Vivifying Propositional Clausal Formulae

Cédric PIETTE[1] and **Youssef HAMADI**[2] and **Lakhdar SAÏS**[1]

Abstract. In this paper, we present a new way to preprocess Boolean formulae in Conjunctive Normal Form (CNF). In contrast to most of the current pre-processing techniques, our approach aims at improving the filtering power of the original clauses while producing a small number of additional and relevant clauses. More precisely, an incomplete redundancy check is performed on each original clauses through unit propagation, leading to either a sub-clause or to a new relevant one generated by the clause learning scheme. This preprocessor is empirically compared to the best existing one in terms of size reduction and the ability to improve a state-of-the-art satisfiability solver.

1 INTRODUCTION

Since a few years, preliminary computations on CNF formulae have been more and more studied by the SAT community. This renewal of interest can be explained by different factors. First, reducing the huge size of the SAT instances encoding real world problems increases the robustness of SAT solvers. Secondly, these instances contain different kinds of structures that can be handled more efficiently before search.

One of the most effective preprocessing techniques (`SatElite`) is currently integrated in state-of-the-art SAT solvers such as `Minisat` and `Rsat`. It is now well acknowledged that the performances of these solvers is usually greatly improved by this particular preprocessing, up to the point where `SatElite` is often used by SAT competitors.

Thus, preprocessing a formula before solving is now known as an important step, and a lot of preprocessors have already been proposed. One of the first and efficient preprocessing algorithm, called `3-Resolution` was incorporated to the `Satz` solver [10]. It consists in adding to the formula all resolvent clauses of size less or equal to 3, until saturation. `2-SIMPLIFY`, a less computationally heavy preprocessor was proposed in [2]. It has been developed to better manage real-world benchmarks, which often contain a lot of binary clauses. Roughly, the idea is thus to use those binary clauses to construct an implication graph, from which unit clauses can be deduced by computing the transitive closure. If unit clauses have been obtained, they are propagated and this process is iterated until a fix point is reached. Later on, `HyPre` generalized `2-SIMPLIFY` by computing hyper-binary resolution to deduce new binary clauses [1]. Moreover, `HyPre` is able to detect and substitute equivalent literals incrementally. The classical DP procedure, based on variable elimination through resolution, has also been used in a limited way as a preprocessing step. A weaker schema has been adopted by the `NiVER` procedure [13]. This one eliminates variables by resolution if this computation does not increase the number of literals of the CNF formula. `NiVER` has been improved later by a so-called substitution rule, together with the use of clause signatures and touched lists to define the recent `SatElite` preprocessor [6].

However, only preprocessors that eliminate variables by a limited application of resolution are now grafted to modern SAT solvers. Indeed, the other kinds of preprocessors aim at modifying the CNF formula with some addition and/or removal of clauses, keeping generally the same set of variables. The main problem of these preprocessors is that it is difficult to measure the relevance of each added or eliminated clause with respect to the resolution step. One can eliminate clauses and can derive an harder sub-formula. Similarly, adding new clauses might lead to an increase in the space complexity, without reducing the search space. Indeed, the added clauses can only clutter the solver by creating redundant information.

In this paper, we revisit this kind of preprocessing, using only forms of resolution that aims at substituting existing clauses by more constrained ones. In other words, our main goal is to strengthen, or *to vivify*, the redundant clauses from the original formula. To this end, we apply a limited check of redundancy on each clause of the CNF formula in order to derive or to approximate one of its minimally redundant sub-clauses. Interestingly, our proposed approach can also take advantage of modern learning scheme to produce new resolvents that are conditionally added to the formula.

This paper is organized as follow: in the next section basic notations and definitions about propositional clausal formulae and SAT are provided. In section 3, different simplification techniques and their practical usefulness are discussed. Next, particular forms of resolution hidden by unit propagation are presented, and an incomplete method which can produce them is presented. The resulting preprocessor is detailed and evaluated in section 4. Finally, we conclude the paper by some perspectives and further works.

