Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads *

Marijn J.H. Heule[1,2], Oliver Kullmann[3], Siert Wieringa[4], and Armin Biere[2]

[1] Delft University of Technology, The Netherlands
[2] Johannes Kepler University Linz, Austria
[3] Swansea University, United Kingdom
[4] Aalto University Helsinki, Finland

Abstract. Satisfiability (SAT) is considered as one of the most important core technologies in formal verification and related areas. Even though there is steady progress in improving practical SAT solving, there are limits on scalability of SAT solvers. We address this issue and present a new approach, called *cube-and-conquer*, targeted at reducing solving time on hard instances. This two-phase approach partitions a problem into many thousands (or millions) of cubes using lookahead techniques. Afterwards, a conflict-driven solver tackles the problem, using the cubes to guide the search. On several hard competition benchmarks, our hybrid approach outperforms both lookahead and conflict-driven solvers. Moreover, because *cube-and-conquer* is natural to parallelize, it is a competitive alternative for solving SAT problems in parallel.

1 Introduction

Satisfiability (SAT) solvers have become very powerful tools to tackle problems ranging from industrial formal verification [4] to hard combinatorial challenges [27]. The most successful tools are known as *conflict-driven clause learning* (CDCL) solvers [24]. These solvers have data-structures optimized for huge instances and focus reasoning on learning new clauses from emerging conflicts. Although there exist several approaches to parallelize CDCL solvers [10], it appears hard to significantly improve performance on most industrial problems.

On the other hand, *lookahead* solvers [14] focus on small hard problems which require sophisticated heuristics to solve them efficiently. These solvers can be parallelized naturally and effectively. Yet, even with many cores at hand, they cannot compete with single core CDCL solvers on industrial problems.

While developing a method for computing van der Waerden numbers, Kullmann observed that CDCL and lookahead solvers can be interleaved in such a way that the combination outperforms both pure methods. In short, lookahead is used to assign a certain fraction of the variables, and afterwards CDCL tackles the reduced problem. For optimal performance the lookahead solver partitions the original problem into thousands (sometimes millions) of cubes. The CDCL solver iteratively assumes each cube to be true and solves the simplified instance.

In order to apply this method, called *cube-and-conquer*, on a large spectrum of problems, we present a mechanism that determines dynamically when to cut

* The first and the fourth author are supported by the Austrian Science Foundation (FWF) NFN Grant S11408-N23 (RiSE). The third author is supported by Academy of Finland project 139402.

off a branch in the search-tree of a lookahead solver to send it to a CDCL solver. Using this mechanism, several hard industrial problems can be solved more efficiently using the combination of solvers than with a stand-alone SAT solver. Additionally, the combined solving method can be parallelized naturally as well. Therefore, using a parallel implementation of our method, we are able to solve some hard instances faster than alternative methods.

Our approach is based on the following intuition. Obviously the reduced formulas, after applying some decisions, become easier to solve. Furthermore, at least empirically, CDCL solvers are effective on solving instances which are rather easy for their size, utilizing *local* heuristics including those based on variable activities. On the other hand, lookahead solvers are considered to be better at picking good decisions at the top-level, by using more global heuristics. There has to be a transition between hard and easy subproblems. So we try to switch from lookahead to CDCL solving when the subproblem seems to become easy.

The outline of this paper is as follows. After some preliminaries in Section 2, an overview of the cube-and-conquer method is provided in Section 3 as well as a description of both solver types. Section 4, discussing the above application to Ramsey theory, offers a motivating study of the method. Then a general methodology is developed. The details of the first phase, the "cube"-phase (partitioning the problem) are discussed in Section 5, and the details of the second phase, the "conquer"-phase (solving the sub-problems) in Section 6. Experimental results are presented in Section 7 and some conclusions are drawn in Section 8.

2 Preliminaries

For a Boolean variable x, there are two *literals*, the positive literal, denoted by x, and the negative literal, denoted by $\neg x$. A *clause* is a disjunction of literals, and a *CNF formula* is a conjunction of clauses. A clause can be seen as a finite set of literals, and a CNF formula as a finite set of clauses. A *unit clause* contains exactly one literal. A truth assignment for a CNF formula F is a function φ that maps variables in F to $\{\mathbf{t}, \mathbf{f}\}$. If $\varphi(x) = v$, then $\varphi(\neg x) = \neg v$, where $\neg\mathbf{t} = \mathbf{f}$ and $\neg\mathbf{f} = \mathbf{t}$. A clause C is satisfied by φ if $\varphi(l) = \mathbf{t}$ for some $l \in C$. An assignment φ satisfies F if it satisfies every clause in F. A *cube* is a conjunction of literals and a *DNF formula* a disjunction of cubes. A cube can be seen as a finite set of literals and a DNF formula as a finite set of cubes. If $c = (l_1 \wedge \ldots \wedge l_k)$ is a cube, then $\neg c = (\neg l_1 \vee \ldots \vee \neg l_k)$ is a clause. A truth assignment φ can be seen as the cube of literals l for which $\varphi(l) = \mathbf{t}$. A cube c is satisfied by φ if $\varphi(l) = \mathbf{t}$ for all $l \in c$. An assignment φ satisfies DNF formula D if it satisfies some cube in D. A DNF formula D is called a *tautology* if every full assignment φ satisfies D. For a CNF formula F, *Boolean constraint propagation* (BCP) (or *unit propagation*) propagates all unit clauses, i.e., repeats the following until fix-point: if there is a unit clause $(l) \in F$, remove from $F \setminus \{(l)\}$ all clauses that contain the literal l, and remove the literal $\neg l$ from all clauses in F. The resulting formula is referred to as $\mathrm{BCP}(F)$. If $\emptyset \in \mathrm{BCP}(F)$, we say that BCP derives a conflict.

