

Eliminating redundant clauses in SAT instances

Olivier Fourdrinoy Éric Grégoire Bertrand Mazure Lakhdar Saïs

CRIL CNRS & IRCICA
Université d'Artois
Rue Jean Souvraz SP18
F-62307 Lens Cedex France
{fourdrinoy, gregoire, mazure, saïs}@cril.fr

Abstract. In this paper, we investigate to which extent the elimination of a class of redundant clauses in SAT instances could improve the efficiency of modern satisfiability provers. Since testing whether a SAT instance does not contain any redundant clause is NP-complete, a logically incomplete but polynomial-time procedure to remove redundant clauses is proposed as a pre-treatment of SAT solvers. It relies on the use of the linear-time unit propagation technique and often allows for significant performance improvements of the subsequent satisfiability checking procedure for really difficult real-world instances.

1 Introduction

The SAT problem, namely the issue of checking whether a set of Boolean clauses is satisfiable or not, is a central issue in many computer science and artificial intelligence domains, like e.g. theorem proving, planning, non-monotonic reasoning, VLSI correctness checking and knowledge-bases verification and validation. These last two decades, many approaches have been proposed to solve hard SAT instances, based on -logically complete or not- algorithms. Both local-search techniques (e.g. [1]) and elaborate variants of the Davis-Putnam-Loveland-Logemann's DPLL procedure [2] (e.g. [3, 4]) can now solve many families of hard huge SAT instances from real-world applications.

Recently, several authors have focused on detecting possible hidden structural information inside real-world SAT instances (e.g. backbones [5], backdoors [6], equivalences [7] and functional dependencies [8]), allowing to explain and improve the efficiency of SAT solvers on large real-world hard instances. Especially, the conjunctive normal form (CNF) encoding can conduct the structural information of the initial problem to be hidden [8]. More generally, it appears that many real-world SAT instances contain redundant information that can be safely removed in the sense that equivalent but easier to solve instances could be generated.

In this paper, we investigate to which extent the elimination of a class of redundant clauses in real-world SAT instances could improve the efficiency of modern satisfiability provers. A redundant clause is a clause that can be removed from the instance while keeping the ability to derive it from the remaining part of the instance. Since testing whether a SAT instance does not contain any redundant clause is NP-complete [7], an incomplete but polynomial-time procedure to remove redundant clauses is proposed

as a pre-treatment of SAT solvers. It relies on the use of the linear time unit propagation technique. Interestingly, we show that it often allows for significant performance improvements of the subsequent satisfiability checking procedure for hard real-world instances.

The rest of the paper is organized as follows. After basic logical and SAT-related concepts are provided in Section 2, Section 3 focuses on redundancy in SAT instances, and the concept of redundancy modulo unit propagation is developed. In Section 4, our experimental studies are presented and analyzed. Related works are discussed in Section 5 before conclusive remarks and prospective future research works are given in the last Section.

2 Technical background

Let \mathcal{L} be a standard Boolean logical language built on a finite set of Boolean variables, denoted x, y , etc. Formulas will be denoted using letters such as c . Sets of formulas will be represented using Greek letters like Γ or Σ . An interpretation is a truth assignment function that assigns values from $\{true, false\}$ to every Boolean variable. A formula is consistent or satisfiable when there is at least one interpretation that satisfies it, i.e. that makes it become *true*. Such an interpretation is called a model of the instance. An interpretation will be denoted by upper-case letters like I and will be represented by the set of literals that it satisfies. Actually, any formula in \mathcal{L} can be represented (while preserving satisfiability) using a set (interpreted as a conjunction) of clauses, where a clause is a finite disjunction of literals, where a literal is a Boolean variable that can be negated. A clause will also be represented by the set formed with its literals. Accordingly, the size of a clause c is given by the number of literals that it contains, and is noted $|c|$.

SAT is the NP-complete problem [9] that consists in checking whether a finite set of Boolean clauses of \mathcal{L} is satisfiable or not, i.e. whether there exists an interpretation that satisfies all clauses in the set or not.

Logical entailment will be noted using the \models symbol: let c be a clause of \mathcal{L} , $\Sigma \models c$ iff c is *true* in all models of Σ . The empty clause will represent inconsistency and is noted \perp .

In the following, we often refer to the binary and Horn fragments of \mathcal{L} for which the SAT issue can be solved in polynomial time [10–12]. A binary clause is a clause formed with at most two literals whereas a Horn clause is a clause containing at most one positive literal. A unit clause is a single literal clause.

In this paper, the *Unit Propagation* algorithm (in short UP) plays a central role. UP recursively simplifies a SAT instance by propagating -throughout the instance- the truth-value of unit clauses whose variables have been already assigned, as shown in algorithm 1.

We define entailment modulo Unit Propagation, noted \models_{UP} , the entailment relationship \models restricted to the Unit Propagation technique.

Definition 1. Let Σ be a SAT instance and c be a clause of \mathcal{L} , $\Sigma \models_{UP} c$ if and only if $\Sigma \wedge \neg c \models_{UP} \perp$ if and only if $UP(\Sigma \wedge \neg c)$ contains an empty clause.