2 DEFINITIONS AND NOTATIONS

We briefly state here some definitions and notations used in the rest of this paper. A propositional formula is in conjunctive normal form (CNF for short), if it can be represented using a set (interpreted as a conjunction) of clauses, where a clause is a set (interpreted as a disjunction) of literals, a literal being a propositional variable, or its negation. The set of variables that appear in a CNF formula Σ will be denoted

[1] Université Lille-Nord de France, Artois, CRIL-CNRS UMR 8188, F-62307 Lens, email: {piette,sais}@cril.fr

[2] Microsoft Research, 7 J J Thomson Avenue, Cambridge, United Kingdom email: youssefh@microsoft.com

$Var(\Sigma)$. $Lit(\Sigma)$ is defined as the set $\{x, \neg x | x \in Var(\Sigma)\}$. For a set of literals L, $\bar{L}$ is defined as $\{\bar{l} | l \in L\}$.

An *interpretation* ρ of a CNF formula Σ is an application from $Var(\Sigma)$ to the set of truth values $\{true, false\}$. It is called a *model* iff it provides the value *true* to Σ (in short $\rho \models \Sigma$). SAT is the problem of deciding whether a given CNF formula admits a model, or not.

Let $c_a = \{l_{a_1}, \dots, l_{a_n}, l\}$ and $c_b = \{l_{b_1}, \dots, l_{b_m}, \neg l\}$ be two clauses. The clause $c = \{l_{a_1}, \dots, l_{a_n}, l_{b_1}, \dots, l_{b_m}\}$ is a logical consequence (called *resolvent*) of c_a and c_b. This production rule is called *resolution* and is denoted $\otimes_R$. We note the resolvent c as $c_a \otimes_R c_b$. Most of the techniques used for solving SAT (e.g. DP-like procedure, unit propagation, learning schemes, etc.) are based on implicit or explicit application of resolution. This is clearly the case for most preprocessors, including the one presented in this paper.

Let c and c' be two clauses of Σ. We say that a clause c' (resp. c) subsumes (resp. is subsumed by) c (resp. c') iff $c' \subset c$. Subsumed clauses c can be removed from Σ while preserving satisfiability. Given $x \in Lit(\Sigma)$, we define $\Sigma|_x$, the formula simplified by the assignment of x to *true*. We recursively define $UP(\Sigma)$ as follows : (1) $UP(\Sigma) = \Sigma$ if Σ does not contain unit clauses, (2) $UP(\Sigma) = \perp$ if Σ contains two unit-clauses $\{x\}$ and $\{\neg x\}$, (3) otherwise, $UP(\Sigma) = UP(\Sigma|_x)$ with x a unit literal appearing in a unit clause of Σ. A clause c is implied by unit propagation from Σ, denoted $\Sigma \models_{UP} c$, if $UP(\Sigma|_{\bar{c}}) = \perp$.

In the next section, the main preprocessing strategies are discussed, and a limited form of resolution that produces more constrained clauses than the original ones is presented.

3 PREPROCESSING CNF FORMULAE

3.1 Adding and/or removing clauses?

Two main categories of preprocessors have been proposed: the first one aims at eliminating variables through a partial application of the DP procedure [5]. Actually, only variables which can be eliminated keeping the formula within a "reasonable" size (w.r.t. the original size) are exhaustively processed by resolution. `SatElite` belongs to this category of preprocessors. The principle of the second category is to modify the original formula by adding and/or removing clauses, usually keeping the whole set of variables. Most of the time, the production of new clauses is made by resolution. For instance, `HyPre` performs hyper resolution to produce binary clauses [1] that are added to the formula. These new clauses represent redundant information with respect to the original CNF formula, and this information seems to generally help solvers.

Recently, a new approach introduced in [7] aims at removing from a formula some of the redundant clauses, namely clauses c of Σ s.t. $\Sigma \backslash \{c\} \models c$. Obviously enough, performing such a test is computationally intractable. Therefore, this redundancy is only checked through unit propagation. As a consequence, this approach is incomplete, but it is able to remove some redundant clauses in polynomial time. As other clause-filtering techniques, the resulting preprocessor can sometimes slow down the whole resolution process because of the removal of some important redundant clauses.

