Dynamic Polynomial Watchdog Encoding for Solving Weighted MaxSAT

Tobias Paxian, Sven Reimer, Bernd Becker

Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 051
79110 Freiburg, Germany
{ paxiant | reimer | becker }@informatik.uni-freiburg.de

Abstract In this paper we present a novel pseudo-Boolean (PB) constraint encoding for solving the weighted MaxSAT problem with iterative SAT-based methods based on the Polynomial Watchdog (PW) CNF encoding. The watchdog of the PW encoding indicates whether the bound of the PB constraint holds. In our approach, we lift this static watchdog concept to a dynamic one allowing an incremental convergence to the optimal result. Consequently, we formulate and implement a SAT-based algorithm for our new Dynamic Polynomial Watchdog (DPW) encoding which can be applied for solving the MaxSAT problem. Furthermore, we introduce three fundamental optimizations of the PW encoding also suited for the original version leading to significantly less encoding size. Our experimental results show that our encoding and algorithm is competitive with state-of-the-art encodings as utilized in QMaxSAT (2nd place in last MaxSAT Evaluation 2017). Our encoding dominates two of the QMaxSAT encodings, and at the same time is able to solve unique instances. We integrated our new encoding into QMaxSAT and adapt the heuristic to choose between the only remaining encoding of QMaxSAT and our approach. This combined version solves 19 (4%) more instances in overall 30% less run time on the benchmark set of the MaxSAT Evaluation 2017. Compared to each encoding of QMaxSAT used in the evaluation, our encoding leads to an algorithm that is on average at least $2X$ faster.

1 Introduction

MaxSAT and its variations are SAT-related optimization problems seeking for a truth assignment to a Boolean formula in Conjunctive Normal Form (CNF) such that the satisfiability of the formula is maximized. Maximizing the satisfiability in a pure MaxSAT problem is to maximize the number of simultaneously satisfied clauses in the CNF. In the weighted MaxSAT variation for each clause a positive integer weight is appended and hence, the maximization of the formula is yielded if the accumulated weights of the satisfied clauses are maximized.

This work is partially supported by the DFG project “Algebraic Fault Attacks” (funding id PO 1220/7-1, BE 1176 20/1, KR 1907/6-1).

There exists a wide range of different solving techniques [1] such as branch and bound algorithms [2], iterative SAT solving [3], unsat core based techniques [4] and ILP solver [5], to name a few. A very successful approach is iterative SAT-based solving. The core idea is to adjust the bounds for the maximized result by iterative (and incremental) SAT solver calls. One possibility to do so is a direct encoding of Pseudo-Boolean (PB) constraints of the maximization objective into the SAT instance, such that the truth assignment of the whole formula directly represents the result of the maximization. By forcing the current optimization result to be larger than the last one found, this approach runs iteratively towards the optimum. The recent MaxSAT Evaluation [6] indicates that this technique can be successfully employed for unweighted and weighted MaxSAT, as the iterative SAT-based solver QMaxSAT [7] demonstrates. QMaxSAT adopts four different variations of the totalizer network [8,9,10,11] and one adder network [12] as PB encoding. A simple heuristic selects between these encodings.

In this paper we introduce a new encoding and algorithm based on the Polynomial Watchdog (PW) encoding [13]. The PW is an efficient encoding for PB constraints, though it is not designed to be employed in an iterative MaxSAT approach. Hence, we modified the original encoding by replacing the static watchdog of [13] by a dynamic one allowing to adjust the optimization goal. Based on this encoding, we provide a complete algorithm for deciding the weighted MaxSAT problem. Additionally, we introduce three fundamental optimizations/heuristics leading to significantly smaller PW encodings.

To demonstrate the effectiveness we adjoin our new encoding to the QMaxSAT solver. Experimental results on the benchmark set of the Evaluation 2017 [6] show that our encoding leads to an algorithm that is 1.) competitive in solved instances and 2.) on average $2X$ faster than existing ones. In particular, our approach is clearly superior to [8,11], especially for weighted MaxSAT instances with large clause weights. Moreover, our approach solves complementary instances with the employed adder network [12] and thus, we adjust the heuristics for deciding the used network leading to 4% more solved instances on the benchmark set of [6].

The remaining paper is structured as follows: We present related work on totalizer networks and the weighted MaxSAT application in Section 2. In Section 3 we introduce the weighted MaxSAT problem as well as the totalizer and PW encoding. In Section 4 we present our proposed dynamic PW encoding and algorithm for iterative MaxSAT solving and propose further optimizations in Section 5. Finally, we demonstrate the applicability of our new encoding and the optimizations in Section 6 and conclude the paper in Section 7.

2 Related Work

Since the original version of the totalizer network is published in 2003 [8], many different variations have been investigated since then. Some of them are also employed in context of (weighted) MaxSAT. In particular, the iterative MaxSAT solver QMaxSAT [7] uses many different variations of the totalizer network.

Namely, the original totalizer sorting network [8], weighted or generalized totalizer networks [9], mixed radix weighted totalizer [10], and modulo totalizer [11].

The original totalizer is well suited for unweighted instances. However, for weighted instances a naïve implementation does not scale in the encoding size.

The generalized totalizer [9] allows a more direct encoding of weighed inputs. This encoding is integrated into the recursive rules constructing the totalizer. The mixed radix weighted totalizer [10] is an extension of the generalized totalizer combined with the concept of mixed radix base [14].

The modulo totalizer [11] was initially developed for unweighted MaxSAT instances. It reduces the number of used clauses for the encoding by counting fulfilled clauses with modulo operations. As our experimental results show, the modulo totalizer still has scaling issues for a large sum of weights.

Our encoding is based on the Polynomial Watchdog (PW) encoding [13] which also uses totalizer sorting networks. Essentially the PW encoding employs multiple totalizer networks to perform an addition with carry on the sorted outputs. The sorting network based encoding of minisat+ described in [15] has similarities, the differences to the PW encoding are described in detail in [13]. In particular, minisat+ introduces additional logic to observe the exact bounds of the current constraint. Whereas the PW encoding utilizes additional inputs to control and observe the current bounds. Hence, we employ the PW encoding as the additional inputs are easier to manipulate for our dynamic approach.

Apart from the totalizer, other encodings for PB constraints are successfully employed for mapping the MaxSAT constraints. E.g., QMaxSAT uses an adder network [12]. This type of network is better suited for a large sum of input values than totalizer networks as adder have linear complexity in encoding size – in contrast to at least $\mathcal{O}(n \log n)$ for sorting networks.

