Reflections on "Incremental Cardinality Constraints for MaxSAT"

Ruben Martins[1], Saurabh Joshi[2], Vasco Manquinho[3], and Inês Lynce[3]

[1] Department of Computer Science, Carnegie Mellon University, United States
rubenm@andrew.cmu.edu
[2] Department of Computer Science and Engineering, IIT Hyderabad, India
sbjoshi@iith.ac.in
[3] INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Portugal
{vasco.manquinho,ines.lynce}@tecnico.ulisboa.pt

Abstract. To celebrate the first 25 years of the International Conference on Principles and Practice of Constraint Programming (CP) the editors invited the authors of the most cited paper of each year to write a commentary on their paper. This report describes our reflections on the CP 2014 paper "Incremental Cardinality Constraints for MaxSAT" and its impact on the Maximum Satisfiability community and beyond.

1 Introduction

During the first decade of the century, advances in Propositional Satisfiability (SAT) algorithms resulted in the proposal of solving Maximum Satisfiability (MaxSAT) using iterative calls to a SAT solver. Previously, most MaxSAT solvers used a branch and bound approach enhanced with lower bounding procedures and MaxSAT inference rules [16].

However, the ability of SAT solvers to provide an unsatisfiable subformula whenever an unsatisfiable call is made enabled the proposal of new algorithms for MaxSAT [8,10,18,23]. These new algorithms were orders of magnitude faster when solving several sets of real-world problem instances.

Meanwhile, the incremental usage of SAT solvers [4, 7, 26, 32] had provided significant gains in some domains where SAT algorithms were iteratively being used. However, incrementality had not yet been fully exploited in MaxSAT solving, except for algorithms using a Sat-Unsat approach.

In Unsat-Sat algorithms for MaxSAT, at each iteration, a new instance of the SAT solver was created and the formula was rebuilt from scratch. As a result, almost all the knowledge from the previous iteration was lost. The main reason for rebuilding the SAT solver is that some constraints from previous iterations are no longer valid. Moreover, removing these constraints would not be enough, as learned constraints would also have to be removed. This cleaning process is not easy to perform efficiently.

Algorithm 1 presents the MSU3 algorithm for partial MaxSAT. Observe that, at each iteration, the cardinality constraint in line 3 from one iteration is not

Algorithm 1: MSU3 Algorithm for partial MaxSAT

Input: $\varphi = \varphi_h \cup \varphi_s$
Output: satisfying assignment to φ

```
1  (φ_W, V_R, λ) ← (φ, ∅, 0)
2  while true do
3      (st, ν, φ_C) ← SAT(φ_W ∪ {CNF(Σ_{r∈V_R} r ≤ λ)})
4      if st = SAT then
5          return ν                          // satisfying assignment to φ
6      foreach c ∈ (φ_C ∩ φ_s) do
7          V_R ← V_R ∪ {r}                          // r is a new variable
8          c_R ← c ∨ r                 // clause c was not previously relaxed
9          φ_W ← (φ_W \ {c}) ∪ {c_R}
10     λ ← λ + 1
```

valid to the next iteration. The set of literals might have changed and the lower bound λ is incremented.

In our paper [21], we proposed three techniques that allow Unsat-Sat MaxSAT algorithms (such as Algorithm 1) to use the same SAT solver between iterations. In particular, we propose the usage of (i) incremental blocking, (ii) incremental weakening, and (iii) iterative encoding of cardinality constraints. Note that rebuilding the SAT solver, includes having to rebuild the CNF encoding of cardinality constraints. However, the iterative encoding of cardinality constraints allows to gradually encode the cardinality constraint into CNF, maintaining the auxiliary variables already used in the previous iteration. Moreover, as a result of using incrementality, the internal state of the SAT solver is maintained, as well as learned clauses discovered in the previous iterations. Experimental results from our paper and subsequent MaxSAT evaluations clearly show the effectiveness of the proposed techniques. We also note that current state of the art MaxSAT solvers extensively use the incremental techniques originally proposed in the paper.

