Conflict Directed Lazy Decomposition

Ignasi Abío[1] and Peter J. Stuckey[2]

[1] Technical University of Catalonia (UPC), Barcelona, Spain
[2] Department of Computing and Information Systems, and
NICTA Victoria Laboratory, The University of Melbourne, Australia

Abstract. Two competing approaches to handling complex constraints in satisfaction and optimization problems using SAT and LCG/SMT technology are: *decompose* the complex constraint into a set of clauses; or *(theory) propagate* the complex constraint using a standalone algorithm and explain the propagation. Each approach has its benefits. The decomposition approach is prone to an explosion in size to represent the problem, while the propagation approach may require exponentially more search since it does not have access to intermediate literals for explanation. In this paper we show how we can obtain the best of both worlds by *lazily decomposing* a complex constraint propagator using conflicts to direct it. If intermediate literals are not helpful for conflicts then it will act like the propagation approach, but if they are helpful it will act like the decomposition approach. Experimental results show that it is never much worse than the better of the decomposition and propagation approaches, and sometimes better than both.

1 Introduction

Compared with other systematic constraint solving techniques, SAT solvers have many advantages for non-expert users. They are extremely efficient off-the-shelf black boxes that require no tuning regarding variable (or value) selection heuristics. However, propositional logic cannot directly deal with complex constraints: we need either *to enrich the language* in which the problems are defined, or *to reduce the complex constraints* to propositional logic.

Lazy clause generation (LCG) or *SAT Modulo Theories* (SMT) approaches correspond to an enrichment of the language: the problem can be expressed in first-order logic instead of propositional logic. A specific theory solver for that (kind of) constraint, called a *propagator*, takes care of the non-propositional part of the problem, propagating and explaining the propagations, whereas the SAT Solver deals with the propositional part. On the other hand, reducing the constraints to propositional logic corresponds to *encoding* or *decomposing* the constraints into SAT: the complex constraints are replaced by an equivalent set of auxiliary variables and clauses.

The advantages of the propagator approach is that the size of the propagator and its data structures are typically quite small (in the size of the constraint) compared to the size of a decomposition, and we can make use of specific global algorithms for efficient propagation. The advantages of the decomposition approach are that the resulting propagation uses efficient SAT data structures and are inherently incremental, and more importantly, the auxiliary variables give the solver more scope for learning appropriate reusable nogoods.

In this paper we examine how to get the best of each approach, and illustrate our method on two fundamental constraints: cardinality and pseudo-Boolean constraints.

An important class of constraints are the so-called *cardinality constraints*, that is, constraints of the form $x_1 + \cdots + x_n \# K$, where the K is an integer, the x_i are Boolean (0/1) variables, and the relation operator $\#$ belongs to $\{\leqslant, \geqslant, =\}$. Cardinality constraints are omnipresent in practical SAT applications such as timetabling [1] and scheduling constraint solving [2]. Some optimization problems, such as MaxSAT or close-solutions problems (see [3]), can be reduced to a set of problems with a single cardinality constraint (see Section 4.1).

The two different approaches for solving complex constraints have both been studied for cardinality constraints. In the literature one can find different decompositions using adders [4], binary trees [5] or sorting networks [6], among others. The best decomposition, to our knowledge, is the cardinality network-based encoding [7]. On the other hand, we can use a propagator for deal with these constraints, and using either an SMT Solver [8] or LCG Solver [9].

Another important class of constraints are the *pseudo-Boolean (PB) constraints*, that is, constraints of the form $a_1x_1 + \cdots + a_nx_n \# K$, where K and a_i are integers, the x_i are Boolean (0/1) variables, and the relation operator $\#$ belongs to $\{\leqslant, \geqslant, =\}$. These constraints are very important and appear frequently in application areas such as cumulative scheduling [10], logic synthesis [11] or verification [12].

In the literature one can find different decompositions of PB constraints using adders [4,6], BDDs or similar tree-like structures [6,13,14] or sorting networks [6]. As before, LCG and SMT approaches are also possible.

To see why both approaches, both propagator and decomposition, have advantages consider the following two scenarios:

– Consider a problem with hundreds of large cardinality constraints where all but 1 never cause failure during search. Decomposing each of these constraints will cause a huge burden on the SAT solver, adding many new variables and clauses, all of which are actually useless. The propagation approach will propagate much faster, and indeed just the decomposition step could overload the SAT solver.
– Consider the problem with the cardinality constraint $x_1 + \cdots + x_n \leqslant K$ and some propositional clauses implying $x_1 + \cdots + x_n \geqslant K + 1$. The problem is obviously unsatisfiable, but if we use a propagator for the cardinality constraint, it will need to generate all the ${}^nC_k$ explanations possible in order to prove the unsatisfiability. However with a decomposition approach the problem can be solved in polynomial time due to the auxiliary variables.

In conclusion it seems likely that in every problem there are *some* auxiliary variables that will produce more general nogoods and will help the SAT solver, and *some* other variables that will only increase the search space size, making the problem more difficult. The intuitive idea of Lazy Decomposition is to try to generate only the *useful* auxiliary variables. The solver initially behaves as a basic LCG solver. If it observes that an auxiliary variable would appear in many nogoods, the solver generates it.

While there is plenty of research on combining SAT and propagation-based methods, for example all of SAT modulo theories and lazy clause generation, we are unaware of any previous work where a complex constraint is partially decomposed. There is some recent work [15] where the authors implement an incremental method for solving pseudo-Boolean constraints with SAT, by decomposing the pseudo-Booleans one by one. However, they do not use propagators for dealing with the non-decomposed constraints, and the decomposition is done in one step for a single constraint.

The remainder of the paper is organized as follows. In the next section we give SAT and LCG/SMT solving as well as decompositions and propagator definitions for both cardinality and psuedo-Boolean constraints. In Section 3 we define a framework for lazy decomposition propagators, and instantiate it for cardinality and psuedo-Boolean constraints. In Section 4 we show results of experiments, and in Section 5 we conclude.

