The Interaction Between Simplification and Search in Propositional Satisfiability

Inês Lynce and João Marques-Silva

Department of Informatics,
Technical University of Lisbon,
IST/INESC/CEL, Lisbon, Portugal
{ines,jpms}@sat.inesc.pt

Abstract. Simplification techniques have been extensively applied to formulas in Conjunctive Normal Form (CNF). Consequently, different models of the same problem can be obtained by using different techniques, namely by *reducing* either the number of variables or clauses, and by *inferring* new clauses. Moreover, alternative formulations are expected to affect the amount of search performed by backtrack search algorithms. The main objectives of this paper are to review existing simplification techniques for CNF formulas, and to evaluate the effectiveness of the different simplification techniques, based on existing propositional simplifiers and state-of-the-art SAT solvers. Experimental results, obtained on representative hard real-world problem instances, indicate that simplification can play a very *puzzling* role in Propositional Satisfiability.

1 Introduction

Recent years have seen the proposal of several effective approaches for solving Propositional Satisfiability (SAT). Different approaches have allowed efficiently solving different classes of instances of SAT. It is generally accepted that whereas most approaches can be competitive in proving satisfiability for different classes of instances, backtrack search is preferred when the objective is to prove unsatisfiability. Nevertheless, formula simplification solutions are also known to be competitive for proving unsatisfiability in specific contexts [10].

The main goal of this paper is to briefly review existing formula simplification techniques, categorize these techniques, and illustrate the effectiveness of the simplification techniques as a preprocessing tool for SAT algorithms. Moreover, we analyze the application of the proposed simplification techniques in real-world instances of SAT. The obtained experimental results indicate that simplification can play a puzzling role in Propositional Satisfiability. The main experimental observation is that reducing the number of variables or inferring additional clauses is not necessarily useful in terms of reducing the amount of search performed by backtrack search SAT algorithms. The practical usefulness of adding variables or inferring clauses depends on which techniques are actually utilized by the simplification algorithm. As a result, one conclusion that can be drawn is that simplification techniques ought to be evaluated in the context in which their are to be used, and not solely by their intrinsic simplification ability.

The paper is organized as follows. We start by addressing formula simplification, and propose a categorization of the different techniques. Section 3 analyzes the existing propositional simplifiers. Experimental results on applying these simplifiers on real-world instances of SAT are presented in Section 4. Moreover, the obtained results are compared and discussed. Finally, Section 5 concludes the paper.

2 Simplification Techniques

In this section we review existing techniques for CNF formula simplification. We start with the most well-known and fundamental techniques, that include the unit clause rule, the pure literal rule, restricted resolution and two-variable equivalence conditions. Next, we analyze pattern-based simplification techniques, namely equivalency reasoning and inference of binary clauses.

Simple simplifications. We start by observing that a clause containing a literal l_j and its complement $\neg l_j$ is satisfied, i.e. $l_j \vee \neg l_j \equiv 1$. Moreover, if a clause contains two copies of a literal l_j, then one copy can be removed, i.e. $l_j \vee l_j \equiv l_j$. Another simplification consists in removing duplicated clauses, which often occurs due to simplifications on the original formula.

Unit clause rule and pure literal rule. The most widely used formula simplification procedures in backtrack search SAT algorithms are certainly the *unit clause rule* and the *pure literal rule* [6].

The unit clause rule applies whenever a unit clause $\omega_i = (l_j)$ is identified. In this case, all clauses containing literal l_j are declared satisfied, and literal $\neg l_j$ is removed from the clauses containing it. The unit clause rule can be iterated, and this process is usually referred to as *Boolean Constraint Propagation (BCP)* [18].

The pure literal rule applies whenever a variable x_j only occurs as a pure literal, either $l_j = x_j$ or $l_j = \neg x_j$. In this situation, all clauses containing literal l_j can be declared satisfied. (Notice that literal l_j can be assigned truth value 1, satisfying all of its clauses.)