The remainder of the paper is organized as follows. Section 2 describes the problem that motivated this approach and its application to MaxSAT. Next, we study and discuss the effectiveness of these incremental techniques. Finally, in section 4, the impact of the proposed ideas on subsequent MaxSAT technology is revised.

2 Origin

The motivation for incremental cardinality constraints [21] comes from a problem of a completely different domain of automated program repair for weak memory models.

Modern multicore CPUs implement optimizations such as *store buffers* and *invalidate queues*. These features result in weaker memory consistency guarantees than sequential consistency (SC) [14]. Though such hardware optimizations

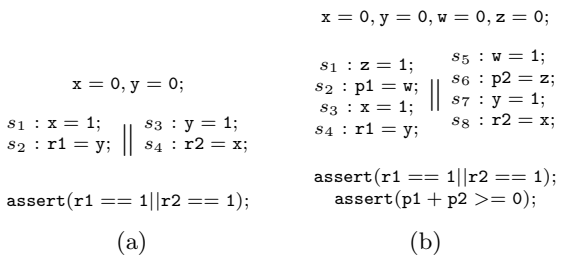


Fig. 1: (a) Reordering in TSO. (b) A program with *innocent* and *culprit* reorderings

offer better performance, the weaker consistency has the drawback of intricate and subtle semantics, thus making it harder for programmers to anticipate how their program might behave when running on such architectures. For example, a pair of statements can appear to have been executed out of the program order.

Consider the program given in Fig. 1a. Here, `x` and `y` are shared variables whereas `r1` and `r2` are thread-local variables or registers. Statements s_1 and s_3 perform write operations. Because of store buffering, these write operations may not be reflected immediately in the memory. Next, both threads may proceed to perform the read operations s_2 and s_4. Since the write operations might still not have hit the memory, stale values for `x` and `y` may be read in `r2` and `r1`, respectively. This will cause the assertion to fail. Such behavior is possible with architectures that implement *Total Store Order (TSO)*, which allows write-read reordering. Note that on an hypothetical architecture that guarantees sequential consistency, this would never happen. However, due to store buffering, a global observer might witness that the statements are executed in the order (s_2, s_4, s_1, s_3), which results in the assertion failure. We say that (s_1, s_2) and (s_3, s_4) have been reordered.

Since these reorderings are non-deterministic, architectures usually provide *fence* (or *memory barrier*) instructions to allow a programmer to restrict such reorderings. In this case, a fence between (s_1, s_2) and (s_3, s_4) is needed to ensure that the assertion does not fail.

We distinguish approaches that aim to restore sequential consistency (SC) and approaches that aim to ensure that a user-provided assertion holds. Since every fence incurs in a performance penalty, it is desirable to keep the number of fences to a minimum.

Consider the example given in Fig. 1b. Here, `x,y,z,w` are shared variables initialized to 0. All other variables are thread-local. A processor that implements total store ordering (TSO) permits a read of a global variable to precede a write operation to a different global variable when there are no dependencies between the two statements. Note that if (s_3, s_4) or (s_7, s_8) is reordered, the assertion will be violated. We shall call such pairs of statements *culprit pairs*. By contrast, the pairs (s_1, s_2) and (s_5, s_6) do not lead to an assertion violation irrespective of the

order in which their statements execute. We shall call such pairs of statements *innocent pairs*. A tool that restores SC would insert four fences, one for each pair mentioned earlier. However, only two fences (between s_3, s_4 and s_7, s_8) are necessary to avoid the assertion violation.

There are approaches which look at a counterexample provided by a model checker for such programs and tries to avoid some minimal set of reorderings in order to avoid assertion violations. For most of these approaches, a large number of innocent pairs is a bane and would result in a lot of expensive queries to the underlying model checker. Reorder Bounded Model Checking (ROBMC) [12] addresses this issue by exploring only those behaviors where the number of behaviors is bounded by some parameter k. For every pair of statements s_i, s_j which can potentially get reordered, a new variable a_{ij} is introduced such that a reordering is allowed only if a_{ij} is true. Then, a cardinality constraint $a_{ij} \leq k$ enforces the bound on reordering. In [12], it is shown that ROBMC results in much lesser number of queries to the model checker, thus making it much more efficient. Initially, $k = 1$ so that all the counterexamples with only one culprit pair can be eliminated.