2 Preliminaries

2.1 SAT Solving

Let $\mathcal{X} = \{x_1, x_2, \ldots\}$ be a fixed set of propositional *variables.* If $x \in \mathcal{X}$ then x and $\overline{x}$ are *positive and negative literals*, respectively. The *negation* of a literal l, written $\overline{l}$, denotes $\overline{x}$ if l is x, and x if l is $\overline{x}$. A *clause* is a disjunction of literals $\overline{x}_1 \vee \ldots \vee \overline{x}_p \vee x_{p+1} \vee \ldots \vee x_n$, sometimes written as $x_1 \wedge \ldots \wedge x_p \rightarrow x_{p+1} \vee \ldots \vee x_n$. A *CNF formula* is a conjunction of clauses.

A (partial) *assignment* A is a set of literals such that $\{x, \overline{x}\} \not\subseteq A$ for any x, i.e., no contradictory literals appear. A literal l is *true* in A if $l \in A$, is *false* in A if $\overline{l} \in A$, and is *undefined* in A otherwise. True, false or undefined is the *polarity* of the literal l. A clause C is true in A if at least one of its literals is true in A. A formula F is true in A if all its clauses are true in A. In that case, A is a *model* of F. Systems that decide whether a formula F has any model are called SAT-solvers, and the main inference rule they implement is *unit propagation*: given a CNF F and an assignment A, find a clause in F such that all its literals are false in A except one, say l, which is undefined, add l to A and repeat the process until reaching a fix-point.

Clauses are not the only constraints that can be defined over the propositional variables. Sometimes some clauses of the formula are expressed more compactly as a single complex constraint.

2.2 SMT/LCG Solver

An SMT solver or a LCG solver[1] is a system for finding models of a formula F and a set of complex constraints $\{c_i\}$. It is composed of two parts: a SAT

[1] In this paper we do not distinguish between SMT solvers and LCG solvers. The two techniques are very similar and both of fit the sketch presented here, although arguably the propagator centric view is more like LCG.

solver engine and a *propagator* for every constraint c_i. The SAT solver searches a model of the formula and the propagators infer consequences of the assignment and the set of constraints (this is, *propagate*), and, on demand of the SAT solver, provide the reason of some of the propagated literals (called the *explanation*).

2.3 Cardinality Constraints

A cardinality constraint takes the form $x_1 + \cdots + x_n \# K$, where the K is an integer, the x_i are literals, and the relation operator $\#$ belongs to $\{\leqslant, \geqslant, =\}$.

Propagation. For a $\leqslant$ constraint, the propagator keeps a count of the number of literals of the constraint which are *true* in the current assignment. The propagator increments this value every time the SAT solver assigns *true* a literal of the constraint. The count is decremented when the SAT solver unassigns one of these literals. When this value is equal to K, no other literal can be *true*: the propagator sets to *false* all the remaining literals. The explanation for setting a literal x_j to *false* can be built by searching for the K literals $\{x_{i_1}, \ldots, x_{i_K}\}$ of the constraint which are *true* to give the explanation $x_{i_1} \wedge x_{i_2} \wedge \cdots \wedge x_{i_K} \rightarrow \overline{x_j}$.

Similarly, in a $\geqslant$ constraint the propagator keeps a count of the literals which are *false* in the current assignment. When this value is equal to $n - K$, the propagator sets to *true* the non-propagated literals. A propagator for an equality constraint keeps track of both values.

Cardinality Network Decomposition. A *k-cardinality network* of size n is a logical circuit with n inputs and n outputs satisfying two properties:

1. The number of *true* outputs of the network equals the number of *true* inputs.
2. For every i with $1 \leqslant i \leqslant k$, the i-th output of the network is *true* if and only if there were at least i *true* inputs.

An example of cardinality networks are sorting networks e.g. [6], with size $O(n \log^2 n)$. An example is shown in Figure 1(b). The smallest decomposition for a k-cardinality network is $O(n \log^2 k)$ [7].

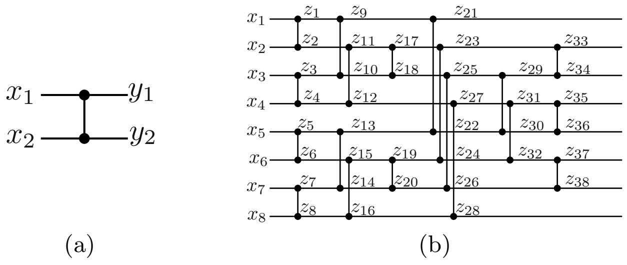
x_1
x_2
y_1
y_2
(a)
x_1
x_2
x_3
x_4
x_5
x_6
x_7
x_8
z_1 z_2 z_3 z_4 z_5 z_6 z_7 z_8
z_9 z_{10} z_{11} z_{12} z_{13} z_{14} z_{15} z_{16}
z_{17} z_{18} z_{19} z_{20}
z_{21} z_{22} z_{23} z_{24} z_{25} z_{26} z_{27} z_{28}
z_{29} z_{30} z_{31} z_{32}
z_{33} z_{34} z_{35} z_{36} z_{37} z_{38}
(b)

Fig. 1. (A) A 2-comparator $\mathsf{2comp}(x_1, x_2, y_1, y_2)$ is shown as a vertical bar joining two lines. (b) An odd-even merge sorting network for $n = 8$. Each line segment (broken at nodes) represents a Boolean variable.