Restricted resolution. Resolution is known to be a sound and complete proof procedure for Propositional Satisfiability [15]. It is in general not computationally feasible to utilize unrestricted forms of resolution. Hence, different algorithms for simplifying propositional formulas have used different forms of constraining the application of the resolution operation [17, 8]. For example, we can consider *restricted* forms of resolution by which *only* simplified clauses can be derived. Considering clauses $\omega_1 = (x_1 \vee x_2 \vee \neg x_3)$ and $\omega_2 = (\neg x_1 \vee x_2 \vee \neg x_3)$, resolution allows deriving clause $(x_2 \vee \neg x_3)$. This can be written as $\omega_3 = \rho(\omega_1, \omega_2, x_1) = (x_2 \vee \neg x_3)$. In general, given $\omega_r = (x_i \vee \alpha)$ and $\omega_s = (\neg x_i \vee \alpha)$, where α is a disjunction of literals, resolution allows deriving the clause, $\omega_t = \rho(\omega_r, \omega_s, x_i) = \rho((x_i \vee \alpha), (\neg x_i \vee \alpha), x_i) = (\alpha)$.

Two-variable equivalence. Two-variable equivalence is a well-known formula simplification procedure. Let us consider the pair of clauses $(\neg x \vee y) \wedge (x \vee \neg y)$, that can be represented as $(x \rightarrow y) \wedge (y \rightarrow x)$ (and also as $(\neg y \rightarrow \neg x) \wedge (\neg x \rightarrow \neg y)$). Then x and y are said to be equivalent, i.e. for any assignment that satisfies the two clauses, the truth values of x and y must be the same. Consequently, we can replace y by x on *all* occurrences of y, thus eliminating variable y from the CNF formula. Moreover, additional two-variable equivalence conditions can be established, namely based on loops of binary implications [1] or obtaining the *2-closure* of the 2CNF sub-formula [7].

Subsumption. A final technique analyzed in this section is *subsumption*. Suppose that our CNF formula φ contains two clauses $\omega_a = (\alpha)$ and $\omega_b = (\alpha \vee \beta)$, where α and β are disjunctions of literals. In this case, clause ω_b can be eliminated from the CNF formula, since any assignment that satisfies ω_a also satisfies ω_b. Within the set of simplification techniques proposed in this paper, subsumption is only to be used after *all* techniques have been applied and yield no further simplification of the formula. It can be easily shown that subsumption can only remove clauses, but will not induce additional simplifications.

Equivalency Reasoning. Recent work on pattern-based simplification was done by C.-M. Li [10] and is referred to as *equivalency reasoning*. This work motivated and was superseded by other pattern-based formula simplification techniques [12], which are described in the following paragraphs. Nevertheless, equivalency reasoning is applied during search and pattern-based formula simplification techniques are currently solely applied during preprocessing.

Inference of Binary Clauses. In what follows, we analyze subsets of clauses for inferring binary clauses [12]. We should note that all proposed conditions can be explained by resorting to resolution. Nonetheless, the objective of studying sets of clauses is to indirectly select to which sets of clauses the resolution operation should be applied to.

In general, let us consider a sub-formula of the form $\varphi_s = (l_1 \vee \neg l_2) \wedge (l_1 \vee \neg l_3) \wedge (l_2 \vee l_3 \vee l_4)$, where $l_1, \ldots, l_4$ are any literals. The application of resolution allows deriving the binary clause $(l_1 \vee l_4)$. Since 2 binary clauses and 1 ternary clause are involved in deriving the resulting binary clause, we say that 2B/1T reasoning was applied.

Additional forms of mB/nT reasoning can be established. As an example we analyze 0B/4T reasoning. Suppose the following set of four ternary clauses: $(l_1 \vee l_2 \vee l_3) \wedge (\neg l_1 \vee \neg l_2 \vee l_3) \wedge (l_1 \vee \neg l_2 \vee \neg l_4) \wedge (\neg l_1 \vee l_2 \vee \neg l_4)$. Then the binary clause $(l_3 \vee \neg l_4)$ can be inferred, again by restricted resolution.

