_2, \ldots)$ represents (in unary) a value that is multiplied by w_m, thus, by adding $\neg y_{b_m}$ to *assump* in line 15, we enforce the value to be less than b_m. That ends the proof that the SAT-solver call in line 16 returns SAT if and only if both $g(\bar{x}) < bound$ and ϕ are satisfied.

4 Experimental Evaluation

Our extension of MiniSat+, based on the features explained in this paper and in the previous one [17], is available online[1]. We call it KP-MiniSat+ (**KP-MSP**, in short). It should be linked with a slightly modified COMiniSatPS[2], where the patch is also given at the link above (See footnote 1). The latest addition to the patch is an assumptions processing improvement due to Hickey and Bacchus [15]. Detailed results of the experimental evaluation are also available online[3].

The set of instances we use is from the Pseudo-Boolean Competition 2016[4]. We use instances with linear Pseudo-Boolean constraints that encode optimization problems. To this end, two categories from the competition have been selected:

– **OPT-BIGINT-LIN** - 1109 instances of optimization problems with big coefficients in the constraints (at least one constraint with a sum of coefficients greater than 2^{20}). An objective function is present. The solver must find a solution with the best possible value of the objective function.
– **OPT-SMALLINT-LIN** - 1600 instances of optimization problems. Like OPT-BIGINT-LIN but with small coefficients in the constraints (no constraint with sum of coefficients greater than 2^{20}).

We compare our solver with two state-of-the-art general purpose constraint solvers. The first one is pbSolver from PBLib ver. 1.2.1, by Tobias Philipp and Peter Steinke [28] (abbreviated to **PBLib** in the results). This solver implements a plethora of encodings for three types of constraints: at-most-one, at-most-k (cardinality constraints) and Pseudo-Boolean constraints. PBLib automatically normalizes the input constraints and decides which encoder provides the most effective translation. We have launched the program `./BasicPBSolver/pbsolver` of PBLib on each instance with the default parameters.

The second solver is NaPS ver. 1.02b by Masahiko Sakai and Hidetomo Nabeshima [30] which implements improved ROBDD structure for encoding constraints in band form, as well as other optimizations. This solver is also built on the top of MiniSat+. NaPS won two of the optimization categories in the Pseudo-Boolean Competition 2016: OPT-BIGINT-LIN and OPT-SMALLINT-LIN. We have launched the main program of NaPS on each instance, with parameters `-a -s -nm`.

We also compare our solver with the original MiniSat+ in two different versions, one using the original MiniSat SAT-solver and the other using COMiniSatPS. We label these **MS+** and **MS+COM** in the results. We present results for **MS+COM** in order to show that the advantage of using our solver does not come simply from changing the underlying SAT-solver.

1 See https://github.com/karpiu/kp-minisatp.
2 See https://baldur.iti.kit.edu/sat-competition-2016/solvers.
3 See http://www.ii.uni.wroc.pl/~karp/sat/2020.html.
4 See http://www.cril.univ-artois.fr/PB16/.

Table 1. Results summary for the OPT-BIGINT-LIN category

solver	solved	Opt	UnSat	cpu (s)	scpu (s)	avg(scpu)	smem (MB)	avg(smem)
KP-MSP++	**468**	**395**	**73**	1046518	44424	94.9	208035	444.5
KP-MSP+-	467	**395**	72	**1037085**	44886	96.1	213973	458.2
KP-MSP--	461	389	72	1039499	**37672**	**81.7**	283681	615.4
NaPS	383	314	69	1314536	51557	134.6	245533	641.1
MS+	220	149	71	1647958	47759	217.1	**42181**	191.7
MS+COM	245	174	71	1609433	54234	221.4	46336	**189.1**

Table 2. Results summary for the OPT-SMALLINT-LIN category

solver	solved	Opt	UnSat	cpu (s)	scpu (s)	avg(scpu)	smem (MB)	avg(smem)
KP-MSP++	**894**	808	86	1282788	43556	48.7	164223	183.7
KP-MSP+-	893	806	**87**	1278926	38474	43.1	162405	181.9
KP-MSP--	893	**809**	84	**1278722**	**37747**	**42.3**	153619	172.0
NaPS	887	803	84	1310006	40376	45.5	186760	210.6
PBLib	747	691	56	1611247	74993	100.4	112993	151.3
MS+	788	715	73	1515166	53566	68.0	113606	144.2
MS+COM	805	734	71	1491269	60270	74.9	**106886**	**132.8**