Cardinality networks are composed of 2-comparators. A *2-comparator* is a circuit $\mathsf{2comp}(x_1, x_2, y_1, y_2)$ with inputs x_1, x_2 and outputs $y_1 = x_1 \vee x_2$ and $y_2 = x_1 \wedge x_2$ illustrated in Figure 1(a). 2-comparators can be easily encoded into SAT through the Tseitin transformations [16] using the clauses: $x_1 \rightarrow y_1$, $x_2 \rightarrow y_1$, $x_1 \wedge x_2 \rightarrow y_2$, $\overline{x_1} \rightarrow \overline{y_2}$, $\overline{x_2} \rightarrow \overline{y_2}$ and $\overline{x_1} \wedge \overline{x_2} \rightarrow \overline{y_1}$. A cardinality network can be decomposed into SAT by encoding all its 2-comparators.

Cardinality constraints can be decomposed into SAT through cardinality networks. For instance, a constraint $x_1 + \cdots + x_n \leqslant K$ can be decomposed in two steps: firstly, we build a $K+1$-cardinality network and encode it into SAT. Secondly, we add the clause $\overline{y_{K+1}}$, where y_{K+1} is the $K+1$-th output of the network. Notice that this implies that no $K+1$ inputs $(x_1, x_2, \ldots, x_n)$ are *true*, since the $K+1$-th output of a $K+1$-cardinality network is *true* if and only if there are at least $K+1$ *true* inputs.

Similarly, the constraint $x_1 + \cdots + x_n \geqslant K$ can be decomposed into a K-cardinality network by adding the clause y_K, and $x_1 + \cdots + x_n = K$ can be decomposed with a $K+1$-Cardinality Network adding the clauses y_K and $\overline{y_{K+1}}$.[2]

Example 1. Figure 1 shows an 8-cardinality network. Constraint $x_1 + \cdots + x_8 \leqslant 3$ can be decomposed into SAT by adding the auxiliary variables $z_1, z_2, \ldots, z_{38}$; the definition clauses $x_1 \rightarrow z_1, x_2 \rightarrow z_1, x_1 \wedge x_2 \rightarrow z_2, x_3 \rightarrow z_3, \ldots$; and the unit clause $\overline{z_{35}}$.

2.4 Pseudo-Boolean Constraints

PB constraints are another kind of complex constraint. They take the form $a_1x_1 + \cdots + a_nx_n \# K$, where K and a_i are integers, the x_i are literals, and the relation operator $\#$ belongs to $\{\leqslant, \geqslant, =\}$. In this paper we assume that the operator $\#$ is $\leqslant$ and the coefficients a_i and K are positive. Other cases can be easily reduced to this one (see [6]).

PB Propagator. The propagator must keep the current sum s during the search, defined as the sum of all coefficients a_i for which x_i is true. This value can be easily incrementally computed: every time the SAT solver sets a literal x_i of the constraint to true, the propagator adds a_i to s, and when the literal is unassigned by the SAT solver it subtracts a_i. For each $i \in \{1, \ldots, n\}$ such that x_i is unassigned and $K - s < a_i$, the propagator sets x_i to false. The propagator can produce explanations in the same way as in the cardinality case: if it has propagated x_j to false, $x_{i_1} \wedge \cdots \wedge x_{i_r} \rightarrow \overline{x_j}$, is returned as the explanation, where $x_{i_1}, \cdots, x_{i_r}$ are all the literals of the constraint with true polarity.

PB Decomposition. PB decomposition into SAT can be made in two steps: first, we build the reduced ordered binary decision diagram (BDD) [17] of the PB constraint; second, we decompose the BDD into SAT.

[2] Actually, cardinality networks for $\leqslant$ constraints can be encoded into SAT only adding the first 3 clauses of every 2-comparator. Similarly, in $\geqslant$ constraints we only need the last 3 clauses of 2-comparators [7].

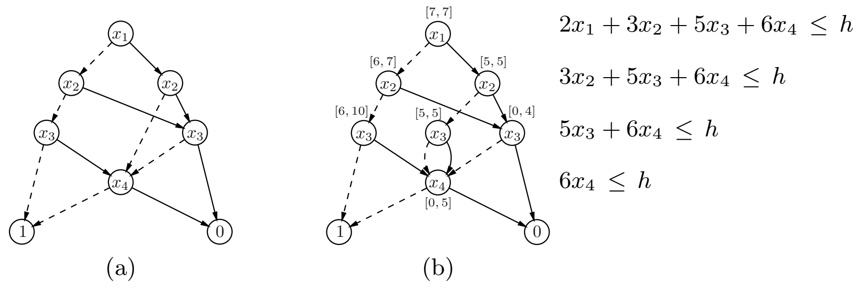

Fig. 2. (a) The BDD of PB constraint $2x_1+3x_2+5x_3+6x_4 \leqslant 7$, and (b) the BDD with long arcs to internal nodes replaced using intermediate nodes and with the intervals given for each node. Each node represents the constraint shown to the right of the BDD for the values of h given by the range.

A BDD of a PB constraint $a_1x_1 + \cdots + a_nx_n \leqslant K$ is a decision diagram that represents this constraint: it has a root node with *selector variable* x_1, the BDD of $a_2x_2 + \cdots + a_nx_n \leqslant K - a_1$ as a *true child* and the BDD of $a_2x_2 + \cdots + a_nx_n \leqslant K$ as *false child.* Moreover, it is *reduced*, i.e., there is no node with identical true and false child, and there are no isomorphic subtrees.

Example 2. Let us consider the PB constraint $c \equiv 2x_1 + 3x_2 + 5x_3 + 6x_4 \leqslant 7$. The BDD of that constraint is shown in Figure 2(a). False (true) children are indicated with dashed (solid) arrows. Terminal node 0 (1) represents the BDDs of Boolean false (true) function.