In general, there could be assertion violations, which will be triggered only if more than 1 culprit pairs are reordered. Therefore, for soundness, after making the program safe for $k = 1$, the bound k must be increased to a higher value to check if there are counter examples for this higher bound. Note that to check for assertion violation at a higher bound k', only constraint that is required to change is from $(\sum a_{ij} \leq k)$ to $(\sum a_{ij} \leq k')$. The rest of the formula which encodes all possible program behaviors remains the same. We move to a higher bound k' only when at a lower bound k the program is declared safe, which is usually indicated by the corresponding formula representing program behaviors being unsatisfiable. This is where, *incremental cardinality constraints* play a crucial role by allowing us to increase the upper bound of the cardinality constraints when approaching from an unsatisfiable region.

3 Looking Back

The experimental evaluation conducted in the CP'14 paper [21] showed a clear improvement when using incremental cardinality encodings. Not only the number of solved instances increased but, on average, the incremental version was 3.6× faster than the corresponding non-incremental version. However, Ansotegui et al. [2] stated that the improvement may not be due to incrementality but rather to the way the cardinality encoding is built. Note that, in the incremental approach, the cardinality encoding is built taking into consideration the structure of the unsatisfiable subformulas (cores) found by the algorithm.

Looking back to our original experiments, we did not fully explore the reason behind the improvements. In this section, we perform additional experiments that show that our original insights were correct and that the *primary reason for the performance gain is incrementality* and not the way the cardinality encoding is built. However, the way the cardinality encoding is built may also have a small

Version	Incremental	Encoding Structure	Reuse Cores
ni	No	Complete Rebuild	No
ni-c	No	Complete Rebuild	From incremental version
ni-s	No	Structure of the Unsat cores	No
ni-s-c	No	Structure of the Unsat cores	From incremental version
i	Yes	Structure of the Unsat cores	No

Table 1: Different versions of MSU3

benefit for the performance of the solver and should be the target of further study.

We restrict ourselves to the non-incremental and incremental versions of the MSU3 algorithm since this was the most efficient algorithm presented in our CP'14 paper [21]. To clarify the nature of the improvement, we ran five variants described in Tab. 1 and implemented on top of the `open-wbo` framework [22]. Column "Incremental" indicates if the SAT solver is reused between iterations, column "Encoding Structure" indicates if the cardinality constraint is rebuilt from scratch in each iteration and column "Reuse Cores" indicates if the cardinality constraint is built using the Unsat cores found in the incremental version.

More specifically: Version (ni) corresponds to the classic implementation of the MSU3 algorithm with no incrementality where the SAT solver is not reused and the cardinality constraint is rebuilt in each iteration; Version (i) corresponds to the fully incremental version proposed in the paper; Version (ni-c) does not reuses the SAT solver between iterations, but the cardinality constraint is non-incrementally rebuilt using the unsatisfiable cores from the incremental version (i); Version (ni-s) also does not reuses the SAT solver, but the cardinality constraint is built according to the structure of the unsatisfiable cores found by the algorithm (i.e. the structure of the cardinality encoding follows the structure of the cores found); Version (ni-s-c) also does not reuses the SAT solver, but the cardinality constraint is built according to the structure of the unsatisfiable cores found by algorithm (i). The goal of testing all these variants is to clarify if the improvement is coming due to the way the encoding is built or due to the incrementality of the approach. All experiments were run on StarExec [33] using Intel Xeon E5-2609 processors (2.40GHz) with a memory limit of 32GB and time limit of 1,800 seconds. We used the same benchmarks as in our CP'14 paper, which corresponds to the 627 partial industrial MaxSAT instances from the MaxSAT Evaluation 2013.

