CGSS in the 2022 MaxSAT Evaluation

Hannes Ihalainen, Jeremias Berg, Matti Järvisalo
HIIT, Department of Computer Science, University of Helsinki, Finland

I. Introduction

We overview the CGSS solver as it participated in the 2022 Evaluation. In short, CGSS implements the core-guided OLL algorithm for MaxSAT, extended with weight aware core extraction, structure sharing and selective addition of equivalences as described in [7] and [3]. Additionally, the solver makes use of stratification, hardening and the so-called core-exhaustion and intrinsic atmost1 techniques described in [6].

The solver is implemented in python, on top of PySAT [5] and the RC2 solver [6]. The authors would like to thank the developers of RC2 for their work. If you use CGSS in your research, we kindly ask you cite [7].

II. Preliminaries

We assume familiarity with conjunctive normal form (CNF) formulas and weighted partial maximum satisfiability (MaxSAT). Treating a CNF formula as a set of clauses, a MaxSAT instance $\mathcal{F}$ consists of two CNF formulas, the hard clauses F_h and the soft clauses F_s, as well a weight $w(C)$ associated with each $C \in F_s$. A solution to $\mathcal{F}$ is an assignment τ that satisfies F_h. The cost of a solution τ is the sum of weights of the soft clauses falsified by τ. An optimal solution is one with minimum cost over all solutions. An unsatisfiable core κ of $\mathcal{F}$ is a subset of soft clauses s.t. $F_h \wedge \kappa$ is unsatisfiable.

Without loss of generality we assume that each soft clause is unit, containing the negation of a variable. We say that a variable b is a *blocking variable* (of the instance $\mathcal{F}$) if $(\neg b) \in F_s$. As assigning a blocking variable to 1 corresponds to falsifying a soft clause, we will in the rest of the text view cores as sets of blocking variables and extend the weight function to blocking variables via $w(b) = w((\neg b))$.

III. Main Features

We overview the main features of CGSS. For a more detailed description, we refer the reader to [7].

OLL: When solving an instance $\mathcal{F}$ the (basic form of the) OLL algorithm [8], [1] iteratively extracts unsatisfiable cores of $\mathcal{F}$ using a SAT-solver, and then reformulates the instance in a way that allows exactly one of the blocking variables in the core to be assigned to 1 (corresponding to falsifying a soft clause) in subsequent iterations. This continues until the SAT solver reports the reformulated instance to be satisfiable and returns an optimal solution of the original instance.

Core reformulation. For reformulating a core $\kappa = \{b_1, \ldots, b_n\}$, the CGSS solver uses the so called *totalizer* [2] CNF encoding of cardinality constraints. The totalizer encoding can be viewed as a tree structure similar to those depicted in Figure 1. The leafs of the tree correspond to the variables in the core. An internal node that is the root of a subtree with the set $S \subset \kappa$ as leaves defines $|S| = m$ new variables $b^S_0, \ldots, b^S_{m-1}$ defined with clauses equivalent to $\left(\sum_{b \in S} b \geq k+1\right) \rightarrow b^S_k$. Specificallym the root of the full tree then defines a set $b^\kappa_0, \ldots b^\kappa_{n-1}$ that count the number of variables of κ set to true by assignments satisfying the totalizer.

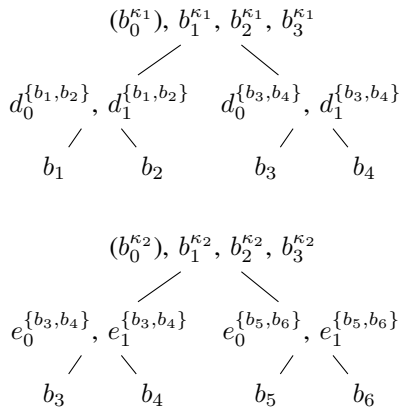

Fig. 1: The structure of totalizers built when relaxing cores $\kappa_1 = \{b_1, b_2, b_3, b_4\}$ (above) and $\kappa_2 = \{b_3, b_4, b_5, b_6\}$ (below).

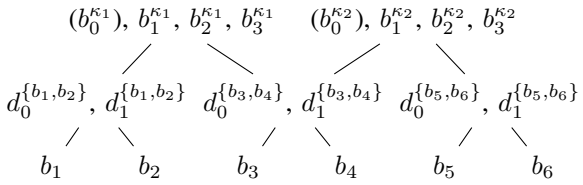

Fig. 2: The structure of totalizers when relaxing the cores κ_1 and κ_2 with structure sharing.

Weight aware core extraction (WCE) [4] is a heuristic designed to delay the core-reformulation steps performed by a solver implementing OLL for as long as possible. When extracting a new core κ, a solver using WCE will lower the weight of each variable $b \in \kappa$ by $w^\kappa = \min\{w(b) \mid b \in \kappa\}$ (this correspond to the so called clause cloning step). Afterwards, the core is stored and the SAT-solver asked for another core containing variables with positive weight. The stored cores are only reformulated when no new cores can

be found. Note that in the unweighted case (i.e. when the weight of each variable is 1) WCE is equivalent to the so called disjoint core technique that extracts a disjoint set of cores before reformulating.

