

Using Graph-Based Representation to Derive Hard SAT instances

Gilles Audemard, Saïd Jabbour, and Lakhdar Saïs

CRIL - Université d'Artois - Lens, France
{audemard,jabbour,sais}@cril.fr

Abstract. In this paper a new class of hard SAT instances is proposed. These instances are built using a new graph based representation of boolean formula in conjunctive normal form (CNF). It extends the well known implication graph of binary clauses (2-SAT) to the general case. Every clause is represented as a set of possible (conditional) implications and encoded with different arcs labeled with a set of literals, called contexts. This representation allows us to reformulate classical resolution as a transitive closure on the graph. This result combined with the underlying structure of the graph are then used to derive hard SAT instances with respect to the state-of-the-art SAT solvers.

1 Introduction

The SAT problem, namely the issue of checking whether a boolean formula in conjunctive normal form is satisfiable or not, is central in many computer science and artificial intelligence domains, like theorem proving, planning, non-monotonic reasoning, VLSI correctness checking and knowledge-bases verification and validation. During 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 [20] and elaborate variants of the Davis-Putnam-Loveland-Logemann's DPLL procedure [9] ([19, 13]) can now solve efficiently many families of large SAT instances encoding real-world applications. Other kind of instances have been proposed with the main objective to defeat all existing solvers. For example, randomly generated unsatisfiable SAT instances still the most difficult ones [7]. Other instances such as Urquart [22], parity-32 [8], pigeon hole [15] can be solved thanks to an efficient handling of their particular structure. These last kind of instances has been for a long time a subject of active research. Many interesting improvements has been derived. Among them, one can cite heuristics [11], look-ahead [18, 4], symmetry breaking [3, 1], equivalence processing [17], functional dependencies [14].

In this paper, we propose a new class of hard SAT instances. These instances are highly structured but randomly generated. As we show in section 5, these instances are difficult to solve using the state-of-the-art SAT solvers such as CDCL (Conflict Driven Clause Learning) solvers (MINISAT [12] and ZCHAFF [19]) or other approaches such as (MARCH_EQ [16]). These instances are built using a new SAT graph-based representation which extends in an original way the well known implication graph of binary CNF formula (2-SAT) to the general case. Every clause is represented as a set of possible (conditional) implications and encoded with different arcs labeled with a set of

literals, called contexts (or conditions). Our proposed representation allows us to extend many interesting features of the 2-SAT polynomial time algorithm. Among them, classical resolution is formulated using the transitive closure of the graph. Paths from nodes labeled with opposite literals lead to a new definition of conditional backbone i.e. the literal is implied under a given context or condition. However, in the general case finding the (minimal) context under which a given literal is implied is computationally intractable.

Our SAT graph representation allows to generate small but hard structured satisfiable and unsatisfiable SAT instances. Our generator use different parameters leading to instances with different clauses length.

The rest of the paper is organized as follows. After some preliminary definitions, our sat graph-based representation is presented in section 3. Then, we introduce our new class of difficult benchmarks in section 4. Before concluding, an extensive experimental study is presented in section 5.

2 Technical preliminaries

Let $\mathcal{B}$ be a Boolean (i.e. propositional) language of formulas built in the standard way, using usual connectives ($\vee, \wedge, \neg, \rightarrow, \leftrightarrow$) and a set of propositional variables.

A *CNF formula* Σ is a set (interpreted as a conjunction) of *clauses*, where a clause is a set (interpreted as a disjunction) of *literals*. A literal is a positive or negated propositional variable. For a literal x (resp. a clause c), $\mathcal{V}(x)$ (resp. $\mathcal{V}(c)$) denotes the propositional variable associated to a literal x (resp. the set of variables associated to literals of c). We note $\mathcal{V}(\Sigma)$ (resp. $\mathcal{L}(\Sigma)$) the set of variables (resp. literals) occurring in Σ . The size of a clause c , denoted $|c|$, is equal to the number of literals it contains. A *unit clause* is a clause formed of size one. A *unit literal* is the unique literal of a unit clause. *Binary clauses* contains two literals. A *Horn clause* contains at most one positive literal. A clause is called *positive* (resp. *negative*) if all of its literals are positive (resp. negative). A *Horn-SAT* (resp. *2-SAT*) is a formula composed of only Horn (resp. binary) clauses. These two fragments of SAT are known to be solvable in linear time [2, 10].

In addition to these usual set-based notations, we define the negation of a set of literals S as the set of the corresponding opposite literals $\neg S = \{\neg l \mid l \in S\}$.

An *interpretation* I of a Boolean formula is an assignment of truth values $\{true, false\}$ to its variables. I can be represented as a conjunction (a set) of literals. Let $l \in \mathcal{L}(\phi)$, the formula obtained by assigning l to true, is $\phi(l) = \{c - \{\neg l\} \mid c \in \phi, \neg l \in c\} \cup \{c \mid c \in \phi, l \notin c, \neg l \notin c\}$. For an interpretation $I = \{l_1, l_2, \dots, l_n\}$, $\phi(I) = \phi(l_1)(l_2) \dots (l_n)$. A *model* of a formula is an interpretation I that satisfies the formula i.e. $\phi(I) = \emptyset$. Accordingly, SAT consists in finding a model of a CNF formula when such a model does exist or in proving that such a model does not exist.