Algorithm 1: Unit_Propagation

Input: a SAT instance Σ

Output: an UP-irredundant SAT instance Γ equivalent to Σ w.r.t. satisfiability s.t. Γ does not contain any unit clause

```
1 begin
2   if  $\Sigma$  contains an empty clause then return  $\Sigma$ ;
3   else
4     if  $\Sigma$  contains a unit clause  $c = \{l\}$  then
5       remove all clauses containing  $l$  from  $\Sigma$ ;
6       foreach  $c \in \Sigma$  s.t.  $\neg l \in c$  do
7          $c \leftarrow c \setminus \{\neg l\}$ 
8       return Unit_Propagation( $\Sigma$ );
9     else
10      return  $\Sigma$ ;
11 end
```

Clearly, \models_{UP} is logically incomplete. It can be checked in polynomial time since UP is a linear-time process. Let c_1 and c_2 be two clauses of \mathcal{L} . When $c_1 \subseteq c_2$, we have that $c_1 \models c_2$ and c_1 (resp. c_2) is said to subsume (resp. be subsumed by) c_2 (resp. c_1). A clause $c_1 \in \Sigma$ is subsumed in Σ iff there exists $c_2 \in \Sigma$ s.t. $c_1 \neq c_2$ and c_2 subsumes c_1 . Σ is closed under subsumption iff for all $c \in \Sigma$, c is not subsumed in Σ .

3 Redundancy in SAT instances

Intuitively, a SAT instance is redundant if it contains parts that can be logically inferred from it. Removing redundant parts of SAT instances in order to improve the satisfiability checking process entails at least two fundamental issues.

- First, it is not clear whether removing such parts will make satisfiability checking easier, or not. Some redundant information can actually improve the efficiency of SAT solvers (see e.g. [13, 14]). For example, it is well-known that redundancy can help local search to escape from local minima.
- It is well-known that checking whether a SAT instance is irredundant or not is itself NP-complete [7]. It is thus as hard as solving the SAT instance itself.

In order to address these issues, we consider an incomplete algorithm that allows some redundant clauses to be detected and that remains polynomial. Intuitively, we should not aim at removing all kinds of redundant clauses. Some types of clauses are expected to facilitate the satisfiability testing since they belong to polynomial fragments of SAT, especially the binary and the Horn ones. Accordingly, we propose an approach that appears to be a two-levels trade-off: on the one hand, we run a redundancy detection and removal algorithm that is both fast and incomplete. On the other hand, we investigate whether it proves useful to eliminate redundant binary and Horn clauses or not.

Algorithm 2: Compute an UP-irredundant formula

Input: a SAT instance Σ **Output:** an UP-irredundant SAT instance Γ equivalent to Σ w.r.t. satisfiability

```
1 begin
2    $\Gamma \leftarrow \Sigma$ ;
3   forall clauses  $c = \{l_1, \dots, l_n\} \in \Sigma$  sorted according to their decreasing sizes do
4     if  $UP(\Gamma \setminus \{c\} \cup \{\neg l_1\} \cup \dots \cup \{\neg l_n\})$  contains an empty clause then
5        $\Gamma \leftarrow \Gamma \setminus \{c\}$ ;
6   return  $\Gamma$ ;
7 end
```

Definition 2.

Let Σ be a SAT instance and let $c \in \Sigma$, c is redundant in Σ if and only if $\Sigma \setminus \{c\} \models c$.

Clearly, redundancy can be checked using a refutation procedure. Namely, c is redundant in Σ iff $\Sigma \setminus \{c\} \cup \{\neg c\} \models \perp$. We thus strengthen this refutation procedure by replacing \models by \models_{UP} in order to get an incomplete but polynomial-time redundancy checking approach.

Definition 3.

Let Σ be a SAT instance and let $c \in \Sigma$, c is UP-redundant in Σ if and only if $\Sigma \setminus \{c\} \models_{UP} c$.

Accordingly, checking the UP-redundancy of c in Σ amounts to propagate the opposite of every literal of c throughout $\Sigma \setminus \{c\}$.

Let us consider Example 1, as depicted below. In this example, it is easy to show that $w \vee x$ is UP-redundant in Σ , while it is not subsumed in Σ . Let us consider $\Sigma \setminus \{w \vee x\} \wedge \neg w \wedge \neg x$. Clearly, $w \vee \neg y$ and $x \vee \neg z$ reduce to $\neg y \wedge \neg z$, respectively. Propagating these two literals generates a contradiction, showing that $w \vee x$ is UP-redundant in Σ . On the other hand, $w \vee x$ is clearly not subsumed in Σ since there is no other clause $c' \in \Sigma$ s.t. $c' \subseteq c$.

Example 1.

$$\Sigma = \begin{cases} w \vee x \\ y \vee z \\ w \vee \neg y \\ x \vee \neg z \\ \dots \end{cases}$$

Accordingly, in Algorithm 2, a (basic) UP pre-treatment is described and can be motivated as follows. In the general case, there exists a possibly exponential number of different sets of irredundant formulas that can be extracted from the initial instance. Indeed, irredundancy and minimally inconsistency coincide on unsatisfiable formulas [7]. Clearly, the specific resulting simplified instance delivered by the Algorithm 1 depends on the order according to which clauses from Σ are considered. As small-size

clauses allow one to reduce the search space in a more dramatic manner than longer ones, we have implemented a policy that checks longer clauses for redundancy, first. Accordingly, this amounts to considering the clauses Σ according to their decreasing sizes.