Structure sharing attempts to reduce the number of equivalent variables introduced by the core reformulation steps by identifying subtrees that can be shared between several different totalizers. For a concrete example, figure 1 demonstrates two possible totalizer structures that can be built when relaxing the cores $\kappa_1 = \{b_1, b_2, b_3, b_4\}$ (above) and $\kappa_2 = \{b_3, b_4, b_5, b_6\}$ (below). Both of these structures include a subtree having b_3 and b_4 as leaves. The root of each of these subtrees define separate variables ($d_i^{\{b_3,b_4\}}$ for the top tree, $e_i^{\{b_3,b_4\}}$ for the bottom) that count the number of variables from the set $\{b_3, b_4\}$ set to true by assignments satisfying the totalizers. These variables will be equivalent in all solutions to the instance. Stated in another way, the two totalizer structures depicted in Figure 1 are equivalent to the smaller single structure depicted in Figure 2.

When relaxing a set of cores obtained via WCE, the CGSS solver uses a heuristic set-covering algorithm for identifying maximal sets of literals shared by as many cores as possible and building totalizers that share these sets as subtrees.

Selective addition of equivalences seeks to identify counting variables of the (shared) totalizers to which it would be useful to add equivalence constraints that facilitate more propagation.

More precisely, consider a count variable b_k^S corresponding to an internal note of a tree that is the root of a subtree with the variables in S as leaves. For correctness of the OLL algorithm, it suffices to add clauses equivalent to the implication $\left(\sum_{b\in S} b \geq k+1\right) \rightarrow b_k^S$. While adding the other direction of the implication (i.e. $b_k^S \rightarrow \left(\sum_{b\in S} b \geq k+1\right)$) could allow the SAT solver do perform more propagation, the large number of clauses required in order to do so for every internal node might instead result in overall deterioration of performance.

In order to balance the potential benefits and overhead (in the form of extra clauses) of adding both sides of the equivalence defining the variables in a totalizer, CGSS attempts to identify nodes for which the equivalence constraints are more likely to lead to further propagation. More specifically, for each leaf and root of a shared subtree, two values are computed: (a) the number of additional clauses needed for defining the full equivalence and (b) how many decisions need to performed by the SAT solver in before the additional constraints result in propagation. If both of these values are below some user provided threshold the equivalence constraints for that particular node are added.

Bounds: The use of WCE and stratification leads to CGSS obtaining intermediate solutions to the instance during search. The cost of any such solution is an upper bound on the optimal cost of the instance. CGSS stores these solutions, effectively turning it into an any-time MaxSAT solver. At the same time, the cores extracted during search can be used to compute a lower bound on the optimal cost by summing the minimum weight of variables appearing in each extracted core. The use of both an upper and a lower bound can allow the solver to terminate as soon as the bounds match, sometimes even before reformulating all of the extracted cores.

IV. Compilation and Usage

CGSS is implemented on top of RC2 in the PySAT framework [5], [6] in a mixture of Python and C++ and can be found at https://bitbucket.org/coreo-group/cgss/src/master/ or the Evaluation website. Installing and running CGSS resembles installing and running RC2, please follow the readme of the repository for more details. The readme also details the command line parameters, the evaluation version of CGSS is invoked by running:

```
python rc2.py -lamWnP [instance.wcnf.gz]
```

from the examples subfolder of the repository base folder.

References

[1] B. Andres, B. Kaufmann, O. Matheis, and T. Schaub, "Unsatisfiability-based optimization in clasp," in *Proc. ICLP Technical Communications*, ser. LIPIcs, vol. 17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012, pp. 211–221.

[2] O. Bailleux and Y. Boufkhad, "Efficient CNF encoding of boolean cardinality constraints," in *Proc. CP*, ser. Lecture Notes in Computer Science, vol. 2833. Springer, 2003, pp. 108–122.

[3] J. Berg and M. Järvisalo, "Weight-aware core extraction in SAT-based MaxSAT solving," in *Proc. CP*, ser. Lecture Notes in Computer Science, 2017, to appear.

[4] J. Berg and M. Järvisalo, "Weight-aware core extraction in SAT-based MaxSAT solving," in *Proc. CP*, ser. Lecture Notes in Computer Science, vol. 10416. Springer, 2017, pp. 652–670.

[5] A. Ignatiev, A. Morgado, and J. Marques-Silva, "PySAT: A Python toolkit for prototyping with SAT oracles," in *SAT*, 2018, pp. 428–437. [Online]. Available: https://doi.org/10.1007/978-3-319-94144-8_26

[6] ——, "RC2: An efficient MaxSAT solver," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 11, no. 1, pp. 53–64, 2019.

[7] H. Ihalainen, J. Berg, and M. Järvisalo, "Refined core relaxation for core-guided maxsat solving," in *CP*, ser. LIPIcs, vol. 210. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 28:1–28:19.

[8] A. Morgado, C. Dodaro, and J. Marques-Silva, "Core-guided MaxSAT with soft cardinality constraints," in *Proc. CP*, ser. Lecture Notes in Computer Science, vol. 8656. Springer, 2014, pp. 564–573.