Let c_1 be a clause containing a literal a and c_2 a clause containing the opposite literal $\neg a$, one *resolvent* of c_1 and c_2 is the disjunction of all literals of c_1 and c_2 less a and $\neg a$, noted $res(a, c_1, c_2)$. A resolvent is called *tautological* when it contains opposite literals; fundamental otherwise.

3 SAT-based graph representation

Different graph representations of CNF formula has been proposed previously and used for different purposes. Among them, one can cite graph implication for learning schemes [19], preprocessing [5], , survey propagation for solving satisfiable 3-SAT random formulas [6], visualization [21] and so on. Our proposed representation can be seen as an extension of the 2-SAT graph representation to the general case [2].

3.1 From CNF to labeled graph

Before introducing our approach, let us first recall the well known graph representation of 2-SAT formula.

Let Φ be a 2-SAT formula. The graph representation of ϕ , is defined as $G_\phi = (S, E)$, where $S = \{x, \neg x | x \in \mathcal{L}(\phi)\}$ and $E = \{(\neg x, y), (\neg y, x) | (x \vee y) \in \phi\}$. The polynomial time algorithm used to solve ϕ is based on applying the computation of the transitive closure of G_ϕ , $G_\phi^c = (S, E')$ and checking whether there exists a literal $x \in S$ such that $(x, \neg x) \in E'$ and $(\neg x, x) \in E'$. Obviously, computing transitive closure on the graph G_ϕ is equivalent to the saturating ϕ by resolution. Indeed, for (x, y) and (y, z) of E a new edge (x, z) is generated and added to the graph. Such application corresponds to $res(y, (\neg x \vee y), (\neg y \vee z)) = (\neg x \vee z)$.

For general CNF formula, a natural representation can be obtained using a hypergraph where vertices correspond to literals, and hyper edges to clauses. In the following, a SAT graph based representation of general CNF formula is presented. It extends in an original way the 2-SAT graph representation described below.

Definition 1. Let c be a clause such that $|c| \geq 2$. A context η_c associated to c is a conjunction of literals such that $\neg\eta_c \subset c$ and $|c - \{\neg\eta_c\}| = 2$ i.e when η_c is true the clause c becomes a binary clause.

Example 1. Let $c = (a \vee \neg b \vee c \vee d)$ be a clause. One possible context associated to c is $\eta_c = (b \wedge \neg c)$. The clause c can be rewritten as $((\neg a \wedge \eta_c) \rightarrow d)$.

For a clause c of size k , we have $|\eta_c| = k - 2$. The context associated to a binary clause is empty, whereas for ternary clauses the context is a unit literal. The number of possible contexts of c is equal to $\frac{k(k-1)}{2}$. A clause c can be rewritten in $k(k-1)$ different ways. When $k = 1$ the clause is unit, no context is possible. Using the clause c given in the example 1, we obtain 6 possible contexts of size 2, and 12 different ways for rewriting c .

Let us now define our SAT graph based representation of CNF formula.

Definition 2 (graph SAT representation). Let ϕ be a CNF formula. we define $G_\phi = (S, E, v)$ the graph SAT associated to Φ as the directed labeled graph defined as follows:

- $S = \{x, \neg x | x \in \mathcal{L}(\phi)\}$
- $E = \{a = (\neg x, y) \text{ such that } \exists c \in \Phi, c = (x \vee \neg\eta_c \vee y) \text{ and } v(a) = \eta_c\}$

In the sequel, for clarity reasons, we sometimes note a labeled arc $a = (x, y)$ as $(x, y, v(a))$. We also note $cla(a) = (\neg x \vee \neg v(a) \vee y)$ as the clause associated to a .

Clearly, the definition 2 generalizes the classical 2-SAT graph representation. Indeed, if all the contexts are empty i.e. all clauses of ϕ are binary, then all the arcs of G_ϕ are labeled with an empty set of literals.

3.2 Transitive closure / Resolution

In this section, a connection between classical resolution and graph transitive closure is described. Let us introduce some necessary definitions. Henceforth, we only consider formula ϕ without tautological clauses, and $G_\phi = (S, A, v)$ the SAT graph representation of ϕ .

Definition 3. Let $G_\phi = (S, A, v)$ a graph, $a_1 = (x, y, v(a_1)) \in A$ and $a_2 = (y, z, v(a_2)) \in A$. We define $tr(a_1, a_2) = a_3$ st. $a_3 = (x, z, v(a_1) \cup v(a_2) \setminus \{x, \neg z\})$.

In the definition above, the elimination of $\{x, \neg z\}$ from the context associated to a_3 guaranties that the clause $cla(a_3)$ do not contains several occurrences of the same literal. It corresponds to the application of the classical fusion rule.