This BDD represents the constraint in the following sense: assume x_1 and x_4 are true and x_2 is false. The constraint is false no matter the polarity of x_3, since $2x_1 + 3x_2 + 5x_3 + 6x_4 \geqslant 2x_1 + 6x_4 = 8 > 7$. In the BDD of the figure 2(a), if we follow the first solid arrow (since x_1 is true), then the dashed arrow (x_2 is false) and finally the solid one (x_4 is true), we arrive to the false terminal node: that is, the assignment does not satisfy the constraint.

For lazy decomposition we will decompose the BDD one layer at a time from the bottom-up. To simplify this process we create a (non-reduced) BDD which does not have any arcs that skip a level unless they go direct to a terminal node, by introducing artificial nodes. Figure 2(b) shows the resulting BDD for c.

Given a node ν with selector variable x_i, we define *the interval* of ν as the set of integers h such that $a_ix_i + \cdots + a_nx_n \leqslant h$ is represented by the BDD rooted at node ν. This set is always an interval (see [14]). Figure 2(b) shows the intervals of constraint $2x_1 + 3x_2 + 5x_3 + 6x_4 \leqslant 7$. The BDDs for a PB constraint can be efficiently built, as shown in [18]. The algorithm, moreover, returns the interval of every BDD's node.

We follow the encoding proposed in [14]: for every node, we introduce a fresh variable. Let ν be a node with selector variable x_j and true and false children t and f. We add the clauses $\nu \to f$ and $\nu \wedge x_j \to t$. We also add a unit clause for setting the root of the BDD to true, and unit clauses setting the true and false terminal nodes to true and false respectively.

This encoding has the following property. Let A be a partial assignment of the variables $x_1, x_2, \ldots, x_n$, and let ν be the node of the BDD of the PB constraint $c \equiv a_1x_1 + \ldots + a_nx_n \leqslant K$, with selector variable x_i and interval $[\alpha, \beta]$. Then, the unit propagation of the partial assignment A and the encoding of the constraint c produces:

– ν if and only if $c \wedge A \models (a_ix_i + \cdots + a_nx_n \leq \beta)$.
– $\overline{\nu}$ if and only if $c \wedge A \models (a_ix_i + \cdots + a_nx_n > \beta)$.

In other words, if $a_1x_1 + \cdots + a_{i-1}x_{i-1} \geqslant K - \beta$ in a partial assignment, unit propagation sets ν to true. If ν is false, unit propagation assures that $a_1x_1 + \cdots + a_{i-1}x_{i-1} \leqslant K - \beta - 1$.

The way of ordering the constraint before constructing the BDD has a big impact on the BDD size. Computing the optimal ordering with respect the BDD size is a NP-hard problem [19], but experimentally the increasing order ($a_1 \leq a_2 \leq \cdots \leq a_n$) is shown to be a good choice. In this paper we use this order.

3 Lazy Decomposition

The idea of lazy decomposition is quite simple: a Lazy Decomposition (LD) solver is, in some sense, a combination of a Lazy Clause Generation solver and an eager decomposition. LD solvers, as LCG solvers, are composed of a SAT solver engine (that deals with the propositional part of the problem) and propagators, each one in charge of a complex constraint. The difference between LCG and LD solvers lies in the role of the propagators: LCG propagators only propagate and give explanations. LD propagators, in addition, detect which variables of the decomposition would be helpful. These variables and the clauses from the eager decomposition involving them are added to the SAT solver engine.

LD is not specific to a few complex constraints, but a general methodology. Given a complex constraint type and an eager decomposition method for it, an LD propagator must be able to perform the following actions:

– **Identify (dynamically) which parts of the decomposition would be helpful to learning:** LD can be seen as a combined methodology that aims to take advantage of the most profitable aspects of LCG and eager decomposition. This point assures that the solver moves to the decomposition when it is the best option.
– **Propagate the constraint when any subset of the decomposition has been added:** The propagator must work either without decomposition or with a part of it.
– **Avoid propagation for the constraint which is handled by the current decomposition:** auxiliary variables from the eager decomposition have their own meanings. The propagator must use these meanings in order to efficiently propagate the constraint when it is partially decomposed. For example, if the entire decomposition is added, we want the propagator to do no work at all.

In this paper we present two examples of LD propagators: the first one, for cardinality constraints, is based on the eager decomposition of Cardinality Networks [7]. The second one, a propagator for pseudo-Boolean constraints, is based on a BDD decomposition [14].

3.1 Lazy Decomposition Propagator for Cardinality Constraints

In this section we describe the LD propagator for a cardinality constraint of the form $x_1 + x_2 + \ldots + x_n \leqslant K$. LD propagators for $\geqslant$ or $=$ cardinality constraints can be defined in a similar way.

According to Section 2.3, the decomposition of a cardinality constraint based on cardinality networks consists in the encoding of 2-comparators into SAT. A key property of the 2-comparator $\mathsf{2comp}(x_1, x_2, y_1, y_2)$ of Figure 1(a) is that $x_1 + x_2 = y_1 + y_2$. This holds since $y_1 = x_1 \vee x_2$ and $y_2 = x_1 \wedge x_2$. Thus we can define a *2-comparator decomposition step* for 2-comparator $\mathsf{2comp}(x_1, x_2, y_1, y_2)$ as replacing the current cardinality constraint $x_1 + x_2 + x_3 + \ldots + x_n \leqslant K$ by $y_1 + y_2 + x_3 + \ldots + x_n \leqslant K$ and adding a SAT decomposition for the 2-comparator. The resulting constraint system is clearly equivalent. The decomposition introduces the new variables y_1 and y_2 to the SAT Solver engine.

The propagation of the LD propagator works just as in the LCG case. As decomposition occurs the cardinality constraint that is being propagated changes by substituting newly defined decomposition variables for older variables.

Example 3. Figure 1 shows an 8-cardinality network. A LD propagator for the constraint $x_1 + \ldots + x_8 \leqslant 3$ initially behaves as an LCG propagator for that constraint. When variables $z_1, z_2, \ldots, z_{12}$ are introduced by decomposing the corresponding six 2-comparators, the substitutions result in the cardinality constraint $z_5 + z_6 + z_7 + z_8 + z_9 + z_{10} + z_{11} + z_{12} \leqslant 3$.

