Minimizing Learned Clauses

Niklas Sörensson[1] and Armin Biere[2]

[1] Chalmers University of Technology, Göteborg, Sweden.
[2] Johannes Kepler University, Linz, Austria.

Abstract. Minimizing learned clauses is an effective technique to reduce memory usage and also speed up solving time. It has been implemented in MiniSat since 2005 and is now adopted by most modern SAT solvers in academia, even though it has not been described in the literature properly yet. With this paper we intend to close this gap and also provide a thorough experimental analysis of it's effectiveness for the first time.

Introduction

Learning clauses [9] is an essential part in modern SAT solvers [6, 8, 10]. Learning is used for forward pruning search space and in combination with a conflict-driven assignment loop [8, 9] also allows to skip redundant decisions during backtracking. The 1-UIP learning scheme [9] is considered to work best [4, 18]. It is possible to increase the efficiency of the 1-UIP scheme, by removing additional literals from learned clauses. This can either be done *locally* [2] or *recursively* [11, 16]. The latter was first implemented in MiniSAT in 2005 but has not been properly described in the literature yet.

Learning is usually explained with the help of implication graphs [8], which has assigned variables as nodes connected through *antecedents* [8]. The analysis starts with a clause in which all literals are assigned to false, formally the antecedent of the conflict κ [8], and resolves in antecedents from its *implied* [8] variables recursively. Several termination conditions are possible [18]. In the simplest scheme the process continues until only decisions are left. The standard algorithm [9] makes sure that learned clauses derived this way contain exactly one literal from the current decisions level. This is usually referred to as that the learned clause must be *asserting*.

If antecedents are resolved in reverse assignment order, the first derived asserting clause, is called the *first unique implication point* (1-UIP) clause [8, 9]. Learning 1-UIP clauses is considered to be the best learning scheme [4, 18]. In extensions [4, 7, 9, 13] additional learned clauses are not asserting, and are added as complement to the 1-UIP clause. However, a proper subset of the 1-UIP clause will necessarily be more successful at pruning future search. This is our original motivation for the algorithms in this paper.

The 1-UIP clause is *minimized* by resolving more antecedents without adding literals. A similar idea appears in [2], which we call *local minimization*. A general version was discovered independently by the first author and implemented in MiniSat 1.13 [16]. This *recursive minimization* is now part of many SAT solvers.