Other encoding schemes are investigated in [15], where adder, sorting network [16] and BDD [17] implementations are compared. A BDD preserves generalized arc consistency (GAC) for PB constraints, if it can be constructed [15] – in contrast to sorting networks and adders in general. However, the enhanced encoding scheme of the Local Polynomial Watchdog (LPW) in [13] preserves GAC at the cost of encoding complexity. Another GAC preserving encoding is presented in [18] which employs a different kind of sorting networks.

All mentioned totalizer modifications only adjust the recursive rules of the totalizer. In contrast, our proposed encoding utilizes the standard totalizer and modifies the cascading version of [13].

3 Preliminaries

In this section we introduce the foundations of MaxSAT and in particular iterative PB encoding based approaches. Furthermore, we introduce the totalizer network [8] and the Polynomial Watchdog (PW) encoding [13] which are the fundamental encodings utilized in our approach.

3.1 MaxSAT

First, we introduce some basic terminologies which will be used within this paper. The input of MaxSAT problems is a Boolean formula in *Conjunctive Normal Form (CNF)*. A CNF is a conjunction of *clauses*, where a clause is a disjunction of *literals*. A clause which contains one literal is called *unit* (clause). In the following, we adopt the commonly used notation that clauses are sets of literals and a CNF is a set of clauses. A *SAT solver* decides the satisfiability problem, i.e. whether a Boolean formula φ in CNF is satisfiable. In this case, the solver returns a satisfying assignment for all variables, which is also called *model* of φ.

MaxSAT is a SAT-related optimization problem seeking for an assignment *maximizing* the number of simultaneously satisfied clauses of a CNF formula φ. The *partial* MaxSAT problem consists also of so-called *hard* clauses, which *must* be satisfied. All other clauses are called *soft* clauses. Thus, a MaxSAT formula can be formulated as follows: $\varphi = S \cup H$, where S denotes the set of soft clauses and H the set of hard clauses. The *weighted* (partial) MaxSAT problem is a generalization, where each soft clause is denoted with an integer weight w_j. The optimization goal is to maximize the accumulated weight of satisfied soft clauses.

A common approach for solving the MaxSAT problem is the *iterative SAT-based* algorithm [3] which incrementally employs a SAT solver. To do so, a *pseudo-Boolean (PB) constraint* C is directly encoded into CNF, where C is defined as $\Sigma_j a_j \cdot x_j \rhd M$ over Boolean variables x_j and positive integers a_j and M. $\rhd$ is one of the relational operators $\rhd \in \{=, >, \geq, <, \leq\}$. By using only constraints of the form $\Sigma_j a_j \cdot x_j \geq M$, the MaxSAT problem can be reduced to the question of finding a maximum value M^* still satisfying C. To do so, the soft clauses are directly connected to the PB constraint network, where x_j is true if and only if the soft clause s_j is true and the weight w_j is connected to a_j of C.

There are various methods and schemes for the encoding of PB constraints as discussed in Section 2. State-of-the-art iterative MaxSAT solvers like QMaxSAT [7] use various and customized CNF encodings. For instance, the QMaxSAT version used in the MaxSAT evaluation effectively employs three different encodings: totalizer network [8], modulo totalizer network [11] and Warners adder network [12].

Regardless of the employed encoding, the iterative approach works as follows: A PB constraint network is encoded as described above and added as hard clauses to the original CNF. For each soft clause $s_j \in S$ a so-called *relaxation* literal r_j is introduced: $s'_j = s_j \vee \overline{r_j}$ (i.e. setting r_j forces s_j to be true) and connected to the PB encoding. Let S' be the set of all modified clauses s'_j, then a SAT solver decides the $\varphi' = S' \cup H$. The returned model allows to determine the current sum of satisfied weights M. The CNF is iteratively modified by adding a constraint demanding a larger optimization result than M. The SAT solver is called incrementally with this new constraint. The whole procedure is repeated until the SAT solver returns "unsatisfiable", i.e. the last added constraint represents a result which is just larger than the optimal result. Hence, the result of the last satisfiable SAT solver call corresponds to the optimization result M^* of the MaxSAT instance.

3.2 Totalizer Network

The *totalizer network* as introduced in [8] is a unary sorting network $\Phi : \{0,1\}^n \rightarrow \{0,1\}^n$, arranging a binary input vector such that the output vector is sorted in descending order. E.g., the input vector $\langle 1,0,1,1,0\rangle$ will be processed as follows: $\Phi(\langle 1,0,1,1,0\rangle) = \langle 1,1,1,0,0\rangle$. The output vector V represents a natural number v in unary representation: if the ith entry of the output vector V is one, the unary representation matches $v \geq i$.

The totalizer sorting network allows an efficient propagation of output values on CNF. The network divides the input vector recursively into two parts until the resulting vector consists only of one element which is sorted by definition. Two unary sorted vectors are merged together by the formula Ψ. Let $U = \langle u_1, \ldots, u_k\rangle$ and $V = \langle v_1, \ldots, v_l\rangle$ be sorted vectors corresponding to unary representations of natural numbers u and v, respectively. Ψ assures that the resulting vector $W = \langle w_1, \ldots, w_{k+l}\rangle$ is the unary representation of w with $w = u + v$.

The totalizer encoding consists of two mirrored parts $D_1(a,b) = (\overline{u_a} \vee \overline{v_b} \vee w_{a+b})$ and $D_2(a,b) = (u_{a+1} \vee v_{b+1} \vee \overline{w_{a+b+1}})$, where $0 \leq a \leq k$ and $0 \leq b \leq l$. By definition $u_0 = v_0 = w_0 = 1$ and $u_{k+1} = v_{l+1} = w_{k+l+1} = 0$ holds. As stated in [8] the resulting formula $\Psi(W = U \oplus V)$ represents the relation $w = u + v$:

$$\Psi(W = U \oplus V) = \bigwedge_{a=0}^{k} \bigwedge_{b=0}^{l} D_1(a,b) \wedge D_2(a,b) \tag{1}$$

Note, by using only the encoding D_1 we guarantee $w \geq a + b$, hence, we are able to set an upper bound for w. Likewise D_2 guarantees $w \leq a + b$, i.e. a lower bound for w is represented. Note that in each case the other direction does not hold. The input vector is split up recursively in two equally sized parts U and V connected by Ψ until U or V has a size of one, which is sorted by definition. The complete encoding of all Ψ parts is called Φ.