Example 2. Let $a_1 = (x, y, \{\neg z, e\})$ and $a_2 = (y, z, \{x, f\})$. The two clauses encoded in a_1 and a_2 are $c_1 = (\neg x \vee z \vee \neg e \vee y)$ and $c_2 = (\neg y \vee \neg x \vee \neg f \vee z)$ respectively. We obtain $tr(a_1, a_2) = (x, z, \{\neg e, \neg f\})$ and $cla(tr(a_1, a_2)) = (\neg x \vee \neg e \vee \neg f \vee z)$. This last clause does not contain redundant literals. Using resolution $res(c_1, c_2, y)$ we obtain $c_3 = (\neg x \vee z \vee \neg e \vee \neg x \vee \neg f \vee z)$. Applying fusion rule on c_3 we eliminate one occurrence of $\neg x$ and z . We obtain $res(c_1, c_2, y) = cla(tr(a_1, a_2)) = (\neg x \vee \neg e \vee \neg f \vee z)$

Property 1. Let $c_1 = (x \vee \eta_{c_1} \vee y) \in \phi$, $c_2 = (\neg y \vee \eta_{c_2} \vee z) \in \phi$, $a_1 = (x, y, \eta_{c_1}) \in A$ and $a_2 = (y, z, \eta_{c_2}) \in A$. We have $res(c_1, c_2, y) = cla(tr(a_1, a_2))$

The proof of the property 1 is a direct consequence of the definitions 2 and 3.

Definition 4 (path). A path $p(x, y)$ between x and y in G_ϕ , is defined as follow : $p(x, y) = [x_{i_1}, x_{i_2}, \dots, x_{i_k}]$ st. $x_{i_1} = x$ and $x_{i_k} = y$ and $1 < j \leq k$ $(x_{i_{j-1}}, x_{i_j}) \in A$. We define $\eta_p = \bigcup_{1 < j \leq k} v((x_{i_{j-1}}, x_{i_j}))$ as the context associated to $p(x, y)$ and $tr(p(x, y)) = tr(\dots(tr((x_{i_1}, x_{i_2}), (x_{i_2}, x_{i_3})) \dots (x_{i_{k-1}}, x_{i_k})) \dots)$ as the transitive closure associated to $p(x, y)$.

Definition 5 (Fundamental path). Let $p(x, y) = [x = x_{i_1}, x_{i_2}, \dots, x_{i_k} = y]$ be a path between x and y . $p(x, y)$ is called fundamental if it satisfies the following conditions:

- η_p do not contain a literal and its opposite
- $\neg x \notin \eta_p$ and $y \notin \eta_p$

The following property states that for a fundamental path $p(x, y)$, one can derive a fundamental resolvent using the transitive closure.

Property 2. Let $p(x, y)$ be a path. If $p(x, y)$ is fundamental iff $cla(tr(p(x, y)))$ is a fundamental clause.

Proof. For a fundamental path $p(x, y)$, we have η_p do not contain a literal and its opposite. As $cla(tr(p(x, y)))$ can be written as $r = (\neg x \vee \neg \eta_p \vee y)$, r is clearly a fundamental clause. Indeed, from definition 5, η_p do not contain a literal and its opposite (first condition) and $x \in \neg \eta_p$ and $\neg y \notin \neg \eta_p$ (second condition). The converse is also true. Suppose that $cla(tr(p(x, y)))$ is not fundamental. As $tr(p(x, y)) = (x, y, \eta_p)$. Then $cla(tr(p(x, y))) = (\neg x \vee \neg \eta_p \vee y)$ is not fundamental. This mean that either the first or the second condition of the definition 5 is violated.

In the following, we describe how classical resolution can be achieved using graph transitive closure.

Definition 6. Let $G_\phi = (S, A, v)$ a graph representation of ϕ . We define $tr(G_\phi) = (S, A', v')$ such that: $\forall x \in S, \forall y \in S$ such that $\exists p(x, y)$ a fundamental path from x and y , $A' = A \cup \{tr(p(x, y))\}$.

Property 3. Let ϕ be a formula. ϕ is unsatisfiable iff $\exists k$ st. $tr^k(G_\phi) = (S, A', v')$ and $\exists x \in S$ with $(x, \neg x, \emptyset) \in A'$ and $(\neg x, x, \emptyset) \in A'$

Proof. Our graph representation is clearly complete i.e. each clause is encoded $k(k-1)$ times. As shown above, the application of the tr on the arcs of G_ϕ is equivalent to applying resolution on ϕ . The proof can be derived from the refutation completeness result of classical resolution.

In property 3, we have shown how resolution can be performed on the SAT graph representation. Obviously, other SAT techniques (e.g. variable elimination, unit propagation, etc) can be reformulated using our proposed representation. Let us note that our graph representation admits many interesting features. One of the main advantage is that dependencies between clauses are well expressed using our representation. Such structural property is known to be important for the efficiency of SAT solvers. Interestingly enough, our representation can be seen as a state graph, where each state represents a literal and the transition between the state s_1 to s_2 can be performed under a given condition $v((s_1, s_2))$ (a set of literals). Many interesting problems can be formulated more easily using graph. For example, one can be interested in computing the shortest path between a literal and its opposite i.e. computing the minimal condition for a given literal to be implied.