Example 2. Let Σ be the following SAT instance:

$$\Sigma = \begin{cases} w \vee x \\ w \vee x \vee y \vee z \\ w \vee \neg y \\ x \vee \neg z \end{cases}$$

- not considering clauses according to their decreasing sizes, but starting with the $w \vee x$ clause, the resulting UP-irredundant set of clauses would be the following Σ_1 :

$$\Sigma_1 = \begin{cases} w \vee x \vee y \vee z \\ w \vee \neg y \\ x \vee \neg z \end{cases}$$

- whereas Algorithm 1, which considers $w \vee x \vee y \vee z$ first, delivers for this example a different -but same size- final set Σ_2 of clauses.

$$\Sigma_2 = \begin{cases} w \vee x \\ w \vee \neg y \\ x \vee \neg z \end{cases}$$

Starting with larger size clauses allows to obtain the smallest set of clauses in terms of number of clauses and also in terms of number of literals. We are sure that all subsumed clauses are removed in this way and only the subsuming clauses are preserved because the larger ones are first tested. As this example illustrates, subsumed clauses are removed, leading to shorter clauses in Σ_2 , which is thus more constrained and, to some extent, easier to solve.

Property 1. Let Σ be a CNF formula. If Σ' is a formula obtained from Σ by applying Algorithm 2 then Σ' is closed under subsumption.

Proof. Suppose that there exist two clauses c and c' of Σ' such that c' subsumes c . We can deduce that $|c'| \leq |c|$. As the clauses of Σ checked for UP-redundancy are ordered according to their decreasing sizes, we deduce that c is UP-redundant. Consequently, $c \notin \Sigma'$, which contradicts the hypothesis. \square

The converse of Property 1 is false. Indeed, the formula $\Sigma = \Sigma_2 \cup \{(y \vee e), (z \vee \neg e)\}$ is closed under subsumption but is not UP-irredundant.

Clearly, in the best cases, the pre-treatment could allow us to get rid of all non-polynomial clauses, and reduce the instance into a polynomial fragment. Since the size of the instances can be huge, we investigate whether polynomial fragments of \mathcal{L} should be protected from redundancy checking or not. As a comparison study, several possible fragments have been considered for UP-redundancy checking: namely Σ , all non

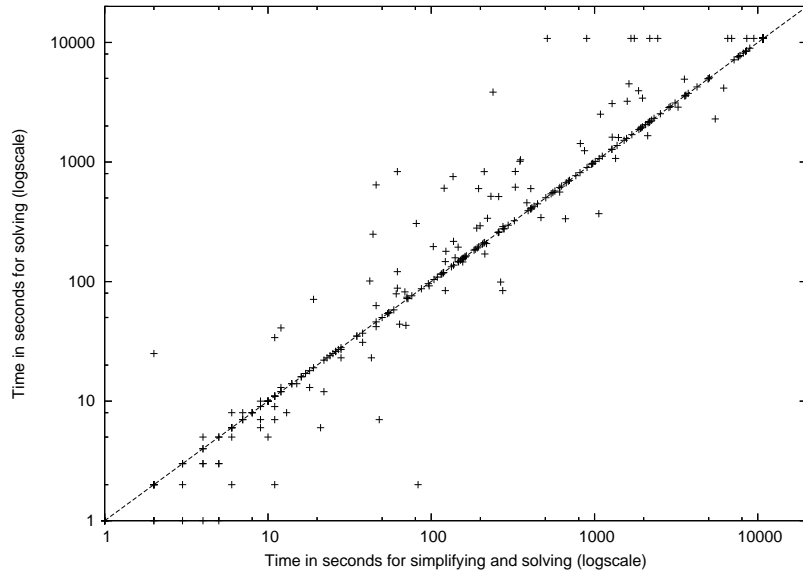


Fig. 1. Results for the 700 tested instances

Horn clauses from Σ , all non-binary clauses from Σ , and all non-Horn and non-binary clauses from Σ .

Also, we have experimented an approach trying to maximize the number of UP triggerings. The intuition is as follows. A clause that contains literals that occur in many binary clauses will lead to a cascade of UP steps. For example, let $c = x \vee y \vee z$. When x occurs in several binary clauses and when we check whether c is redundant using UP, each such binary clause will be simplified into a unit clause, recursively leading to other UP steps. Accordingly, we define a weight w associated to a clause c , noted $w(c)$ as the sum of the weights of each literal of c , where the weight of a literal is given by the number of binary clauses to which it belongs. Let us note that in order for a UP propagation step to occur when a clause $c_1 \in \Sigma$ is checked for redundancy using UP, there must be another clause $c_2 \in \Sigma$ s.t. $|c_2 - \{c_1 \cap c_2\}| = 1$. Clearly, when $|c_2 - \{c_1 \cap c_2\}| = 0$, c_1 is UP-redundant. Since computing and recording this necessary condition can be resource-consuming, we have implemented a more restrictive and easier-to-compute criterion based on the aforementioned weights. When c_2 is a binary clause, the previous condition is satisfied if and only if c_1 (to be checked for redundancy) contains a literal from c_2 . Accordingly, when $w(c) = 0$, c is not checked for redundancy. This weight-heuristic has been mixed with a policy allowing clauses from polynomial classes (binary, Horn, binary and Horn) to be protected from redundancy checking.