3 Formula Simplification Algorithms

Having described most of the existing simplification techniques in the previous section, we now characterize different formula simplification algorithms in terms of these techniques. For this purpose we selected state-of-the-art simplification algorithms, namely `Compact` [5], `Compactor` [11] [1], `Simplify` [12] and `CompressLite` [3][2].

Before describing the techniques specific to each simplification algorithm, we enumerate the ones common to all the algorithms considered:

- Boolean constraint propagation (BCP).
- Pure literal rule (PLR).
- Elimination of duplicated clauses and literals in clauses, and elimination of satisfied clauses.

Compact. `Compact` is a propositional simplifier developed by James Crawford [5], that implements the following techniques:

- *Singleton rule*: if a variable literal has a single occurrence, then apply resolution to eliminate that variable.
- *Failed literal rule*: if the assignment $l_1 = 1$ yields a conflict (due to BCP), then we can assign $l_1 = 0$.
- *Binary failed literal rule*: if the assignments $l_1 = 1$ and $l_2 = 1$ yield a conflict (due to BCP), then we can create the binary clause $(\neg l_1 \vee \neg l_2)$.

[1] Available from http://www.laria.u-picardie.fr/~cli/my_compact.c.
[2] Available from the SATLIVE! web site, http://www.satlive.org/.

Table 1. Example Instances

Application Domain	Instance	Satisfiable?
Bounded Model Checking (BMC) [4]	barrel6	N
	longmult6	N
	queueinvar16	N
Combinational Equivalence Checking (CEC) [9]	c1908	N
	c2670_bug	Y
Superscalar Processor Verification (SPV) [16]	dlx2_cc_bug02	Y
	dlx2_cc_bug08	Y
Data Encryption Standard (DES) [14]	cnf-r3-b3-k1.2	Y
	cnf-r3-b4-k1.2	Y

Compactor. `Compactor` corresponds to the SATZ algorithm [11], without the backtrack search phase. `Compactor` includes the following simplification techniques:

- Failed literal rule.
- Application of resolution for deriving resolvents such that each resolvent has less than four literals.

Simplify. `Simplify` is another simplification algorithm that incorporates the following simplification techniques:

- Two-variable equivalence.
- Restricted resolution (i.e. only applied to binary clauses and to infer unit clauses).
- Clause inference conditions, based on the 2B/1T and 0B/4T clause patterns (see Section 2).

CompressLite. `CompressLite` [3] is a recent simplifier that implements alternative forms of variable probing:

- Utilization of BCP for probing variable assignments. The main objective is to identify two-variable equivalences. If $x = 1$ implies $y = 1$ and $x = 0$ implies $y = 0$, then x and y can be deemed equivalent.

4 Experimental Results

This section evaluates the practical application of the different simplification techniques described in this paper, according to two main metrics:

- The ability to effectively simplify the original formula.
- The effective reduction in the amount of search, when using formula simplification techniques as a preprocessing engine for backtrack search SAT algorithms.

With the objective of evaluating the formula simplification ability of the simplifiers described in Section 3 (`Compact`, `Compactor`, `Simplify` and `CompressLite`), several representative problem instances from different real-world applications were used. The problem instances are shown in Table 1, and can be obtained from the different authors. Moreover, for all the results presented in this section, a PIII-866MHz Linux machine with 512 MByte of RAM was used.

Table 2. Formula Simplification — Number of Variables and Clauses

Instance	Initial Formula		`Compact`		`Compactor`		`Simplify`		`CompressLite`	
	Var^s	Cl^s	Var^s	Cl^s	Var^s	Cl^s	Var^s	Cl^s	Var^s	Cl^s
barrel6	2306	8931	1722	7151	2222	10035	515	2193	1730	7053
c1908	1917	5096	1891	5148	1916	5069	658	2248	985	3208
c2670_bug	2708	6696	2443	6261	2553	6443	1162	3553	1361	4023
cnf-r3-b3-k1.2	676896	27503	—		3661	23773	4418	27901	4318	27146
cnf-r3-b4-k1.2	939040	35963	—		4697	30812	5721	36484	5610	35566
dlx2_cc_bug02	1515	12808	1405	16042	1489	13021	1486	13950	1504	12742
dlx2_cc_bug08	1515	12808	1405	16005	1489	13021	1486	13875	1504	12742
longmult6	2848	8853	1824	5890	2279	6987	1824	6376	1404	4646
queueinvar16	1168	6496	1074	6830	1134	6336	1088	7114	1099	6358