4 Generating hard instances

In this section, we show how our graph SAT representation can be used to generate difficult SAT instances. Our construction is based on graph traversal following paths $p(a, \neg a)$ from a literal a to its opposite $\neg a$. Let us note, that when the cumulated set of context is empty, then $\neg a$ is an implied literal i.e. $cla(tr(p(a, \neg a))) = \neg a$. However, deciding if a given literal is implied or finding the minimal condition for its implication is computationally intractable. Our idea consists in constructing a particular graph

where the implication of $\neg a$ is easy to prove but difficult to obtain with a given solver. Our proposed generator combine in an original way the graph representation with the well known hyper resolution.

Definition 7. Let $c_s = (y_1 \vee y_2 \vee \dots \vee y_m)$ be a clause, called side clause and ϕ_h be a CNF formula $(\neg y_1 \vee \alpha) \wedge (\neg y_2 \vee \alpha) \wedge \dots \wedge (\neg y_m \vee \alpha)$, where α is a sub-clause. Applying hyper-resolution on c_s and ϕ_h , in short $hr(c_s, \phi_h)$, we derive the sub-clause α , called hyper-resolvent.

Hyper-resolution can be defined using classical resolution. Indeed, α can also be derived using several resolution steps.

In the following definition, we introduce a new notion, called block-resolution, which extend hyper-resolution. Block-resolution can be seen as an application of several hyper resolution steps.

Definition 8. Let $c_s = (y_1 \vee y_2 \vee \dots \vee y_m)$ be a side clause and $c_r = (\neg x_1 \vee x_2 \vee \dots \vee x_k)$ a clause where $1 < k \leq m + 1$. Let $\phi_b = \phi_{h_1} \wedge \phi_{h_2} \wedge \dots \wedge \phi_{h_{k-1}}$ be a CNF formula such that $\phi_{h_1} = (\neg y_1 \vee \neg x_1 \vee x_2) \wedge \dots \wedge (\neg y_{\frac{m}{k-1}} \vee \neg x_1 \vee x_2)$ and $\phi_{h_{k-1}} = (\neg y_{m - \frac{m}{k-1} + 1} \vee \neg x_1 \vee x_k) \wedge \dots \wedge (\neg y_m \vee \neg x_1 \vee x_k)$. We define block resolution on c_s and ϕ_b as : $br(c_s, \phi_b) = hr(\dots hr(hr(c_s, \phi_{h_1}), \phi_{h_2}) \dots \phi_{h_{k-1}}) = c_r$, called a block resolvent. We define $B_k^m(x_1, x_2, \dots x_k) = c_s \wedge \phi_b$ as a block formula.

It is important to note, that in the definition 8, for $k = 1$, block-resolution corresponds to hyper-resolution.

To illustrate the above definition, in figure 1 an example of block formula $B_3^m(x_1, x_2, x_3)$ and its graph representation are given. On the left hand side of the figure, the side clause is shown in gray scales, and the other clauses of $B_3^m(x_1, x_2, x_3)$ are represented using our SAT graph based representation i.e. each arc from x_1 to x_i with $1 < i \leq 3$ is labeled with a different literal from the side clause.

Remark 1. Applying resolution between the side clause and the m other clauses, we can derive the clause $c_r = (\neg x_1 \vee x_2 \vee x_3)$ representing the implication $(x_1 \rightarrow (x_2 \vee x_3))$. Such a resolvent c_r can be obtained with one block-resolution step.

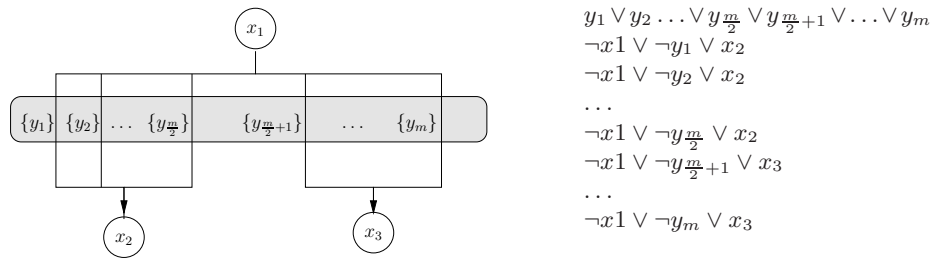


Fig. 1. A block B_3^m and its associated set of clauses

By connecting blocks in a hierarchical form, we want to build a graph in a way to highlight that a literal $\neg a$ is implied. Our graph is built as shown in figure 2. In this figure, box in gray scales represent side clauses. Starting from the literal a , we use an incremental construction. Using block-resolution, at each step we generate one more node until a given level l is reached. Then, at each step, we remove one node. The last one is the node associated to $\neg a$. We formalize this construction in definition 9.

Definition 9. We define $\Phi_l^m(a) =^C \Phi_l^m(a) \wedge^D \Phi_l^m(a)$ the set of clauses obtained with the partial SAT graph represented in figure 2. Each gray scale represents a B_3^m block. Φ_l^m is called diamond formula.

The following property ensures that the literal $\neg a$ is implied by the set of clauses $\Phi_l^m(a)$.