An LD propagator must determine parts of the decomposition that should be added to the SAT solver. For efficiency, our LD solver adds variables only when it performs a restart: restarts occurs often enough for generating the important variables *not too late*, but occasionally enough to not significantly affect solver performance. Moreover, it is much easier to add variables and clauses to the solver at the root of search.

The propagator assigns a natural number act_i, the *activity*, to every literal x_i of the constraint. Every time a nogood is constructed, the activity of the literals belonging to the nogood is incremented by one. Each time the solver restarts the propagator checks if the activities of the literals of the constraint are greater than λN, where N is the number of conflicts since the last restart and λ is a parameter of the LD solver.

If $act_i \leqslant \lambda N$ then $act_i := act_i/2$. This is done in order to focus on the recent activity. If $act_i > \lambda N$, there are three possibilities:

- If x_i is not the input of a 2-comparator (i.e. an output of the cardinality network) nothing is done.

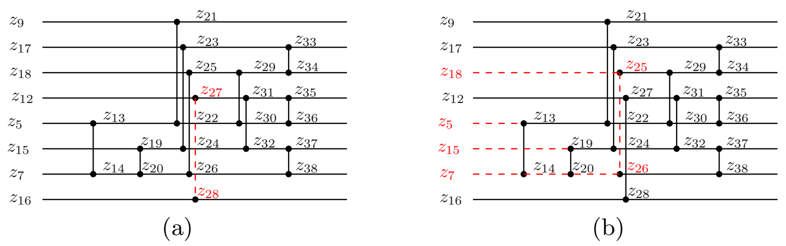

Fig. 3. The remaining undecomposed sorting network after decomposing some 2-comparators with (a) $\mathsf{2comp}(z_{12}, z_{16}, z_{27}, z_{28})$ shown dotted, and (b) inputs leading to $\mathsf{2comp}(z_{18}, z_{20}, z_{25}, z_{26})$ shown dotted.

- If x_i is an input of a 2-comparator $\mathsf{2comp}(x_i, x_j, y_1, y_2)$, and its other input x_j has been already generated by the decomposition, we perform a decomposition step on the comparator.
- If x_i is an input of a 2-comparator $\mathsf{2comp}(x_i, x_j, y_1, y_2)$, and its other input x_j has not been generated by the decomposition yet, we proceed as follows: let $S = \{x_{k_1}, x_{k_2}, \ldots, x_{k_s}\}$ be the literals in the current constraint that, after some decomposition steps, can reach x_j. We perform a decomposition step on all the comparators whose inputs both appear in S. Thus x_j is "closer" to being generated by decomposition.

Example 4. Assume the LD propagator for the constraint $x_1 + \ldots + x_8 \leqslant 3$ has generated some variables, so the current constraint is $z_9 + z_{17} + z_{18} + z_{12} + z_5 + z_{15} + z_7 + z_{16} \leqslant 3$. The remaining undecomposed cardinality network is shown in Figure 3.1(a).

In a restart, if the activity of z_{12} is greater than λN we decompose the comparator $\mathsf{2comp}(z_{12}, z_{16}, z_{27}, z_{28})$ generating new literals z_{27} and z_{28} and using them to replace z_{12} and z_{16} in the constraint.

However, if the activity of z_{18} is greater than λN, we cannot decompose $\mathsf{2comp}(z_{18}, z_{20}, z_{25}, z_{26})$ since z_{20} has not been generated yet. The literals reaching z_{20} are z_5, z_{15} and z_7 (see Figure 3.1(b)). Since z_5 and z_7 are the inputs of a 2-comparator $\mathsf{2comp}(z_5, z_7, z_{13}, z_{14})$, this comparator is encoded: z_{13} and z_{14} are introduced and they replace z_5 and z_7 in the constraint.

3.2 Lazy Decomposition Propagator for PB Constraints

In this section we describe the LD propagator for a PB constraint of the form $c \equiv a_1x_1 + \cdots + a_nx_n \leqslant K$ with $a_i > 0$, since other PB constraints can be reduced to this one.

Suppose $\mathcal{B}$ is the BDD for the PB constraint c. The decomposition of the constraint works as follows: if ν is a node with selector variable x_i and interval $[\alpha, \beta]$, ν is set to true if $a_1x_1 + \cdots + \ldots + a_{i-1}x_{i-1} \geqslant K - \beta$. If ν is set to false, the encoding assures that $a_1x_1 + \cdots + a_{i-1}x_{i-1} \leqslant K - \beta + 1$. The LD propagator must maintain this property for nodes ν which have been created as a literal via decomposition.

In our LD propagator, the BDD is lazily encoded from bottom to top: all nodes with the same selector variable are encoded together, thus removing a layer from the bottom of the BDD. Therefore, the LD propagator must deal with the nodes ν at some level i which all represent expressions of the form $a_i x_i + \cdots + a_n x_n \leqslant \beta_\nu$ or equivalently $a_1 x_1 + \cdots + a_{i-1} x_{i-1} \geqslant K - \beta_\nu$. Suppose ν' is the node at level i with highest β_ν where ν' is currently false. The decomposed part of the original PB constraint thus requires that $a_1 x_1 + \cdots + a_{i-1} x_{i-1} \leqslant K - \beta_{\nu'} - 1$. Define $K_i = K - \beta_{\nu'} - 1$, and $node_i = \nu'$

The LD propagator works as follows. The propagator maintains the current sum (lower bound) of the expression $s = a_1 x_1 + \cdots + a_{i-1} x_{i-1}$, just as in the LCG case. If this value is greater than $K - \beta_\nu$ for some leaf node ν with selector variable x_i and interval $[\alpha_\nu, \beta_\nu]$, this node variable ν is set to true. If some leaf node ν (with selector variable x_i and interval $[\alpha_\nu, \beta_\nu]$) is set to false, we set $K_i = K = \beta_\nu - 1$ and $node_i = \nu$. If, at some moment, $s + a_j$ for some $1 \leqslant j < i$ where x_j is undefined is greater than K_i, the propagator sets x_j to false. The explanation is the literals in $x_1, \ldots, x_{i-1}$ that are true and $node_i$.