3 Combining CDCL and Lookahead

The main complete SAT solver types are *conflict-driven clause learning* (CDCL) solvers [24] and *lookahead* solvers [14]. In short, CDCL solvers are optimized for

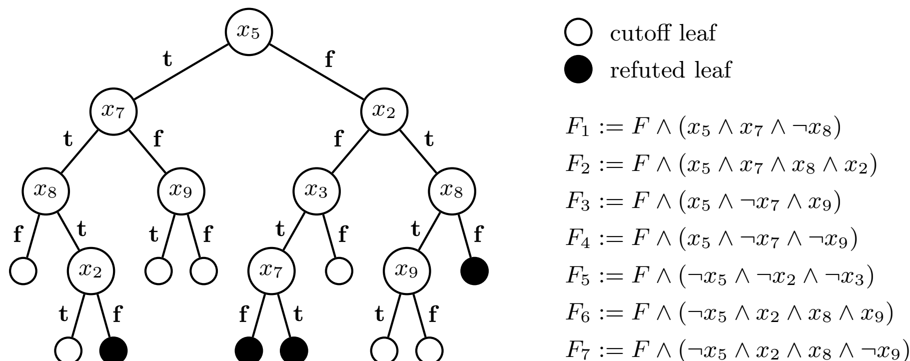

Fig. 1. A partition of a CNF formula F into seven subformulas F_i. The binary search tree on the left is constructed by a lookahead solver. It shows in the internal nodes the decision variable, and on the edges the truth value of a branch. Black leaves represent refuted leaves, while white leaves are cutoff leafs. The decisions of cutoff leaves yield a cube of assumptions that together with F forms a subformula F_i.

large industrial problems and consequently use inexpensive decision heuristics. In contrast, lookahead solvers focus on small hard problems on which it pays off to compute sophisticated decision heuristics. This section describes the main features of these solvers, and how we want to combine both types.

Overview The central approach in this paper deals with a lookahead solver that partitions a formula into many subformulas which in turn are solved by a CDCL solver. The sophisticated decision heuristics of lookahead solvers are used to compute important decision variables. These decisions are provided to the CDCL solver to guide the search process.

Figure 1 illustrates this approach by an example. The left shows a binary search tree produced by a lookahead solver. Internal nodes contain a decision variable. On the edges the truth value is shown to which a decision variable is set to reach a child node. There are two possible leaf nodes. Either the lookahead solver refuted the branch because a conflict emerged, or the *cutoff heuristic* suggests that this branch should be solved by a CDCL solver. This heuristic (discussed in detail in Section 5) is crucial for the effectiveness of the approach.

The cutoff branches can be described as a cube of the decisions on the path to the leaf. A CDCL solver can solve the branch by either adding the decisions as unit clauses, or by adding them as *assumptions* (see the Incremental SAT solving paragraph below). In case one of the branches is satisfiable, the original formula is satisfiable (and hence remaining branches could be neglected). If all cutoff branches are unsatisfiable, the original formula is unsatisfiable.

The use of lookahead heuristics to partition a formula have been proposed by Hyvärinen *et al.* [15]. In [15] formulas are partitioned into dozens of subformulas which are distributed on a grid to be solved in parallel. The starting point of this paper is now the discovery, discussed in Section 4, that some hard combinatorial problems can be efficiently solved by partitioning them into many thousands of subformulas (millions for harder problems). Inspired by these results we focus on the latter approach. We also use more sophisticated lookahead techniques as employed in state-of-the-art lookahead solvers.

Lookahead solvers Since CDCL is currently the dominant approach in practical SAT solving, we assume the reader already knows how CDCL solvers work, and otherwise refer to [24] for more details.

Lookahead solvers combine the David-Putnam-Logemann-Loveland (DPLL) algorithm [7] with *lookaheads*; for a general discussion see [14,19], while we describe here an exemplary scheme. Given a CNF formula F, a lookahead on literal x works as follows: First, x is assigned to $\mathtt{t}$, followed by BCP. Second, in case there was no conflict, the difference between F and the reduced formula F' is measured. The quality of lookahead techniques depends heavily on the used measurement. A frequently used method weighs the clauses in $F' \setminus F$ (the ones that are reduced but not satisfied). Third, all simplifications are reversed to get back to F. If a conflict was detected during the lookahead, then x is forced to $\mathbf{f}$ and is called a *failed literal*. The measurements are used to determine the decision variable in each node of the search tree. In general a variable x is chosen for which both the lookahead on x and $\neg x$ result in a large reduction of the formula. We remark that this scheme combines reduction (elimination of failed literals) and lookahead (estimating the quality of a branch by considering its development in the future), while in general these processes can be different.

State-of-the-art lookahead solvers are `kcnfs` [8], `march` [25], `OKsolver` [18], and `satz` [23]. These solvers show strong performances on hard random k-SAT formulas, but they cannot compete with CDCL solvers on large industrial instances. Apart from random instances, lookahead techniques are also useful for combinatorial problems; these problems have some form of structure to be exploited, and yield relatively small but typically very hard SAT problems.

While measuring the reduction of the formula F, most lookahead solvers also perform *local learning*. In contrast to the learning in CDCL solvers, local learning computes clauses (mostly unary and binary) that can be added to the formula for further reduction, but that have to be removed again during backtracking to the parent node in the search tree. An example of local learning is hyper binary resolution [2]. Current state-of-the-art lookahead solvers do not implement conflict clause learning as in CDCL solvers, and mostly not even backjumping (except of the `OKsolver`). For an overview of local learning we refer to [14].

Incremental SAT solving A frequently used feature of CDCL solvers is *incremental SAT solving* [9]. The solver provides an interface to (i) add clauses to the formula and (ii) to solve the formula under a cube of assumptions (decisions at level 0). Both techniques are very useful for tools that integrate SAT solvers. The input of an incremental solver can be seen as a sequence consisting of both clauses and cubes, where each cube defines a *job* which is the conjunction of that cube and all clauses preceding it in the sequence. In the context of cube-and-conquer we solve one formula under a set of cubes, thus all clauses precede all cubes in the solver input. A useful feature of incremental SAT solvers is that if a formula has no solutions under a given cube c, then the solver returns a subset $c' \subseteq c$ that was required to prove unsatisfiability. The clause $\neg c'$ can then be added to the formula to improve performance on other cubes.

As an example of the above, let us return to Figure 1. Now, consider a CDCL solver solving F_2, which is F assuming cube $(x_5 \wedge x_7 \wedge x_8 \wedge x_2)$. If however actually only $(x_8 \wedge x_2)$ is required to proof unsatisfiability, then we can add $(\neg x_8 \vee \neg x_2)$ to the formula. This binary clause is conflicting with F_6 and F_7, so by adding it, these cubes are immediately refuted.