Fig. 2 shows a cactus plot with the running times of the different non-incremental and incremental versions. We can see that the incremental version clearly outperforms the non-incremental versions and it is the main reason for the performance of the incremental MSU3 algorithm. On the other hand, all non-incremental versions solve a similar number of instances.

Fig. 3 shows scatter plots that compare the non-incremental versions with and without using the structure of the cores when building the cardinality encod-

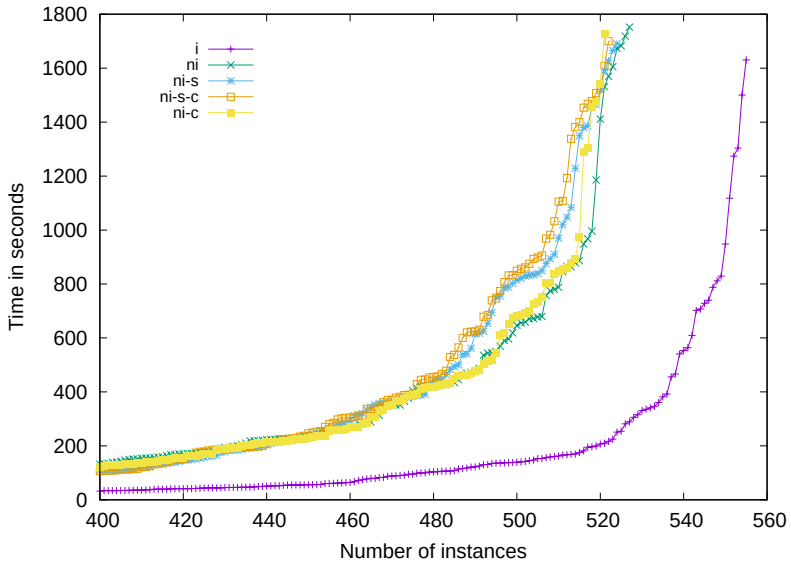
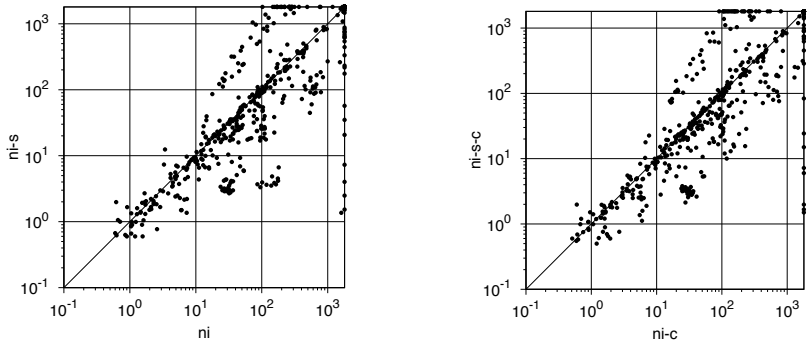


Fig. 2: Running times of the different non-incremental and incremental versions



(a) With and without structure with different cores

(b) With and without structure with the same cores

Fig. 3: Impact of the structure of the encoding on the performance of the solver

ing. Each point in the plot corresponds to a problem instance, where the x-axis corresponds to the run time required by the non-incremental versions that do not use the structure and the y-axis corresponds to the run time required by the non-incremental versions that use core structure when building the cardinality encoding. The (ni) and (ni-s) versions solve 527 and 524 instances, respectively. However, when looking at the performance of the algorithm, we can see that the majority of the instances are solved faster when using the structure of the

cores to build the encoding. Nevertheless, there are several outliers where this behavior is not observed. The (ni-c) and (ni-s-c) versions solve 521 and 522 instances, respectively. These versions were run on the 555 instances solved by the incremental version and reuse the same cores found by the incremental version. Therefore, the only difference between (ni-c) and (ni-s-c) is on how the encoding is built. In this case, we can also observe a similar scenario as before where (ni-s-c) is faster on the majority of the instances. Even though the structure of the cores when building the cardinality encoding has a minor effect on the number of solved instances, it does seem to improve the running time of the solver.

