Everything You Always Wanted to Know About Blocked Sets (But Were Afraid to Ask)

Tomáš Balyo[1], Andreas Fröhlich[2], Marijn J. H. Heule[3], Armin Biere[2] [*]

[1] Charles University, Prague, Czech Republic
Faculty of Mathematics and Physics
[2] Johannes Kepler University, Linz, Austria
Institute for Formal Models and Verification
[3] The University of Texas at Austin
Department of Computer Science

Abstract. Blocked clause elimination is a powerful technique in SAT solving. In recent work, it has been shown that it is possible to decompose any propositional formula into two subsets (blocked sets) such that both can be solved by blocked clause elimination. We extend this work in several ways. First, we prove new theoretical properties of blocked sets. We then present additional and improved ways to efficiently solve blocked sets. Further, we propose novel decomposition algorithms for faster decomposition or which produce blocked sets with desirable attributes. We use decompositions to reencode CNF formulas and to obtain circuits, such as AIGs, which can then be simplified by algorithms from circuit synthesis and encoded back to CNF. Our experiments demonstrate that these techniques can increase the performance of the SAT solver Lingeling on hard to solve application benchmarks.

1 Introduction

Boolean satisfiability (SAT) solvers have seen lots of progress in the last two decades. They have become a core component in many different areas connected to the analysis of systems, both in hardware and software. Also, the performance of state-of-the-art satisfiability modulo theories (SMT) solvers often heavily relies on an integrated SAT solver or, as for example when bit-blasting bit-vectors, just encodes the SMT problem into SAT and then uses the SAT solver as reasoning engine. This gives motivation to develop even more efficient SAT techniques.

One crucial factor for the performance of state-of-the-art SAT solvers are sophisticated preprocessing and inprocessing techniques [1]. Conjunctive normal form (CNF) level simplification techniques, such as blocked clause elimination (BCE) [2,3,4], play an important role in the solving process. A novel use of blocked clauses was proposed in [5], which showed, that it is possible to decompose any propositional formula into two so called blocked sets, both of which can

be solved solely by blocked clause elimination. This blocked clause decomposition was then used to efficiently find backbone variables [6] and implied binary equivalences through SAT sweeping. Using this technique the performance of the state-of-the-art SAT solver Lingeling [7] could be improved on hard application benchmarks from the last SAT Competition 2013.

The success of these techniques gives reason to investigate blocked clauses in more detail. In this paper, we revisit topics from previous work [5] and present several new results. The paper is structured as follows. In Sect. 2, we first give definitions used throughout the rest of the paper. We introduce the notion of *blocked sets*, and, in Sect. 3, present new properties of blocked sets. We then discuss several new and improved ways to efficiently solve blocked sets in Sect. 4. In Sect. 5, we revisit *blocked clause decomposition* (BCD), as earlier proposed in [5], and suggest extensions to increase performance of decomposition algorithms and to generate blocked sets with certain useful attributes. We apply these techniques in Sect. 6 to extract compact circuit descriptions in AIG format from arbitrary CNF inputs, allowing the use of sophisticated circuit simplification techniques as implemented in state-of-the-art synthesis tools and model checkers like ABC [8,9]. As described in Sect. 7, this extraction mechanism can also be used to reencode the original formula into a different CNF which can sometimes be solved more efficiently by existing SAT solvers. All our experimental results are discussed in Sect. 8. We conclude our paper in Sect. 9.

2 Preliminaries

In this section, we give the necessary background and definitions for existing concepts used throughout our paper.

CNF   Let F be a Boolean formula. Given a Boolean variable x, we use x and $\overline{x}$ (alternatively $\neg x$) to denote the corresponding positive and negative literal, respectively. A clause $C := (x_1 \vee \cdots \vee x_k)$ is a disjunction of literals. We say F is in conjunctive normal form (CNF) if $F := C_1 \wedge \cdots \wedge C_m$ is a conjunction of clauses. Clauses can also be seen as a set of literals. A formula in CNF can be seen as a set of clauses. A clause is called a unit clause if it contains exactly one literal. A clause is a tautology if it contains both x and $\overline{x}$ for some variable x. The sets of variables and literals occurring in a formula F are denoted by $vars(F)$ and $lits(F)$, respectively. A literal l is pure within a formula F if and only if $\overline{l} \notin lits(F)$.

AIGs   An *and-inverter-graph* (AIG) [10] is a directed acyclic graph that represents a structural implementation of a circuit. Each node corresponds to a logical and-gate and each edge can either be positive or negative, representing whether the gate is negated. Since $\{\neg, \wedge\}$ is a functionally complete set of Boolean operators, obviously all Boolean formulas can be represented by AIGs.

(Partial) Assignments   An *assignment* for a formula F is a function α that maps all variables in F to a value $v \in \{1, 0\}$. We extend α to literals, clauses and formulas by using the common semantics of propositional logic. Therefore $\alpha(F)$ corresponds to the truth value of F under the assignment α. Similarly, a *partial*

assignment is a function β that maps only some variables of F to $v \in \{1, 0\}$. The value of the remaining variables in F is undefined and we write $\beta(x) = *$. Again, we extend β to literals, clauses and formulas by using the common semantics and simplification rules of propositional logic. Therefore $\beta(F)$ denotes the resulting formula under the partial assignment β. If we only want to assign one specific variable, we also use $F_{x=v}$ to represent the simplified formula.