Property 4. Let $\Phi_l^m(a)$ be a diamond formula. We have $\Phi_l^m(a) \models \neg a$

Proof. Sketch of the proof : let us remind that for a block $B_3^m(x_1, x_2, x_3)$ can derive $x_1 \rightarrow x_2 \vee x_3$ using block resolution. The traversal of the graph (see figure 2 from top to bottom, and applying block-resolution at each step, one can derive successively the following implications : $a \rightarrow x_2^1 \vee x_2^2$, $a \rightarrow x_3^1 \vee x_3^2 \vee x_3^3 \dots$, $a \rightarrow x_{2 \times l-2}^1 \vee x_{2 \times l-2}^2$ and $a \rightarrow \neg a$

From this property, it is easy to generate unsatisfiable SAT instances by generating formulas $\Phi_l^m(a) \wedge \Phi_l^m(\neg a)$. However, we observed that these instances are quite easy to solve. Since, we want to generate difficult formulas, we operate a very simple change. Instead to build two graphs, one for the implication of a and another one for the implication of $\neg a$, we choose two different literals and we keep two graphs, one for the implication of $\neg a$ and the other one for the implication of $\neg b$. Then generated formulas are of the form $\Phi_l^m(a) \wedge \Phi_l^m(b)$. We can now introduce the generation process used by our generator.

- Each generated formula $\mathcal{G}_l^m$ has the form $\mathcal{G}_l^m = \Phi_l^m(a) \wedge \Phi_l^m(b)$ where a and b are two different literals.
- All variables are randomly generated
- $\forall 1 \leq k, h, i, j \leq l, x_k^i \neq x_h^j$
- Each side clause is a positive one
- Variables appearing in side clauses does not appear in nodes and *vice versa*

Of course, there exists a large number of ways to construct hard formulas using our proposed blocks. We use these restrictions for $\mathcal{G}_l^m$ as it derive the hardest SAT instances. Our generator is available at <http://www.cril.fr/~jabbour>. As we can see in the next section 5 our proposed approach allows to generate small but hard unsatisfiable instances. Furthermore, we can remark that each generated instance has some special characteristics:

- It does not contain pure literal
- It contains only clauses of size 3 and m
- It does not contain equivalent literals.

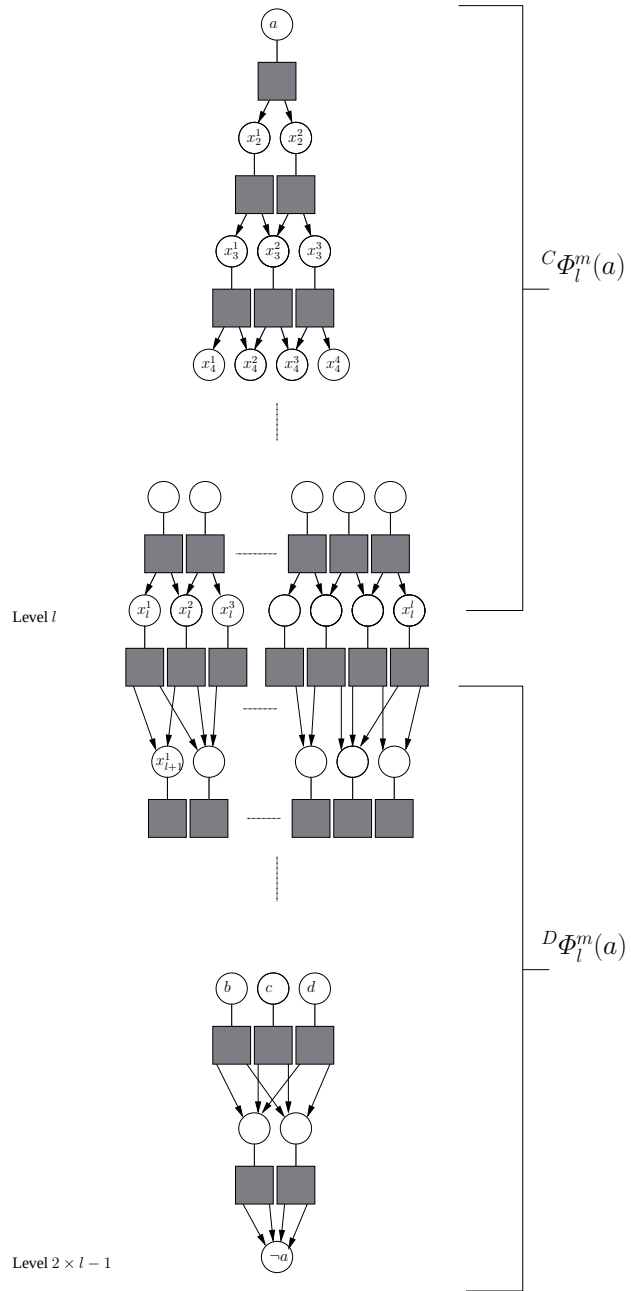


Fig. 2. Graph based generation : a hierarchy of block formulas

5 Experiments

The experimental results reported in this section are obtained on a Xeon 3.2 GHz (2 GB RAM). CPU time is limited to 1800 seconds.

We generate different classes of formulas. The first ones are $\mathcal{G}_l^4$ ($m = 4$) instances, whereas the second ones are $\mathcal{G}_l^5$ ($m = 5$). For each class, the number of levels (l) stands from 14 to 24, and different formulas are generated by varying the number of variables. Then, at least 100 instances are generated for each level.