The main problem with those techniques is that it is hard to characterize which redundant clauses are useful. Indeed, a tradeoff has to be made between the management of a large number of clauses, which slows down DPLL implementations, and their relevance, namely their ability to trigger propagations. Indeed, it is well-known that redundant information can actually help SAT solvers; for instance, the powerful learning scheme, which produces a particular resolvent clause after each conflict, can be viewed as a dynamic addition of redundant clauses during search. This learning strategy is now known to be one of the key features of modern solvers, which proves the interest of redundant information with respect to practical SAT resolution. Nevertheless, a simple experiment which consists in adding all learnt clauses to a CNF formula after its resolution shows that this new redundant information makes the formula generally more difficult to solve. Hence, how can we ensure that a particular clause-adding approach can effectively boost a given SAT solver?

A priori, one interesting option would consider the efficient generation of sub-clauses from the original CNF. In this way, there is neither addition nor removal of any clause, but the substitution of existing clauses by more constrained ones. In current solvers, this computation would have great advantages: it could not only increase the number of unit propagations with no more clauses to manage, but would also lead to shorter learnt clauses during the search by reducing the *reason* of the propagated literals. Several techniques have already been proposed to generate sub-clauses. For instance, it is proposed in [4] to explore the implication graph to generate resolvent clauses and to only take into account the ones which subsume at least one original clause of the CNF formula, in order to substitute this latter clause by the shorter produced one. Actually, this computation is exponential in the worst case, and a weaker polynomial version restricted to a single literal assignment is proposed. In the next section, a new approach that aims at checking more systematically whether a clause can be shortened or not, is presented.

3.2 One answer: shorten existing clauses

The way a problem is encoded in CNF is crucial for its practical resolution, and can lead to exponential differences in resources requirement. Analyzing the different kinds of modelling is now an active path of research (see e.g. [8]). However, even with "good" modelling, some clauses might be redundant. A clause is redundant if it can be inferred from the remaining part of the CNF formula. In our approach redundancy check is only used to shorten clauses by eliminating some redundant literals.

However, checking whether a clause is redundant is CoNP-complete [11]. Hence, an incomplete but linear time deduction strategy has been adopted. Indeed, this check is performed with respect to unit propagation, only. More formally, a clause c of Σ is redundant modulo unit propagation (in short $Red_{UP}(\Sigma, c)$), iff $\Sigma \backslash \{c\} \models_{UP} c$. Obviously, if $Red_{UP}(\Sigma, c')$, and $c' \subset c$, then we also have $Red_{UP}(\Sigma, c)$. The converse is not true. This observation lead us to a new definition of minimal redundancy of clauses. We say that a clause c of Σ is minimally redundant modulo UP iff $\nexists c' \subset c$ s.t. $Red_{UP}(\Sigma, c')$.

One of the main goals behind our vivification process is to find for each redundant clause, one of its minimal redundant sub-clauses. Actually, a clause checked to be shortened is removed from the CNF formula, and the opposite of its literals are assigned one by one according to their lexicographic ordering. Given a CNF formula Σ and $c = \{l_1, l_2, \ldots, l_n\}$ a clause from Σ. Assuming that the order in which the literals are assigned is $(\neg l_1, \ldots, \neg l_n)$, two possible cases may occur:

1. $\exists i \in \{1, \ldots, n-1\}$ s.t. $\Sigma \backslash \{c\} \cup \{\neg l_1, \ldots, \neg l_i\} \vDash_{UP} \bot$
   In this case, we have $\Sigma \backslash \{c\} \vDash_{UP} c'$ with $c' = (l_1 \vee \ldots \vee l_i)$ This new clause c' strictly subsumes c. Hence, the original clause can be substituted by the new deduced one. Obviously, c' is not necessarily minimally redundant modulo UP. Indeed, another ordering on the literals $\{l_1, l_2, \ldots, l_i\}$ might lead to an even shorter sub-clause. Thanks to a conflict analysis, the deduced sub-clause c' could be shortened again leading to an even smaller sub-clause. Indeed, a new clause η can be generated by a *complete* traversal of the implication graph associated to Σ and to the assignments of the literals $\{\neg l_1, \ldots, \neg l_i\}$. The complete traversal of the implication graph ensure that the clause η contains only literals from c'. Thereby, η is a sub-clause of $(l_1 \vee \ldots \vee l_i)$.
2. Otherwise, as unit propagation is performed after each assignment, if one of the remaining literals is assigned by this filtering operation, then a sub-clause is produced. Trivially, when this phenomenon occurs, the propagated literal is either assigned positively (it satisfies the removed clause of the CNF formula) or negatively (it is falsified in this clause). Considering i and j with $1 \leq i < j \leq n$, the two possible cases are:
   - $\Sigma \backslash \{c\} \cup \{\neg l_1, \ldots, \neg l_i\} \vDash_{UP} \neg l_j$
     In this case, we can deduce: $\Sigma \backslash \{c\} \vDash_{UP} (l_1 \vee \ldots \vee l_i \vee \neg l_j)$ Applying resolution between this new clause and c (using the variable l_j), we obtain:
     $(l_1 \vee \ldots \vee l_j \vee \ldots \vee l_n) \otimes_R (l_1 \vee \ldots \vee l_i \vee \neg l_j) = (l_1 \vee \ldots \vee l_{j-1} \vee l_{j+1} \vee \ldots \vee l_n)$. This new clause clearly subsumes c. Hence, the original clause can be substituted by the new deduced one.
   - $\Sigma \backslash \{c\} \cup \{\neg l_1, \ldots, \neg l_i\} \vDash_{UP} l_j$
     In this case, we can deduce: $\Sigma \backslash \{c\} \vDash_{UP} (l_1 \vee \ldots \vee l_i \vee l_j)$ In this case too, the produced clause subsumes c and enables to "remove" literals from it.

Accordingly, from the iterative assignments of the opposite literals of a clause, one reduced clause could be produced. This computation can clearly be integrated into a modern SAT solver, and benefit from lazy data structures to be performed. Moreover, during such a search, some assignments could lead to a conflict. As explained above, when this case occurs, the procedure can use the conflict analysis implemented in current solvers to produce smaller sub-clauses in a polynomial time.

Using the previous rules and the learning feature of SAT solvers, a CNF formula can be *vivified*, namely made easier to solve. In the next section, we present the practical implementation that has been made, based on the previous ideas.

4 CNF FORMULAE VIVIFICATION

4.1 Technical choices

In this section, different practical parameters are discussed, some of them resulting from extensive experiments.

First, the ideas proposed in the previous section imply to test the clauses of a formula to shorten some of them. However, if a literal is actually removed from a clause, new propagations can be performed using this clause, meaning that all the failed tests made on previous clauses could then succeed with this shortened clause. Hence, whenever a test succeeds to produce a sub-clause, all other clauses are checked again with a new *iteration* of the procedure.

Second, the presented sub-clause production technique supposes that the order in which the literals are assigned is important. Clearly, to ensure a maximal clause reduction, one has to check all possible orders of literals. However, this could lead to a pretty heavy computation; then, an incomplete strategy that consists in trying only one particular order has been adopted. Actually, a variant of the MOMS branching heuristic [9] is used to sort the literals in order to maximize the number of implied literals by unit propagation.

Yet, using only this heuristic makes the order very similar from one iteration to the other. As said previously, a clause is tested again only if at least one other clause has been shortened. However, keeping only the MOMS ordering does not appear as a good solution, because the procedure would not benefit of the potential multiple iterations made on each clause. To diversify the search, some randomization is used as follows: assuming that the literals of a clause are sorted with respect to MOMS, two of them are selected randomly and are exchanged in this ordering.

Finally, when a conflict occurs, the tested clause $c = (l_1 \vee \ldots \vee l_n)$ is substituted by its sub-clause $c' = (l_1 \vee \ldots \vee l_i)$. As mentioned above, a complete traversal of the implication graph could lead to an even more reduced clause, but for efficiency purposes, this computation is not performed. In our implementation, classical learning scheme is used to generate a nogood η corresponding to the first UIP. If this new clause η subsumes the sub-clause c', then c' is now substituted by η; otherwise η is only added to the formula if its size (in term of number of literals) is strictly less than the size of the original clause. As the results show, this strategy only adds a few number of nogoods ($< 5\%$ of the number of original clauses), which prove useful for the future exhaustive search.