4 Experimental results

We have experimented the aforementioned UP-based pre-treatment extensively on the last SAT competitions benchmarks (<http://www.satcompetition.org>). We have tested more than 700 SAT instances that stem from various domains (industrial, random, hand-made, graph-coloring, etc.). Three of the winners of the last three competitions, namely ZChaff, Minisat and SatElite have been selected as SAT solvers. All experiments have been conducted on Pentium IV, 3Ghz under linux Fedora Core 4. The complete list of our experimental data and results are available at:

http://www.cril.fr/~fourdrinoy/eliminating_redundant_clauses.html

First, we have run the three SAT solvers on all benchmarks, collecting their computing times to solve each instance. A time-out was set to 3 hours. Then, we have run the UP pre-treatment on those benchmarks and collected both the simplification run-times and the subsequent run-times spent by each of the aforementioned solvers on the simplified instances. More precisely, we have experimented a series of different forms of UP pre-treatment. In the first one, we have applied the UP-redundancy removing technique on all clauses. In the other ones, non-binary, non Horn clauses have been targeted, successively. We have also targeted clauses that are neither Horn nor binary. Finally, all those experimentations have been replayed using the additional triggering heuristic $w(c) > 0$.

In Fig.1, we show the gain on all 700 tested instances. On the x -axis, we represent the time for simplifying and solving an instance with its best policy. On the y -axis the best time for solving the initial -not yet simplified- instance is given. Accordingly, the line of centers represents the borderline of actual gain. Instances that are above the line of centers benefit from the simplification policy. Clearly, this figure shows that our technique is best employed for difficult instances requiring large amounts of CPU-time to solve them. Indeed, for those instances we obtain significant gains most often. In particular, all the dots horizontally aligned at 10000 seconds on the y -axis represent SAT instances that can not be solved -by no solver- without UP-redundant simplification.

Instances	short name	#C	#V
gensys-icl004.shuffled-as.sat05-3825.cnf	gensys	15960	2401
rand_net60-30-5.miter.shuffled.cnf	rand_net	10681	3000
f3-b25-s0-10.cnf	f3-b25	12677	2125
ip50.shuffled-as.sat03-434.cnf	ip50	214786	66131
f2clk_40.shuffled-as.sat03-424.cnf	f2clk	80439	27568
f15-b3-s3-0.cnf	f15-b3	469519	132555
IBM_FV_2004_rule_batch_23_SAT_dat.k45.cnf	IBM_k45	381355	92106
IBM_FV_2004_rule_batch_22_SAT_dat.k70.cnf	IBM_k70	327635	63923
f15-b2-s2-0.cnf	f15-b2	425316	120367
IBM_FV_2004_rule_batch_22_SAT_dat.k75.cnf	IBM_k75	979230	246053
okgen-c2000-v400-s553070738-553070738.cnf	okgen	2000	400

Table 1. Some typical instances

Not surprisingly, our experiments show us that applying the simplification method on all clauses is often more time-consuming than focusing on non-polynomial clauses only and delivers the smallest simplified instances. However, this gain in size does not lead to a systematic run-time gain in solving the instances, including the simplification time. Indeed, these polynomial clauses might allow an efficient resolution of these instances.

Globally, our experiments show us that the best policy consists in applying the weight-based heuristic on all clauses.

In Tables 1 to 5, a more detailed typical sample of size-reduction of instances by the several aforementioned methods is provided. In Table 1, we provide for each instance its numbers of clauses ($\#C$) and variables ($\#V$) and a short name to facilitate the presentation of results. In Table 2, the CPU time in seconds (T_s) needed to simplify the instance is given, together with the obtained size reduction, expressed in number of clauses ($\#c_r$) and expressed in percents. In the second column results are given for a policy that considers all clauses for simplification. In the next ones *Horn*, *Bin*, *Horn&Bin* represent the classes that are protected from simplification. The last columns provide the results for the same policies augmented with the weight heuristics.

In Tables 3 to 5, satisfiability checking run-times are provided for the same instances, using Zchaff, Minisat and SatElite, respectively. TO means “time-out”. In the first column, T_b is the CPU-time to solve the initial instance. In the subsequent columns, the CPU-time to solve the simplified instance (T_r) is given together with the efficiency gains with respect to satisfiability checking: $\%_p$ and $\%_g$ being the gains without and with taking the simplification run-time into account. The symbol ∞ (resp. $-\infty$) represents the gain when the pre-processor-less (resp. pre-processor-based) approach fails to deliver a result while the pre-processor-based (resp. pre-processor-less) succeeds. When both approaches fail, the gain is represented by $-$.

5 Related work

Introducing a fast -polynomial time- pre-processing step inside logically complete SAT solvers is not a new idea by itself. Mainly, C-SAT [15] was provided with a pre-processor that made use a form of bounded resolution procedure computing all resolvents whose size are smaller than the size of their parents. At that time C-SAT was the most powerful solver to solve random k -SAT instances. Satz uses the same technique, but with a resolvent size limited to three [16]. Recently, Niklas Eén and Armin Biere have introduced a variable elimination technique with subsumption, self-subsuming resolution and variable elimination by substitution [17] as a pre-processing step for modern SAT solvers, extending a previous pre-processor NiVER by [18]. Another polynomial-time preprocessor has been introduced by James Crawford and is available in [19]. In [20], an algorithm is described that maintains a subsumption-free CNF clauses database by efficiently detecting and removing subsumption as the clauses are being added. An interesting path for future research would consist in comparing our approach with those other pre-processors from both theoretical and experimental points of view.