Table 3. Formula Simplification — CPU Time

Instance	`Compact`	`Compactor`	`Simplify`	`CompressLite`
barrel6	0.43	0.52	2.68	0.05
c1908	0.07	0.06	0.19	1.23
c2670_bug	0.07	0.07	0.22	0.56
cnf-r3-b3-k1.2	>1000	3.79	5.30	448.79
cnf-r3-b4-k1.2	>1000	6.74	9.01	693.37
dlx2_cc_bug02	0.54	1.14	2.18	0.05
dlx2_cc_bug08	0.52	1.14	2.16	0.05
longmult6	5.45	0.26	0.34	2.18
queueinvar16	0.21	0.65	0.78	0.13

Formula Simplification. The first experiment consisted in applying the proposed formula simplification techniques to the different problem instances. Table 2 shows the results, namely the number of variables and clauses before and after applying the simplification techniques for the four simplifiers. (The maximum allowed CPU time was set to 1,000 seconds.)

As can be concluded, for most of the instances considered, large reductions in the number of variables and clauses can be achieved. In some cases (e.g. *barrel6* for `Simplify`) the final number of variables is one fourth of the original number [3]. Similar reductions can be observed in the number of clauses. Nevertheless, for some instances (e.g. *queueinvar16* and *dlx2_cc_bug08*) the amount of simplification in the number of variables is negligible. It is also interesting to note that some simplifiers detected some of the instances to be sat/unsat in this preprocessing phase. Furthermore, for some instances (e.g. *dlx2_cc_bug02* and *dlx2_cc_bug08*) it is possible to observe an increase in the number of clauses, due to the application of clause inference techniques. These clauses can eventually be utilized later to reduce the search effort.

Table 3 shows the CPU Time for simplifying each instance by each simplifier. Clearly, the results depend on the instance and on the simplifier. Nonetheless, and considering in concrete the DES instances, and the `Compact` and `CompressLite` simplifiers, we can conclude that some simplification techniques can sometimes introduce so much computational overhead that their utilization may not be worthwhile.

Search Results. The next experiment consisted of evaluating the actual usefulness of the simplification techniques in solving instances of SAT. Table 4 shows the results of running GRASP [13], rel_sat [2], SATZ [11] and EQSATZ [10] on the different problem

[3] For the DES instances the reductions in the number of variables are even more dramatic, but all these variables are known to be trivially removed.

Table 4. Time Results before Formula Simplification

Instance	GRASP	rel_sat	SATZ	EQSATZ
barrel6	>3000	643.0	1100.1	0.5
c1908	2465.5	>3000	>3000	525.3
c2670_bug	6.4	1.0	0.4	>3000
cnf-r3-b3-k1.2	1858.0	>3000	—	—
cnf-r3-b4-k1.2	1520.4	>3000	—	—
dlx2_cc_bug02	649.9	>3000	>3000	>3000
dlx2_cc_bug08	2.6	>3000	>3000	8.4
longmult6	354.9	64.0	36.7	18.9
queueinvar16	21.1	15.0	705.7	2.8

Table 5. Time Results after Formula Simplification — `Compact` and `Compactor`

Instance	`Compact`				`Compactor`			
	GRASP	rel_sat	SATZ	EQSATZ	GRASP	rel_sat	SATZ	EQSATZ
barrel6	>3000	296.0	674.4	0.3	>3000	667.0	1099.1	0.2
c1908	>3000	>3000	>3000	1358.9	>3000	>3000	>3000	393.6
c2670_bug	263.3	1.0	0.5	44.3	0.3	0.0	0.5	>3000
cnf-r3-b3-k1.2	—	—	—	—	16.3	10.0	>3000	3.0
cnf-r3-b4-k1.2	—	—	—	—	12.8	13.0	7.5	5.2
dlx2_cc_bug02	760.4	>3000	>3000	>3000	1408.8	2133.0	>3000	>3000
dlx2_cc_bug08	97.7	>3000	>3000	>3000	17.6	9.0	>3000	8.3
longmult6	263.6	>3000	>3000	>3000	241.5	67.0	35.4	18.2
queueinvar16	20.1	14.0	739.6	2.4	21.2	18.0	704.4	2.8

instances *before* applying the simplification techniques [4]. The maximum allowed CPU time was set to 3,000 seconds. The results obtained *after* applying the simplification techniques are shown in Tables 5 [5] and 6.