The policy for lazy decomposition is as follows. Every time a nogood is generated that requires explanation from the PB constraint c, an activity act_c for the constraint c is incremented. If at restart $act_c \geqslant \mu N$ where N is the number of nogoods since last restart we decompose the bottom layer of c and set $act_c = 0$. Otherwise $act_c := act_c/2$.

Note that the fact that the coefficients a_i in c are in increasing order is important. Big coefficients are more important to the constraint and hence their corresponding variables are likely to be the most valuable for decomposition.

4 Experimental Results

The goals of this section are, first, to check that Lazy Decompositions solvers do in fact significantly reduce the number of auxiliary variables generated and, secondly, to compare them to the LCG and eager decomposition solving approaches. For some problems we include other related solving approaches to illustrate we are not optimizing a very slow system.

All the methods are programmed in the Barcelogic SAT solver [20]. All experiments were performed on a 2Ghz Linux Quad-Core AMD. All the experiments used a value of $\lambda = 0.3$ and $\mu = 0.1$. We experimented with different values and found values for λ between 0.1–0.5 give similar performance, while values for μ between 0.05–0.5 also give similar performance. While there is more to investigate here, it is clear that no problem specific tuning of these parameters is required.

4.1 Cardinality Optimization Problems

Many of the benchmarks on which we have experimented are pure SAT problems with an optimal cardinality function (i.e., an objective function $x_1 + \cdots + x_n$) to minimize.

These problems can be solved by branch and bound: first, we search for an initial solution solving the SAT problem. Let O be the value of $x_1 + \cdots + x_n$ in this solution. Then, we include the cardinality constraint $x_1 + \cdots + x_n \leqslant O - 1$. We repeatedly solve replacing the cardinality constraint by $x_1 + \cdots + x_n \leqslant O - 1$, where O is the last solution found. The process finishes when the last problem is unsatisfiable, which means that O is the optimal solution.

Notice that this process can be used for all approaches considered. In the cardinality network decomposition approach, the encoding is not re-generated every time a new solution is found: we just have to add a unit clause setting the O-th output variable of the network to false. LCG and LD solvers can also easily be adapted as branch and bound solvers, by modifying the bound on the constraint.

For all the benchmarks of this section we have compared the LCG solver for cardinality constraints (LCG), the eager cardinality constraint decomposition approach (DEC), our Lazy Decomposition solver for cardinality constraints (LD), and the three best solvers for industrial partial MaxSAT problems in the past Partial MaxSAT Evaluation 2011: versions 1.1 (QMaxSAT1.1) and 4.0 (QMaxSAT4.0) of QMaxSAT [21] and Pwbo solver, version 1.2 (Pwbo) [22].

Partial MaxSAT. The first set of benchmarks we used were obtained from the MaxSAT Evaluation 2011 (http://maxsat.ia.udl.cat/introduction/), industrial partial MaxSAT category. The benchmarks are encodings of different problems: filter design, logic synthesis, minimum-size test pattern generation, haplotype inference or maximum-quartet consistency.

We can easily transform these problems into SAT problems by introducing one fresh variable to any soft clause. The objective function is the sum of all these new variables. Time limit was set to 1800 seconds per benchmark as in the Evaluation. Table 1(a) shows the number of problems (up to 497) solved by the different methods after, respectively, 15 seconds, 1 minute, etc.

In these problems the eager decomposition approach is much better than the LCG solver. Our LD approach has a similar behavior to the decomposition approach, but LD is faster in the easiest problems. Notice that with these results we would be the best solver in the evaluation, even though our method for solving these problems (adding a fresh variable per soft clause) is a very naive one!

Table 1. Number of instances solved of (a) 497 partial MaxSAT benchmarks and (b) 600 DES benchmarks

Method	15s	1m	5m	15m	30m
DEC	211	296	367	**382**	386
LCG	144	209	265	275	279
LD	**252**	**319**	**375**	381	**386**
QMaxSAT4.0	191	274	352	370	377
Pwbo	141	185	260	325	354
QMaxSAT1.1	185	278	356	373	383

(a)

Method	15s	1m	5m	15m
DEC	409	490	530	541
LCG	151	186	206	228
LD	370	482	528	539
QMaxSAT4.0	275	421	534	**557**
Pwbo	265	361	423	446
QMaxSAT1.1	378	488	**537**	556
SARA-09	**411**	**501**	**537**	549

(b)

Discrete-Event System Diagnosis Suite. The next benchmarks we used are for discrete-event system (DES) diagnosis as presented in [23]. In these problems, we consider a plant modeled by a finite automaton. Its transitions are labeled by the events that occur when the transition is triggered. A sequence of states and transitions on the DES is called a trajectory; it models a behavior of the plant. Some events are observable, that is, an observation is emitted when they occur. The goal of the problem is, knowing that there is a set of faulty events in the DES, find a trajectory consistent with the observations that minimizes the number of faults. As all the problems in this subsection, it is modeled by a set of SAT clauses and a cardinality function to minimize.

In addition to the previously mentioned methods, we have also compared the best SAT encoding proposed in [23] (denoted by SARA-09). It is a specific encoding for these problems. Table 1(b) shows the number of benchmarks solved by the different methods after 15 seconds, 1 minute, etc.