Threats to validity. Even though our results support our original insights that incrementality is the main reason for the observed improvement, some factors may have led us to wrong conclusions. In particular, maybe if we used a different set of benchmarks, then the results would be different. However, we did a preliminary study with different benchmarks [29] and obtained similar results. Another threat to our conclusions is the fact that we only tested our approach using the MSU3 algorithm. It may be that the structure of the cores when building the cardinality encoding is more important for other MaxSAT algorithms (e.g., WPM3 [2], RC2 [11]) than for MSU3. As future work, we plan to investigate this impact on different algorithms and see if our conclusions still hold.

4 Impact on MaxSAT and beyond

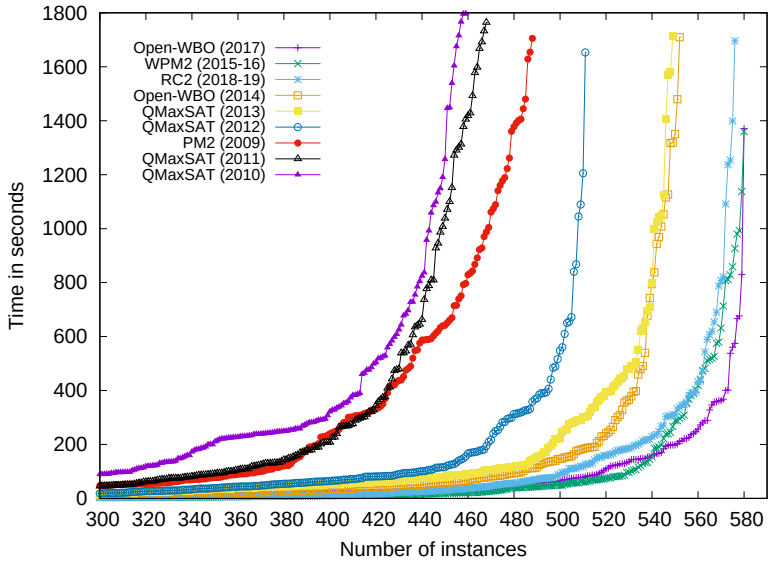


Fig. 4: Evolution of MaxSAT solvers for partial MaxSAT

Fig. 4 shows a cactus plot with the evolution of MaxSAT solvers for partial MaxSAT since 2009 until 2019. We selected the non-portfolio solvers with most instances solved in the MaxSAT Evaluation of each year. All solvers were run on StarExec with a time limit of 1,800 seconds and a memory limit of 32GB on the 627 partial industrial benchmarks from MaxSAT Evaluation 2013. The solvers in Fig. 4 are: PM2 [1] (2009), QMaxSAT [13, 28] (2010-2013), Open-WBO [21, 27] (2014, 2017), WPM3 [2] (2015-16), and RC2 [11] (2018-19). We can see a significant improvement of MaxSAT solvers in the last decade.

The techniques proposed in the CP'14 paper were implemented on top of the `open-wbo` framework [22] and won the partial industrial MaxSAT Evaluation 2014 category. This framework is open-source where any minisat-like SAT solver can be pluged-in. Since the proposed techniques became available in an open-source solver, it became easier to be adopted by the research community. As a result, other state of the art MaxSAT solvers have also adopted many of the techniques, in particular the iterative encoding of cardinality constraints [24,25]. In 2015, it was shown that the iterative encoding could also be used for encoding pseudo-Boolean constraints into SAT [20]. Nowadays, top performing Unsat-Sat MaxSAT algorithms use at least one of the three techniques proposed in the paper either for cardinality of for pseudo-Boolean constraints [3, 11, 27]. Moreover, a new generation of incomplete solvers for MaxSAT also take advantage of algorithms using these techniques [6, 9, 15].