4 Creating Cubes: The basic method

In this section we describe cube-and-conquer in its simplest form, as it came out of investigations into van-der-Waerden-like numbers ([21,1,22]). The principle aim is to solve extremely hard instances, which would take many years on a single machine. Thus a natural splitting of the problem into sub-problems is applied, and since lookahead solvers are competitive on these instances, it is natural to use lookahead for this task. The great surprise now is that on these (easy) sub-problems, conflict-driven solvers are very fast, and via this collaboration a *total speed-up* (regarding the *total running time*) of at least a factor of two (compared always to the best single solver available) is achieved. So even on a single machine the problems are solved at least twice as fast, and additionally the splitting is ideal for parallelization (via clusters for example; no communication is needed between the processes). This was the birth of “cube-and-conquer”. The lookahead solver is the `OKsolver`, which participated successfully at the SAT 2002 competition and aims at being as “theoretically clean” as possible; see [18,19] for further information, and see the `OKlibrary` ([20]) for the renovated source code. It uses complete elimination of failed literals, and autarky reduction for the partial assignment at hand (see [17]). The distance along a branch is, as discussed above, a weighted sum of the number of new clauses, while the heuristics is the product of these values for the two branches (to be maximized); again (as for the reduction), all variables are (always) considered.

Computing the cubes is rather simple: cubes are partial assignments, corresponding to initial parts of the paths from the root to leaves in the splitting (branching) tree, and the task is to “cut off” these paths at the right place. Two methods are implemented, interpreting a depth parameter $D \geq 0$: either the branches are cut off when exactly D decisions have been made (method A), or when the total number of assigned variables (decisions, unit propagations, failed literals, autarkies) is at least D (method B).

The interface to the sub-solver is here as simple as possible: a complete decoupling is achieved by *applying* the partial assignments, and the sub-solver just gets the results. So each sub-instance is solved completely independent of each other, and the sub-solver only sees the sub-instance. For method A as well as for method B, the partial assignments contain everything: the decisions, the unit-propagation, the failed literals, the autarkies found (including pure literals).

On the implementation side, there are two simple data formats: either storing each partial assignment in its own file in DIMACS format (this is used for the experiments below), or creating an iCNF file[5], which here is basically just the concatenation of the instance and the partial assignments, put into one big file. Processing runs through the partial assignments, applies them to the original CNF, and calls the sub-solver on the sub-instance. Since only unsatisfiable instances are considered in this section, and the sub-instances are independent of each other, the order of the instances does not matter. All methods and all data are available in the `OKlibrary`, see [20]. The cutoff (the above parameter D) is determined ad-hoc such that sub-instances only take *a few seconds* (this seems to be around the optimum, but with less overhead, as achieved by the system discussed in Section 5, one can partition further — the more cubes the better).

[5] http://users.ics.tkk.fi/swiering/icnf/

We report here only on two instance classes, determining unsatisfiability of van-der-Waerden (vdW) instances and palindromic vdW instances, using in both cases two colors, and thus the instances have a canonical translation into boolean CNF. Such problems are explained (resp. introduced in the palindromic case) in [1], and they were also part of the SAT 2011 competition. The standard (boolean) vdW-problems are given by equations $\mathrm{vdw}(k_1, k_2) = n$, for natural numbers $k_1 \leq k_2 \leq n$, meaning that whenever partitioning $\{1, \ldots, n\}$ into two parts, it holds that the first part contains an arithmetic progression (ap for short) of size k_1 or the second part contains an ap of size k_2 (and n is minimal with this property). This gives a CNF with n variables $v_1, \ldots, v_n$ and with two clause-sizes k_1, k_2, where the clauses of length k_1 are all the ap's of size k_1, as positive clauses, and the clauses of length k_2 are all the ap's of size k_2, as negative clauses. The *palindromic* (boolean) vdW-problems are given by equations $\mathrm{vdw}^{\mathrm{pd}}(k_1, k_2) = (n_1, n_2)$ $(n_1 < n_2)$, with a similar meaning, only that now only palindromic partitions are allowed, thus regarding the partition as a bit-string of length n, given by the values of $v_1, \ldots, v_n$, and requiring that $(v_1, \ldots, v_n) = (v_n, \ldots, v_1)$. By these equations, the number of variables is halved, replacing v_n by v_1 and so on, and shorter clauses are obtained. Subsumption elimination is performed on the instances. There are now two unsatisfiable problems, one using $\frac{n_1+1}{2}$ variables, with $n = n_1 + 1$ as the smallest n with unsatisfiable problem, and one with $\frac{n_2+1}{2}$ variables, based on the smallest $n = n_2$ such that *all* $n' \geq n$ yield unsatisfiable problems. For standard vdW-instances, lookahead solvers can perform better than conflict-driven solvers, while for palindromic vdW-instances conflict-driven solvers are much better (here we are not speaking about cube-and-conquer, but about standard SAT solving). Method (B) for determining the cutoff was vastly superior (diminishing the variability of the sub-instances enormously), and is only considered here. As the sub-solver, `minisat-2.2.0` performed very well here and is used throughout. All times are on a single core with about 2 GHz (parallelization has not been used), and the times for the cube-and-conquer approach is the total time, including all computations (writing each sub-instance to file etc.). All solvers mentioned below for comparison seem best performing (as ordinary SAT solvers, on the original (full) instances).

For $\mathrm{vdw}(3, 15) = 218$ (yielding 13362 clauses) the lookahead solver `satz` (version 215) needs about 20h, while with $D = 35$ (yielding 32331 cubes) it is solved in about 4h. The maximal time per job is 5 seconds, enabling trivial optimal parallelization with more than 2000 processors (by just distributing the jobs for the sub-problems to the first available processor). For $\mathrm{vdw}(4, 8) = 146$ (yielding 4930 clauses) `picosat` (version 913) takes 8h. Setting $D = 20$ (yielding 65270 cubes), it is solved in 4h, with maximal job-time of 22s. `picosat` for $\mathrm{vdw}(5, 6) = 206$ was aborted after a week, while with $D = 20$ (yielding 91001 cubes) it was solved in about one day. For $\mathrm{vdw}^{\mathrm{pd}}(3, 25) = (586, 607)$ (yielding 45779 resp. 49427 clauses), `precosat` (version 570) used in both cases about 13 days, while with $D = 45$ (yielding 9120 resp. 13462 cubes) the problems were solved in about 6.5h resp. 2 days. For $\mathrm{vdw}^{\mathrm{pd}}(4, 12) = (387, 394)$ (yielding 15544 resp. 15889 clauses) `minisat` version 2.2.0, was aborted after 2 weeks, while setting $D = 30$ resp. $D = 34$ (yielding 132131 resp. 147237 cubes) solved the problems in 2 days resp. 8h. Finally, for $\mathrm{vdw}^{\mathrm{pd}}(5, 8) = (312, 323)$ (yielding 9121 resp. 9973 clauses), `minisat` used 3 1/2 days resp. 53 days, while setting $D = 20$ in both cases (yielding 22482 resp. 87667 cubes) solved it in 5h resp. 40h.