The best method is that described in [23]. However, DEC and LD methods are not far from it. This is a strong argument for these methods, since SARA-09 is a specific method for these problems while eager and lazy decomposition are general methods. On the other hand, LCG does not perform well in these problems, and LD performs more or less as DEC. Both versions of QMaxSAT also performs very well on these problems.

Close Solution Problems. Another type of optimization problems is suggested in [3]. In these problems, we have a set of SAT clauses and a model, and we want to find the most similar solution (w.r.t the Hamming distance) to the given model if we add some few extra clauses. Table 2(a) contains the number of solved instances of the original paper after different times.

For the original problems LD is slightly better than eager decomposition (DEC) and much better than the other approaches.

Since the number of instances of the original paper was small, we created more instances. We selected the 55 satisfiable instances from SAT Competition 2011, industrial division, that we could solve in 10 minutes. For each of these 55 problems, we generated 10 close-solution benchmarks adding a single randomly generated new clause (with at most 5 literals) that falsified the previous model. 100 of the 550 benchmarks were unsatisfiable, so we removed them (searching

Table 2. Number of instances solved of the (a) 40 original close-solution problems and (b) 450 new close-solution problems

Method	15s	1m	5m	15m	60m
DEC	18	24	**31**	**34**	34
LCG	16	18	24	27	30
LD	**19**	**26**	**31**	**34**	**36**
QMaxSAT4.0	9	14	18	20	22
Pwbo	5	6	7	7	7
QMaxSAT1.1	6	11	16	17	19

(a)

Method	15s	1m	5m	15m	60m
DEC	143	168	208	226	243
LCG	181	223	242	255	268
LD	**187**	**230**	**252**	**262**	**279**
QMaxSAT4.0	55	55	63	69	80
Pwbo	102	144	179	204	215
QMaxSAT1.1	54	55	57	57	64

(b)

the closest solution does not make sense in an unsatisfiable problems). Table 2(b) shows the results on the remaining 450 instances.

For the new problems LCG and LD are the best methods with similar behaviour. Notice that for these problems the cardinality constraint size involves all the variables of the problem, so it can be huge. In a few cases, the encoding approach runs out of memory since the encoding needed more than 2^{25} variables. We considered these cases as a timeout.

4.2 MSU4

Another type of cardinality benchmarks also comes from the MaxSAT Evaluation 2008. In this case we solved them using the *msu4* algorithm [24], which transforms a partial MaxSAT problem into a set of SAT problems with multiple cardinality constraints.[3]

We have grouped all the problems that came from the same partial MaxSAT problem, and we set a timeout of 900 seconds for solving all the family of problems. We had 1883 families of problems (i.e., there were originally 1883 partial MaxSAT problems), but in many cases all the problems of the family could be solved by any method in less than 5 seconds, so we removed them. Table 3(a) contains the results on the remaining 479 benchmarks.

Table 3. (a) Number of families solved from 479 non-trivial MSU4 problems, and (b) number of instances solved from 669 problems PB Competition-2011

Method	15s	1m	5m	15m
DEC	190	282	352	411
LCG	123	168	212	241
LD	**263**	**336**	**410**	**435**

(a)

Method	15s	1m	5m	15m	60m
DEC	318	354	390	407	427
LCG	**372**	387	400	415	433
LD	369	382	401	423	439
borg	280	**406**	**438**	**445**	**467**

(b)

In these problems the LD approach is clearly the best, particularly in the first minute. The reason is that for most of the problems, DEC is faster than LCG, and LD performs similarly to DEC. But there are some problems where LCG is much faster than DEC: in these cases, LD is also faster than LCG, so in total it beats both other methods. Moreover, in some problems there are some *important* constraints which should be decomposed and some other which shouldn't. The LD approach can do this, while DEC and LCG methods either decompose all the constraints or none.

4.3 PB Competition Problems

To compare pseudo-Boolean propagation approaches we used benchmarks from the pseudo-Boolean Competition 2011 (http://www.cril.univ-artois.fr/PB11/),

[3] We thanks Albert Oliveras and Carlos Ansótegui for his assistance with these benchmarks.

category *DEC-SMALLINT-LIN* (no optimisation, small integers, linear constraints). In this problems we have compared the LCG, DEC and LD approaches for PB constraints and the winner of the pseudo-Boolean Competition 2011, the solver borg (borg) [25] version pb-dec-11.04.03. Table 3(b) contains the number of solved instances (up to 669) after 15 seconds, 1 minute, etc.

In this case, LCG approach is better than DEC, while LD is slightly better than LCG and much better than DEC since presumably it is worth decomposing some of the PB constraints to improve learning, but not all of them. The borg solver is clearly the best, but again it is a tuned portfolio solver specific for pseudo-Boolean problems and makes use of techniques (as in linear programming solvers) which treat all PB constraints simultaneously.

4.4 Variables Generated

One of the goals of Lazy Decomposition is to reduce the search space of the problem. In this section we examine the "raw" search space size in terms of the number of Boolean variables in the model.

Table 4 shows the results of all the problem classes. DEC gives the multiplication factor of Boolean variables created by eager decomposition. For example if the original problem has 100 Boolean variables and the decomposition adds 150 auxiliary variables, we have 250 Boolean variables in total and the multiplication factor will be 2.5. LD gives the multiplication factor of Boolean variables resulting from lazy decomposition. Finally *aux. %* gives the percentage of auxiliary decomposition variables actually created using lazy decomposition. The values in the table are the average over all the problems in that class.

In the optimization problems, there is just one cardinality constraint and most of the time is devoted to proving the optimality of the best solution. Therefore, the cardinality constraint appears in most nogoods since we require many explanations to prove the optimality of the solution. For these classes, the number of auxiliary variables we need is high 35-60 %. Still this reduction is significant.