Considering these choices, a new polynomial preprocessor called `ReVivAl` (for pReprocessing based on Vivification Algorithm) has been developed. This method is described in the Algorithm 1. Roughly, for each clause c of an input CNF Σ, c is removed from Σ and the opposite of each literal is assigned alternatively with unit propagation (loop from line 5 to 29). Moreover, different checks about the remaining literals (that "should" be unassigned) and the presence of a conflict are performed, as presented in section 3.2 (tests on lines 11, 13, 17, 19, 23 and 27). The order in which the literals are selected for assignment is given by the function *select_a_literal* which just selects the highest literal with respect to our randomized MOMS-like score, where two randomly chosen literals have their score reversed. As long as one of the clauses has been reduced (*change* set to *true*), the process continues with all the other clauses. Let us note that our implementation has been integrated into a modern SAT solver, which enables the use of most recent data structures and mecanisms designed for SAT resolution. Hence, the redundancy test of each clause, performed by a serie of assignments, takes advantage of the efficiency of *watched literals*. In the same way, the conditional add of clauses is achieved through the "classical" learning functions, usually called by the solver after each conflict. Exploiting those structures and techniques implemented into exhaustive methods not only leads to an easy

Algorithm 1: Vivification of a CNF formula

```
Input: $\Sigma$ : a CNF formula
Output: a vivified CNF formula
1  begin
2      change $\longleftarrow$ $true$; ;
3      while change do
4          change $\longleftarrow$ $false$ ;
5          foreach $c \in \Sigma$ do
6              $\Sigma \longleftarrow \Sigma \backslash \{c\}$ ;    $\Sigma_b \longleftarrow \Sigma$ ;
7              $c_b \longleftarrow \emptyset$ ;   shortened $\longleftarrow$ $false$ ;
8              while (Not(shortened) And ($c \neq c_b$)) do
9                  $l \longleftarrow$ select_a_literal($c \backslash c_b$) ;
10                 $c_b \longleftarrow c_b \cup \{l\}$ ; $\Sigma_b \longleftarrow (\Sigma_b \cup \{\neg l\})$ ;
11                 if $\bot \in$ UP($\Sigma_b$) then
12                     $c_l \longleftarrow$ conflict_analyze_and_learn() ;
13                     if $c_l \subset c$ then
14                         $\Sigma \longleftarrow \Sigma \cup \{c_l\}$ ;
15                         shortened $\longleftarrow$ $true$ ;
16                     else
17                         if $|c_l| < |c|$ then
18                             $\Sigma \longleftarrow \Sigma \cup \{c_l\}$ ; $c_b \longleftarrow c$ ;
19                         if $c \neq c_b$ then
20                             $\Sigma \longleftarrow \Sigma \cup \{c_b\}$ ;
21                             shortened $\longleftarrow$ $true$ ;
22                 else
23                     if $\exists (l_s \in (c \backslash c_b))$ $s.t.$ $l_s \in$ UP($\Sigma_b$) then
24                         if $(c \backslash c_b) \neq \{l_s\}$ then
25                             $\Sigma \longleftarrow \Sigma \cup \{c_b \cup \{l_s\}\}$ ;
26                             shortened $\longleftarrow$ $true$ ;
27                     if $\exists (l_s \in (c \backslash c_b))$ $s.t.$ $\neg l_s \in$ UP($\Sigma_b$)
                       then
28                         $\Sigma \longleftarrow \Sigma \cup \{c \backslash \{l_s\}\}$ ;
29                         shortened $\longleftarrow$ $true$ ;
30             if Not(shortened) then  $\Sigma \longleftarrow \Sigma \cup \{c\}$ ;
31             else  change $\longleftarrow$ $true$ ;
32     return $\Sigma$ ;
33 end
```

implementation of our method within most of current solvers, but also provides our approach with their effectiveness for the different performed tests. Our approach is thoroughly evaluated in the following section.