3.3 Polynomial Watchdog Encoding Scheme

In this section we briefly introduce the *Polynomial Watchdog (PW)* encoding scheme for PB constraints C. For more details the interested reader is referred to [13]. The PW encoding uses the functions Ψ and Φ of the totalizer in order to represent one PB constraint as depicted in Fig. 1. A *Global Polynomial Watchdog (GPW)* is introduced allowing to efficiently detect a violation of M in C, with C: $\Sigma_j a_j \cdot x_j < M$. To do so, a so-called *watchdog* ω is introduced. Essentially, whenever C is falsified the literal ω will be assigned to true. I.e. a lower bound is defined and only the D_1 clauses of Ψ are needed. The GPW is defined as

$$GPW(C) = PW(C) \wedge \overline{\omega} \tag{2}$$

which guarantees the previous mentioned property.

The PW is a binary addition with carry of the weights. The coefficients of the constraints are split into their binary representation, the bits with the same weight 2^i are added to one totalizer Φ called *top bucket* TB_i with weight 2^i. The most significant bit position of all coefficients equals the number of top buckets p. These top buckets are connected appropriately representing the carry of the unary addition, where two buckets with weight 2^i and 2^{i+1} are merged applying Ψ. To do so, only every second output of the bucket of weight 2^i and every output of the 2^{i+1} bucket has to be connected into a so-called *bottom bucket* BB_{i+1} with weight 2^{i+1}. Generally, the first top buckets TB_0 and TB_1 are merged resulting in BB_1 and for all other buckets TB_{i+1} and BB_i are merged into BB_{i+1}. The bottom bucket with the largest index p is also called *last bucket*. The naming of top, bottom and last bucket is motivated by the graphical representation as seen in Fig. 1. Each output ω_m of the last bucket represents a weight which is a multiple $m = \lceil \frac{M}{2^p} \rceil$ of 2^p. Since only every second output of the low ordered buckets are used, the actual satisfied weight M is represented by $m \cdot 2^p$ minus a tare sum t of size $0 \leq t < 2^p$. This tare is added to the TB_i's using its binary representation as tare variables T_i with weight 2^i for $0 \leq i < p$. I.e. $t = \Sigma_{T_i=1} 2^i = 2^p - (M \bmod 2^p)$. Hence M can be reformulated as:

$$M = m \cdot 2^p - \Sigma_{T_i=1} 2^i \tag{3}$$

In summary, GPW adds $\overline{\omega_m}$ to guarantee a solution smaller than $m \cdot 2^p$. The exact target weight of the PB constraint C is achieved by calculating the tare values T_i a priori[1] according to Eq. 3. Consequently, the constraint $\overline{\omega_m}$ guarantees that any solution with weight $\geq M$ instantly results in a conflict.

Example 1. Given the constraint $C : 2x_1 + 3x_2 + 5x_3 + 7x_4 < 11$, the a_j values are separated due to their binary representation. As $\lfloor \log_2 7 \rfloor = 2$ holds, the largest bucket size is 2^2 and hence $p = 2$. The position of the watchdog can be achieved by: $m = \lceil \frac{M}{2^p} \rceil = \lceil \frac{11}{2^2} \rceil = 3$. The tare values can be calculated with: $t = \Sigma_{T_i=1} 2^i = 2^p - (M \bmod 2^p) = 2^2 - (11 \bmod 2^2) = 1$, i.e. the binary representation of the tare values is 1_{10} leading to: $T_0 = 1, T_1 = 0$. We can check our result by applying them into Eq. 3: $M = m \cdot 2^p - \Sigma_{T_i=1} 2^i = 3 \cdot 2^2 - (1 \cdot 2^0 + 0 \cdot 2^1) = 11$. This leads to the following PW encoding in Figure 1. Note, the blue dashed lines in this figure are not present in the actual encoding, since $T_1 = 0$.

In [13] it is stated that merging two totalizers of size n requires $\mathcal{O}(n^2)$ clauses and for the whole totalizer $\mathcal{O}(n^2 \log(n))$ clauses are required. The complete PW encoding complexity is given by $\mathcal{O}(n^2 \log(n) \log(a_{max}))$ clauses, where a_{max} is the largest integer weight of all clauses of the MaxSAT problem.

However, as already stated in [8] the number of encoded clauses of the whole totalizer can be bounded by $\mathcal{O}(n^2)$ and not $\mathcal{O}(n^2 \log(n))$. Since we have no doubt about the remaining argumentation of [13], we conclude that the number of clauses of the (G)PW encoding is actually in $\mathcal{O}(n^2 \log(a_{max}))$.

Note, in [13] the concept of a *Local Polynomial Watchdog (LPW)* is introduced which preserves GAC. The complexity is given by $\mathcal{O}(n^3 \log n \log(a_{max}))$. Likewise, we propose that also this complexity has to be corrected to $\mathcal{O}(n^3 \log(a_{max}))$. Still,

[1] Note that tare values of zero do not have any influence. Hence, only the tare bits $T_i = 1$ are added to TB_i, and are also directly set to 1.

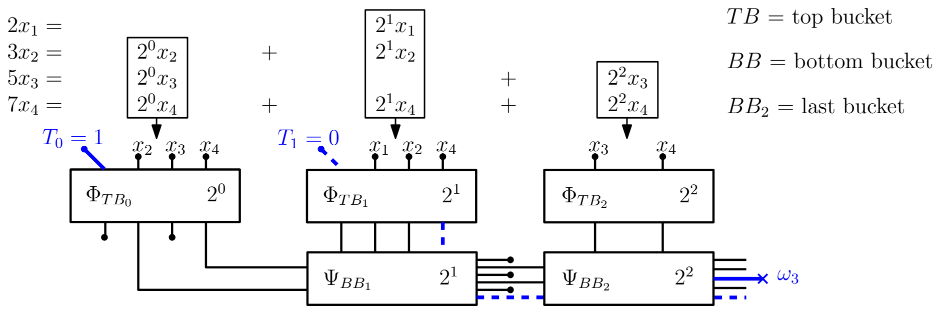


Figure 1: Polynomial Watchdog Encoding Scheme

we do not apply the LPW since the encoding size is not feasible for our application. (G)PW in contrast does not maintain GAC.