Due to its linear-time character, the unit propagation algorithm has been exploited in several ways in the context of SAT, in addition to being a key component of DPLL-like

instances	No Restriction		Horn		Binary		Horn& Binary	
	T_s	$\#c_r(\%)$	T_s	$\#c_r(\%)$	T_s	$\#c_r(\%)$	T_s	$\#c_r(\%)$
gensys	1.26	2861(17.92)	1.18	2208(13.83)	1.21	2560(16.04)	1.13	2171(13.60)
rand_net	1.80	608(5.69)	0.62	178(1.66)	0.62	0(0)	0.30	0(0)
f3-b25	1.66	1502(11.84)	1.04	926(7.30)	1.64	1502(11.84)	1.03	926(7.30)
ip50	65.06	1823(0.84)	22.02	504(0.23)	14.11	194(0.09)	9.10	110(0.05)
f2clk	6.39	344(0.42)	2.08	119(0.14)	1.29	77(0.09)	0.75	54(0.06)
f15-b3	359.52	55816(11.88)	116.73	14167(3.01)	55.38	1010(0.21)	37.62	640(0.13)
IBM.k45	53.26	32796(8.59)	10.33	2122(0.55)	6.22	717(0.18)	5.03	715(0.18)
IBM.k70	36.68	22720(6.93)	5.36	4628(1.41)	3.81	0(0)	2.78	0(0)
f15-b2	306.06	50717(11.92)	100.14	12909(3.03)	47.74	979(0.23)	34.00	609(0.14)
IBM.k75	172.47	116841(11.93)	40.14	5597(0.57)	24.64	3912(0.39)	22.34	3911(0.39)
okgen	0.00	0(0)	0.00	0(0)	0.00	0(0)	0.00	0(0)

Weighting $w(c) > 0$

	No Restriction		Horn		Binary		Horn & Binary	
	T_s	$\#c_r(\%)$	T_s	$\#c_r(\%)$	T_s	$\#c_r(\%)$	T_s	$\#c_r(\%)$
gensys	0.65	2560(16.04)	0.63	2171(13.60)	0.64	2560(16.04)	0.64	2171(13.60)
rand_net	1.66	514(4.81)	0.58	148(1.38)	0.62	0(0)	0.30	0(0)
f3-b25	0.21	60(0.47)	0.15	44(0.34)	0.21	60(0.47)	0.14	44(0.34)
ip50	53.95	1823(0.84)	19.43	504(0.23)	14.24	194(0.09)	9.21	110(0.05)
f2clk	6.06	267(0.33)	1.96	100(0.12)	1.30	77(0.09)	0.80	54(0.06)
f15-b3	229.84	24384(5.19)	83.46	6393(1.36)	55.69	1010(0.21)	37.72	640(0.13)
IBM.k45	34.53	11049(2.89)	10.36	2122(0.55)	6.23	717(0.18)	4.98	715(0.18)
IBM.k70	15.25	11464(3.49)	5.36	4616(1.40)	3.77	0(0)	2.77	0(0)
f15-b2	209.55	22217(5.22)	77.22	5709(1.34)	51.04	979(0.23)	33.32	609(0.14)
IBM.k75	125.26	39640(4.04)	38.15	5597(0.57)	26.02	3912(0.39)	22.32	3911(0.39)
okgen	0	0(0)	0.00	0(0)	0.00	0(0)	0.00	0(0)

Table 2. Simplification time and size reduction

procedures. For example, C-SAT and Satz used a local treatment during important steps of the exploration of the search space, based on UP, to derive implied literals and detect local inconsistencies, and guide the selection of the next variable to be assigned [15, 16]. In [21], a double UP schema is explored in the context of SAT solving. In [22, 8], UP has been used as an efficient tool to detect functional dependencies in SAT instances. The UP technique has also been exploited in [23] in order to derive subclauses by using the UP implication graph of the SAT instance, and speed up the resolution process.

Bailleux, Roussel and Boufkhad have studied how clauses redundancy affects the resolution of random k -sat instances [24]. However, their study is specific to random instances and cannot be exported to real-world ones. Moreover, they have explored redundancy and not UP-redundancy; their objective being to measure the redundancy degree of random 3-SAT instances at the crossover point. From a complexity point of view, a full study of redundancy in the Boolean framework is given in [25].

To some extent, our approach is also close to compilation techniques [26–29] where the goal is to transform the Boolean instances into equivalent albeit *easier* ones to check or to infer from. The idea is to allow much time to be spent in the pre-processing step, transforming the instance into a polynomial-size new instance made of clauses belonging polynomial-time fragments of \mathcal{L} , only. Likewise, our approach aims to reduce the size of the non-polynomial fragments of the instances. However, it is a partial reduction since all clauses belonging to non-polynomial fragments are not necessary removed. Moreover, whereas compilation techniques allow a possibly exponential time

instances	T_b	No Restriction $T_r(\%p, \%g)$	Horn $T_r(\%p, \%g)$	Binary $T_r(\%p, \%g)$	Horn & Binary $T_r(\%p, \%g)$
gensys	(3418.17)	2847.1(16.70,16.66)	3353.69(1.88,1.85)	1988.37(41.82,41.79)	3683.58(-7.76,-7.79)
rand_net	(1334.19)	942.15(29.38,29.24)	1067.01(20.02,19.97)	1334.19(0,-0.04)	1334.19(0,-0.02)
f3-b25	(790.96)	137.26(82.64,82.43)	155.32(80.36,80.23)	134.91(82.94,82.73)	157.44(80.09,79.96)
ip50	(2571.18)	675.11(73.74,71.21)	474.64(81.53,80.68)	1023.82(60.18,59.63)	1945.87(24.31,23.96)
f2clk	(6447.19)	4542.32(29.54,29.44)	9457.25(-46.68,-46.72)	3978.14(38.29,38.27)	3848.02(40.31,40.30)
f15-b3	(7627.95)	5620.25(26.32,21.60)	2926.38(61.63,60.10)	10157.9(-33.16,-33.89)	2419.52(68.28,67.78)
IBM.k45	(5962.46)	2833.1(52.48,51.59)	3656.05(38.68,38.50)	3244.61(45.58,45.47)	4751.79(20.30,20.22)
IBM.k70	(TO)	514.83(∞, ∞)	5377.91(∞, ∞)	TO(-,-)	TO(-,-)
f15-b2	(TO)	2287.73(∞, ∞)	8891.1(∞, ∞)	TO(-,-)	3969.27(∞, ∞)
IBM.k75	(TO)	TO(-,-)	TO(-,-)	TO(-,-)	TO(-,-)
okgen	(1309.66)	1309.66(0,-0.00)	1309.66(0,-0.00)	1309.66(0,-0.00)	1309.66(0,-0.00)