Discussion of the Results. The obtained experimental results are somewhat surprising, since formula simplification by different simplifiers may yield instances that require fairly different run times for each SAT solver. As a result, we can establish general trends, denoting which simplification techniques are indeed effective in reducing the CPU times of SAT solvers.

We start by analyzing the results for different simplifiers and solvers, and afterwards highlight which simplification techniques are in general useful for reducing the run times of SAT solvers.

For the `Compact` simplifier, we can point out two main conclusions:

1. A solver can sometimes need more time to solve an instance after applying formula simplification (e.g. *c1908* instance).
2. The reductions in the number of variables and clauses on the original formula do not necessarily imply a reduction in the search effort (e.g *dlx2_cc_bug* instances, for which `Compact` obtains the best reductions for the original formula).

In contrast, `Compactor` *never* presents the best reductions in the number of variables and clauses for simplifying a given instance, when compared with the other simplifiers. Nevertheless, `Compactor` is more effective than `Compact` in reducing the search after formula simplification.

[4] SATZ and EQSATZ stopped with error conditions when solving the DES instances.
[5] Instances that were solved during preprocessing are marked with *simp*.

Table 6. Time Results after Formula Simplification — `Simplify` and `CompressLite`

Instance	Simplify				CompressLite			
	GRASP	rel_sat	SATZ	EQSATZ	GRASP	rel_sat	SATZ	EQSATZ
barrel6	1.2	0.0	0.1	0.1	1999.5	276.0	>3000	485.4
c1908	10.0	4.0	>3000	0.9	>3000	>3000	>3000	408.8
c2670_bug	0.1	0.0	0.1	0.5	0.1	0.0	0.2	1.9
cnf-r3-b3-k1.2	82.8	15.0	15.0	4.0	68.4	14.0	149.8	4.4
cnf-r3-b4-k1.2	80.7	16.0	>3000	7.2	29.9	14.0	>3000	7.9
dlx2_cc_bug02	107.4	869.0	>3000	>3000	>3000	361.0	>3000	>3000
dlx2_cc_bug08	0.5	520.0	>3000	3.8	19.5	3.0	>3000	>3000
longmult6	249.2	43.0	19.8	48.2	150.2	53.0	15.5	26.5
queueinvar16	20.4	11.0	430.4	3.1	19.1	14.0	416.9	2.8

Formula simplifcation with `Simplify` allows solving problem instances that, otherwise, the different SAT solvers would be unable to solve in the allowed CPU time. In concrete, *all* the instances that were not solved in 3,000 seconds by GRASP and rel_sat before formula simplification are now solved in the allowed CPU time. In addition, all the simplified formulas are in general solved faster than the original formulas. There are however a few exceptions. For example, for one instance (*dlx2_cc_bug08*), and for rel_sat, the small reduction in the number of variables and the increase in the number of clauses, causes an increase in the amount of search. This result is explained by the dynamic nature of rel_sat's branching heuristic. Moreover, it is interesting to observe that the reduction in the number of variables and clauses allows problem instance *barrel6* to be solved by GRASP in a few seconds, in contrast with the original formula, for which the GRASP SAT solver quits after 3,000 CPU seconds. Finally, and with respect to EQSATZ, it is also interesting to observe that formula simplification applied by `Simplify` during preprocessing cooperates effectively with the techniques for formula simplification applied by EQSATZ during the search. Moreover, one instance (*c2670_bug*) would not be solved by EQSATZ in 3,000 CPU seconds if it was not for the simplifications produced by `Simplify`.