5 Creating Cubes: a general methodology

This section shows how to modify a lookahead solver into a partitioning tool. First, we explain where to modify the code, Section 5.1. Second, we present an adaptive mechanism to cut off branches in Section 5.2. We conclude with some important heuristics in Section 5.3. The automatic partitioning provided here essentially is able to simulate the splitting characteristics from Section 4.

5.1 General framework

The procedure *CreateCubes*, a modified lookahead solver for partitioning, shown in Figure 2, takes as input a CNF formula F and outputs two sets. The first set $\mathcal{A}$ is a disjunction of cubes for which each cube represents a set of assumptions that describe a cutoff branch in the DPLL tree. The cubes in $\mathcal{A}$ cover all subproblems of F that have not been refuted during the partition procedure. The second set $\mathcal{C}$ is a conjunction of clauses. Each of these (learnt) clauses are implied by F and represent refuted branches in the DPLL tree. Hence the clauses in $\mathcal{C}$ can be added to F to obtain a logically equivalent formula $F' := F \cup \mathcal{C}$.

The recursive procedure has five inputs. Besides F, $\mathcal{A}$, and $\mathcal{C}$, it passes on the set of *decision literals* (denoted by φ_{dec}) and the set of *implied literals* (denoted φ_{imp}). Implied literals are assignments that were forced by BCP or some form of learning such as failed literal reasoning. Initially, *CreateCubes* is called with the input formula F and all the other parameters as empty sets.

CreateCubes (CNF F, DNF $\mathcal{A}$, CNF $\mathcal{C}$, dec. lits. φ_{dec}, imp. lits. φ_{imp})

1 $\langle F, \varphi_{\text{imp}} \rangle :=$ *LAsimplify_and_learn* (F, φ_{dec}, φ_{imp})
2 **if** $\varphi_{\text{dec}} \cup \varphi_{\text{imp}}$ falsify a clause in F **then return** $\langle \mathcal{A}, \mathcal{C} \cup \{\neg\varphi_{\text{dec}}\} \rangle$
3 **if** cutoff heuristic is triggered **then return** $\langle \mathcal{A} \cup \{\varphi_{\text{dec}}\}, \mathcal{C} \rangle$
4 $l_{\text{dec}} :=$ *LAdecide* (F, φ_{dec}, φ_{imp})
5 $\langle \mathcal{A}, \mathcal{C} \rangle :=$ *CreateCubes* $(F, \mathcal{A}, \mathcal{C}, \varphi_{\text{dec}} \cup \{l_{\text{dec}}\}, \varphi_{\text{imp}})$
6 **return** *CreateCubes* $(F, \mathcal{A}, \mathcal{C}, \varphi_{\text{dec}} \cup \{\neg l_{\text{dec}}\}, \varphi_{\text{imp}})$

Fig. 2. The general framework of the recursive procedure *CreateCubes*.

In line 1 of the procedure, the method *LAsimplify_and_learn* is called. This method simplifies the formula by BCP and lookaheads, forcing some variables to certain truth values. All assigned variables are added to φ_{imp}. Additionally, it produces *local learnt clauses* which are added to F. In case the current assignment falsifies F then a conflict clause is learnt. This clause consists of the complements of the decisions and is added to $\mathcal{C}$ (line 2). Line 3 deals with cutting off branching which is further discussed in the next subsection. The procedure *LAdecide* on line 4 determines the next decision variable and preferred truth value based on lookaheads. There exists a vast body of work on these decision heuristics [19]. Section 5.3 offers the details of this produce.

After *CreateCubes* is terminated, $\mathcal{A}$ and $\mathcal{C}$ are optimized. First, the clauses in $\mathcal{C}$ are reduced in size by applying self-subsumption resolution. For instance, back to the example in Figure 1 with $(x_5 \vee x_2 \vee \neg x_3 \vee x_7), (x_5 \vee x_2 \vee \neg x_3 \vee \neg x_7) \in \mathcal{C}$, then the resolvent $(x_5 \vee x_2 \vee \neg x_3)$ replaces both antecedent clauses. When $\mathcal{C}$ is fully optimized, this set of conflict clauses is used to remove assumptions in $\mathcal{A}$. For instance if $(\neg x_5 \wedge x_2 \wedge x_8 \wedge x_9) \in \mathcal{A}$, and $(x_5 \vee \neg x_2 \vee x_8) \in \mathcal{C}$, then x_8 is removed as an assumption because it will be forced by BCP after $\mathcal{C}$ is added to F. After these optimizations until fix-point, $\mathcal{A}$ is a tautology.

5.2 Cutoff heuristic

The heuristic that triggers the cutoff of a branch is of crucial importance to create an effective partition. Ideally, this heuristic partitions the original problem into several subproblems such that 1) the runtimes to solve each of the subproblems are comparable and 2) the sum of these runtimes (at least) does not exceed the runtime of solving the original instance.

A (simplifying) interpretation of the results discussed in Section 4 is that for some hard combinatorial problems both objectives can be achieved by cutting off a branch if a certain fraction (say 10%) of the variables is assigned — this measure is much easier to handle than the solution time for the sub-instances, which for the experiments reported in Section 4 was determined in an ad-hoc manner. There actually the total solution time for the subproblems was not just not bigger than the original solution time, but much smaller. So this metric is very useful for several small hard problems. However, for the larger industrial instances, the number of decisions appears to be also of important to determine the hardness of a subproblem. Additionally, for these formulas sometimes a single decision assigns 10% of the variables, while for other formulas it requires over 100 decisions. In the former case the number of partitions becomes too small, while in the latter case the number of partitions becomes too large.