We use three state-of-the-art SAT solvers, ZCHAFF (version MARCH_EQ 2007.3.12), MINISAT (version 2.0), which are conflict directed clause driven (cdcl) solvers, and MARCH_EQ (version 010). These three solvers are compared on the generated instances.

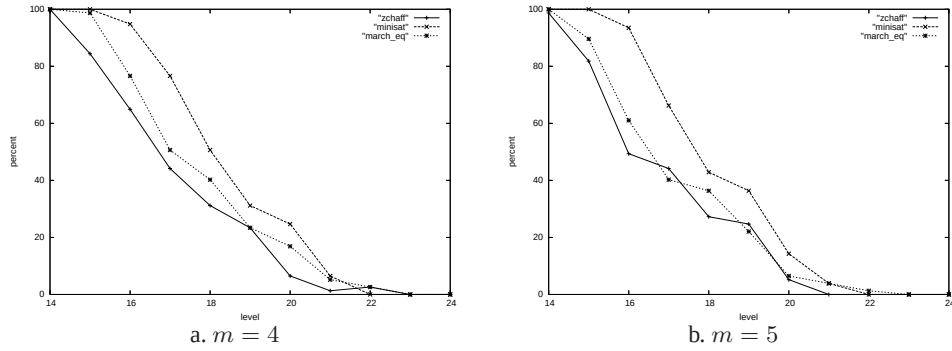


Fig. 3. percent of instances solved wrt level

The figure 3 gives the percent of solved instances by the three solvers for different levels. Figure 3.a exhibits this result for $m = 4$. It is clear that the greater the level the harder are the instances. From level $l = 16$, instances become hard for the three solvers. At level $l = 23$, the instances are very difficult and none solver is able to solve an instance at level 23. Let us note that, at this level, instances contains less 6000 clauses and the number of variables does not exceed 2000. For $m = 5$ (figure 3.b), generated instances are more difficult. Indeed, at level $l = 20$, ZCHAFF and MARCH_EQ solve 5% of the instances, where MINISAT solves 15%. Here, the number of variables does not exceed 1700.

For a given level, we want to know the impact of the number of variables in the difficulty of instances. As shown in figure 4 (where $m = 4$), we observe a threshold phenomenon when the number of variables changes. Indeed, for a given level, it exists a critical number of variables which produce difficult generated problems. This number is near from 600 for a level $l = 15$ (figure 4.a) and near 700 for a level $l = 16$. Like for random 3-SAT instances, as far from this critical number of variables we are as easier are the instances. We can also observe that MINISAT is the best on these instances and ZCHAFF obtain the worst results.

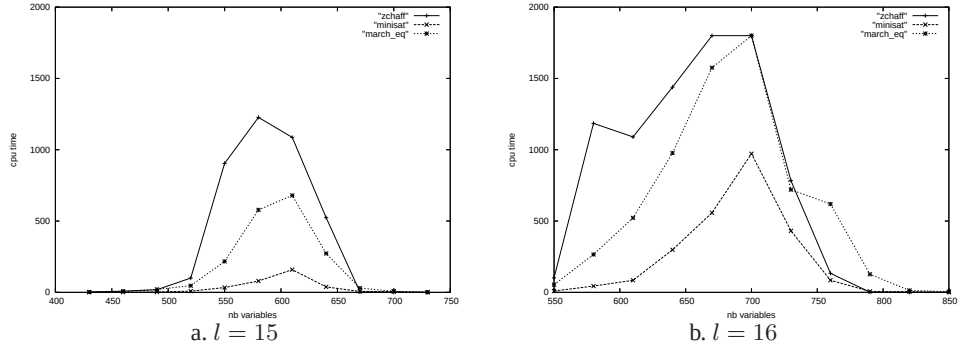


Fig. 4. CPU time comparison ($m = 4$)

This threshold phenomenon is observed also for $\mathcal{G}_l^5$ instances as we can see on figure 5. In this case, the critical point is near 580 for the level $l = 15$, where is near to 680 for level $l = 16$.

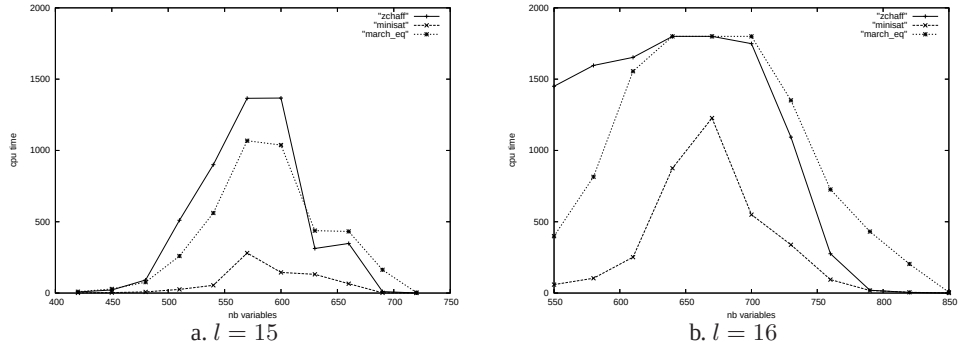


Fig. 5. CPU time comparison ($m = 5$)

Tables 1 and 2 exhibit that our generator can produce interesting and difficult satisfiable and unsatisfiable instances. The average time reported in these figures takes only into account solved instances. For a given level, we produces difficult satisfiable instances (table 1) for all solvers. Results reported in table 2 exhibit that our generator is able to produce quite short unsatisfiable and difficult benchmarks.