From the experimental results and previous observations, we can conclude that the *singleton rule* and the binary *failed literal rule* used in `Compact` do not in general contribute to reducing the run times of SAT solvers (quite the contrary). The restricted form of probing utilized by `CompressLite` is also not essential for reducing the run times of SAT solvers. In contrast, the failed-literal rule, two-variable equivalence, restricted resolution (as used in `Simplify`), and the 2B/1T and 0B/4T clause inference conditions, yield formula simplifications that effectively help SAT solvers reducing their run times.

5 Conclusions

This paper analyzes existing techniques for the simplification of propositional formulas, and conducts an experimental evaluation of these techniques, with the goal of deciding which ones are indeed useful in the context of solving hard real-world instances of SAT.

The experimental results, obtained for different existing simplifiers and on representative real-world instances of SAT, clearly demonstrate the effectiveness of these simplification techniques, when formula simplification is the main objective. However, and interestingly, the experimental results also indicate that some formula simplification techniques, despite yielding significant reductions in the number of variables and allowing the inference of a large number of clauses, may not necessarily contribute to solving specific classes of problem instances. One conclusion of this experimental data is that formula simplification techniques should be evaluated not only on their ability to reduce the number of variables or to infer new clauses, but also on their ability to yield CNF formulas that SAT solvers can more easily solve. This suggests a two-phase evaluation process, both for evaluating new formula simplication techniques and for selecting which of the existing techniques are indeed helpful for SAT solvers.

The presented experimental results depend on the problem instances considered. The next natural step is to conduct a thorough evaluation of the existing formula simplication techniques on *all* existing suites of real-world problem instances, of which the problem instances considered in this paper are representative examples.

References

1. B. Aspvall, M. F. Plass, and R. E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, March 1979.
2. R. Bayardo Jr. and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the National Conference on Artificial Intelligence*, pages 203–208, 1997.
3. D. L. Berre. Exploiting the real power of unit propagation lookahead. In *LICS Workshop on Theory and Applications of Satisfiability Testing*, June 2001.
4. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In W. R. Cleaveland, editor, *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1579. Springer-Verlag, March 1999.
5. J. Crawford and L. Auton. Experimental results on the cross-over point in satisfiability problems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 22–28, 1993.
6. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.
7. A. V. Gelder and Y. K. Tsuji. Satisfiability testing with more reasoning and less guessing. In D. S. Johnson and M. A. Trick, editors, *Second DIMACS Implementation Challenge*. American Mathematical Society, 1993.
8. J. Groote and J. Warners. The propositional formula checker heerhugo. In I. Gent, H. van Maaren, and T. Walsh, editors, *SAT 2000*, pages 261–281. IOS Press, 2000.
9. W. Kunz and D. Stoffel. *Reasoning in Boolean Networks*. Kluwer Academic Publishers, 1997.
10. C.-M. Li. Integrating equivalency reasoning into davis-putnam procedure. In *Proceedings of the National Conference on Artificial Intelligence*, pages 291–296, July 2000.
11. C. M. Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of International Conference on Principles and Practice of Constraint Programming*, 1997.
12. J. P. Marques-Silva. Algebraic simplification techniques for propositional satisfiability. In *International Conference on Principles and Practice of Constraint Programming*, pages 537–542, September 2000.
13. J. P. Marques-Silva and K. A. Sakallah. GRASP-A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.
14. F. Massacci. Using Walk-SAT and Rel-SAT for cryptographic key search. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 290–295, 1999. (DES instances available from URL http://www.uni-koblenz.de/~massacci).
15. S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, 1995.
16. M. N. Velev and R. E. Bryant. Superscalar processor verification using efficient reductions from the logic of equality with uninterpreted functions to propositional logic. In *Proceedings of Correct Hardware Design and Verification Methods*, LNCS 1703, pages 37–53, September 1999.
17. F. Vlach. Simplification in a satisfiability checker for VLSI applications. *Journal of Automated Reasoning*, 10(1):115–136, February 1993.
18. R. Zabih and D. A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the National Conference on Artificial Intelligence*, pages 155–160, 1988.