4.2 Empirical Evaluation

We have compared `ReVivAl` against the preprocessor which is actually considered as the best approach, namely `SatElite`. The state-of-the-art SAT solver `RSAT` [12] has been selected, since it has been recognized in the last competition as very adapted for structured problems. All our experiments have been conducted on Intel Xeon 3GHz under Linux CentOS 4.1. (kernel 2.6.9) with a RAM limit of 2GB. For all experiments, a timeout of 3 hours has been respected.

We have compared the preprocessors both on their size reduction and their impact on the efficiency of RSAT. Actually, this comparison has been conducted on a very large set of benchmarks from the SAT competitions, SAT Race, SATLIB and other sources; more than 5000 instances have been used for those experiments that have needed about 600 days of CPU time. A sample of experiments where examples will be referred in the following is proposed in Table 1, but the exhaustive results are available at:
`http://www.cril.fr/~piette/preprocessor.html`.

The first main part of Table 1 provides the name of the tested problem together with the number of clauses (*#cla*) and literals (*#lit*) it contains.The two other parts of the table are similar (one for each preprocessor), and contain the *time* of preprocessing in seconds, the size of the resulting formula in term of number of literals and clauses after the corresponding preliminary computation, and the *solving time* (in seconds) needed to solve the CNF formula after simplification. In addition, for `ReVivAl`, the number of performed iterations and learnt clauses are provided in the columns "#ite" and "#learnt", respectively. The best preprocessing on a given instance corresponds to the best one in term of cumulated preprocessing and solving time (reported in boldface).

First, let us focus on benchmarks that can be actually solved being only preprocessed. Indeed, it exists such CNF formulae, including some instances proposed for the SAT competitions and/or the SAT Races. Given the features of the presented preprocessing approaches, when one of them (or both) succeed(s) to prove (un)satisfiability of a CNF formula, this clearly means that the CNF formula is solvable in polytime (indicated *Polynomial* in the table). The interest of such formulae can be questioned for solvers empirical evaluations, because they do not exhibit any computational difficulties, which *should* be the key point of comparison between exhaustive procedures. Among the tested formulae, `SatElite` (resp. `ReVivAl`) proves 35 (resp. 167) instances polynomial. Moreover, note that for both preprocessors, those computations are most often performed within a few seconds (see e.g. `SAT_dat.k1`, `ezfact16_3`).

Second, let us consider the size of CNF formula after being preprocessed. Some differences can be observed between both approaches. Indeed, on the first hand, the purpose of `SatElite` is to eliminate variables without increasing the size of the CNF formula. Thus, resulting CNF formula can have about the same number of clauses, but they can exhibit a higher number of literals. On the opposite, `ReVivAl` tries to minimize the size of clauses and to add limited relevant ones. As a consequence, the simplified formulae sometimes contain a little more clauses than the original ones, but in general the average number of literals per clause is reduced, making them more exploitable for the solver's unit-propagation mechanism. As an example, on the benchmark `alu4mul.miter` which exhibits 30465 clauses and 103040 literals (ratio #lit/#cla = 3.38), `SatElite` eliminates variables keeping about the same number of clauses and literals whereas `ReVivAl` returns a smaller CNF formula in number of clauses (28992) for a ratio equal to 3.11. Cases where `SatElite` provides a formula with a largely bigger ratio can occur (see e.g. `3pipe_3_ooo` and `3bitadd_31`), but not with `ReVivAl`.