Resolution The resolution rule states that, given two clauses $C_1 = (l \vee a_1 \vee \cdots \vee a_{k_1})$ and $C_2 = (\bar{l} \vee b_1 \vee \cdots \vee b_{k_2})$, the clause $C = (a_1 \vee \cdots \vee a_{k_1} \vee b_1 \vee \cdots \vee b_{k_2})$, called the *resolvent* of C_1 and C_2, can be inferred by resolving on the literal l. This is denoted by $C = C_1 \otimes_l C_2$. A special case of resolution where C_1 or C_2 is a unit clause is called *unit resolution*.

Unit Propagation If a unit clause $C = (x)$ or $C = (\bar{x})$ is part of a formula F, then F is equivalent to $F_{x=1}$ or $F_{x=0}$, respectively, and can be replaced by it. This process is called *unit propagation*. By $\mathrm{UP}(F)$ we denote the fixpoint obtained by iteratively performing unit propagation until no more unit clauses are part of the formula.

Blocked Clauses Given a CNF formula F, a clause C, and a literal $l \in C$, l blocks C w.r.t. F if (i) for each clause $C' \in F$ with $\bar{l} \in C'$, $C \otimes_l C'$ is a tautology, or (ii) $\bar{l} \in C$, i.e., C is itself a tautology. A clause C is blocked w.r.t. a given formula F if there is a literal that blocks C w.r.t. F. Removal of such blocked clauses preserves satisfiability [2]. For a CNF formula F, *blocked clause elimination* (BCE) repeats the following until fixpoint: If there is a blocked clause $C \in F$ w.r.t. F, let $F := F \setminus \{C\}$. BCE is confluent and in general does not preserve logical equivalence [3,4]. The CNF formula resulting from applying BCE on F is denoted by $\mathrm{BCE}(F)$. We say that BCE can solve a formula F if and only if $\mathrm{BCE}(F) = \emptyset$. Such an F is called a *blocked set*. We define $\mathcal{BS} := \{F \mid \mathrm{BCE}(F) = \emptyset\}$. A *pure literal* in F is a literal which occurs only positively. It blocks the clauses in which it occurs. As special case of blocked clause elimination, eliminating pure literals removes clauses with pure literals until fixpoint.

3 Properties

In this section, we revisit two monotonicity properties of blocked sets from previous work [5,4] and then prove several new properties. One basic property of blocked sets is the following monotonicity property: If $G \subset F$ and C is blocked w.r.t. F, then C is blocked w.r.t. G [5,4]. A direct implication is given by the following second version: If $F \in \mathcal{BS}$ and $G \subseteq F$ then $G \in \mathcal{BS}$.

Resolution does not affect the set of solutions of a Boolean formula. In CDCL solvers resolution is often used to learn additional information to help in the solving process. Therefore, it is interesting to see, that adding resolvents to a blocked set can destroy this property, which might thus make the formula much harder to solve:

Proposition 1. *Blocked sets are not closed under resolution, not even unit resolution.*

Proof. We give a counter-example:

$$F = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_3) \wedge (\overline{x}_2 \vee x_3) \wedge (\overline{x}_3)$$

We can check that $\mathrm{BCE}(F) = \emptyset$ by removing the clauses in the order in which they appear in F using the first literal of each clause as blocking literal. However, $F \wedge (x_1 \vee x_2)$ (with $(x_1 \vee x_2)$ obtained by unit resolution from the first and the last clause) is not a blocked set anymore. $\qquad\square$

As being one of the most important techniques in SAT solvers, it is of interest whether unit propagation can be combined with blocked clause elimination without destroying blockedness. In contrast to (unit) resolution, it is possible to apply unit propagation to blocked sets without the risk of obtaining a set of clauses that is not blocked anymore:

Proposition 2. *Blocked sets are closed under unit propagation.*

Proof. Since $F \in \mathcal{BS}$, $\mathrm{BCE}(F) = \emptyset$. Let $C_1, \ldots, C_m$ be the clauses of F in the order in which they are removed by blocked clause elimination. Consider the formula F' obtained from F by unit propagation of a single unit clause $C_i = \{x\}$ and the sequence of formulas $F_i = \{C_i, \ldots, C_m\}$. We remove all clauses from F' by BCE in the same order as the corresponding clauses in F were removed by BCE. For each C_j, we have to consider three different cases. Case 1: $\overline{x}$ was the blocking literal of C_j. This is however actually not possible for $j < i$. Note that C_j is not blocked in F_j since C_i is still in F_i and the resolvent of C_j with C_i on $\overline{x}$ is not a tautology. Similarly observe for $j > i$ that the clause C_i is not blocked in F_i. Finally for $j = i$, the literal $\overline{x}$ is not part of C_i. Case 2: x was the blocking literal of C_j. In this case the clause is not part of F' anymore and therefore does not need to be removed. Case 3: $l \notin \{x, \overline{x}\}$ was the blocking literal of C_j. In order to remove C_j, we have to show that all resolvents of C_j with other clauses $C_k \in F_i$ on l with $j \leq k \leq m$ are still tautologies. Since unit propagation did not remove any literals other than $\overline{x}$ from clauses in the formula, only resolvents being tautologies due to containing both x and $\bar{x}$ can be affected. However, in any case, either $x \in C_j$ or $x \in C_k$, and one of the two clauses has been removed from F due to unit propagation and thus this resolution does not have to be considered anymore. $\qquad\square$

Given a set of clauses $F \in \mathcal{BS}$, we know that F has at least one satisfying assignment. We could try to find this satisfying assignment by choosing a random unassigned variable x, setting the variable to both possible values and checking which of the reduced formulas $F_{x=0}$ and $F_{x=1}$ is still satisfiable. One could hope that the satisfiability of $F_{x=v}$ could again be proved by showing that $\mathrm{BCE}(F_{x=v}) = \emptyset$. However, this does not hold:

Proposition 3. *Blocked sets are not closed under partially assigning variables, furthermore, a blocked set may become non-blocked even in the subspace where this formula remains satisfiable.*

Proof. Consider the following example:

$$F = (\overline{x}_3 \vee \overline{x}_1 \vee x_4) \wedge (x_3 \vee x_2 \vee \overline{x}_4) \wedge (\overline{x}_2 \vee \overline{x}_1) \wedge (x_1 \vee x_4) \wedge (x_1 \vee x_5) \wedge (\overline{x}_5 \vee \overline{x}_4)$$

It is easy to verify that $\mathrm{BCE}(F) = \emptyset$ by removing the clauses in the order in which they appear in F with each clause being blocked on its first literal. If we now assign a value to x_3, we get the following two formulas:

$$F_{x3=0} = (x_2 \vee \overline{x}_4) \wedge (\overline{x}_2 \vee \overline{x}_1) \wedge (x_1 \vee x_4) \wedge (x_1 \vee x_5) \wedge (\overline{x}_5 \vee \overline{x}_4)$$
$$F_{x3=1} = (\overline{x}_1 \vee x_4) \wedge (\overline{x}_2 \vee \overline{x}_1) \wedge (x_1 \vee x_4) \wedge (x_1 \vee x_5) \wedge (\overline{x}_5 \vee \overline{x}_4)$$

Neither of the resulting formulas is blocked and both are satisfiable. □

The difference between unit propagation and applying partial assignment is that in the former we assume the unit clause to be part of the blocked set. In this case the result after unit propagation or just applying the corresponding assignment can not destroy blockedness, while adding an arbitrary unit clause to a blocked set does not have this property.

4   Solving Blocked Sets

Blocked sets are always satisfiable and it is easy to find a satisfying assignment for them [4]. However, as shown in [5] it is also possible to efficiently find multiple satisfying assignments in a bit-parallel fashion by generalizing the original solution extraction algorithm [4].

In the original algorithm, we start from a random full truth assignment α for a blocked set B. We then traverse the clauses $C \in B$ in the reverse order of their elimination, e.g. the clause eliminated first will be considered last. For each clause C, we check if $\alpha(C) = 1$ and, if it is not, we flip the truth value of the blocking literal variable in α.

Our new generalized version of the reconstruction algorithm presented next, uses ternary logic consisting of values $v \in \{1, 0, *\}$. Instead of starting from a random (full) assignment, we start from a partial assignment β with $\beta(x) := *$ for all $x \in vars(B)$. We then traverse the clauses $C \in B$ in the same order as the original algorithm. If the clause is satisfied under the current assignment, we continue with the next clause. Otherwise, we do the following:

1. if there is a literal $l \in C$ with variable x and $\beta(x) = *$,
   then we set the variable to 1 or 0 which satisfies the clause.
2. otherwise, we flip the assignment of the blocking literal variable.

This algorithm is presented in Fig. 1. It will terminate with a partial satisfying assignment (some variables might have the value $*$). The values of the $*$ variables can be chosen arbitrarily. In this way, the algorithm finds several solutions at the same time. If the number of $*$ variables is k then the algorithm found 2^k solutions. Note, that any unassigned variable in a partial satisfying assignment can not be in the backbone nor part of an implied equivalence.

```
        Solve (Blocked set B)
S1      β := [*, *, . . . , *]
S2      for Clause C ∈ reverse(eliminationStack) do
S3          if C is satisfied under β then continue
S4          V := getUnassignedVars(C)
S5          if V = ∅ then flip the blocking literal of C in β
S6          else
S7              select x ∈ V
S8              set x in β to a value that satisfies C
S9      return β
```

Fig. 1. Pseudo-code of the generalized blocked set solution reconstruction algorithm. The clauses are traversed in the reverse order of their elimination.

Making different choices on line S7 will give us different solutions. In fact, the non-determinism allows the algorithm to find all satisfying assignments of a blocked set. We show this in the following proposition.

Proposition 4. *The generalized reconstruction algorithm can find all the solutions of a blocked set.*

Proof. Let β be an arbitrary satisfying assignment of a blocked set B. When the reconstruction algorithm encounters an unsatisfied clause C, it can choose to satisfy C using the same variable value pair as in β. The case that the blocking literal needs to be flipped will never occur, since unsatisfied clauses with no $*$ variable will not be encountered. This is due to the fact that we set all variables to their proper values when first changing the value from $*$. The algorithm will terminate with the solution β (possibly some $*$ values remain). □

The next proposition demonstrates the correctness and time complexity of the generalized reconstruction algorithm.

Proposition 5. *The generalized reconstruction algorithm always terminates in linear time with a satisfying assignment of the blocked set.*

Proof. The algorithm visits each clause exactly once and performs a constant number of operations for each of its literals. Therefore the algorithm runs in linear time in the size of the blocked set.