In the MSU4 and PB problems, on the other hand, there are lots of complex constraints. Most of them have little impact in the problem (i.e., during the search they cause few propagations and conflicts). These constraints are not

Table 4. The average variable multiplication factor for (DEC) eager decomposition and (LD) lazy decomposition, and the average percentage of auxiliary decomposition variables created by lazy decomposition

Class of problems	DEC	LD	*aux. %*
Partial MaxSAT	7.46	5.41	61.72
DES	1.55	1.16	26.62
Original close-solution	12.21	7.48	45.33
New close-solution	24.55	12.38	35.88
MSU4	1.77	1.01	2.18
PB Competition	44.21	17.52	3.24

decomposed in the lazy approach. The LD solver only decomposes part of the most active constraints, so, the number of auxiliary variables generated in these problems is highly reduced.

5 Conclusions and Future Work

We have introduced a new general approach for dealing with complex constraints in complete methods for combinatorial optimization, that combines the advantages of decomposition and global constraint propagation. We illustrate this approach on two different constraints: cardinality and pseudo-Boolean constraints. The results show that, in both cases, our new approach is nearly as good as the best of the eager decomposition and global propagation approaches, and often better. Note that the strongest results for lazy decomposition arise when we have many complex constraints, since many of them will not be important for solving the problem, and hence decomposition is completely wasteful. But we can see that for the important constraints it is worthwhile to decompose.

There are many directions for future work. First we can clearly improve our policies for when and what part of a constraint to decompose. We will also investigate how to decide the right form of decomposition for a constraint during execution rather than fixing on a decomposition prior to search. We also plan to create lazy decomposition propagators for other complex constraints such as linear integer constraints, regular, lex, and incorporate the technology into a full lazy clause generation solver.

Acknowledgments. The first author is partially supported by Spanish Min. of Educ. and Science through the LogicTools-2 project (TIN2007-68093-C02-01) and by an FPU grant. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

1. Asín Achá, R., Nieuwenhuis, R.: Curriculum-based course timetabling with SAT and MaxSAT. Annals of Operations Research, 1–21 (February 2012)
2. Mcaloon, K., Tretkoff, C., Wetzel, G.: Sports league scheduling. In: Proceedings of the 3th Ilog International Users Meeting (1997)
3. Abío, I., Deters, M., Nieuwenhuis, R., Stuckey, P.J.: Reducing Chaos in SAT-Like Search: Finding Solutions Close to a Given One. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 273–286. Springer, Heidelberg (2011)
4. Warners, J.P.: A Linear-Time Transformation of Linear Inequalities into Conjunctive Normal Form. Information Processing Letters 68(2), 63–69 (1998)
5. Bailleux, O., Boufkhad, Y.: Efficient CNF Encoding of Boolean Cardinality Constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 108–122. Springer, Heidelberg (2003)
6. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2, 1–26 (2006)

7. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks: a theoretical and empirical study. Constraints 16(2), 195–221 (2011)
8. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). Journal of the ACM, JACM 53(6), 937–977 (2006)
9. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009)
10. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Why Cumulative Decomposition Is Not as Bad as It Sounds. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 746–761. Springer, Heidelberg (2009)
11. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Generic ILP versus specialized 0-1 ILP: an update. In: Pileggi, L.T., Kuehlmann, A. (eds.) 2002 International Conference on Computer-aided Design, ICCAD 2002, pp. 450–457. ACM (2002)
12. Bryant, R.E., Lahiri, S.K., Seshia, S.A.: Deciding CLU Logic Formulas via Boolean and Pseudo-Boolean Encodings. In: Proc. Intl. Workshop on Constraints in Formal Verification (September 2002); Associated with Intl. Conf. on Principles and Practice of Constraint Programming (CP 2002)
13. Bailleux, O., Boufkhad, Y., Roussel, O.: A Translation of Pseudo Boolean Constraints to SAT. Journal on Satisfiability, Boolean Modeling and Computation, JSAT 2(1-4), 191–200 (2006)
14. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: BDDs for Pseudo-Boolean Constraints – Revisited. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 61–75. Springer, Heidelberg (2011)
15. Manolios, P., Papavasileiou, V.: Pseudo-boolean solving by incremental translation to SAT. In: Formal Methods in Computer-Aided Design, FMCAD (2011)
16. Tseitin, G.S.: On the Complexity of Derivation in the Propositional Calculus. Zapiski Nauchnykh Seminarov LOMI 8, 234–259 (1968)
17. Bryant, R.E.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers 35(8), 677–691 (1986)
18. Mayer-Eichberger, V.: Towards Solving a System of Pseudo Boolean Constraints with Binary Decision Diagrams. Master's thesis, Lisbon (2008)
19. Tani, S., Hamaguchi, K., Yajima, S.: The Complexity of the Optimal Variable Ordering Problems of Shared Binary Decision Diagrams. In: Ng, K.W., Balasubramanian, N.V., Raghavan, P., Chin, F.Y.L. (eds.) ISAAC 1993. LNCS, vol. 762, pp. 389–398. Springer, Heidelberg (1993)
20. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: The Barcelogic SMT Solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 294–298. Springer, Heidelberg (2008)
21. Koshimura, M., Zhang, T., Fujita, H., Hasegawa, R.: Qmaxsat: A partial max-sat solver. JSAT 8(1/2), 95–100 (2012)
22. Martins, R., Manquinho, V., Lynce, I.: Parallel Search for Boolean Optimization. In: RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (2011)
23. Anbulagan, Grastien, A.: Importance of Variables Semantic in CNF Encoding of Cardinality Constraints. In: Bulitko, V., Beck, J.C. (eds.) Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009. AAAI (2009)
24. Manquinho, V., Marques-Silva, J., Planes, J.: Algorithms for Weighted Boolean Optimization. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 495–508. Springer, Heidelberg (2009)
25. Silverthorn, B., Miikkulainen, R.: Latent class models for algorithm portfolio methods. In: Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (2010)