We are providing results for three versions of KP-MSP: (1) **KP-MSP++** that contains our algorithms and the latest modification to COMiniSatPS, (2) **KP-MSP+-** that also contains the algorithms but not the modification, and (3) **KP-MSP--** - without the algorithms and the modification, but still with optimizations of KP-MSP described in [17] (in particular, in encodings of constraints on a goal function, it reuses clauses from previous encoding by the "shared-formulas" original technique of MiniSat+). We would like to see what is the impact of new techniques on the number of solved instances and the average times and spaces used.

All the three versions of KP-MSP used default parameters, except for the parameter `-gs`, which forces the algorithm to always encode the goal function using our selection network (and the direct encoding for small sub-networks). This means that other constraints can sometimes be encoded using either BDDs or adder networks, and the original MiniSat+'s heuristics (slightly modified by us to strongly prefer encoding by sorters) decide which method is used. For example, for OPT-BIGINT-LIN instances, in all encoded non-goal constraints: 99.58% were encoded by sorters, 0.34% by BDDs and 0.08% by adders. If we consider only the successfully solved instances then the corresponding numbers are: 99.73%, 0.02% and 0.25%.

All experiments were carried out on machines with Intel(R) Core(TM) i7-2600 CPU @ 3.40 GHz. The timeout limit is set to 1800 s and the memory limit is 15 GB, which are enforced with the following commands: `ulimit -Sv 15000000` and `timeout −k 20 1809 < solver > < parameters > < instance >`.