Weighting $w(c) > 0$

instances	T_b	No Restriction $T_r(\%p, \%g)$	Horn $T_r(\%p, \%g)$	Binary $T_r(\%p, \%g)$	Horn & Binary $T_r(\%p, \%g)$
gensys	(3418.17)	1986.98(41.87,41.85)	3896.93(-14.00,-14.02)	1967.22(42.44,42.42)	3873.89(-13.33,-13.35)
rand_net	(1334.19)	555.13(58.39,58.26)	614.75(53.92,53.87)	1334.19(0,-0.04)	1334.19(0,-0.02)
f3-b25	(790.96)	839.54(-6.14,-6.16)	811.37(-2.58,-2.59)	791.97(-0.12,-0.15)	813.05(-2.79,-2.81)
ip50	(2571.18)	708.81(72.43,70.33)	465.13(81.90,81.15)	1091.54(57.54,56.99)	1958(23.84,23.48)
f2clk	(6447.19)	5196.02(19.40,19.31)	5766.78(10.55,10.52)	4042.8(37.29,37.27)	3965.68(38.48,38.47)
f15-b3	(7627.95)	TO(- $\infty, -\infty$)	TO(- $\infty, -\infty$)	10024(-31.41,-32.14)	2448.27(67.90,67.40)
IBM.k45	(5962.46)	4447.15(25.41,24.83)	3698.1(37.97,37.80)	3283.2(44.93,44.83)	4925.58(17.39,17.30)
IBM.k70	(TO)	4131.58(∞, ∞)	5564.5(∞, ∞)	TO(-,-)	TO(-,-)
f15-b2	(TO)	4456.24(∞, ∞)	2880.15(∞, ∞)	TO(-,-)	4028.04(∞, ∞)
IBM.k75	(TO)	TO(-,-)	TO(-,-)	TO(-,-)	TO(-,-)
okgen	(1309.66)	1309.66(0,0)	1309.66(0,-0.00)	1309.66(0,-0.00)	1309.66(0,-0.00)

Table 3. Zchaff results

instances	T_b	No Restriction $T_r(\%p, \%g)$	Horn $T_r(\%p, \%g)$	Binary $T_r(\%p, \%g)$	Horn & Binary $T_r(\%p, \%g)$
gensys	(7543.42)	4357.66(42.23,42.21)	7078.84(6.15,6.14)	4722.35(37.39,37.38)	7370.48(2.29,2.27)
rand_net	(41.15)	11.00(73.26,68.87)	35.12(14.66,13.14)	41.15(0,-1.52)	41.15(0,-0.73)
f3-b25	(755.90)	225.45(70.17,69.95)	246.97(67.32,67.18)	233.73(69.07,68.86)	243.39(67.80,67.66)
ip50	(88.43)	61.95(29.94,-43.62)	138.90(-57.06,-81.97)	64.47(27.09,11.13)	60.13(31.99,21.70)
f2clk	(280.88)	290.41(-3.39,-5.66)	188.53(32.87,32.13)	325.87(-16.01,-16.48)	274.77(2.17,1.90)
f15-b3	(875.37)	531.31(39.30,-1.76)	561.40(35.86,22.53)	759.43(13.24,6.91)	555.47(36.54,32.24)
IBM.k45	(3940.82)	3729.01(5.37,4.02)	3568.43(9.44,9.18)	3625.13(8.01,7.85)	3541.57(10.13,10.00)
IBM.k70	(643.67)	76.16(88.16,82.46)	527.58(18.03,17.20)	643.67(0,-0.59)	643.67(0,-0.43)
f15-b2	(516.36)	334.07(35.30,-23.96)	257.93(50.04,30.65)	452.94(12.28,3.03)	369.69(28.40,21.81)
IBM.k75	(4035.72)	5096.73(-26.29,-30.56)	6324.08(-56.70,-57.69)	5782.49(-43.28,-43.89)	5534.54(-37.13,-37.69)
okgen	(46.43)	46.43(0,-0.02)	46.43(0,-0.01)	46.43(0,-0.01)	46.43(0,-0.01)