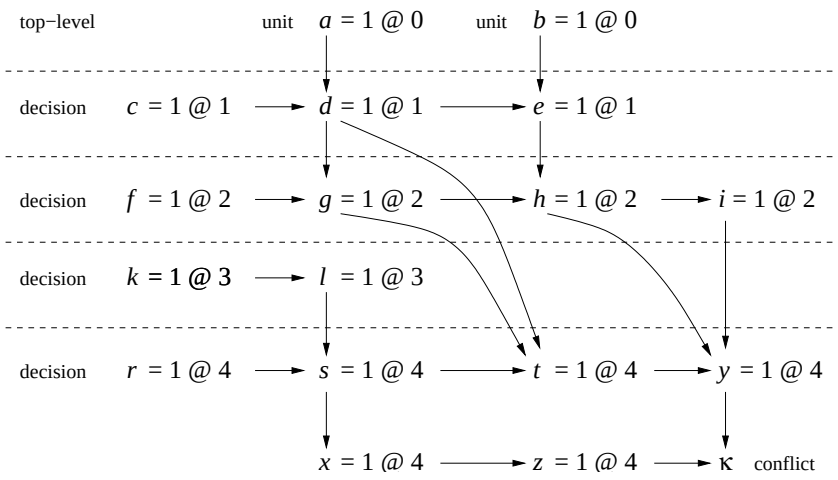

Fig. 1. An implication graph with two top-level unit clauses, and four decisions. The 1-UIP scheme, resolves from the antecedent $(\overline{y} \vee \overline{z})$ of the conflict as few as possible literals until exactly one literal from the current decision level, the 1-UIP $\overline{s}$, is left. The resulting 1-UIP clause is $(\overline{d} \vee \overline{g} \vee \overline{h} \vee \overline{i} \vee \overline{s})$. Depending on the definition, e.g. whether all literals in the derived clause are decisions, or just the UIP on the current decision level, the decision UIP clause either is comprised of the negations of all the decisions $(\overline{c} \vee \overline{f} \vee \overline{k} \vee \overline{r})$ or is obtained from the 1-UIP clause, replacing $\overline{s}$ by $\overline{l} \vee \overline{r}$. In any case, all four non top-level decision levels are "pulled" into the decision UIP clause, while the 1-UIP clause allows to jump over the decision level of k. Local minimization of the 1-UIP clause removes $\overline{i}$, since its single antecedent literal $\overline{h}$ already occurs in the 1-UIP clause. This is an instance of self-subsuming resolution, resolving $(\overline{d} \vee \overline{g} \vee \overline{h} \vee \overline{i} \vee \overline{s})$ with $(\overline{h} \vee i)$ to obtain $(\overline{d} \vee \overline{g} \vee \overline{h} \vee \overline{s})$. No other local minimization is possible. Top-level assigned literals can be ignored. Thus the nodes d and g together dominate h. As a consequence $\overline{h}$ can be deleted using recursive minimization to obtain $(\overline{d} \vee \overline{g} \vee \overline{s})$.

Minimization

The example in Fig. 1 applies the original 1-UIP scheme and the decision scheme. It also explains how the 1-UIP clause can be minimized locally or recursively. More precisely, the 1-UIP clause can be *minimized locally* by resolving out a literal, which has other literals in its antecedent already in the 1-UIP clause. This gives a first version of an algorithm for locally minimizing learned clauses:

> *Generate the 1-UIP clause. Apply self-subsuming resolution, in reverse assignment order, using antecedent clauses for self-subsuming resolution.*

This algorithm actually produces a regular and linear resolution derivation of the minimized clause. In general, resolving antecedents can not introduce cycles, even if resolved out-of-order with respect to assignment order, in contrast to [1]. Furthermore, tree-like resolution can be made regular [17]. Thus literals can actually be deleted in an arbitrary order. This simplifies implementation considerably and results in the following modified algorithm for local minimization:

Generate the 1-UIP clause. Mark all its literals. Remove those implied variables which have all their antecedent literals marked.

On the current decision level, enforcing traversal in assignment order presents no overhead, since literals of the current decision level have to be unassigned anyhow. Traversing the *trail* [6] backward gives the desired topological ordering. Traversing all literals on the trail of previous decision levels is more costly.

It is possible to continue resolving out literals, as long all newly introduced literals are *eventually* resolved out. A literal can be deleted if its antecedent literals are, in the implication graph, dominated by other literals from the 1-UIP clause. The recursive minimization algorithm can be formulated as follows:

Generate the 1-UIP clause. Mark its literals. Implied variables in 1-UIP clause are candidates for removal. Search implication graph. Start from antecedent literals of candidate. Stop at marked literals or decisions. If search always ends at marked literals then the candidate can be removed.

Soundness can be proven by simulating graph traversal with resolution. The only issue is, if literals are resolved out, not respecting the reverse chronological assignment order. Again these irregularities can be eliminated by reorganizing the derivation [17]. The result is a regular tree-shaped resolution derivation.

As optimizations successful removals should be cached and we can terminate the search through the implication graph early as soon as a literal from a decision level that is not present in the 1-UIP clause is encountered. This early termination condition can be implemented by marking decision levels of the 1-UIP clause, if decision levels are represented explicitly in the SAT solver, or as in MiniSat and PicoSat, by an over-approximation technique based on signatures as in subsumption algorithms [5].

Experiments

To empirically compare the effectiveness of recursive versus local minimization versus no minimization of learned clauses at all, we used the same set of 100 benchmarks as in the SAT Race'08 [14]. The run times were obtained on our 15 node cluster with Pentium 4 CPUs running at 2.8 GHz with 2 GB main memory. The space limit was 1.5 GB and the time limit 1800 seconds.

In order to obtain a statistically valid evaluation, we first employed two different SAT solvers. Additionally we used two versions of each SAT solver, one with preprocessing enabled and one version in which it was disabled. Second we use all 100 SAT Race'08 instances. Third, we injected noise using a random number generator to influence decision heuristics. The same set of benchmarks was then run 10 times with different seeds for each of the four configurations, i.e. one out of two SAT solvers with and without preprocessing. Altogether we have 4000 runs for each of the three variants of the minimization algorithm. The worst case accumulated execution time would have been $3 \cdot 4000 \cdot 1800$ seconds = 6000 hours. Since many instances finished before the time limit was reached it actually only took 2513 hours of compute time to finish all 12000 runs.