We will prove that at the end of each for-cycle iteration all the clauses that have been traversed so far are satisfied by the assignment β. From this, the correctness of the returned solution follows. We need to examine three cases. If the clause is already satisfied by β or it is satisfied by setting a value of a $*$ variable, then none of the traversed clauses can become unsatisfied. Otherwise, the clause becomes satisfied by flipping the blocked literal. In this case, we know from the definition of the blocked set that all previous clauses which contain the negation of the blocking literal are already satisfied by another literal. Namely, by the literal whose negation appears in the current clause and is false here. □

We have seen that one can easily find solutions of blocked sets using a simple algorithm which in a way resembles a local search SAT solver (i.e. by "flipping" assignments to certain variables). This gives rise to the question of how state-of-the-art stochastic local search (SLS) SAT solvers perform on blocked sets. We conducted experiments using several well known SLS solvers and added our own specialized local search solver which always flips the blocking literal. From the results in Fig. 2, we can conclude that standard local search solvers struggle with solving blocked sets. However, when only the blocking literal is flipped, the problems are easily solved. This suggests that the crucial point in solving blocked sets is to know which is the blocked literal of a clause and the order in which the clauses are addressed is less important.
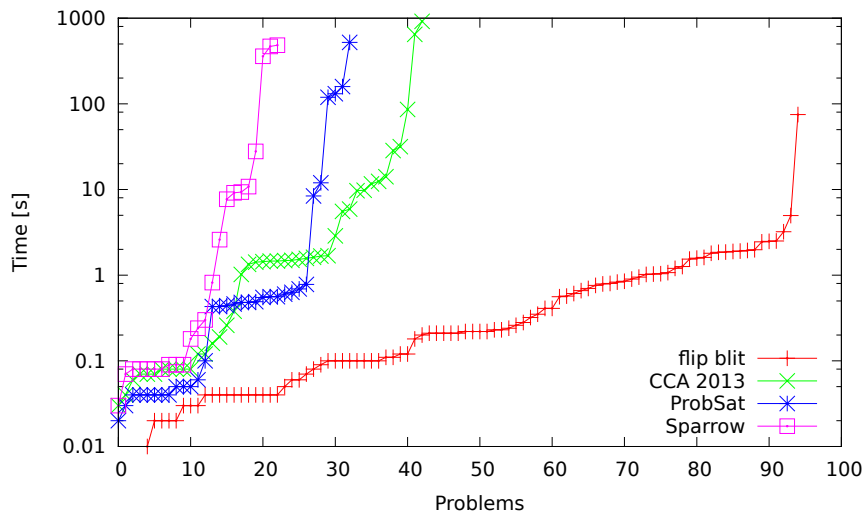

Fig. 2. Performance of the stochastic local search SAT solvers ProbSat [11], Sparrow [12], CCA2013 [13], and a modified ProbSat which always flips the blocking literal. The used instances are the 95 benchmarks described in Section 5.1 (instances from the 2013 SAT competition where unit decomposition is successful). The experiments were run on a PC with Intel Core i7-2600 3.4GHz CPU and 16GB of memory. The time limit for each instance was 1000 seconds.

5   Blocked Clause Decomposition

Blocked clause decomposition (BCD) [5] is the process of splitting a clause set of a CNF formula F into two sets such that at least one of them is a blocked set. If both sets are blocked, the decomposition is called symmetric.

Usually, we want to decompose the formula in a way that one of the blocked sets is large (i.e. contains many clauses) while the other is small. The motivation

is that at least one of the blocked sets (the large one) should resemble the original formula as much as possible. Therefore, we will use the size difference of the two sets as a measure of the *quality of the decomposition*. We denote the larger blocked set by L (large set) and the smaller one by R (rest of the clauses). Symmetric BCD can be defined by the following formula:

$$\text{BCD}(F) = (L, R); \ F = L \cup R; \ L, R \in \mathcal{BS}; \ |L| \geq |R|$$

The simplest way of doing BCD is *pure decomposition* (described in [5]). First, we start with two empty clause sets L and R. Then, for each variable $x \in vars(F)$, we do the following. Let F_x be the set of clauses of F where x occurs positively and $F_{\overline{x}}$ where it occurs negatively. We add the larger of F_x and $F_{\overline{x}}$ to L and the smaller to R. We remove F_x and $F_{\overline{x}}$ from F and continue with the next variable. This algorithm produces two blocked sets which can be solved by pure literal elimination, hence the name pure decomposition.

Pure decomposition can be easily implemented to run in linear time and, therefore, is very fast on all formulas. A drawback is given by the fact that the difference in the sizes of L and R is usually not very big. This disadvantage can be addressed by post-processing, i.e., moving clauses from R to L as long as L (and R) remain blocked. An obvious way of post-processing (already described in [5]) is to move blocked (with respect to the current L) clauses from R to L. We extend this method by also moving so called *blockable* clauses.

Definition 1. *A clause C is* blockable *w.r.t. a blocked set L if each literal $l \in C$ potentially blocks C where a literal $l \in C$ potentially blocks C if each clause $C' \in L$ containing $\overline{l}$ has a different blocking literal (its blocking literal is not $\overline{l}$).*

It is easy to observe that a blockable clause will not prevent any other clause in the blocked set from being eliminated by blocked clause elimination and, therefore, adding it to the blocked set will not destroy its blocked status. A blockable clause can be easily detected if we maintain a literal occurrence list for the clauses in the blocked set. Using this data structure, and remembering for each clause which is its blocking literal, allows an efficient checking of the blockable property.

Moving blocked and blockable clauses can be considered to be lightweight post-processing methods since they are fast but often cannot move too many clauses. Another kind of post-processing algorithm is based on the following idea. If $\text{BCE}(L \cup S) = \emptyset$ for some $S \subset R$, all clauses in S can be moved from R to L. We refer to S as a candidate set. This kind of post-processing is more time consuming but also much more powerful. Different strategies can be employed for the selection of S such as *QuickDecompose* [5], which continues to move clauses from R to L until no more clauses can be moved. L is then called a maximal blocking set. Since *QuickDecompose* requires a lot of time for many instances, we decided to use a heuristic approach called *EagerMover*, which is described in Fig. 3. The algorithm tries to move $1/4$ of all clauses in R until none of the 4 parts can be moved. Trying to move smaller parts than $1/4$ in each step makes the algorithm slower but possibly more clauses can be moved. Experiments revealed that $1/4$ is a good compromise.

```
         EagerMover (Blocked set L, R)
EM1          moved := True
EM2          while moved do
EM3            moved := False
EM4            for i := 0 to 3 do
EM5              S := R : {i · 0.25 · |R|, (i + 1) · 0.25 · |R|}
EM6              if BCE(L ∪ S) = ∅ then
EM7                L := L ∪ S; R := R \ S; moved := True
```

Fig. 3. Pseudo-code of the *EagerMover* post-processing algorithm, $R : \{a, b\}$ is the subsequence of R starting with the a-th element until the b-th element.

