HyperSAT

Domagoj Babić and Alan J. Hu
Department of Computer Science, University of British Columbia
{babic,ajh}@cs.ubc.ca

1 Introduction

HyperSAT is a DPLL solver with a new search space pruning technique based on the theory of B-Cubing [3, 2, 1], powerful preprocessing, and standard 1-UIP learning [10]. The core of the solver is based on a simple watched literal scheme as implemented in LIMMAT [4], with some minor optimizations and extended to support equivalence clauses.

The decision heuristic is fairly complex. The solver keeps counters for each literal. Counters are incremented by certain values, depending on the properties of the literal and the conflict, and frequently scaled (every 300 conflicts). The priority of each variable is computed according to the following heuristic function:

$$\begin{aligned} \mathsf{pri}(v) &= 8\,\mathsf{abs}(\mathsf{pos}(v) - \mathsf{neg}(v)) \\ &\quad +(\mathsf{pos}(v)+1)(\mathsf{neg}(v)+1) \end{aligned}$$

Literal counters are accessed through the $\mathsf{pos}()$ and $\mathsf{neg}()$ functions. The first term rewards variables that more frequently appear in conflicts with only one phase. Such asymmetry can lead to a quick conflict detection. Variables that are present in many clauses often represent a good choice for a decision, so the second term greedily increases the priority of such variables.

Once the variable has been selected for a case split (decision), the solver picks the phase that corresponds to the literal with a smaller counter value. This tends to increase the number of implications and the probability of discovering a conflict.

2 B-Cubing

Intuitively, a SAT solver works by case-splitting: it decides to assign a value to a variable, checks to see if the resulting subproblem is satisfiable, and if not, tries the other assignment. This case-splitting naturally corresponds to a search tree that considers all possible assignments to the variables (Fig. 1).

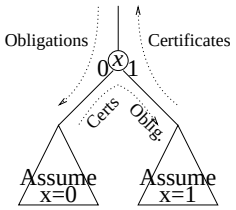

Figure 1: In B-Cubing, information from the left subtree is used in the right subtree.

For B-Cubing, we develop a framework that allows new kinds of pruning information. In the general framework, a node in the search tree inherits from its parent the obligation to prove a part of the search space unsatisfiable. When it is done, it will return to its parent some sort of certificate that a (possibly larger) part of the search space was indeed unsatisfiable. New pruning opportunities arise at the node, because the certificates returned from exploring one branch can be combined with the obligations inherited from above to be used in pruning the other branch. An advantage of keeping some pruning information local to a node is that the solver can perform pruning that is only applicable locally (or, alternatively, that the solver need not store the context information to determine exactly when the pruning information is usable, since the context is implicit in the search tree). Furthermore, the pruning information can be discarded when it is no longer needed.

B-Cubing requires different backtracking mechanism (as explained in [3]). Instead of indirectly flipping 1-UIP literal through assertion clauses, as in ZChaff [7], the solver backtracks to the last decision that was involved in at least one conflict and contains a non-conflicting list of scheduled implied literals. Scheduled implied literals are literals that would

This work was supported in part by a research grant from the Natural Science and Engineering Research Council of Canada and a graduate fellowship from the University of British Columbia.

have been implied at the corresponding decision level if the solver behaved like ZChaff.

3 Preprocessing

The solver can also handle equivalence clauses of the form $(a_1 \Leftrightarrow a_2 \Leftrightarrow \cdots \Leftrightarrow a_n)$. Equivalence clauses are discovered during the preprocessing phase. The preprocessing loop repeats application of the pure literal rule [5], detection and propagation of binary equivalences, propagation of unit literals [9], elimination of duplicate clauses, limited ground resolution, equivalence reasoning, and detection of tautologies as long as there are any changes to the original formula. The procedure is guaranteed to terminate as each pass reduces the size of the original formula.

Binary equivalence propagation is based on a graph algorithm that records all binary equivalences and detects inconsistencies in the theory. All binary equivalences detected in one pass are represented as a graph. Two types of edges represent positive and negative equivalences. The algorithm then finds a representative of each equivalence class and replaces all other variables in the class. Duplicate clauses that are found in the original formula or generated in the simplification loop are eliminated after each pass. Limited ground resolution infers clause (A) from two clauses of type $(A \vee a)$ and $(A \vee \bar{a})$. Equivalence reasoning and tautology clauses are handled in a similar way as in March [8, 6].

4 Learning

Learned clauses correspond to 1-UIP cuts in the implication graph [10]. When the clause cache becomes full, about half of the clauses are deleted, and the size of the cache is increased by a small monotonically decreasing percentage. The maximum size of the cache has been set to 2^{19} clauses. Clauses in the cache are sorted by participation in conflicts and length. The longer and less used a clause is, the more likely it is to be deleted.

5 Restarts

The basic version of HyperSAT is not randomized and never restarts, but randomized restarts were added in HyperSAT-RR version as an attempt to increase the robustness of our solver. In order to prevent unneeded restarts, the solver tracks its own progress. There are two progress metrics, one corresponds to the first decision level at which some decision variable has been explored with both phases and the second is a signature that roughly corresponds to the size of the search space that has been searched. If there is progress being made according to the first metric, the solver will not restart, otherwise it will inspect the second metric. If there is no progress either, it will restart. Depending on how much search space has been explored, the solver will occasionally restart even if there is some progress according to the second metric. The closer to the solution the solver thinks it is, the smaller the probability of restarting.

References

[1] D. Babić, J. Bingham, and A. J. Hu. Better SAT-Solving Using B-Cubing: Theory and Experiments. 2005. Submitted to SAT 2005.

[2] D. Babić, J. Bingham, and A. J. Hu. Efficient SAT Solving: Beyond Supercubes. 2005. Submitted to DAC 2005.

[3] D. Babić and A. J. Hu. Integration of Supercubing and Learning in a SAT Solver. In *Asia South Pacific Design Automation Conference*, pages 438–444. ACM/IEEE, 2005.

[4] A. Biere. The Evolution from LIMMAT to NANOSAT. Technical Report 444, Dept. of Computer Science, ETH Zürich, 2004.

[5] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM*, 7(3):201–215, 1960.

[6] M. Heule and H. van Maaren. Aligning cnf- and equivalence-reasoning. In *SAT*, pages 174–181, 2004.

[7] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, pages 530–535. ACM Press, 2001.

[8] J. P. Warners and H. van Maaren. Recognition of tractable satisfiability problems through balanced polynomial representations. In *Proceedings of the 5th Twente workshop on on Graphs and combinatorial optimization*, pages 229–244. Elsevier Science Publishers B. V., 2000.

[9] H. Zhang and M. E. Stickel. An efficient Algorithm for Unit Propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.

[10] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the International Conference on Computer-aided Design*, pages 279–285. IEEE Press, 2001.