As first SAT solver we use an internal version of MiniSat [6], a snapshot from November 11, 2008. It is almost identical to the one described in [15] the winner of the SAT Race'08. It additionally allows to perturbate the initial variable ordering slightly using a pseudo random number generator. The second SAT solver is PicoSat [3] version 880, an improved version of PicoSat [3]. The major improvement was to separate garbage collection of learned clauses from restart scheduling. One out of 1000 decisions is a random decision in PicoSat. The seed for the random number generator is specified on the command line. Table 1 shows our main experimental results. In Tab. 2, we focus on two configurations. More details can be found at `http://fmv.jku.at/papers/minimize.7z`.

Conclusion

In this paper we discussed algorithms for minimizing learned clauses. Our extensive experimental analysis proves the effectiveness of clause minimization.

References

1. G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In *Proc. SAT'08*.
2. P. Beame, H Kautz, and A Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22, 2004.
3. A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4, 2008.
4. N. Dershowitz, Z. Hanna, and A. Nadel. Towards a better understanding of the functionality of a conflict-driven SAT solver. In *Proc. SAT'07*.
5. N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. SAT'05*.
6. N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT'03*.
7. H. S. Jin and F. Somenzi. Strong conflict analysis for propositional satisfiability. In *Proc. DATE'06*.
8. J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability*. IOS Press, 2009.
9. J. Marques-Silva and K. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Trans. on Computers*, 48(5), 1999.
10. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proc. DAC'01*.
11. A. Nadel. *Understanding and Improving a Modern SAT Solver*. PhD thesis, Tel Aviv University, 2008. Submitted.
12. S. Pilarski and G. Hu. Speeding up SAT for EDA. In *Proc. DATE'02*.
13. K. Pipatsrisawat and A. Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In *Proc. AAAI'08*.
14. C. Sinz. SAT-Race'08. http://baldur.iti.uka.de/sat-race-2008.
15. N. Sörensson and N. Eén. MS 2.1 and MS++ 1.0 — SAT Race 2008 editions.
16. N. Sörensson and N. Eén. MiniSat v1.13 – A SAT solver with conflict-clause minimization, 2005.
17. A. Urquhart. The complexity of propositional proofs. *Bull. of EATCS*, 64, 1998.
18. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in boolean satisfiability solver. In *Proc. ICCAD'01*.

		solved instances		time in hours		space in GB		out of memory		deleted literals
MiniSat	recur	788	9%	170	11%	198	49%	11	89%	33%
with	local	774	7%	177	8%	298	24%	72	30%	16%
preprocessing	none	726		192		392		103		
MiniSat	recur	705	13%	222	8%	232	59%	11	94%	37%
without	local	642	3%	237	2%	429	24%	145	26%	15%
preprocessing	none	623		242		565		196		
PicoSat	recur	767	10%	182	13%	144	45%	10	60%	31%
with	local	745	6%	190	9%	188	29%	10	60%	15%
preprocessing	none	700		209		263		25		
PicoSat	recur	690	6%	221	8%	105	63%	10	68%	33%
without	local	679	5%	230	5%	194	31%	10	68%	14%
preprocessing	none	649		241		281		31		
	recur	2950	9%	795	10%	679	55%	42	88%	34%
altogether	local	2840	5%	834	6%	1109	26%	237	33%	15%
	none	2698		884		1501		355		

Table 1. Experiments with MiniSat and PicoSat on SAT Race'08 benchmarks. The first column specifies the configuration, e.g. which of the two SAT solvers is used and whether preprocessing is enabled or disabled. The last three rows summarize these four configurations. The next column specifies the minimization algorithm: "recur" is recursive minimization, the default in MiniSat and PicoSat. Then there are rows with "local" minimization for each configuration and "none" denotes the base case, in which learned clauses are not minimized at all. The third column gives the number of solved instances out of 1000, respectively 4000 in the last three rows. Each row corresponds to the 10 runs with different seeds over the 100 instances. The next column gives the improvement in number of solved instances with respect to the base case. The number of solved instances increases by roughly 10% for recursive minimization, and half that much for local minimization only. The difference in run-time, shown in the next two columns, gives a similar picture. The percentage in the 6th column is calculated as the amount of time the minimizing algorithm finishes earlier relative to the base case. An unsolved instance contributes 1800 seconds. In the next two columns we report on memory usage, calculated as the sum of the maximum main memory used in each run, at most 1.5 GB per run. Half of the memory can be saved using recursive minimization, with local minimization one quarter. This effect is even more dramatic with respect to the number of times a run reached the space limit, which is shown in columns 9 and 10 next, particularly in the case of MiniSat. PicoSat uses more compact data structures than MiniSat, for instance to store binary clauses [12]. Minimization also reduced the number of space-outs by more than 60%. Finally, the last column, shows the average number of deleted literals per learned clause. This is calculated with respect to the size of the 1-UIP clause, which would have been generated without minimization, even though the 1-UIP clause is minimized afterwards. This is different from comparing the average length of learned clauses with and without minimization, since these statistics are computed within the minimizing solver. This gives an explanation why memory savings are almost twice as much as savings due to deleted literals only. Minimization not only saves space, but also reduces the search space.