The success of solving MaxSAT using incremental encodings of constraints has provided a boost in performance when solving problem instances in several domains, such as program repair [12], model-based diagnosis [17, 19] or timetabling [5]. Moreover, these ideas have also provided inspiration for similar incremental approaches in solving other problems such as Markov logic networks [31] or software analysis [30].

Acknowledgement. We would like to thank Naveen Pai for a preliminary study on the impact of cores when building incremental cardinality constraints for Independent Studies in Computer Science at Carnegie Mellon University [29].

References

1. Ansótegui, C., Bonet, M.L., Levy, J.: On solving maxsat through SAT. In: International Conference of the Catalan Association for Artificial Intelligence. pp. 284–292. IOS Press (2009)
2. Ansótegui, C., Gabàs, J.: WPM3: an (in)complete algorithm for weighted partial maxsat. Artif. Intell. 250, 37–57 (2017)
3. Ansótegui, C., Gabàs, J., Levy, J.: Exploiting subproblem optimization in sat-based maxsat algorithms. Journal of Heuristics 22(1), 1–53 (2016)
4. Audemard, G., Lagniez, J.M., Simon, L.: Improving Glucose for Incremental SAT Solving with Assumptions: Application to MUS Extraction. In: Järvisalo, M., Gelder, A.V. (eds.) International Conference on Theory and Applications of Satisfiability Testing. pp. 309–317. Springer (2013)
5. Demirović, E., Musliu, N.: Maxsat-based large neighborhood search for high school timetabling. Computers & Operations Research 78, 172–180 (2017)

6. Demirovic, E., Stuckey, P.J.: Linsbps. MaxSAT Evaluation 2018: Solver and Benchmark Descriptions, volume B-2018-2 of Department of Computer Science Series of Publications B, University of Helsinki pp. 8–9 (2018)
7. Eén, N., Sörensson, N.: Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Computer Science 89(4), 543–560 (2003)
8. Fu, Z., Malik, S.: On Solving the Partial MAX-SAT Problem. In: Biere, A., Gomes, C.P. (eds.) International Conference on Theory and Applications of Satisfiability Testing. LNCS, vol. 4121, pp. 252–265. Springer (2006)
9. Guerreiro, A.P., Terra-Neves, M., Lynce, I., Figueira, J.R., Manquinho, V.: Constraint-based techniques in stochastic local search maxsat solving. In: Schiex, T., de Givry, S. (eds.) International Conference on Principles and Practice of Constraint Programming. pp. 232–250. Springer (2019)
10. Heras, F., Morgado, A., Marques-Silva, J.: Core-guided binary search algorithms for maximum satisfiability. In: Burgard, W., Roth, D. (eds.) AAAI Conference on Artificial Intelligence. AAAI Press (2011)
11. Ignatiev, A., Morgado, A., Marques-Silva, J.: Pysat: a python toolkit for prototyping with sat oracles. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) International Conference on Theory and Applications of Satisfiability Testing. pp. 428–437. Springer (2018)
12. Joshi, S., Kroening, D.: Property-driven fence insertion using reorder bounded model checking. In: Bjørner, N., de Boer, F.S. (eds.) International Symposium on Formal Methods. pp. 291–307. Springer (2015)
13. Koshimura, M., Zhang, T., Fujita, H., Hasegawa, R.: QMaxSAT: A Partial Max-SAT Solver. Journal on Satisfiability, Boolean Modeling and Computation 8, 95–100 (2012)
14. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers 28, 690–691 (09 1979)
15. Lei, Z., Cai, S.: Satlike-c. MaxSAT Evaluation 2018: Solver and Benchmark Descriptions, volume B-2018-2 of Department of Computer Science Series of Publications B, University of Helsinki pp. 24–25 (2018)
16. Li, C.M., Manyà, F.: MaxSAT, Hard and Soft Constraints. In: Handbook of Satisfiability, pp. 613–631. IOS Press (2009)
17. Liu, M., Ouyang, D., Cai, S., Zhang, L.: Efficient zonal diagnosis with maximum satisfiability. Science China Information Sciences 61(11), 112101 (2018)
18. Manquinho, V., Marques-Silva, J., Planes, J.: Algorithms for Weighted Boolean Optimization. In: Kullmann, O. (ed.) International Conference on Theory and Applications of Satisfiability Testing. pp. 495–508 (2009)
19. Marques-Silva, J., Janota, M., Ignatiev, A., Morgado, A.: Efficient model based diagnosis with maximum satisfiability. In: IJCAI. vol. 15, pp. 1966–1972 (2015)
20. Martins, R., Joshi, S., Manquinho, V., Lynce, I.: On using incremental encodings in unsatisfiability-based maxsat solving. Journal on Satisfiability, Boolean Modeling and Computation 9, 59–81 (2015)
21. Martins, R., Joshi, S., Manquinho, V.M., Lynce, I.: Incremental cardinality constraints for maxsat. In: O'Sullivan, B. (ed.) Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8656, pp. 531–548. Springer (2014)
22. Martins, R., Manquinho, V., Lynce, I.: Open-WBO: a Modular MaxSAT Solver. In: Sinz, C., Egly, U. (eds.) International Conference on Theory and Applications of Satisfiability Testing. LNCS, vol. 8561, pp. 438–445. Springer (2014)