5.1 Unit Decomposition

To achieve good quality decomposition, the following heuristic (called *unit decomposition* in [5]) was introduced: remove unit clauses from the original formula and test if the remaining clauses are a blocked set. If they are a blocked set, put the unit clauses into R and we are done. If the formula is an encoding of a circuit SAT problem, then this approach will always succeed [5]. This heuristic works on 77 of the 300 instances of the application track of the 2013 SAT Competition.

This heuristic can be generalized, by running unit propagation on the input clauses and removing satisfied clauses. Clauses are not simplified by removing false literals since those might be used as blocking literals. Next, we test if the clause set is blocked and, if it is, we put the unit clauses into R and we are done. Our improved heuristic succeeds on 95 instances, 18 more than the original. In the case that unit decomposition is not successful, we use pure decomposition.

After unit decomposition succeeds, there still might be some unit clauses which can be moved from R to L. Therefore, it is useful to run some of the mentioned post-processing algorithms to improve the quality of the decomposition.

5.2 Solitaire Decomposition

In this section, we define a special type of blocked clause decomposition called *solitaire blocked clause decomposition* (SBCD). In solitaire decomposition, we require that the small set R contains only a single unit clause. We will use this concept to translate a SAT problem into a circuit SAT problem (see Sect. 6). Solitaire decomposition cannot be achieved for every formula unless we allow an additional variable. A formal definition of solitaire decomposition follows.

Definition 2. *Let F be CNF formula. $\mathrm{SBCD}(F) = (L, \{l\})$ where $L \in \mathcal{BS}$ and l is a literal, F and $L \cup \{l\}$ have the same set of satisfying assignments on the variables of F.*

A trivial solitaire decomposition of an arbitrary CNF is obtained by adding a new fresh variable x to each clause of the formula and then using $l := \overline{x}$ as the only literal in the small set:

$$\mathrm{SBCD}(C_1 \wedge C_2 \wedge \cdots \wedge C_m) = (\{x \vee C_1\} \wedge \{x \vee C_2\} \wedge \cdots \wedge \{x \vee C_m\}, \{\overline{x}\})$$

It is easy to see that $\{x \vee C_1\} \wedge \{x \vee C_2\} \wedge \cdots \wedge \{x \vee C_m\}$ is a blocked set with x being the blocking literal for each clause. As improvement, perform blocked clause decomposition and then add a new fresh variable only to the clauses in the small set R. These clauses can then be moved to L and the new R will contain only the negation of the new variable.

6 Extracting Circuits

Using the reconstruction algorithm discussed in Sect. 4, it is possible to construct a circuit representation of a blocked set. Let F be a blocked set and let $C_1, \ldots, C_m$ be the clauses of F in the order in which they are removed by blocked clause elimination. During the reconstruction algorithm, we iterate through all the clauses of F starting from C_m to C_1 and flip the blocking literal of a clause if and only if that clause is not satisfied, i.e., all its literals are set to 0.

For each original variable $x_1, \ldots, x_n$ of F, we will have several new variables called *versions*. By $x_{i,k}$, we denote the k-th version of x_i ($x_{i,0}$ is x_i) and by $x_{i,\$}$, the latest version of x_i ($x_{i,\$} = x_{i,k}$ if k is the largest integer such that $x_{i,k}$ is defined). The notation is extended to literals in the same way.

Starting with C_m, we traverse the clauses in the reverse order of their elimination. For each clause $C = (x_i \vee y_{j_1} \vee \cdots \vee y_{j_k})$, where x_i is the blocking literal, we create a new version of x_i. We do this to represent the fact that the literal might have been flipped and can have a different value in the following iterations. The value of the new version is given by the following definition.

$$x_{i,\$+1} := x_{i,\$} \vee (\overline{y}_{j_1,\$} \wedge \cdots \wedge \overline{y}_{j_k,\$})$$

The following example demonstrates the definitions obtained from a blocked set.

Example 1. Let $F = (\overline{x}_1 \vee x_3 \vee x_2) \wedge (x_3 \vee x_4 \vee \overline{x}_1) \wedge (x_1 \vee \overline{x}_2 \vee \overline{x}_3)$. Using BCE, the clauses can be eliminated in the order of their appearance with the first literal being the blocking literal. We proceed in the reverse order and obtain the following definitions:

$$x_{1,1} := x_{1,0} \vee (x_{2,0} \wedge x_{3,0})$$
$$x_{3,1} := x_{3,0} \vee (\overline{x}_{4,0} \wedge x_{1,1})$$
$$\overline{x}_{1,2} := \overline{x}_{1,1} \vee (x_{3,1} \wedge x_{2,0})$$

These equations can be directly implemented as a circuit or, w.l.o.g., as an AIG. The first versions of the variables ($x_{i,0}$) are the inputs of the circuit and the higher versions are defined by the definitions. Using this construction, together with the solitaire decomposition as defined in Sect. 5, we can translate an arbitrary CNF (i.e. $F \notin \mathcal{BS}$) into an instance of the circuit SAT problem in the form of an AIG as follows: The large set L of the decomposition (with $L \in \mathcal{BS}$) is first encoded into an AIG G as already described. Then the output is defined to be the conjunction of G with the latest version of the unit literal corresponding to

the small set. By doing this, one can potentially apply simplification techniques known from circuit synthesis and model checking (e.g. use ABC [9] to rewrite the circuit) and potentially profit from the similarity of blocked sets to circuits.