Weighting $w(c) > 0$

instances	T_b	No Restriction $T_r(\%p, \%g)$	Horn $T_r(\%p, \%g)$	Binary $T_r(\%p, \%g)$	Horn & Binary $T_r(\%p, \%g)$
gensys	(7543.42)	4742.55(37.12,37.12)	7434.11(1.44,1.44)	4615.46(38.81,38.80)	7610.16(-0.88,-0.89)
rand_net	(41.15)	27.00(34.38,30.33)	25.60(37.79,36.36)	41.15(0,-1.52)	41.15(0,-0.73)
f3-b25	(755.90)	745.80(1.33,1.30)	738.83(2.25,2.23)	773.19(-2.28,-2.31)	779.37(-3.10,-3.12)
ip50	(88.43)	54.28(38.61,-22.39)	138.28(-56.35,-78.33)	67.17(24.04,7.93)	52.71(40.39,29.97)
f2clk	(280.88)	224.82(19.95,17.79)	221.20(21.24,20.54)	299.23(-6.53,-6.99)	289.93(-3.22,-3.50)
f15-b3	(875.37)	543.86(37.87,11.61)	687.06(21.51,11.97)	751.79(14.11,7.75)	498.80(43.01,38.70)
IBM.k45	(3940.82)	1826.6(53.64,52.77)	3324.82(15.63,15.36)	3637.35(7.70,7.54)	3714.29(5.74,5.62)
IBM.k70	(643.67)	31.51(95.10,92.73)	518.48(19.44,18.61)	643.67(0,-0.58)	643.67(0,-0.43)
f15-b2	(516.36)	378.88(26.62,-13.95)	155.70(69.84,54.89)	483.39(6.38,-3.49)	371.51(28.05,21.59)
IBM.k75	(4035.72)	4202.16(-4.12,-7.22)	5782.1(-43.27,-44.21)	5927.94(-46.88,-47.53)	5655.36(-40.13,-40.68)
okgen	(46.43)	46.43(0,0)	46.43(0,-0.00)	46.43(0,-0.00)	46.43(0,-0.00)

Table 4. Minisat results

instances	T_b	No Restriction $T_r(\%_p, \%_g)$	Horn $T_r(\%_p, \%_g)$	Binary $T_r(\%_p, \%_g)$	Horn & Binary $T_r(\%_p, \%_g)$
gensys	(5181.22)	4672.43(9.81,9.79)	7879.9(-52.08,-52.10)	5146.42(0.67,0.64)	7964.11(-53.71,-53.73)
rand_net	(131.01)	173.97(-32.79,-34.16)	110.02(16.01,15.54)	131.01(0,-0.48)	131.01(0,-0.23)
f3-b25	(1664.94)	2935.61(-76.31,-76.41)	1926.54(-15.71,-15.77)	2934.61(-76.25,-76.35)	1924.24(-15.57,-15.63)
ip50	(79.45)	110.37(-38.91,-120.79)	143.30(-80.36,-108.08)	150.33(-89.20,-106.97)	58.31(26.60,15.15)
f2clk	(323.15)	270.61(16.25,14.28)	291.93(9.66,9.01)	453.03(-40.19,-40.59)	518.12(-60.33,-60.56)
15-b3	(833.17)	244.23(70.68,27.53)	976.24(-17.17,-31.18)	281.31(66.23,59.58)	760.59(8.71,4.19)
IBM.k45	(7829.02)	4111.24(47.48,46.80)	4382.78(44.01,43.88)	3457.46(55.83,55.75)	3314.55(57.66,57.59)
IBM.k70	(712.10)	225.22(68.37,63.22)	757.96(-6.43,-7.19)	712.10(0,-0.53)	712.10(0,-0.39)
f15-b2	(794.85)	261.70(67.07,28.56)	575.30(27.62,15.02)	556.98(29.92,23.91)	441.52(44.45,40.17)
IBM.k75	(2877.51)	3714.45(-29.08,-35.07)	4099.39(-42.46,-43.85)	5505.93(-91.34,-92.20)	3796.42(-31.93,-32.71)
okgen	(323.06)	323.06(0,-0.00)	323.06(0,-0.00)	323.06(0,-0.00)	323.06(0,-0.00)

Weighting $w(c) > 0$

instances	T_b	No Restriction $T_r(\%_p, \%_g)$	Horn $T_r(\%_p, \%_g)$	Binary $T_r(\%_p, \%_g)$	Horn & Binary $T_r(\%_p, \%_g)$
gensys	(5181.22)	5052.73(2.47,2.46)	8046.2(-55.29,-55.30)	5163.37(0.34,0.33)	8231.05(-58.86,-58.87)
rand_net	(131.01)	63.00(51.91,50.63)	130.23(0.59,0.14)	131.01(0,-0.47)	131.01(0,-0.23)
f3-b25	(1664.94)	1701.57(-2.20,-2.21)	1698.4(-2.00,-2.01)	1745.22(-4.82,-4.83)	1723.25(-3.50,-3.51)
ip50	(79.45)	113.38(-42.70,-110.61)	136.52(-71.82,-96.28)	146.68(-84.61,-102.54)	60.46(23.90,12.30)
f2clk	(323.15)	313.56(2.96,1.09)	238.38(26.23,25.62)	466.84(-44.46,-44.86)	503.75(-55.88,-56.13)
15-b3	(833.17)	675.88(18.87,-8.70)	1276.87(-53.25,-63.27)	271.53(67.40,60.72)	733.52(11.96,7.43)
IBM.k45	(7829.02)	5836.14(25.45,25.01)	4364.69(44.24,44.11)	3341(57.32,57.24)	3311.19(57.70,57.64)
IBM.k70	(712.10)	406.32(42.94,40.79)	740.12(-3.93,-4.68)	712.10(0,-0.53)	712.10(0,-0.38)
f15-b2	(794.85)	316.32(60.20,33.83)	293.82(63.03,53.31)	549.23(30.90,24.47)	469.28(40.95,36.76)
IBM.k75	(2877.51)	3121.35(-8.47,-12.82)	4017.92(-39.63,-40.95)	5170.69(-79.69,-80.59)	3738.34(-29.91,-30.69)
okgen	(323.06)	323.06(0,0)	323.06(0,-0.00)	323.06(0,-0.00)	323.06(0,-0.00)

Table 5. SatElite results

to be spent in the pre-processing step, we make sure that our pre-processing technique remains a polynomial-time one.