	MiniSat with preprocessing							PicoSat with preprocessing						
	seed	solved	time	space	mo	del		seed	solved	time	space	mo	del
recur	8	82	16	19	1	33%	recur	9	79	17	14	1	31%
recur	6	81	17	20	1	33%	recur	0	78	18	14	1	31%
local	0	81	16	29	7	16%	recur	3	78	18	14	1	31%
local	7	80	17	29	8	15%	recur	8	78	18	14	1	31%
recur	4	80	17	20	1	33%	recur	2	77	19	14	1	31%
recur	1	79	17	20	1	33%	local	7	77	19	19	1	15%
recur	9	79	17	20	1	34%	recur	6	77	18	14	1	31%
local	5	78	18	29	7	16%	local	3	77	18	18	1	15%
local	1	78	17	29	6	16%	recur	7	76	18	14	1	31%
recur	0	78	17	20	1	34%	local	4	75	19	19	1	15%
recur	5	78	17	19	1	33%	local	1	75	19	19	1	15%
local	3	77	18	31	7	16%	recur	4	75	18	14	1	31%
local	8	77	18	30	8	16%	recur	5	75	18	14	1	30%
recur	7	77	17	20	1	34%	local	2	74	19	19	1	15%
recur	3	77	17	20	1	34%	local	8	74	19	19	1	15%
recur	2	77	17	20	2	33%	recur	1	74	19	14	1	31%
none	7	76	19	39	9	0%	local	5	74	19	18	1	15%
local	2	76	18	31	8	16%	local	6	73	20	19	1	15%
local	4	76	18	31	7	16%	local	0	73	20	19	1	15%
local	6	76	18	30	7	16%	local	9	73	19	19	1	16%
local	9	75	19	29	7	16%	none	5	72	21	26	4	0%
none	9	74	19	39	10	0%	none	3	72	20	26	3	0%
none	6	73	19	40	12	0%	none	7	72	20	26	2	0%
none	3	73	19	39	10	0%	none	8	71	21	27	2	0%
none	8	72	20	39	11	0%	none	9	71	20	25	3	0%
none	0	72	20	39	11	0%	none	1	70	21	27	1	0%
none	1	72	19	39	9	0%	none	4	69	21	26	2	0%
none	5	72	19	39	10	0%	none	0	69	21	26	4	0%
none	2	71	20	40	11	0%	none	6	68	21	26	2	0%
none	4	71	19	39	10	0%	none	2	66	22	27	2	0%

Table 2. Even for 100 benchmarks there is a great variance for different seeds. The columns are as in Tab. 1. Even though the space reduction and also the percentage of deleted literals is consistent for different seeds, the run-time and the number of solved instances vary widely. Both, for MiniSat and PicoSat, it would be possible to draw the conclusion that local minimization is better than recursive minimization, by picking two specific seeds, for instance MiniSat/local/0 vs. MiniSat/recur/2, and PicoSat/local/7 vs. PicoSat/recur/1. The relative space usage is consistent over different runs of the same algorithm, as is the percentage of deleted literals. For empirical evaluations of heuristics of SAT solvers, we suggest, to enforce identical search behavior [3], or to use a very large set of benchmarks, definitely more than 100. However, it is probably necessary to randomize the algorithms and run them with different seeds a sufficient number of times. Another option is to use secondary statistics directly related to the proposed heuristics, like the number of deleted literals in our case, in addition to the number of solved instances, time usage, or a scatter plot.