7 CNF Reencodings

In this section, we describe several ways of reencoding SAT problems using the result of blocked clause decomposition. Our input as well as our output is a CNF formula. By reencoding, we hope to increase the speed of SAT solving. The idea is similar to the one of circuit extraction described in Sect. 6. In comparison to the AIG encoding, our CNF encoding is more complex and versatile.

First, we describe how to encode the progression of the solution reconstruction for one blocked set $C_1 \wedge \cdots \wedge C_m$. For each variable x_i, we will have several versions as already described in Sect. 6, with $x_{i,\$}$ being the latest version of x_i. As we did previously, we traverse the clauses in the order $C_m, \ldots, C_1$ and introduce a new version for the blocking literal for each clause using the definition

$$x_{i,\$+1} := x_{i,\$} \vee (\overline{y}_{j_1,\$} \wedge \cdots \wedge \overline{y}_{j_k,\$})$$

which can be expressed by the following $k + 2$ clauses:

$$(\overline{x}_{i,\$} \vee x_{i,\$+1}) \wedge (x_{i,\$+1} \vee y_{j_1,\$} \vee \cdots \vee y_{j_k,\$}) \wedge \bigwedge_{l=1}^{k}(\overline{y}_{j_l,\$} \vee \overline{x}_{i,\$+1} \vee x_{i,\$}) \qquad (1)$$

This formula represents the main step of the reconstruction algorithm, which states that the blocking literal is flipped if and only if the clause is not satisfied.

While symbolically encoding the progression of the reconstruction algorithm, we might decide not to have new versions for some of the variables. During reconstruction this corresponds to disallow that their values are flipped. Thus these variables need to have the right truth value from the beginning. Specifying that x_i should not have versions, means that $x_{i,\$+1} = x_{i,\$} = x_i$ which turns $k+1$ of the $k+2$ clauses of (1) into a tautology. The remaining clause is just a copy of the original clause using the latest versions of the variables. If we decided that none of the variables should have new versions, then the result of the reencoding would simply be the original formula.

In the rest of this section we propose several options to reencode a pair of blocked sets. The simplest way is to reencode the large blocked set L and append the clauses of R with variables renamed according to the last versions of variables from the reencoding of L. This can be done even for asymmetric decompositions since we have no requirement on R. Another way is to reencode both blocked sets and then make the last versions of the corresponding variables equal. This can be done by renaming the last versions from one blocked set to match the last versions in the other. As mentioned earlier, we can decide not to have versions for some of the variables. There are several heuristics which can be used to select these variables. In our experiments we found the following two heuristics useful.

1. Have versions only for variables that occur in both sets.
2. Have versions only for variables that occur as a blocking literal in both sets.

The entire reencoding process can be done in several ways. First, we need to choose a decomposition and a post-processing algorithm to obtain the blocked sets. Next, we need to decide which variables should have versions and whether to reencode both blocked sets. The choice of the decomposition and post-processing algorithms strongly influences the runtime but also the quality of the reencoding. As we will see from the experiments, better decompositions usually result in reencoded formulas that are easier to solve. The number of variables which have versions has a strong impact on the size of the reencoded formula which also influences the runtime of SAT solvers. Unfortunately, there is no clear choice since different combinations work best for different benchmark formulas.

8   Experiments

The algorithms described above were implemented in a Java tool which takes CNF formulas as input and produces an AIG or a reencoded CNF formula as output. The tool, its source code, and log files of experiments described in this section are available at `http://ktiml.mff.cuni.cz/~balyo/bcd/`.

We evaluated the proposed methods on the 300 instances from the application track of the SAT Competition 2013. All experiments were run on a 32-node cluster with Intel Core 2 Quad (Q9550 @2.83GHz) processors and 8GB of memory. We used a time limit of 5000 seconds and a memory limit of 7000MB, a similar set-up as in the SAT Competition 2013.

To solve the reencoded instances, we used the SAT solver Lingeling [7], the winner of the application track of the SAT Competition 2013. We used the synthesis and verification system ABC [9] to simplify AIG circuits and the AIGER library to convert between different AIG formats and converting AIG to CNF.

Figure 4 shows the performance of various decomposition methods. The best decomposition can be obtained by using unit decomposition followed by the lightweight and eager post-processing methods. Further, we observe that the eager post-processing method can significantly increase the quality after pure decomposition and often helps the unit decomposition. On the other hand, the eager post-processing can increase the time of decomposition up to 1000 seconds, which takes away a lot of time from the SAT solver.