More generally, discarding the instances that cannot be solved using any of the preprocessors in conjunction to RSAT, a time gap of 18.8% can be observed in favour of `ReVivAl`. Futhermore, using `SatElite` RSAT cannot decide the satisfiability of 2508 instances within 3 hours of CPU time

Instance		SatElite			ReVivAl				
name	(#cla,#lit)	time	(#cla,#lit)	solv. time	time	(#cla,#lit)	solv. time	#ite	#learnt
pbl-00250	(32700,256765)	0.33	(31115,245053)	time out	**59.8**	(34815,219076)	**613.89**	30	2785
velev-fvp-sat-3.0-07	(1012271,2979665)	**4.05**	(998048,3017403)	**9.84**	61.53	(990394,2928889)	16.04	2	19
alu4mul.miter	(30465,103040)	0.1	(30392,102976)	915.91	**17.99**	(28992,90194)	**670.66**	15	502
Composite-024BitPrimes-1	(11158,49842)	0.02	(10087,44762)	7403.21	**0.56**	(10728,35545)	**2013.16**	20	163
Composite-024BitPrimes-0	(11158,49842)	0.02	(10016,44487)	733.2	**0.48**	(10395,34668)	**16.59**	18	132
velev-eng-uns-1.0-04	(66654,188252)	**0.5**	(62239,201530)	**11.67**	60.09	(57518,157260)	14.24	15	89
3bitadd_31	(31310,86676)	0.38	(31186,108004)	5058.73	**10.17**	(33125,83346)	**1.71**	28	1815
SAT_dat.k1	(3868,9928)	0	Polynomial	-	0	Polynomial	-	-	-
c3540mul.miter	(33199,112244)	**0.13**	(33066,112216)	**1635.29**	3.3	(27206,80134)	2186.01	13	815
logistics-rotate-10t5	(338789,680799)	2.45	(317194,687554)	1250.5	**11.67**	(277860,558081)	**151.64**	1	0
ezfact16_3	(1113,4089)	0	(990,3586)	0	0	Polynomial	-	-	-
ezfact48_8	(11001,41369)	0.02	(9215,34333)	61.99	**1.37**	(9582,27880)	**46.71**	18	315
ezfact48_9	(11001,41369)	0.02	(10532,39464)	204.94	**1.26**	(9558,27858)	**82.46**	17	344
ezfact64_1	(19785,74601)	0.06	(16716,62498)	time out	**2.77**	(17265,50532)	**3049.66**	15	549
abb313GPIA-8-c	(426860,2561106)	1.47	(421719,2521104)	366.29	**60.88**	(404007,2340042)	**12.05**	28	236
qg7-10	(33736,89626)	**0.04**	(11038,27452)	**0.01**	0.15	Polynomial	-	-	-
color-10-3	(6475,25200)	**0.08**	(6175,32600)	**1.28**	0.06	(6475,25200)	19.41	1	0
grieu-vmpc-s05-27r	(96849,253854)	0.15	(96849,253854)	104.02	**25.87**	(96482,239730)	**51.76**	9	154
ferry12	(32199,71303)	0.21	(30268,68540)	8.28	**0.5**	(30405,67168)	**1.87**	1	12
mod2c-3cage-unsat-9-1	(464,1856)	**0**	(464,1856)	**2152.31**	0	(472,1533)	2942.17	9	8
544707209399nw	(18031,53975)	0.12	(16032,49234)	1477.7	**2.26**	(14768,34277)	**814.4**	9	628
rand_net40-60-10	(14321,33560)	0.1	(10778,29243)	486.48	**3.74**	(14321,33152)	**421.97**	8	0
abb313GPIA-8-cn	(693640,2080902)	**16.73**	(388404,2307599)	**143**	23.27	(677607,2017201)	2321.44	12	19
equilarge_m1	(11489,33442)	0.07	(11158,41295)	time out	0.12	(11489,33164)	time out	5	0
hanoi5u	(73777,160717)	**0.29**	(61778,135270)	**183.74**	1.18	(59467,129001)	284.83	1	9
shuffling-2-s1765005333	(30465,103040)	0.13	(30392,102976)	1380.07	**23.93**	(28975,89789)	**1023.51**	11	592
lksat-n2200...s1262048766	(7524,22572)	0.04	(6322,19609)	201.38	**0.19**	(7629,20439)	**59.91**	16	105
3pipe_3_ooo	(33270,95618)	**0.19**	(31735,101212)	**9.67**	30.86	(31150,87665)	9.13	33	60
gripper12	(30746,68144)	0.12	(28060,62815)	time out	**0.26**	(27871,61632)	**676.2**	1	2
gripper13	(40461,89385)	0.16	(37437,83384)	1300.36	**0.37**	(37195,82030)	**1183.14**	1	4