An alternative approach by Hyvärinen *et al.* [15] cuts off a branch after k decisions have been made (this was called method A in Section 4). The advantage of this approach is that one can clearly upper-bound the number of partitions in advance. However, branches with the same number of decisions are rarely equally hard to solve. It is often the case, that assigning a decision literal x to $\mathbf{t}$ results in significantly more implied literals than assigning x to $\mathbf{f}$ or vice versa.

We combine both approaches by using the product of the number of decisions and the number of assigned variables, $|\varphi_{\text{dec}}| \cdot |\varphi_{\text{dec}} \cup \varphi_{\text{imp}}|$, as the cutoff metric. Furthermore, the refined procedure *CreateCubes**, Figure 3, includes a dynamic cutoff mechanism. It implements the cutoff of a branch (with the cutoff heuristic discussed above) as shown in line 5 using a threshold parameter θ. Two lines update the value of θ. The first, the *increment rule* on line 1, raises the value by 5% without a condition. This rule aims to restore the value in case it was reduced too much. The second, the *decrement rule* on line 3, lowers the value by 30%. This rule tries to avoid two unfavorable situations described below.

First and most importantly, the value is decreased if the lookahead solver hits a conflict, meaning that the current node is a refuted branch. The rationale of this update is as follows. If the lookahead solver was able to show that the current node is conflicting, then probably a CDCL solver could have found the conflict faster. Additionally, if the CDCL solver would have found the conflict, then it could have analyzed it and possibly computed a smaller reason of this conflict (than all decisions as computed by the lookahead solver). By lowering θ, the mechanism tries to cut off neighboring branches before a conflict emerges.

Secondly, the mechanism prevents the recursive procedure from going too deep into the DPLL tree. For most interesting instances, it appeared useful to decrease θ for all nodes with a depth larger than 20. In case one wants the mechanism to finish creating cubes within a few seconds, then the condition should be dependent on the size of the formula, such as $|\varphi_{\text{dec}}| + \log_2(|F|) > 30$.

Initially, θ should be large enough to ensure that the mechanism will cut off the tree at a reasonable depth. We used $\theta := 1000$ as initial value. Using a value which is a factor 10 larger or smaller hardly influences the resulting partition. Using this initial value, θ will first be decreased before cutting off a branch.

```
    CreateCubes* (CNF F, DNF A, CNF C, dec. lits. φ_dec, imp. lits. φ_imp)
1       θ := 1.05 · θ
2       ⟨F, φ_imp⟩ := LAsimplify_and_learn (F, φ_dec, φ_imp)
3       if φ_dec ∪ φ_imp falsify a clause in F or |φ_dec| > 20 then θ := 0.7 · θ
4       if φ_dec ∪ φ_imp falsify a clause in F then return ⟨A, C ∪ {¬φ_dec}⟩
5       if |φ_dec| · |φ_dec ∪ φ_imp| > θ · |vars(F)| then return ⟨A ∪ {φ_dec}, C⟩
6       l_dec := LAdecide (F, φ_dec, φ_imp)
7       ⟨A, C⟩ := CreateCubes* (F, A, C, φ_dec ∪ {l_dec}, φ_imp)
8       return CreateCubes* (F, A, C, φ_dec ∪ {¬l_dec}, φ_imp)
```

Fig. 3. The recursive procedure *CreateCubes** with the cutoff mechanism.

5.3 Heuristics for splitting

Besides the development of the cutoff mechanism, the standard heuristics for lookahead solvers had to be tweaked in order to realize fast performance.

Decision heuristics. The default and costly lookahead evaluation heuristic (measurement) in most lookahead solvers is based on the clauses that are reduced, but not satisfied during a lookahead. These clauses are weighted depending on their (new) length. In general, a clause of length k has a weight which is a factor five times larger compared to a clause of length $k+1$. A more cheaply heuristic counts the number of variables that are assigned during the lookahead.

For an example of both heuristics, consider the formula F below. Because the longest clauses have length 3, all "new" clauses have length 2, so no weights are required. Let $\mathsf{eval}_{\text{cls}}(x_i)$ denote the clause based heuristic being the (weighted) sum of the reduced, not satisfied clauses and $\mathsf{eval}_{\text{var}}(x_i)$ the variable based heuristic being the number of assigned variables during the lookahead on $x_i = 1$. E.g., $\mathsf{eval}_{\text{var}}(\neg x_6) = 1$ and $\mathsf{eval}_{\text{cls}}(\neg x_6) = 2$ because the lookahead on $x_6 = 0$ reduces two clauses from ternary to binary, and only x_6 is assigned. Notice that the values of the two heuristics are not necessarily related. $\mathsf{eval}_{\text{cls}}(x_i)$ may be much smaller than $\mathsf{eval}_{\text{var}}(x_i)$. For instance $\mathsf{eval}_{\text{cls}}(\neg x_2) = 1$, while $\mathsf{eval}_{\text{var}}(\neg x_2) = 4$.

$$F = (\neg x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee x_3 \vee x_6) \;\wedge \\ (\neg x_1 \vee x_4 \vee \neg x_5) \wedge (x_1 \vee \neg x_6) \wedge (x_4 \vee x_5 \vee x_6) \wedge (x_5 \vee \neg x_6)$$

In general, lookahead solvers rank variables x_i by $\mathsf{eval}(x_i) \cdot \mathsf{eval}(\neg x_i)$. Ties are broken by $\mathsf{eval}(x_i) + \mathsf{eval}(\neg x_i)$. The decision heuristics select in each node of the DPLL tree the variable with the highest rank.

The default heuristics $\mathsf{eval}_{\text{cls}}$ appeared to be quite effective on instances that had none or few binary clauses. This is frequently the case for random and crafted instances used in the SAT competitions. However, we noticed that $\mathsf{eval}_{\text{var}}$ was more effective on industrial instances. An advantage of $\mathsf{eval}_{\text{var}}$ is that it does not require the eager data-structures used in lookahead SAT solvers. Hence, this heuristic can relatively easy be implemented in CDCL solvers.