In Fig. 5 the time required to solve the instances before and after reencoding are shown, as described in Sect. 7. For the experiments, we used different decomposition methods (see Fig. 5) followed by the simple reencoding method (only reencode the large set and rename the small set). Versions were only introduced for variables that occur as a blocking literal in both sets. For most of the problems, Lingeling with the original formula dominates the other methods and it is only for the hard problems (solved after 3500 seconds) that the reencoding starts to pay off. Overall, Lingeling without reencoding solved 232 instances, with Unit+Blockable decomposition it solved 237, and with Pure+Eager 240.
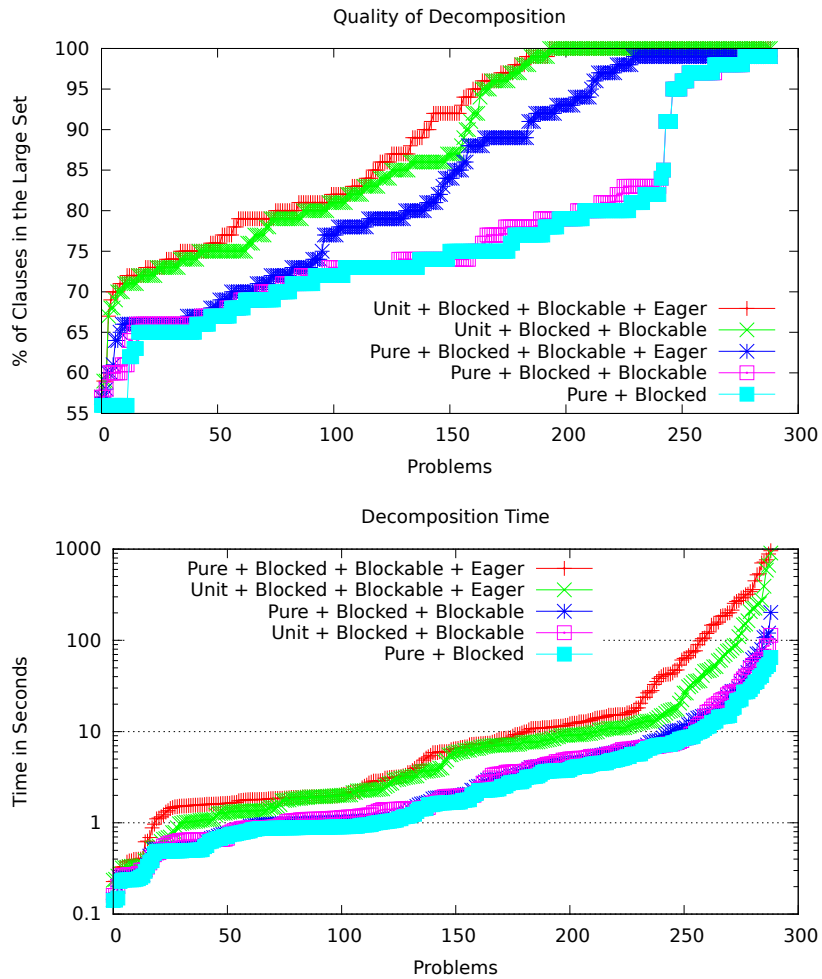
Fig. 4. The quality (percentage of the number of clauses in the large blocked set) and runtime of several decomposition and post-processing algorithm combinations.

Figure 6 demonstrates the usefulness of the circuit extraction and simplification approach. Similarly to the reencoding approach, this is also only useful for harder instances (solved after 3000 seconds). For the hardest 20 problems, our approach clearly takes over and ends up solving 7 more instances. These include two instances which were not solved by any sequential solver in the SAT competition 2013: *kummling_grosmann–pesp/ctl_4291_567_8_unsat_pre.cnf* and *kummling_grosmann–pesp/ctl_4291_567_8_unsat.cnf*. For this experiment, we only used those 135 instances where we can obtain a decomposition with a certain quality,
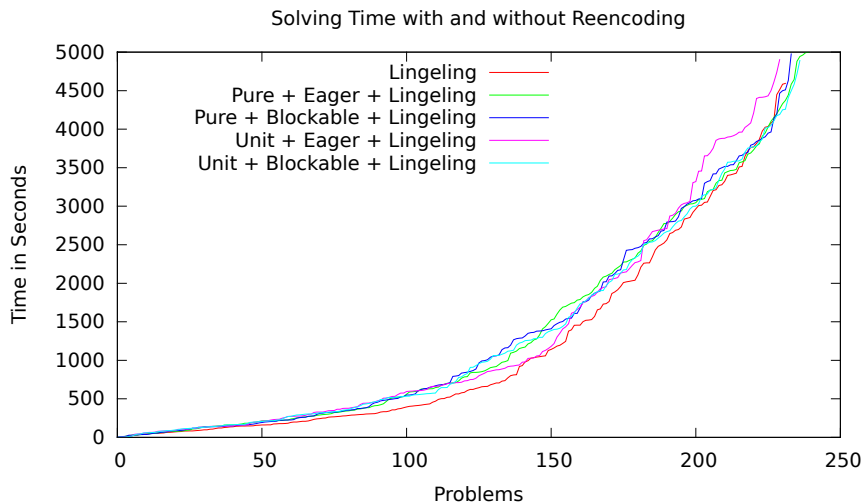
Fig. 5. The time required to solve the benchmark formulas which can be solved in 5000 seconds. The time limit for the reencoded instances was decreased by the time required for the reencoding process.

namely 90% of the clauses must be in the large set. To obtain the blocked sets we used unit decomposition with eager post-processing.

9 Conclusion

Simplification techniques based on blocked clauses can have a big influence on SAT solver performance. In this paper, we looked at blocked clauses in detail once again. We showed that blocked sets are closed under unit propagation while they are not closed under resolution or partial assignments. We proposed new ways for finding solutions of blocked sets and modified existing reconstruction algorithms to work with partial assignments. This enables us to find multiple solutions in one reconstruction run. This allows to rule out backbones and implied binary equivalences faster. Further, we analyzed the performance of local search on blocked sets.