4 Dynamic PW Encoding and GPW Algorithm

In this section we introduce our new dynamic PW encoding scheme. We state details of the encoding adjustments and the employment in an iterative MaxSAT solver. The principle of our approach is to lift the static watchdog as described in [13] to a dynamic version, which allows to set a lower bound M of the PB constraint dynamically for optimizing this bound.

The remainder of this section is as follows: we introduce the adjustments of the original PW encoding in Section 4.1 and present a complete algorithm to solve the MaxSAT problem based on this encoding in Section 4.2.

4.1 Dynamic PW Encoding

In order to employ the PW encoding for representing MaxSAT constraints, we need to allow different watchdog positions and consequently lift the concept of statically a-priori set tare values and watchdog positions to a dynamic one. Thus, we call our modification the *Dynamic Polynomial Watchdog (DPW)* encoding.

As mentioned in Section 3.1 the MaxSAT problem can be reduced to find the optimal M^* in a constraint C with $\Sigma_j a_j \cdot x_j \geq M$, where all x_j's and a_j's are appropriately connected to all r_j's and w_j's of the MaxSAT problem leading to $\Sigma_j w_j \cdot r_j \geq M$ in our approach. Note, r_j implies "s_j is satisfiable" but not vice versa. However, the r_j's still represent a lower bound, we will revisit this issue later. In order to increment M, we need to adjust the watchdog and tare values appropriately. Section 4.2 state further details on this adjustment.

Analogously to [13] we define the *Dynamic Global Polynomial Watchdog (DGPW)* as follows:

$$DGPW_i(C) = DPW(C) \wedge \omega_i \tag{4}$$

Here, i corresponds to the ith output of the last bucket. Note, the *GPW* watchdog as introduced in Eq. 2 represents an *upper bound* for the unary representation, whereas the *DGPW* watchdog in Eq. 4 is a *lower bound*. From a different perspective: the *DGPW* requires a minimum number of ones at the output vector, whereas the *GPW* demands a minimum number of zeros. Thus, our *DPW* encoding only employs the D_2 part of the totalizer.

If $DGPW_m$ is satisfied, we can conclude the *DPW* encoding fulfills a total sum of weights of at least $m \cdot 2^p$. According to Eq. 3 the actual bound M for the constraint is achieved by subtracting the tare values which are set to 1. Note, the DPW encoding adds *all* tare values T_i to the top buckets TB_i. Further, the tare variables are not set to a precomputed value. We rather allow the SAT solver to alter the logic value of these variables. Hence, one crucial part of the algorithm is to efficiently determine exact values for the tares.

Example 2. Reconsider Example 1: The DPW additionally adds the tares T_0 and T_1, i.e. the blue dashed lines are part of the DPW encoding. By using D_2 we change the operator of the underlying constraint C from $<$ to $\geq$. If we use the same watchdog position 3 applying $DGPW_3$ and fix the tare values as in Example 1, the represented constraint is $C : 2x_1 + 3x_2 + 5x_3 + 7x_4 \geq 11$.

Based on this encoding we formulate an algorithm leading to the optimization result of the original MaxSAT formulation.

4.2 Dynamic GPW Algorithm

Our algorithm is separated in two phases. First, we apply a *Coarse Convergence (CC)* and finalize with a *Fine Convergence (FC)* as follows:

CC: The watchdog position is increased until the formula is unsatisfiable.
FC: Refines the result of CC by adjusting the tare variables appropriately.

Coarse Convergence (CC) Algorithm 1 gives an overview of the CC phase. It takes the complete CNF of the MaxSAT problem and DPW encoding as input and returns the maximum position m^* of the last bucket for which the constraint is still satisfied.

First we perform an initial SAT solver call in line 2 without additional (watchdog) constraints returning an initial watchdog position. We increment the watchdog connected to the last bucket until the formula is not satisfiable anymore (cf. lines 3-13): Therefore, we calculate the next watchdog position based on the current model (cf. GetLastSatPos in line 4) seeking for the last position of the last bucket for which the model is set to true. Since we know that this position represents a lower bound of our solution, we increment it and add the resulting watchdog as an assumption for the next SAT solver call (cf. GetWatchdog in lines 5 and 6). If the result is satisfiable we add the last assumption as unit to the CNF allowing the solver to simplify the CNF representation. This is repeated until the solver returns "unsatisfiable", i.e. we found the maximum position m^*.

Algorithm 1 Coarse Convergence

```
1: procedure CoarseConvergence(CNF)
2:     ⟨result, model⟩ ← Solve(CNF);
3:     while true do
4:         position ← GetLastSatPos(model);
5:         assumption ← GetWatchdog(position+1));
6:         ⟨result, model⟩ ← Solve(CNF + assumption);
7:         if result = SAT then
8:             CNF ← AddUnitClause(assumption);
9:         else
10:            CNF ← AddUnitClause(GetWatchdog(position));
11:            return position;                          ▷ Return last SAT position
12:        end if
13:    end while
14: end procedure
```

By doing so, we have determined the first part of Eq. 3 leading to $M^* = m^* \cdot 2^p - \Sigma_{T_i=1} 2^i$. Note, up to this point, no constraints are assumed for the tare variables T_i. Hence, the current model of the tare values does not correspond to the optimal solution. We have to adjust these values as stated in the next subsection. However, the result of the *CC* phase states the possible solution interval for our searched optimal bound M^* as follows: $(m^* - 1) \cdot 2^p < M^* \leq m^* \cdot 2^p$.

We further optimize the GetLastSatPos function of line 4: As mentioned in Section 3.1, s'_j is satisfied if r_j is set to true. Nevertheless, r_j might be false and simultaneously s_j is satisfied, too. This result cannot be seen by our *GPW* encoding since only the r_j values are connected. Thus, we check each and every soft clause if s_j is satisfied regardless of the value of r_j. Finally, we add up the weight of every *actual* satisfied soft clause resulting in an actual current optimal weight $\hat{M}$ for which holds: $\hat{M} \geq M$. By $\hat{m} = \lceil \frac{\hat{M}}{2^p} \rceil$ we obtain the watchdog position $\hat{m}$ of this optimum. Note, in this case we can immediately add another unit clause to the CNF. As $\hat{M}$ might be larger than M we also might skip output positions of the last bucket and hence, we can add the unit corresponding to the position $\hat{m}$ before calling the SAT solver in line 6 of Algorithm 1. Note that we actually do not need this additional "by-chance" concept, if we would consider appropriate constraints representing the relation $\overline{r_j} \rightarrow \overline{s_j}$, where $\overline{s_j}$ indicates that every literal of the soft clause s_j is falsified. By adding these constraints, r_j would be true iff the soft clause s_j is falsified. However, by adding these constraints, we would lose the potential of this by-chance mechanism as all literals of s_j would be immediately set to false whenever r_j is set to true.