Direction heuristics. Given a decision variable x, *direction heuristics* decide which branch (x to $\mathbf{t}$ or x to $\mathbf{f}$) to explore first; see Section 5.3.2 in [14] for more information. Direction heuristics in lookahead solvers aim to improve performance on satisfiable formulas. Therefore, the solver prefers the branch that is most "likely" to be satisfiable. For methods how to estimate such probabilities see Section 7.9 in [19], and see Subsection 4.6.2 in [3] for some discussions in the CSP context. As a cheap approximation one can take the least constraint branch first. This is the complementary strategy of the *first fail principle* [12] which is often used in Constraint Satisfaction. In case $\mathsf{eval}(x) < \mathsf{eval}(\neg x)$, x to $\mathbf{t}$ is explored first. Otherwise x to $\mathbf{f}$ is preferred. For a certain node with decision variable x, we refer to the branch with $\mathsf{eval}(x) < \mathsf{eval}(\neg x)$ as its *left branch*. The other branch we call its *right branch*.

The partition mechanism as described in Section 5.2 seems to be quite robust regarding the direction heuristics. The number of cubes and the average size of the cubes is hardly influenced by exploring the left or the right branch first. However the order in which partitions are visited has a clear impact on performance related to the left and right branches, when considering how the *sub-problems* are solved; see Section 6.1.

6 Solving Cubes

A CDCL solver deals with the second phase of the cube-and-conquer method. The solver takes as input the original formula F, optionally extended with the learnt clauses $\mathcal{C}$, and the set of assumption cubes $\mathcal{A}$. The latter is ordered based on some heuristic. For each cube $c \in \mathcal{A}$ based on this order, the CDCL solver solves $F \wedge c\,(\wedge\,\mathcal{C})$. First, we present how to solve the cubes sequentially (Section 6.1). Second, we discuss a parallel solving approach (Section 6.2).

6.1 Sequential solving

The sequential solving procedure is rather straightforward and shown in Figure 4. Iteratively, a cube $c \in \mathcal{A}$ is selected (line 3) and assumed to be true followed by solving the simplified formula (line 4). In case the result is satisfiable, the original formula is satisfiable and hence the procedure ends. After all cubes have been refuted, the formula is found to be unsatisfiable.

After refuting a cube, most CDCL solvers provide a technique, known as *AnalyzeFinal*, to extract a subset of the cube that was required to proof unsatisfiability. It can be useful to add the clause –the complement of this subset– to the formula (line 5). Adding it can help refuting another cube more easily and the CDCL solver cannot remove it (in contrast to learnt clauses). However, if $|\mathcal{A}|$ is much larger than $|F|$, the addition may significantly slow down performance.

Last, but not least, we observed that removing some learnt clauses after refuting a cube can significantly improve performance of cube-and-conquer. This can be explained by the intuition that the subproblems are relatively independent and hence the learnt clauses of one subproblem can hardly be reused for another subproblem. Removal of learnt clauses is realized by reseting the clause deletion policy after solving a cube (line 6). So the size of the clause database is reduced to its initial size and the least important clauses are kicked out.

```
   SolveCubes (CDCL solver S, CNF F, DNF A)
1      S.Load (F)
2      while A is not empty do
3          get a cube c from A and remove c from A
4          if S.SolveWithAssumptions (c) = satisfiable then return satisfiable
5          S.AnalyzeFinal ()                                      // optional
6          S.ResetClauseDeletionPolicy ()
7      return unsatisfiable
```

Fig. 4. The pseudo-code of the sequential solver using the partition.

Describing the cubes. In the partition procedure CreateCubes, the cube consists only of all decisions (φ_{dec}) from the root to the cutoff. Alternatively, one could describe a cube by all the assigned variables ($\varphi_{\text{dec}} \cup \varphi_{\text{imp}}$). The latter may include several assignments that a CDCL solver cannot reconstruct by BCP, for instance the failed literals. Recall that this approach is used is Section 4 and by Hyvärinen *et al.* [15,16]. However, it seems that communicating implied variables to a CDCL solver does not improve runtime. Throughout our experiments, using cubes consisting of only decision literals resulted in stronger performance.

The order in which the decision literals are assumed in the CDCL solver influence the size of conflict clauses. The natural order –the order in which the decisions were made– appears to be the best alternative.

Ordering the cubes. During the experiments, we observed a relation between the time it requires to refute a cube and the number of right branches between the root and the cutoff of that cube: the more right branches (also known as *discrepancies*), the easier the corresponding subformula. On the other hand, for satisfiable formulas, cubes that cover a solution tend to have *few right branches*. Although we focused mostly on unsatisfiable formulas, we observed that for satisfiable benchmarks it pays off to solve the cubes with few right branches first. This strategy is known as *limited discrepancy search* [13].

There is also another reasoning for preferring this order, namely when solving cubes in parallel (see Section 6.2). In case CreateCubes produces an unbalanced tree, then frequently one or a few cubes will consume most of the computation costs to solve a formula. Therefore, one should solve the hard cubes first: a few cores attack these cubes, while others solve the easy ones. Otherwise, if a hard cube needs to be solved in the end, there would no cubes left for the other cores.

6.2 Parallel solving

A natural extension of the approach in the prior section is to consider solving the partitions in parallel. In existing work on parallel SAT solving [10] two main approaches are distinguishable. The first aims to partition the formula in an attempt to divide the total workload evenly over multiple computation nodes, the second are so called *portfolio* approaches [11]. Rather than partitioning the formula, *portfolio* systems run multiple solvers in parallel, each attempting to solve the same formula, and the system finishes whenever the fastest solver finishes. Often such portfolios consist simply of multiple instances of the same CDCL solver, as those can be made to all traverse the search space in a different

order by as little as using different random seeds. Such parallel solvers thus mostly exploit the lack of robustness of SAT solvers, and can be surprisingly effective. Parallel SAT solvers of both types can be extended with exchange of learnt clauses between computation nodes.