This notion of block allows to generate different kinds of formulas. In this paper, we achieved experiments, with $m = 4$ and $m = 5$, but difficult instances could be generated with $m = 2$. Let us remark that with this value binary clauses are generated. Currently, we are generalizing the side clause to a set of clauses. In this way, we can generate difficult formulas containing 800 variables and 3000 binary clauses and 1000 n-ary clauses.

		ZCHAFF		MINISAT		MARCH_EQ	
level l	nb inst.	nb solved	avg time	nb solved	avg time	nb solved	avg time
15	63	58	68	63	37	60	118
16	78	65	92	78	113	62	218
17	84	67	96	84	139	67	169
18	68	45	137	68	199	59	327
19	52	37	240	52	241	35	247
20	30	1	326	30	421	18	359
21	8	1	872	8	440	7	289
22	3	-	-	-	-	3	1035

Table 1. Some difficult SAT instances ($m = 4, 5$)

		ZCHAFF		MINISAT		MARCH_EQ	
level l	nb inst.	nb solved	avg time	nb solved	avg time	nb solved	avg time
14	106	106	67	106	5	106	40
15	91	70	155	91	55	85	214
16	67	23	592	67	254	44	523
17	26	0	-	26	834	3	1203
18	4	0	-	4	1383	-	-

Table 2. Some short UNSAT instances ($m = 4, 5$)

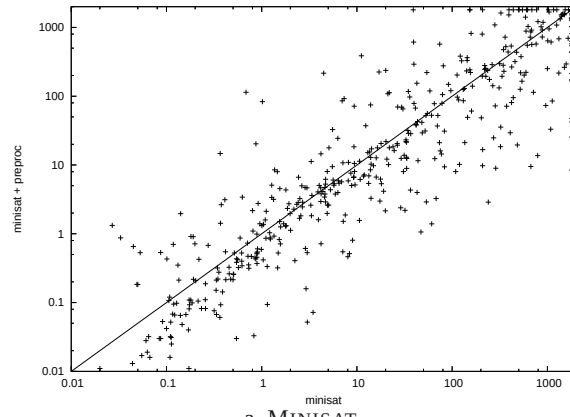
5.1 Preprocessing these hard instances

In this section, we propose a way to simplify these instances by adding some clauses in a preprocessing step. To this end, according to definition 8, for each block $\mathcal{B}_3^m(x_1, x_2, x_3)$, we add the clause $(\neg x_1 \vee x_2 \vee x_3)$.

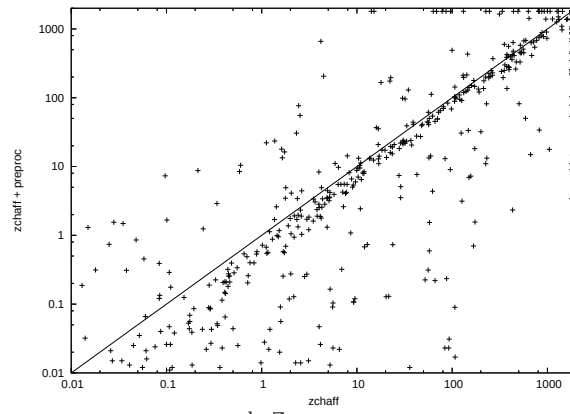
Figure 6 compares results of the three solvers with and without preprocessing. Each scatter plot given in figure 6 illustrates the comparative results of a given solver on each generated instance. The x-axis (resp. y-axis) corresponds to the cpu time tx (resp. ty) obtained by a solver on the original instance (resp. instance augmented by previous clauses). Each dot with (tx, ty) coordinates corresponds to a SAT instance. Dots above (resp. below) the diagonal indicate instances where the original formula is solved faster i.e. $tx < ty$ (resp. slower i.e. $tx > ty$) than the SAT formula with additional clauses.

This preprocessing step improve significantly the performances of MARCH_EQ solver. The performances of ZCHAFF and MINISAT are also improved. Interestingly enough, in a majority of cases where the performances are decreased corresponds to satisfiable instances.

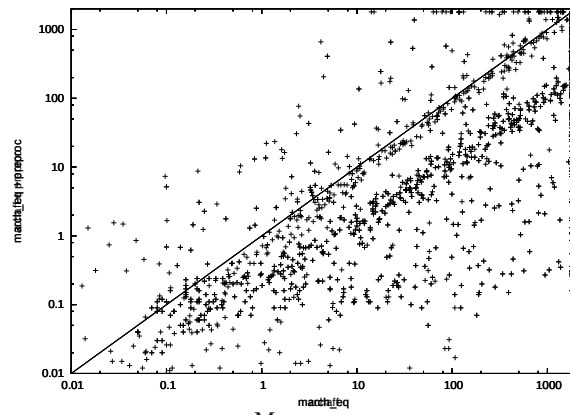
Finally, The plot in figure 7 is obtained as follows: the x-axis represents the number of benchmarks solved by a given solver and the y-axis (in log scale) the time needed to solve this number of problems. In this figure we take into account all generated instances ($\mathcal{G}_l^4$ and $\mathcal{G}_l^5$). It is clear that ZCHAFF and MARCH_EQ are not very good on our benchmarks. Furthermore, MINISAT seems to be the most efficient solver by report/ratio to the two others solvers. Adding the preprocessing step changes the performances. Each solver is able to solve more instances with these additionnal clauses (up to 180



a. MINISAT



b. ZCHAFF



c. MARCH_EQ

Fig. 6. Original formulas vs preprocessing formulas

instances for MARCH_EQ). Furthermore, in this case, MARCH_EQ becomes the best solver.