Fine Convergence (FC) Once, we found the coarse solution interval, we seek for the exact result by adjusting the tare variables as part of the Fine Convergence (FC) phase. The general idea is the same as in the CC phase: we force the SAT solver to find a better solution than the current one. Algorithm 2 summarizes our approach. In addition to the modified CNF resulting from Alg. 1, the procedure also gets the last model from a satisfying SAT solver call from the CC phase as an input.

Algorithm 2 Fine Convergence

```
 1: procedure FineConvergence(CNF, model)
 2:     for ($n = p - 1;\ n \geq 0;\ n = n - 1$) do
 3:         if model[$T_n$] = false then
 4:             CNF ← AddUnitClause(Negate($T_n$));
 5:         else
 6:             assumption ← Negate($T_n$);
 7:             ⟨result, model⟩ ← Solve(CNF + assumption);
 8:             if result = SAT then
 9:                 CNF ← AddUnitClause(Negate($T_n$));
10:             else
11:                 CNF ← AddUnitClause($T_n$));
12:             end if
13:         end if
14:     end for
15: end procedure
```

First, consider the following observation: In Eq. 3 we have defined the upper bound $(m \cdot 2^p)$ as reference point. Instead, we can also define the value of M using the lower bound $(m-1) \cdot 2^p + 1$ as reference:

$$M = (m-1) \cdot 2^p + 1 + \Sigma_{T_i=0} 2^i \tag{5}$$

As Eq. 5 indicates the tare weights for which the corresponding tare is not satisfied is added to the lower bound. Hence, the tare is only a remainder of the sum of all satisfied weights relating to the lower bound. Consequently, in order to maximize the value of M, we have to maximize $\Sigma_{T_i=0} 2^i$.

Algorithm 2 iterates over all tare variables, from the most significant T_{p-1} to the least significant T_0. If the current tare T_n is already 0, we can add the corresponding unit to our CNF (cf. lines 3 and 4). Otherwise, we have to check whether we can set T_n to zero (and thus maximize the sum of weights) by adding the assumption $T_n = false$ to the solver (cf. lines 6 and 7). If the SAT solver returns satisfiable, we proceed as in line 4 by adding the corresponding unit. Otherwise, it is ensured that the PB constraint is always violated with $T_n = false$, and hence we can fix this tare to true. Note, the last property only holds, if we iterate from the most to the least significant tare position. By doing so, essentially a binary search of the maximum possible tare weight is performed.

As in the coarse convergence, we can explicitly calculate the actual current weight $\hat{M}$. By doing so, we may skip tare positions: If $\hat{M}$ implies that the most significant open tare position T_n must be set to zero, we can directly add the corresponding unit (cf. line 4) and proceed with the next tare position.

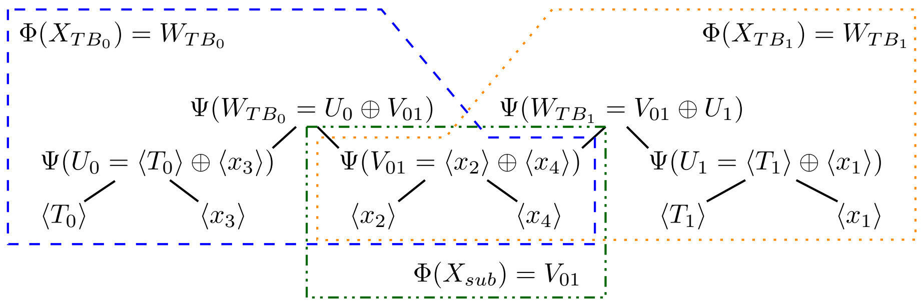

$\Phi(X_{TB_0}) = W_{TB_0}$
$\Phi(X_{TB_1}) = W_{TB_1}$
$\Psi(W_{TB_0} = U_0 \oplus V_{01})$
$\Psi(W_{TB_1} = V_{01} \oplus U_1)$
$\Psi(U_0 = \langle T_0\rangle \oplus \langle x_3\rangle)$
$\Psi(V_{01} = \langle x_2\rangle \oplus \langle x_4\rangle)$
$\Psi(U_1 = \langle T_1\rangle \oplus \langle x_1\rangle)$
$\langle T_0\rangle$
$\langle x_3\rangle$
$\langle x_2\rangle$
$\langle x_4\rangle$
$\langle T_1\rangle$
$\langle x_1\rangle$
$\Phi(X_{sub}) = V_{01}$

Figure 2: Caching of Adder: Φ applied to TB_0 and TB_1 of Example 1. The sorted vector V_{01} can be used for the encoding of TB_0 and TB_1.

5 PW Encoding Optimizations

In this section we state three fundamental optimizations on the PW encoding. In Section 5.1, we describe a technique which allows reusing already encoded parts of the network. Note, this optimization concept is already mentioned in [13] as future work. In Section 5.2, we propose an approach for determining the cone-of-influence of encoded outputs of the network. By doing so, we are able to incrementally build the needed parts of the encoding, leading to a significant reduction of the encoding size. In Section 5.3, we present a concept to set any weight as lower or upper bound. Applying this to the original PW encoding allows to use any operator for the PB constraint directly without converting the formula. Note, all concepts can be utilized in the classical PW encoding and moreover do not affect each other. I.e., utilizing one of the following techniques does not (negatively) influence the efficiency of one of the others, in general.

5.1 Caching of Adder

The PW encoding consists of many shared sub-formulae Ψ among different buckets which actually need be encoded only once [13] as shown in Example 3.

Example 3. Considering Example 1, both TB_0 and TB_1 contain an identical subset of input variables $X_{sub} = \{x_2, x_4\}$. This subset can be encoded once as $\Phi(X_{sub})$ and reused for TB_0 and TB_1 as shown in Figure 2. The dashed and dotted boxes indicate involved buckets, and the dash-dotted box the shared part.