6 Conclusions

Eliminating redundant clauses in SAT instances during a pre-treatment step in order to speed up the subsequent satisfiability checking process is a delicate matter. Indeed, redundancy checking is intractable in the worst case, and some redundant information can actually help to solve the SAT instances more efficiently. In this paper, we have thus proposed and experimented a two-levels trade-off. We rely on the efficiency albeit incomplete character of the unit propagation algorithm to get a fast pre-treatment that allows some -but not all- redundant clauses to be detected. We have shown from an experimental point of view the efficiency of a powerful weight-based heuristics for redundancy extraction under unit propagation. Such a pre-treatment can be envisioned as a compilation process that allows subsequent faster operations on the instances. Interestingly enough, the combined computing time spent by such a pre-treatment and the subsequent SAT checking often outperforms the SAT checking time for the initial instance on very difficult instances.

This piece of research opens other interesting perspectives. For example, such a pre-processing step can play a useful role in the computation of minimally inconsistent subformulas (MUSes) [30]. Also, we have focused on binary and Horn fragments

as polynomial fragments. Considering other fragments like e.g. the reverse Horn and renamable Horn could be a fruitful path for future research.

Acknowledgments

This research has been supported in part by the EC under a Feder grant and by the Région Nord/Pas-de-Calais.

References

1. Selman, B., Levesque, H.J., Mitchell, D.G.: A new method for solving hard satisfiability problems. In: Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92). (1992) 440–446
2. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Communications of the ACM **5**(7) (1962) 394–397
3. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC'01). (2001) 530–535
4. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03). (2003) 502–518
5. Dubois, O., Dequen, G.: A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01). (2001) 248–253
6. Williams, R., Gomes, C.P., Selman, B.: Backdoors to typical case complexity. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03). (2003) 1173–1178
7. Liberatore, P.: The complexity of checking redundancy of CNF propositional formulae. In: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI'02). (2002) 262–266
8. Grégoire, É., Ostrowski, R., Mazure, B., Saïs, L.: Automatic extraction of functional dependencies. In: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04). (2004) 122–132
9. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, New York (USA), Association for Computing Machinery (1971) 151–158
10. Tarjan, R.E.: Depth first search and linear graph algorithms. *SIAM J. Comput.* **1** (1972) 146–160
11. Even, S., Itai, A., Shamir, A.: On the complexity of timetable and multicommodity flow problems. *SIAM J. Comput.* **5** (1976) 691–703
12. Dowling, W.H., Gallier, J.H.: Linear-time algorithms for testing satisfiability of propositional horn formulae. *Journal of Logic Programming* **1**(3) (1984) 267–284
13. Wei, W., Selman, B.: Accelerating random walks. In: Proceedings of 8th International Conference on the Principles and Practices of Constraint Programming (CP'2002). (2002) 216–232
14. Kautz, H.A., Ruan, Y., Achlioptas, D., Gomes, C.P., Selman, B., Stickel, M.E.: Balance and filtering in structured satisfiable problems. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01). (2001) 351–358

15. Dubois, O., André, P., Boufkhad, Y., Carlier, Y.: SAT vs. UNSAT. In: Second DIMACS implementation challenge: cliques, coloring and satisfiability. Volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (1996) 415–436
16. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97). (1997) 366–371
17. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05). (2005) 61–75
18. Subbarayan, S., Pradhan, D.K.: NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. In: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04). (2004) 276–291
19. Crawford, J.: A polynomial-time preprocessor ("compact") (1996) <http://www.cirl.uoregon.edu/crawford/>.
20. Zhang, W.: Configuration landscape analysis and backbone guided local search: Part i: Satisfiability and maximum satisfiability. *Artificial Intelligence* **158**(1) (2004) 1–26
21. Le Berre, D.: Exploiting the real power of unit propagation lookahead. In: Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT'01), Boston University, Massachusetts, USA (2001)
22. Ostrowski, R., Mazure, B., Saïs, L., Grégoire, É.: Eliminating redundancies in SAT search trees. In: Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'2003), Sacramento (2003) 100–104
23. Darras, S., Dequen, G., Devendeville, L., Mazure, B., Ostrowski, R., Saïs, L.: Using Boolean constraint propagation for sub-clauses deduction. In: Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05). (2005) 757–761
24. Boufkhad, Y., Roussel, O.: Redundancy in random SAT formulas. In: Proceedings of the 17th National Conference on Artificial Intelligence (AAAI'00). (2000) 273–278
25. Liberatore, P.: Redundancy in logic i: CNF propositional formulae. *Artificial Intelligence* **163**(2) (2005) 203–232
26. Selman, B., Kautz, H.A.: Knowledge compilation using horn approximations. In: Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91). (1991) 904–909
27. del Val, A.: Tractable databases: How to make propositional unit resolution complete through compilation. In: Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94). (1994) 551–561
28. Marquis, P.: Knowledge compilation using theory prime implicates. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95), Montréal, Canada (1995) 837–843
29. Mazure, B., Marquis, P.: Theory reasoning within implicant cover compilations. In: Proceedings of the ECAI'96 Workshop on Advances in Propositional Deduction, Budapest, Hungary (1996) 65–69
30. Grégoire, É., Mazure, B., Piette, C.: Extracting MUSes. In: Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06), Trento, Italy (2006) 387–391