Table 1. SatElite VS ReVivAl

(preprocessing *and* solver), while the solver fails for only 2457 benchmarks with our approach. This 51 instances difference does not look big, but SAT competitions and Races are usually settled by even smaller gaps. However, even if ReVivAl has in general a better effect on CNFs than SatElite, counter-examples can obviously be provided (see e.g. hanoi5u and abb313GPIA-8-cn). Nevertheless, many classes of SAT instances are typically more sensitive to the ReVivAl process, which is better than SatElite at improving RSAT. For example, on ezfact-*, encoding circuits factorization, Composite-*BitPrimes instances, encoding composite numbers (suggested as a challenge to SAT solvers in 1997 by Cook and Mitchell [3]), gripper* planning instances, our proposed approach clearly outperforms SatElite.

5 CONCLUSION

In this paper, ReVivAl, a new preprocessing based on limited forms of resolution and conflict analysis has been proposed. Our approach, called vivification, makes an original use of clause redundancy checking to produce sub-clauses and to add new relevant clauses obtained thanks to the clause learning scheme. Its efficiency is illustrated through extensive experiments with a state-of-the-art DPLL solver. A comparison with the best known preprocessing technique shows that ReVivAl, achieves interesting improvements, especially on circuits factorization, composite numbers and planning instances.

Our results open many interesting future directions of research. It appears that combining several preprocessors often enables to even better improvements. Indeed, a combination of SatElite and ReVivAl obtained parlicularly interesting results at the SAT-Race 2008 (6^{th} on 19 submitted solvers). A dynamic selection of preprocessors based on automated-tuning approaches is thus a path that should be explored. The periodical use of ReVivAl, for example during restarts, is also a promising future direction.

REFERENCES

[1] F. Bacchus and J. Winter, 'Effective preprocessing with hyper-resolution and equality reduction', in *SAT'03*, pp. 341–355, (2003).

[2] Ronen I. Brafman, 'A simplifier for propositional formulas with many binary clauses', in *IJCAI'01*, pp. 515–522, (2001).

[3] S.A. Cook and D.G. Mitchell, 'Finding hard instances of the satisfiability problem: A survey', *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, **35**, (1997).

[4] S. Darras, G. Dequen, L. Devendeville, B. Mazure, R. Ostrowski, and L. Sais, 'Using boolean constraint propagation for sub-clause deduction', in *CP'05*, pp. 757–761, (2005).

[5] M. Davis and H. Putnam, 'A computing procedure for quantification theory', *Journal of the ACM*, **7**(3), 201–215, (1960).

[6] N. Eén and A. Biere, 'Effective preprocessing in SAT through variable and clause elimination', in *SAT'05*, pp. 61–75, (2005).

[7] O. Fourdrinoy, E. Grégoire, B. Mazure, and L. Sais, 'Eliminating redundant clauses in SAT instances', in *CP-AI-OR'07*, pp. 71–83, (2007).

[8] A. Hertel, P. Hertel, and A. Urquhart, 'Formalizing dangerous SAT encodings', in *SAT'07*, pp. 159–172, (2007).

[9] R. G. Jeroslow and J. Wang, 'Solving propositional satisfiability problems', *Annals of mathematics and artificial intelligence*, **1**, 167–187, (1990).

[10] C. Li and Anbulagan, 'Look-ahead versus look-back for satisfiability problems.', in *CP'97*, pp. 341–355, (1997).

[11] Paolo Liberatore, 'Redundancy in logic I: CNF propositional formulae', *Artif. Intell.*, **163**(2), 203–232, (2005).

[12] K. Pipatsrisawat and A. Darwiche, 'RSAT 2.0: SAT solver description', Technical Report D–153, Automated Reasoning Group, Computer Science Department, UCLA, (2007).

[13] S. Subbarayan and D. Pradhan, 'NiVER: Non increasing variable elimination resolution for preprocessing SAT instances', *SAT'04*, 276–291, (2004).