There is another subset $X_{sub2} = \{x_2, x_3\}$ of TB_0 and TB_2 in Example 1, which could be proceeded likewise. Note, if we try to share both subsets X_{sub} and X_{sub2} at the same time, x_4 of TB_0 will be encoded twice. This additional encoding will degrade the outcome of the method and should therefore be avoided.

The core idea of the Adder Caching (AC) is to reuse the encoding of this shared parts whenever such a sub-formulae is identified. Unfortunately, merging buckets/sub-formulae is quite sophisticated, since caching of one sub-formula influences the upcoming operations, as Example 3 demonstrates. In general, the problem of finding an optimal solution is at least NP-hard as it can be reduced to a set cover problem. Hence, we implemented various heuristics in order to decide which parts to share. Our heuristics rely on the different cost estimations of one caching operation. As measurements, we tried several static parameters which can be calculated a priori: number of encoded clauses, the bucket sizes, number of possible follow up cache operations and number of cache operations, to name a few. Although all heuristics were able to significantly reduce the encoding size, none of them has a significant impact in terms of solved instances or run time.

Instead, we developed a heuristic which is not as effective in terms of encoding size, but has a significant impact on run time as experimental results show. We collocate the sorter inputs according to their corresponding soft clause weight, such that for each weight w, there is a list of corresponding inputs with weight w. Then, for one weight w exactly one totalizer Φ is encoded, i.e. if there are n soft clauses with identical weight w, the totalizer for sorting these n soft clauses is only built once. This sub-formula is shared over all buckets needed for the binary representation of weight w. This is repeated for all weights, and finally the top bucket are constructed by connecting the built sub-formulae using Ψ. In contrast to all other heuristics, multiple cache steps are considered at once as *all* involved adders are merged. We assume that this is one reason why the other heuristics are not as effective and suggest future work on this topic.

5.2 Cone-of-Influence Encoding

As the methodology in the previous section reduces the cost of top buckets, we develop a technique mainly reducing the encoding size of the bottom buckets. Note, the encoding of bottom buckets usually dominates the whole PW encoding as these buckets have more inputs than top buckets. We apply this technique within the CC phase, where the watchdog position is incremented.

The cone-of-influence (COI) encoding converts only needed parts of the (D)PW into CNF. Consider a standard $GPW(C) = PW(C) \wedge \overline{\omega_i}$. We observed that only the output ω_i needs to be encoded for deciding GPW as the encoding of Ψ guarantees *at most* $i-1$ ones at the output. By encoding only ω_i it is ensured to create at most $\mathcal{O}(n)$ clauses for the last bucket with n inputs, instead of $\mathcal{O}(n^2)$. Depending on the position of ω_i in the output vector W, even more encoding size can be saved: positions close to the borders of W lead to smaller encodings than in the middle as the upcoming Ex. 4 will motivate.

We introduce a binary tree, which represents the recursive construction of the totalizer encoding including the information whether a variable is already encoded. Hence, each node represents a snapshot of the current Ψ encoding for all partially sorted outputs. Before encoding a specific (output) variable, the tree nodes directly indicate which variables and clauses have to be added and which are already encoded. We traverse the tree from the root to the leaves collecting all clauses within the cone-of-influence of the demanded variable considering also introduced helper variables. Note, the tree also represents and considers the carry inputs of the bottom buckets where only every second output is encoded.

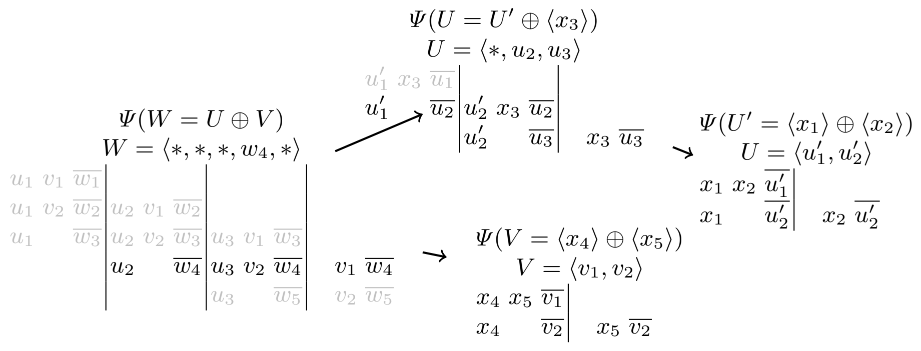

$\Psi(W = U \oplus V)$
$W = \langle *, *, *, w_4, * \rangle$
$\Psi(U = U' \oplus \langle x_3 \rangle)$
$U = \langle *, u_2, u_3 \rangle$
$\Psi(U' = \langle x_1 \rangle \oplus \langle x_2 \rangle)$
$U = \langle u'_1, u'_2 \rangle$
$\Psi(V = \langle x_4 \rangle \oplus \langle x_5 \rangle)$
$V = \langle v_1, v_2 \rangle$

Figure 3: Binary tree representing the cone-of-influence of w_4.

Example 4. Figure 3 illustrates our generated tree. Each node is represented by a table indicating the clauses created by the corresponding Ψ. The ith row in the table shows the needed clauses for the ith output entry according to Ψ. Consider that only w_4 (as seen at the very left) has to be encoded as it might be the current watchdog. By applying our cone-of-influence technique, we only need to consider the 12 highlighted clauses, whereas 11 clauses (in gray) could be saved for the encoding. A $*$ in the vector indicates that the corresponding variable (row of table) is not encoded. Moreover, assume that we may only consider w_5: in this case we just need to encode the very last row of each table.

5.3 Exact Bound Encoding

We explicitly enforce specific values as a lower or upper bound, where we need to encode D_2 for the lower and D_1 for the upper bound. Note, for an exact upper bound $< M$, we need constraints of the form $\overline{r_j} \Rightarrow \overline{s_j}$ and set ω_i to false as in Section 3.3. In both cases, we have to adjust the tare values by utilizing Eq. 3.

We utilize the Exact Bound (EB) encoding for setting the next sum of weights in our CC phase. Instead of incrementing the next watchdog position, we explicitly add sufficient assumptions to enforce the weight $\hat{M} + 1$ for the next solver call. By doing so, instances are easier to solve for the SAT solver (since we add more and specific assumptions leading to the solution), but the number of solver calls is increased in the worst case. However, as experimental results show, we actually often converge faster to the optimum, also due to weights satisfied by soft clauses by chance (cf. computation of $\hat{M}$ in Section 4.2).