23. Morgado, A., Heras, F., Liffiton, M., Planes, J., Marques-Silva, J.: Iterative and core-guided MaxSAT solving: A survey and assessment. Constraints 18(4), 478–534 (2013)
24. Morgado, A., Ignatiev, A., Marques-Silva, J.: Mscg: Robust core-guided maxsat solving. JSAT 9, 129–134 (2014)
25. Nadel, A.: Solving maxsat with bit-vector optimization. In: Beyersdorff, O., Wintersteiger, C.M. (eds.) International Conference on Theory and Applications of Satisfiability Testing. pp. 54–72. Springer (2018)
26. Nadel, A., Ryvchin, V.: Efficient SAT Solving under Assumptions. In: Cimatti, A., Sebastiani, R. (eds.) International Conference on Theory and Applications of Satisfiability Testing. pp. 242–255 (2012)
27. Neves, M., Martins, R., Janota, M., Lynce, I., Manquinho, V.: Exploiting resolution-based representations for maxsat solving. In: Heule, M., Weaver, S. (eds.) Theory and Applications of Satisfiability Testing – SAT 2015. pp. 272–286. Springer International Publishing, Cham (2015)
28. Ogawa, T., Liu, Y., Hasegawa, R., Koshimura, M., Fujita, H.: Modulo Based CNF Encoding of Cardinality Constraints and Its Application to MaxSAT Solvers. In: International Conference on Tools with Artificial Intelligence. pp. 9 – 17. IEEE (2013)
29. Pai, N., Martins, R.: Benefits of Iterative Encoding in MaxSAT. Poster presented at Independent Studies in CS at Carnegie Mellon University. Fall 2018.
30. Si, X., Zhang, X., Grigore, R., Naik, M.: Maximum satisfiability in software analysis: Applications and techniques. In: Majumdar, R., Kuncak, V. (eds.) International Conference on Computer Aided Verification. pp. 68–94. Springer (2017)
31. Si, X., Zhang, X., Manquinho, V., Janota, M., Ignatiev, A., Naik, M.: On incremental core-guided maxsat solving. In: Rueher, M. (ed.) International Conference on Principles and Practice of Constraint Programming. pp. 473–482. Springer (2016)
32. Strichman, O.: Pruning Techniques for the SAT-Based Bounded Model Checking Problem. In: Margaria, T., Melham, T.F. (eds.) Correct Hardware Design and Verification Methods. LNCS, vol. 2144, pp. 58–70. Springer (2001)
33. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A Cross-Community Infrastructure for Logic Solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) International Joint Conference on Automated Reasoning. pp. 367–373. Springer (2014)