In the solving phase of cube-and-conquer many partitions are independently solved and thus it can be easily parallelized. However as we make use of incremental SAT, so one can also think of this phase as one single incremental problem. In [26] two different job assignment strategies for parallel incremental SAT were discussed and implemented in a tool called `Tarmo`. That work was focused on Bounded Model Checking (BMC) but it can be seen as a general framework for parallel incremental SAT solving with clause sharing. The first strategy implemented is the *multijob* approach in which an idle node is assigned the first job that is not already assigned to any other node. When two nodes are idle at the same time the job assignment order is undefined but it is guaranteed that no two nodes ever work on the same job. The second strategy called *multiconv* is inspired by portfolio solvers, and it simply runs a conventional incremental SAT solver on all jobs on all nodes. The latter can be effective for BMC where jobs are difficult and job order is relevant. For cube-and-conquer however we deal with a huge number of jobs, most of which are very easy, which means there are no large deviations in single job run times for the *multiconv* strategy to exploit. For this application *multijob* is a natural choice, although it is not ideal. If the partitioning is uneven a small number of the jobs may make up a large fraction of the run time. Thus using *multijob* nodes given only easy jobs may end up sitting idle waiting for a small number of nodes with hard jobs to finish. In `Tarmo` we experimented also with an extended strategy, *multijob+*, which is like *multijob* except that it will assign a job that is already being solved by some node to nodes that would otherwise become idle. This modified strategy appeared to beneficial for performance of the cube-and-conquer solving phase.

Another feature of `Tarmo` is its ability to share learnt clauses between solver threads. As discussed in [26] different settings are possible for the amount of clauses shared. `Tarmo`'s default setting which shares learnt clauses that have a length which is below average appeared the most effective for this application.

After studying the parallelization of cube-and-conquer's solving phase using various versions of `Tarmo`, a special purpose multithreaded version of the fast SAT solver `lingeling` was created, which uses the basic *multijob* strategy. This special purpose solver called `iLingeling` is faster than `Tarmo` for this application although it does not use clause sharing or the *multijob+* strategy yet.

7 Experimental Results

The experiments focus on the strength of cube-and-conquer on hard application benchmarks. For this paper we used instances from the SAT 09 application category that were not solved during the competition (within the given timeout of 10,000 seconds) – the same set as used in [16]. We modified two existing SAT solvers according to the general method of cube-and-conquer. First, the lookahead SAT solver `march` [25] was converted into a splitting tool called `march_cc`. Second, the CDCL solver `lingeling` was extended to deal with iCNF files. This version called `iLingeling` also supports solving cubes in parallel. The sources of both tools are available on `http://fmv.jku.at/cnc/`.

Phase I of our cube-and-conquer implementation consists of A) simplifying the formula using the preprocessor of `lingeling` (option -s) and B) calling `march_cc` on the result. The cutoff mechanism in `march_cc` is implemented as shown in Figure 3. Three benchmarks in the SAT09 suite (9dlx* and sortnet*) remained too large after simplifying and caused memory problems for `march_cc`. Therefore, we replaced $|\varphi_{\text{dec}}| > 20$ by $|\varphi_{\text{dec}}| > 10$ in the decrement rule for these instances. We used the cheap $\mathsf{eval}_{\text{var}}$ lookahead evaluation, because it resulted in improved performance compared to $\mathsf{eval}_{\text{cls}}$. The reported runtimes in Table 1 for phase I include both preprocessing and partitioning – the latter consuming most of the time. Notice that partitioning is based on lookahead. Hence, this part can relatively easy be parallelized. Since solving cubes requires more time than creating them, this optimization is left for future work. `march_cc` outputs an iCNF file which concatenates the simplified formula and a line for each cube.

For phase II of cube-and-conquer, the iCNF file is provided to `iLingeling`. We used a 12-core-machine during this phase. On such a machine, `iLingeling` starts 12 worker threads using separate `lingeling` solvers. Idle threads ask for the first cube that has not been dealt with by another thread. After receiving a cube, `lingeling` solves the reduced formula of the first phase with the cube as assumptions. After a cube is refuted, the clause database of the corresponding `lingeling` is reduced as discussed in Section 6.1. A thread terminates either when a solution is found by one of the 12 solvers or when no new cube is available. `iLingeling` terminates when all threads are terminated.

Table 1 shows the results of our cube-and-conquer implementation on hard SAT 2009 application instances. The experiments are run on a two 6-core AMD Opteron 2435 machine from 2009. This machine, part of a cluster, has 32GB main memory and each job had a memory limit of 2.5GB per core. Additionally it shows the results of three alternative solvers, which we obtained from [16]:

- `Plingeling` 276, a multi-core portfolio solver using 12 cores [5].
- `ManySAT` 1.5, multi-core portfolio solver using 4 cores [11].
- `PT-Learn`, an iterative partitioning solver with learning running on a grid [16].

The portfolio solvers `Plingeling` and `ManySAT` were run on exactly the same hardware as our implementation, while `PT-Learn` was run on the M-grid environment consisting of nine clusters with CPU's from 2006 to 2009.

When we compare our approach with the two portfolio solvers `Plingeling` and `ManySAT`, then cube-and-conquer solves several more of these hard instances. Portfolio solvers are stronger on the three huge instances 9dlx* and sortnet*. A possible explanation could be that these instances must be "easy" relative to their size. Therefore, lookahead techniques can not really help the CDCL solvers.

The `PT-Learn` solver shows on most instances comparable performance to cube-and-conquer – although the latter is an order of magnitude faster on the eq.atree.braun* and gss* benchmarks. The comparison of both solvers in Table 1 however is biased towards `PT-Learn`: the experiments are run on similar hardware, but `PT-Learn` runs up to 60 jobs at the same time, while cube-and-conquer runs at most 12 jobs. `PT-Learn` suffers a bit from delays, while our solver runs on one machine. So, the presented results are suggesting that cube-and-conquer is actually the strongest solver on these hard application benchmarks.

Additional experiments suggest that our current implementation of cube-and-conquer is not optimal yet. For several instances, we observed improved real time using less than 12 cores. E.g., our 4 core cube-and-conquer experiments dated-5-19-u in 901 seconds. Also, total-10-17-u was solved in 2632 seconds using a single

Table 1. Results on benchmarks of the SAT 2009 application suite that were not solved during that competition. S denotes satisfiable, U denotes unsatisfiable. Phase I uses `lingeling` for preprocessing and `march_cc` for partitioning. The column I shows the total time (in seconds) of both tools on a single core. Phase II uses `iLingeling` to solve the cubes. Both the total time (sum of all threads) and the real time are listed. For the other solvers only the real time is provided which originate from [16]. — denotes that the timeout of 4 hours (14400 seconds) was reached.