We also employ a restricted version of the exact upper bound encoding by setting the first unsatisfied watchdog of the CC phase to $\overline{\omega_{m^*+1}}$. Note, we do not restrict the tare values or add $\overline{r_j} \Rightarrow \overline{s_j}$ constraints. By doing so, we implicitly forbid assignments of the relaxation literals leading to a weight $> m^* \cdot 2^p$ and thus guiding the SAT solver. We only employ the restricted upper bound in combination with the COI encoding of Section 5.2 in order to avoid the additional encoding of $\mathcal{O}(n^2)$ D_1 clauses for the last bucket.

6 Experimental Results

We implemented the new encoding scheme and algorithm in C++ as extension of the QMaxSAT solver [7] as used in the MaxSAT Evaluation 2017 [6].

All experiments were run on one Intel Xeon E5-2650v2 core at 2.60 GHz, with 64 GB of main memory and Ubuntu Linux 16.04 in 64-bit mode as operating system. We aborted all experiments whose computation time exceeded $3,600$ CPU seconds or which required more than 32 GB of memory as in the evaluation of 2017. We also used the benchmark set of [6] consisting of 767 weighted instances.

Tab. 1 shows the efficiency of our optimizations (Section 5) of the (D)GPW. As results show, AC and COF have significant impact on either the number of encoded variables or clauses leading to a encoding size of 50% compared to Plain. Whereas, EB is not designed to decrease the encoding size, but still has a significant impact on the run time. Compared to Plain, DGPW is about $2X$ faster and has almost $3X$ less encoding size in number of clauses and variables wrt. the commonly solved instances. Moreover, 28 more instances could be solved.

Tab. 2 compares the number of solver calls and run time for the two solving phases as well as SAT solver result for the DPGW[2]. In total only 21 SAT solver calls are needed on average, most of them are satisfiable calls in the CC phase. The comparison between the average and median time shows that for easy instances the most time is spent in the CC phase and for hard instances it is the FC phase. The same holds for satisfiable and unsatisfiable solver calls.

In a second experiment, we compare DGPW with QMaxSAT. Tab. 3 shows the results on the QMaxSAT encodings. The AutoQD heuristic chooses between adder and DGPW as (modulo) totalizer are inferior to our encoding (cf. Fig. 4a). Our heuristic chooses DGPW if either the sum of weights is small ($< 400,000$) or large ($> 2,000,000,000$). In all other cases the adder is chosen. The original QMaxSAT AutoQ heuristic also chooses (modulo) totalizers for a small sum of weights ($< 2^{17}$) as they usually outperform adders in this case. In addition, our empirical results show that DGPW dominates the adder for huge weights. Tab. 3 is composed as Tab. 1 comparing two neighboring columns wrt. the commonly solved instances. As expected, the adder needs two orders of magnitudes less clauses due to linear encoding complexity, and the totalizer needs two orders of magnitudes more clauses due to the naive encoding of weights. Fig. 4 underlines our results for the individual encodings in a scatter and cacti plot.

Notably, our results are comparable to the MaxSAT evaluation where AutoQ also solved 503 instances with an average run time of 385.18 seconds [6]. DGPW is competitive in the number of solved instances wrt. the other networks (470 vs. 228, 329, and 491). Remarkably, DGPW is at least $2X$ faster than every other network for the commonly solved instances. The new VBS solves 31 more instances in 60% of the run time, whereas our basic AutoQD solves 19 more instances than the evaluation version of QMaxSAT in overall 70% of the run

[2] The run time difference to Tab. 1 is caused by the time needed for the encoding and the remaining part of our algorithmn (e.g. analyzing the SAT solver model).

Table 1: Comparison of *DGPW* without extensions (Plain) with adder caching (AC), cone-of-influence (COI), exact bound (EB) and a combined (DGPW) version using all optimizations. First, the average run time and the total number of solved instances are given. Two neighboring columns compare an optimization with Plain opposing the average run time, median run time and encoding size wrt. the commonly (com.) solved instances. The encoding size is given by the average number of clauses "#cl" (in millions) and variables "#var".

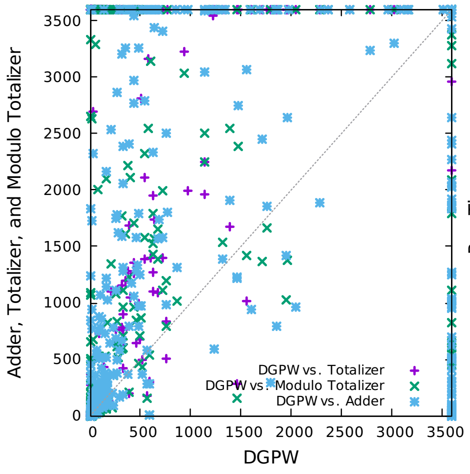
		Plain	AC	Plain	COI	Plain	EB	Plain	DGPW
	#instances	442	453	442	455	442	451	442	470
	avg time	323.00	269.63	323.00	313.93	323.00	297.47	323.00	264.16
	med time	20.56	12.52	20.56	23.26	20.56	19.71	20.56	12.05
com.	#instances	430	430	439	439	439	439	429	429
com.	avg time	302.89	207.07	302.51	288.84	303.08	251.59	306.73	164.60
com.	med time	16.55	10.04	18.48	17.98	18.48	16.48	17.22	7.60
com.	avg #cl	29.19	21.32	28.62	14.66	28.59	28.62	29.26	9.78
com.	avg #var	96,803	53,639	95,324	80,295	95,298	95,512	97,000	38,336

Table 2: DGPW divided by Coarse Convergence (CC), Fine Convergence (FC), satisfiable (SAT) and unsatisfiable (UNSAT) solver calls. For each phase the average/median number of the solver calls and solving time in seconds are given.