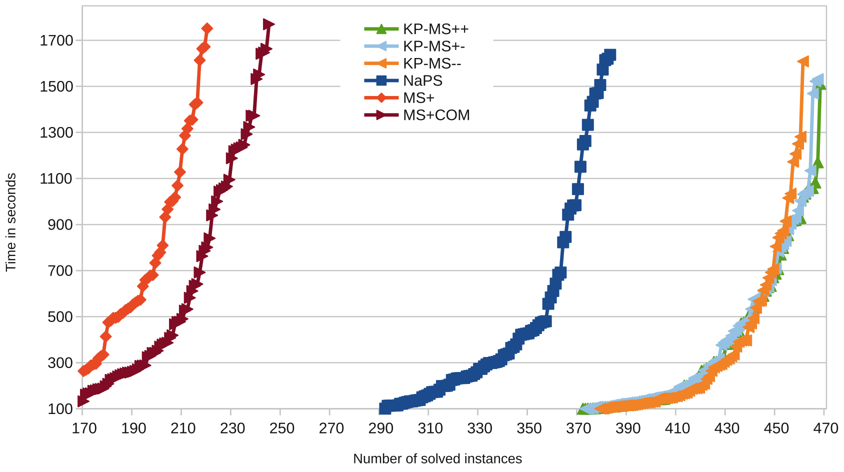

Fig. 4. Cactus plot for OPT-BIGINT-LIN division from the PB16 suite

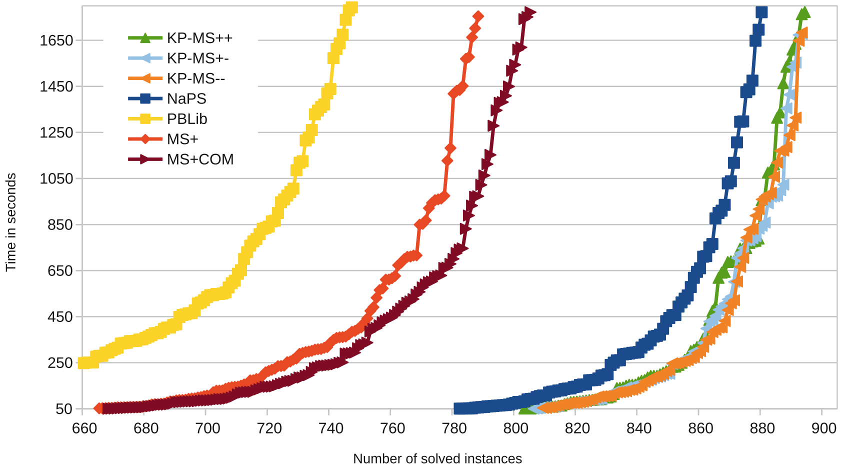

Fig. 5. Cactus plot for OPT-SMALLINT-LIN division from the PB16 suite

In Tables 1 and 2 we present results for categories OPT-BIGINT-LIN and OPT-SMALLINT-LIN, respectively. In the **solved** column we show the total number of solved instances, which is the sum of the number of instances where the optimum was found (the **Opt** column) and the number of unsatisfiable instances found (the **UnSat** column). In the **cpu** column we show the total solving time (in seconds) of the solver over all instances of a given category, and **scpu** is the total solving time over solved instances only. Similarly, **smem** is the total memory space used (in megabytes) during the computation of the solved instances. Averages have been computed as follows: $\mathbf{avg(scpu)} = \mathbf{scpu/solved}$ and $\mathbf{avg(smem)} = \mathbf{smem/solved}$.

Looking at the results, one can observe that new algorithms increase the number of solved instances to 468 in the OPT-BIGINT-LIN category. It is now almost equal to the number of 470 instances solved together by all the competitors of PB Competition 2016. The modification of COMiniSatPS add 1 solved instance and reduces the average time (by 1.2 s) and memory use (by 13.7 MB). The algorithms reduce the average memory use by 157.2 MB (KP-MSP+- versus KP-MSP--). Moreover, one can observe significant improvement in the number of solved instances in comparison to **NaPS** in this category.

In case of OPT-SMALLINT-LIN category, the differences among the results of all three versions of KP-MSP are small. It is understandable, as the coefficients of goal functions are not big in this category, thus, the sizes of mixed-radix bases are small, so the optimization techniques of [17] are equivalently efficient to the new algorithms.

In terms of memory usage **MS+** and **MS+COM** are the most efficient in this evaluation, but their overall performance is poor. Observe also that their average values are computed over much smaller sets of solved instances. Solver **PBLib** had the worst performance in this evaluation. Notice that the results of **PBLib** for OPT-BIGINT-LIN division are not available. This is because **PBLib** is using 64-bit integers in calculations, thus could not be launched with all OPT-BIGINT-LIN instances.

Figures 4 and 5 show cactus plots of the results, which indicate the number of solved instances within the time. We see a clear advantage of our solvers over the competition in the OPT-BIGINT-LIN category.

5 Conclusions

In this paper we showed that comparator networks are still competitive when used in encoding Pseudo-Boolean constraints to SAT. The popular idea of incremental encoding applied to the sorting network encoding of a pseudo-Boolean goal function leads to an increase in the number of solved instances in the OPT-BIGINT-LIN category and reduces the memory use compared to other state-of-the-art methods. The proposed modification is short and easy to implement using any modern SAT-solver which supports *assumptions*.

References

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A parametric approach for smaller and better encodings of cardinality constraints. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 80–96. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_9
2. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Mayer-Eichberger, V.: A new look at bdds for pseudo-boolean constraints. J. Artif. Intell. Res. **45**, 443–480 (2012)
3. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Generic ILP versus specialized 0–1 ILP: an update. In: Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design, pp. 450–457. ACM (2002)

4. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks and their applications. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 167–180. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_18
5. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks: a theoretical and empirical study. Constraints **16**(2), 195–221 (2011)
6. Audemard, G., Lagniez, J.-M., Simon, L.: Improving glucose for incremental SAT solving with assumptions: application to MUS extraction. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 309–317. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39071-5_23
7. Bailleux, O., Boufkhad, Y., Roussel, O.: A translation of pseudo-boolean constraints to SAT. J. Satisfiability Boolean Model. Comput. **2**, 191–200 (2006)
8. Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the 30 April–2 May 1968, Spring Joint Computer Conference, pp. 307–314. ACM (1968)
9. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Deciding CLU logic formulas via boolean and pseudo-boolean encodings. In: Proceedings of the International Workshop on Constraints in Formal Verification (CFV 2002). Citeseer (2002)
10. Codish, M., Zazon-Ivry, M.: Pairwise cardinality networks. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 154–172. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_10
11. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
12. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electron. Notes Theor. Comput. Sci. **89**(4), 543–560 (2003)
13. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. J. Satisfiability Boolean Model. Comput. **2**, 1–26 (2006)
14. Elffers, J., Nordström, J.: Divide and conquer: towards faster pseudo-boolean solving. In: IJCAI, pp. 1291–1299 (2018)
15. Hickey, R., Bacchus, F.: Speeding up assumption-based SAT. In: Janota, M., Lynce, I. (eds.) SAT 2019. LNCS, vol. 11628, pp. 164–182. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-24258-9_11
16. Karpiński, M., Piotrów, M.: Smaller selection networks for cardinality constraints encoding. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 210–225. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_16
17. Karpiński, M., Piotrów, M.: Competitive sorter-based encoding of PB-constraints into SAT. In: Berre, D.L., Järvisalo, M. (eds.) Proceedings of Pragmatics of SAT 2015 and 2018. EPiC Series in Computing, vol. 59, pp. 65–78. EasyChair (2019). https://doi.org/10.29007/hh3v. https://easychair.org/publications/paper/tsHw
18. Karpiński, M., Piotrów, M.: Encoding cardinality constraints using multiway merge selection networks. Constraints **24**(3–4), 234–251 (2019). https://doi.org/10.1007/s10601-019-09302-0
19. Knuth, D.E.: The art of computer programming. In: Sorting and Searching, 2nd edn, vol. 3. Addison Wesley Longman Publishing Co., Inc., Redwood City (1998)
20. Koshimura, M., Zhang, T., Fujita, H., Hasegawa, R.: QMaxSAT: a partial Max-SAT solver. J. Satisfiability Boolean Model. Comput. **8**, 95–100 (2012)
21. Le Berre, D., Parrain, A.: The sat4j library, release 2.2. J. Satisfiability Boolean Model. Comput. **7**(2–3), 59–64 (2010)
22. Manolios, P., Papavasileiou, V.: Pseudo-boolean solving by incremental translation to SAT. In: 2011 Formal Methods in Computer-Aided Design (FMCAD), pp. 41–45. IEEE (2011)

23. Manquinho, V.M., Roussel, O.: The first evaluation of pseudo-boolean solvers (PB'05). J. Satisfiability Boolean Model. Comput. **2**, 103–143 (2006)
24. Martins, R., Joshi, S., Manquinho, V., Lynce, I.: On using incremental encodings in unsatisfiability-based MaxSAT solving. J. Satisfiability Boolean Model. Comput. **9**(1), 59–81 (2014)
25. Nadel, A., Ryvchin, V.: Efficient SAT solving under assumptions. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 242–255. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_19
26. Oh, C.: Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL. Ph.D. thesis, New York University (2016)
27. Paxian, T., Reimer, S., Becker, B.: Dynamic polynomial watchdog encoding for solving weighted MaxSAT. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) SAT 2018. LNCS, vol. 10929, pp. 37–53. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94144-8_3
28. Philipp, T., Steinke, P.: PBLib – a library for encoding pseudo-boolean constraints into CNF. In: Heule, M., Weaver, S. (eds.) SAT 2015. LNCS, vol. 9340, pp. 9–16. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24318-4_2
29. Piotrów, M.: UWrMaxSAT-a new minisat+-based solver in maxsat evaluation 2019. MaxSAT Eval. **2019**, 11 (2019)
30. Sakai, M., Nabeshima, H.: Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. IEICE Trans. Inf. Syst. **98**(6), 1121–1127 (2015)
31. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Why decomposition is not as bad as it sounds. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 746–761. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_58
32. Shtrichman, O.: Pruning techniques for the SAT-based bounded model checking problem. In: Margaria, T., Melham, T. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 58–70. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-44798-9_4
33. Whittemore, J., Kim, J., Sakallah, K.: Satire: a new incremental satisfiability engine. In: Proceedings of the 38th annual Design Automation Conference, pp. 542–545. ACM (2001)
34. Zha, A., Koshimura, M., Fujita, H.: N-level modulo-based CNF encodings of pseudo-boolean constraints for MaxSAT. Constraints **24**(2), 133–161 (2019)