Benchmark	S/U	number of cubes	I total	II total	II real	`Plingeling` real	`ManySAT` real	`PT-Learn` real
9dlx_vliw_at_b_iq8	U	84	284	—	—	3256	2750	—
9dlx_vliw_at_b_iq9	U	40	314	—	—	5164	3731	—
AProVE07-25	U	98320	168	81513	6858	—	—	9967
dated-5-19-u	U	28547	478	5601	2538	4465	18080	2522
eq.atree.braun.12	U	86583	115	3218	269	—	—	4691
eq.atree.braun.13	U	83079	106	17546	1466	—	—	9972
gss-24-s100	S	339398	1853	14265	1191	2930	6575	3492
gss-26-s100	S	493870	1517	66489	5547	18173	—	10347
gus-md5-14	U	78488	649	—	—	—	—	13890
ndhf_xits_09_UNS	U	39351	128	—	—	—	—	9583
rbcl_xits_09_UNK	U	61653	210	132788	16900	—	—	9819
rpoc_xits_09_UNS	U	36733	255	104552	20665	—	—	8635
sortnet-8-ipc5-h19	S	583	271	48147	4023	2700	79010	4304
total-10-17-u	U	19773	948	5927	5561	3672	10755	4447
total-5-15-u	U	7865	192	—	—	—	—	18670

core. This time is almost half the 12 core real time and faster than the other parallel SAT solvers. Notice that for both instances the real time is relatively close to the total time, indicating that solving a certain cube requires most of the computational cost.

8 Conclusions

We presented the novel SAT solving approach cube-and-conquer which is a very powerful method to solve hard CNF formulas. Our approach combines sophisticated lookahead decision heuristics with the efficiency of CDCL solvers. Results on hard van der Waerden benchmarks using our basic method show reduced computational costs up to a factor 20 compared to the fastest "pure" SAT solver. Moreover, using our cutoff mechanism, we were able to apply cube-and-conquer on hard application instances of the SAT competition. As a result, we outperform on most of these benchmarks the state-of-the-art parallel SAT solvers.

While this paper focused on the *offline* version of cube-and-conquer (i.e., a strict separation between both phases), we plan to implement an *online* version in the future. By integrating the method into a single solver, the phases can communicate with each other. For instance, the cube creation phase may select more effective decision literals if it knows which variables were frequently part of *AnalyzeFinal*. Also, if a cube appears hard to solve, the conquer phase can request additional assumptions.

References

1. Ahmed, T., Kullmann, O., Snevily, H.: On the van der Waerden numbers $w(2; 3, t)$. Tech. Rep. arXiv:1102.5433 [math.CO], arXiv (February 2011)

2. Bacchus, F.: Enhancing Davis Putnam with extended binary clause reasoning. In: AAAI 2002. pp. 613–619 (2002)
3. van Beek, P.: Backtracking search algorithms. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, chap. 4, pp. 85–134 (2006)
4. Biere, A.: Bounded model checking. In: Biere et al. [6], chap. 14, pp. 455–481
5. Biere, A.: Lingeling, Plingeling, Picosat and Precosat at SAT race 2010 (2010)
6. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, FAIA, vol. 185. IOS Press (February 2009)
7. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)
8. Dubois, O., Dequen, G.: A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In: Nebel, B. (ed.) IJCAI. pp. 248–253. Morgan Kaufmann (2001)
9. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electr. Notes Theor. Comput. Sci. 89(4), 543–560 (2003)
10. Hamadi, Y.: Conclusion to the special issue on parallel SAT solving. JSAT 6(4), 263 (2009)
11. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: a parallel SAT solver. JSAT 6(4), 245–262 (2009)
12. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. Artif. Intell. 14(3), 263–313 (1980)
13. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: IJCAI 1995. pp. 607–613 (1995)
14. Heule, M.J.H., van Maaren, H.: Look-Ahead Based SAT Solvers, chap. 5, pp. 155–184. Vol. 185 of Biere et al. [6] (February 2009)
15. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Partitioning SAT instances for distributed solving. In: LPAR-17. LNCS, vol. 6397, pp. 372–386 (2010)
16. Hyvärinen, A.E.J., Junttila, T., Niemelä, I.: Grid-based SAT solving with iterative partitioning and clause learning. In: CP 2011. LNCS, vol. 6876 (2011)
17. Kleine Büning, H., Kullmann, O.: Minimal Unsatisfiability and Autarkies, chap. 11, pp. 339–401. Vol. 185 of Biere et al. [6] (February 2009)
18. Kullmann, O.: Investigating the behaviour of a SAT solver on random formulas. Tech. Rep. CSR 23-2002, University of Wales Swansea, Computer Science Report Series (`http://www-compsci.swan.ac.uk/reports/2002.html`) (October 2002), 119 pages
19. Kullmann, O.: Fundaments of Branching Heuristics, chap. 7, pp. 205–244. Vol. 185 of Biere et al. [6] (February 2009)
20. Kullmann, O.: The `OKlibrary`: Introducing a "holistic" research platform for (generalised) SAT solving. Studies in Logic 2(1), 20–53 (2009)
21. Kullmann, O.: Green-Tao numbers and SAT. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 352–362. Springer (2010)
22. Kullmann, O.: Computing ordinary and palindromic van der Waerden numbers via collaboration between look-ahead and conflict-driven SAT solvers (February 2012), in preparation
23. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: IJCAI (1). pp. 366–371 (1997)
24. Marques-Silva, J.P., Lynce, I., Malik, S.: Conflict-Driven Clause Learning SAT Solvers, chap. 4, pp. 131–153. Vol. 185 of Biere et al. [6] (February 2009)
25. Mijnders, S., de Wilde, B., Heule, M.J.H.: Symbiosis of search and heuristics for random 3-SAT. In: Mitchell, D., Ternovska, E. (eds.) LaSh 2010 (2010)
26. Wieringa, S., Niemenmaa, M., Heljanko, K.: Tarmo: A framework for parallelized bounded model checking. In: Brim, L., van de Pol, J. (eds.) PDMC. EPTCS, vol. 14, pp. 62–76 (2009)
27. Zhang, H.: Combinatorial designs by SAT solvers. In: Biere et al. [6], chap. 17, pp. 533–568