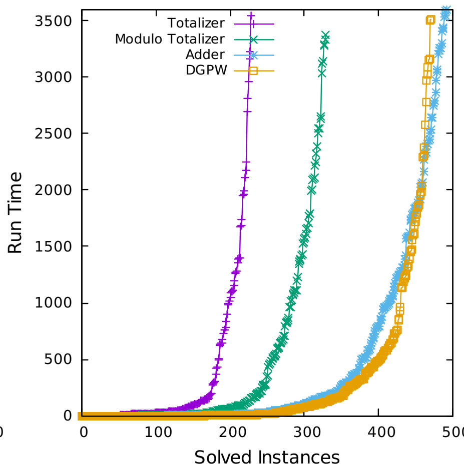
	CC		FC		SAT		UNSAT	
	avg	med	avg	med	avg	med	avg	med
solver calls	14.33	9.00	6.69	3.00	19.29	12.00	1.72	2.00
solving time	120.30	4.93	139.20	1.62	117.73	6.17	141.77	0.87

Table 3: Comparing DGPW with totalizer (Tot), modulo totalizer (ModT) and adder (Add). The virtual best solver (VBS) of the QMaxSAT with integrated DGPW (VBSQD) and without (VBSQ) is depicted. Finally, results are shown for the original heuristic of QMaxSAT (AutoQ) and our adopted one (AutoQD).

		Tot	DGPW	ModT	DGPW	Add	DGPW	VBSQ	VBSQD	AutoQ	AutoQD
	#instances	228	470	329	470	491	470	504	535	503	522
	avg time	326.29	264.16	372.49	264.16	430.39	264.16	381.04	301.44	408.30	334.44
	med time	33.12	12.05	21.60	12.05	29.59	12.05	26.96	19.01	28.68	23.34
com.	#instances	225	225	313	313	428	428	504	504	497	497
com.	avg time	303.45	142.78	330.64	141.23	388.37	163.16	381.04	228.60	380.95	268.35
com.	med time	32.12	5.01	18.48	3.69	17.39	7.53	26.96	13.55	27.62	19.80
com.	avg #cl	28.57	0.60	11.61	2.88	0.17	16.45	1.92	12.37	0.33	6.70
com.	avg #var	63,920	11,901	108,006	23,341	45,768	57,404	60,796	56,062	51,328	48,890



(a) Run time comparison per instance



(b) Solved instances for a given run time

Figure 4: Comparing different QMaxSAT encodings (adder, totalizer and modulo totalizer) with our newly introduced *DGPW* encoding

time, both wrt. the commonly solved instances. Thus, there is still some potential for our heuristic, however we conserve most of the combined capacity.

7 Conclusions

In this paper, we presented a new encoding scheme for mapping the weighted MaxSAT problem to the Polynomial Watchdog (PW) encoding. To do so, we extended the original encoding by the support of dynamic watchdogs and tare variables. Furthermore, we introduced three optimizations for the PW encoding.

As experimental results show, our optimizations lead to $3X$ smaller encoding sizes and $2X$ faster run times on average compared with the original PW encoding in [13]. Furthermore, we showed the applicability of our new encoding scheme while achieving a speed-up of more than $2X$ compared to the competitors. Finally, we integrated our encoding into a state-of-the-art MaxSAT solver and implemented a prototypical heuristic for deciding the encoding used.

The presented encoding could also be used to handle PB constraints in other solvers like Open-WBO [19]. As future work, we want to investigate the minimization of bucket sizes, which is usually the bottleneck of the PW encoding, by multiplying or dividing the weights with a common factor, and thus changing the binary representation. Moreover, we plan to investigate further heuristics for the adder caching as there is even more potential for the LPW presented in [13].

References

1. Menai, M.E.B., Al-Yahya, T.N.: A taxonomy of exact methods for partial Max-SAT. Journal of Computer Science and Technology **28**(2) (2013) 232–246
2. Hansen, P., Jaumard, B.: Algorithms for the maximum satisfiability problem. Computing **44**(4) (1990) 279–303
3. Zhang, H., Shen, H., Manya, F.: Exact algorithms for MAX-SAT. Electronic Notes in Theoretical Computer Science **86**(1) (2003) 190–203
4. Marques-Silva, J., Planes, J.: Algorithms for maximum satisfiability using unsatisfiable cores. In: Advanced Techniques in Logic Synthesis, Optimizations and Applications. Springer (2011) 171–182
5. Ansótegui, C., Gabàs, J.: Solving (weighted) partial MaxSAT with ILP. In: CPAIOR. (2013) 403–409
6. : Twelfth MaxSAT evaluation (2017) `http://mse17.cs.helsinki.fi/index.html`.
7. Koshimura, M., Zhang, T., Fujita, H., Hasegawa, R.: QMaxSAT: A partial Max-SAT solver system description. Journal on Satisfiability, Boolean Modeling and Computation **8** (2012) 95–100
8. Bailleux, O., Boufkhad, Y.: Efficient CNF encoding of boolean cardinality constraints. In: Principles and Practice of Constraint Programming–CP 2003, Springer (2003) 108–122
9. Joshi, S., Martins, R., Manquinho, V.: Generalized totalizer encoding for pseudo-boolean constraints. In: International Conference on Principles and Practice of Constraint Programming, Springer (2015) 200–209
10. Uemura, N., Fujita, H., Koshimura, M., Zha, A.: A sat encoding of pseudo-boolean constraints based on mixed radix (3 2017)
11. Ogawa, T., Liu, Y., Hasegawa, R., Koshimura, M., Fujita, H.: Modulo based CNF encoding of cardinality constraints and its application to MaxSAT solvers. In: Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on, IEEE (2013) 9–17
12. Warners, J.P.: A linear-time transformation of linear inequalities into conjunctive normal form. Information Processing Letters **68**(2) (1998) 63–69
13. Bailleux, O., Boufkhad, Y., Roussel, O.: New encodings of pseudo-boolean constraints into CNF. In: International Conference on Theory and Applications of Satisfiability Testing, Springer (2009) 181–194
14. Codish, M., Fekete, Y., Fuhs, C., Schneider-Kamp, P.: Optimal base encodings for pseudo-boolean constraints. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer (2011) 189–204
15. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. In: Journal on Satisfiability, Boolean Modeling and Computation. Volume 2. (2006) 1–26
16. Batcher, K.E.: Sorting networks and their applications. In: AFIPS Spring Joint Computing Conference, ACM (1968) 307–314
17. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. Computers, IEEE Transactions on **100**(8) (1986) 677–691
18. Manthey, N., Philipp, T., Steinke, P.: A more compact translation of pseudo-boolean constraints into cnf such that generalized arc consistency is maintained. In: Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz), Springer (2014) 123–134
19. Martins, R., Manquinho, V., Lynce, I.: Open-wbo: A modular maxsat solver,. In Sinz, C., Egly, U., eds.: Theory and Applications of Satisfiability Testing – SAT 2014, Cham, Springer International Publishing (2014) 438–445