While existing local search solvers performed rather poorly a simple extension improves the efficiency of local search on blocked sets significantly. We revisited blocked clause decomposition and described several new decomposition techniques as well as improved versions of existing ones. In particular, we showed how unit decomposition heuristics can be extended to be successful on more problems. We defined solitaire decomposition and described how it can be used to translate SAT to circuit SAT. We proposed various reencoding techniques to obtain different CNF representations. In the experimental section, we evaluated the performance of the state-of-the-art SAT solver Lingeling on the reencoded
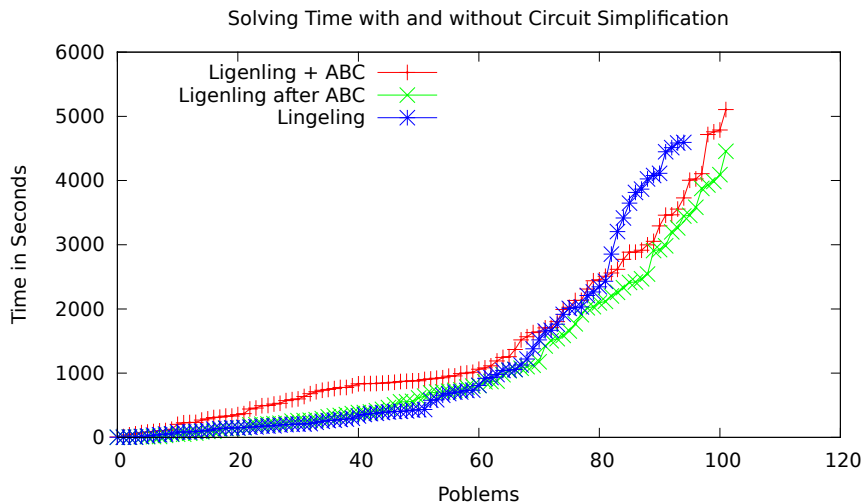
Fig. 6. The time required to solve the benchmark formulas, which can be decomposed with a quality of at least 90% (135 of 300 instances). The plot shows Lingeling on the original formula and the formula obtained by AIG circuit extraction followed by circuit simplification using the dc2 method of ABC (with a 500 seconds time limit) and finally reencoding the simplified circuit back to CNF. The data labeled "Lingeling + ABC" represents the total time required by the solving, reencodings and simplification. Lingeling alone can solve 95 instances and 102 after reencoding and simplification.

benchmarks. Our results showed that Lingeling can benefit from the reencodings being able to solve more formulas in a given time limit. The performance of solving is increased mainly for harder instances.

As future work, we want to optimize our circuit extraction techniques. If parts of a CNF were obtained from a circuit SAT problem, the original representation might help to solve the problem. It is unclear if or how this original structure can best be extracted from a CNF. However, there is a close connection between blocked clause elimination and operations on the circuit structure of a formula [3]. Therefore a better understanding of blocked sets might be an important step into this direction.

From a theoretical point of view, it is still not clear whether or how all solutions of a blocked set of clauses can be enumerated in polynomial time. Although this was conjectured to be possible in [5], it still has not been proven. While this conjecture is interesting from the theoretical point of view, it also would have important implications in practice since it guarantees the efficiency of algorithms on formulas containing blocked subsets with few solutions.

Finally, blocked clause decomposition and solving blocked sets needs a non-negligible portion of time in the solving process.Therefore, further improvements for decomposition techniques or the solving process of blocked sets will also directly affect the number of formulas that can be solved.

References

1. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Proceedings of IJCAR 2012. Volume 7364 of LNCS., Springer (2012) 355–370
2. Kullmann, O.: On a generalization of extended resolution. Discrete Applied Mathematics **96–97** (1999) 149–176
3. Järvisalo, M., Biere, A., Heule, M.J.H.: Blocked clause elimination. In Esparza, J., Majumdar, R., eds.: TACAS '10. Volume 6015 of LNCS., Springer (2010) 129–144
4. Järvisalo, M., Biere, A., Heule, M.J.H.: Simulating circuit-level simplifications on cnf. Journal of Automated Reasoning **49**(4) (2012) 583–619
5. Heule, M.J.H., Biere, A.: Blocked clause decomposition. In: Proceedings of the 19th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'13). (2013)
6. Parkes, A.J.: Clustering at the phase transition. In: In Proc. of the 14th Nat. Conf. on AI, AAAI Press / The MIT Press (1997) 340–345
7. Biere, A.: Lingeling, plingeling and treengeling entering the sat competition 2013. In: In Proceedings of SAT Competition 2013, A. Balint, A. Belov, M. J. H. Heule, M. Järvisalo (editors), vol. B-2013-1 of Department of Computer Science Series of Publications B pages 51-52, University of Helsinki, 2013. (2013)
8. Mishchenko, A., Chatterjee, S., Brayton, R.K.: DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In Sentovich, E., ed.: Proceedings of the 43rd Design Automation Conference (DAC 2006), ACM (2006) 532–535
9. Brayton, R.K., Mishchenko, A.: Abc: An academic industrial-strength verification tool. In: Proc. CAV'10. (2010) 24–40
10. Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust boolean reasoning for equivalence checking and functional property verification. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **21**(12) (2002)
11. Balint, A., Schöning, U.: Choosing probability distributions for stochastic local search and the role of make versus break. In Cimatti, A., Sebastiani, R., eds.: SAT. Volume 7317 of LNCS., Springer (2012) 16–29
12. Balint, A., Fröhlich, A.: Improving stochastic local search for sat with a new probability distribution. In Strichman, O., Szeider, S., eds.: SAT. Volume 6175 of LNCS., Springer (2010) 10–15
13. Li, C., Fan, Y.: Cca2013. In: Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions. (2013)