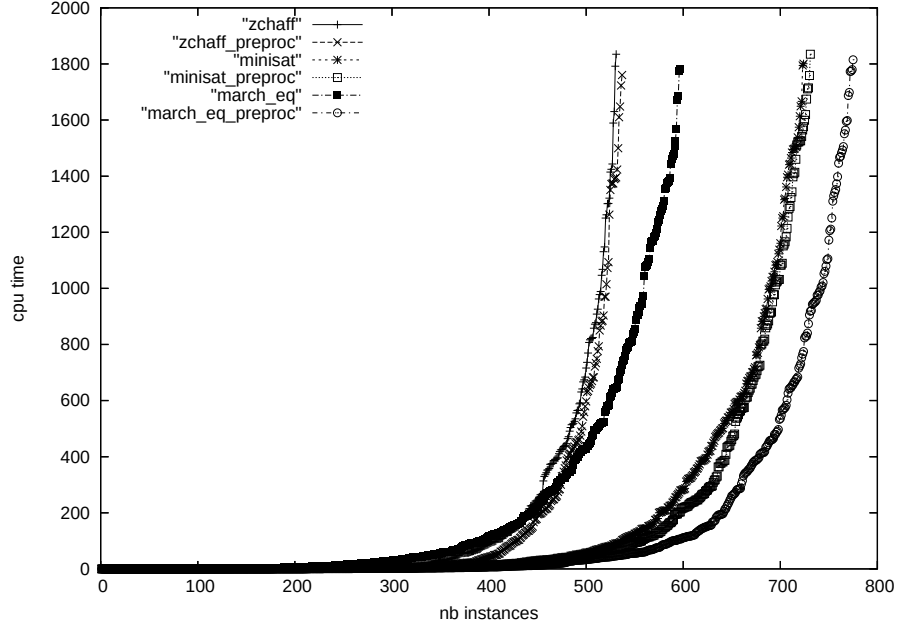


Fig. 7. Number of instances solved vs CPU time

6 Conclusion

In this paper we have presented a new class of difficult SAT benchmarks. These benchmarks are built using a new graph based representation of boolean formula in conjunctive normal form. It originally extends the well known 2-SAT implication graph. This new representation offers many interesting features. The structure of the formula (variables dependencies) is clearly well expressed. Experiments carried out on three state of the art solvers exhibits the difficulty of proposed instances. This graph sat representation and these difficult instances open interesting paths for future research. Among them, we plan to extend our generator in order to derive hard benchmarks containing less variables and clauses. Another interesting theoretical work concerns the use of the SAT graph properties to characterize new SAT polynomial fragments.

References

1. F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Solving difficult instances of boolean satisfiability in the presence of symmetry. *Transactions on Computer Aided Design*, 2003.

2. B. Aspvall, M. Plass, and R. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
3. B. Benhamou and L. Sais. Tractability through symmetries in propositional calculus. *Journal of Automated Reasoning*, 12(1):89–102, February 1994.
4. D. Le Berre. Exploiting the real power of unit propagation lookahead. In *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, 2001.
5. R. Brafman. A simplifier for propositional formulas with many binary clauses. In *proceedings of IJCAI*, pages 515–522, 2001.
6. A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Struct. Algorithms*, 27(2):201–226, 2005.
7. V. Chvátal and E. Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35:759–768, October 1988.
8. J. Crawford. Instances of learning parity function . <http://www.cs.ubc.ca/hoos/SATLIB/Benchmarks/SAT/DIMACS/PARITY/descr.html>.
9. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.
10. W. Dowling and J. Gallier. Linear-time algorithms for testing satisfiability of propositional horn formulae. *journal of logic programming*, 3:267–284, 1984.
11. O. Dubois and G. Dequen. 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)*, August 4–10 2001.
12. N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In *proceedings of SAT*, pages 61–75, 2005.
13. N. Eén and N. Sörensson. An extensible sat-solver. In *proceedings of SAT*, pages 502–518, 2003.
14. E. Gregoire, B. Mazure, R. Ostrowski, and L. Sais. Automatic extraction of functional dependencies. In *proceedings of SAT*, volume 3542 of LNCS, pages 122–132, 2005.
15. A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.
16. M. Heule, M. Dufour J. van Zwieten, and H. van Maaren. March_{eq}: Implementing additional reasoning into an efficient lookahead sat solver. In *proceedings of SAT*, number 3542 in LNCS, pages 345–359, 2005.
17. C. Min Li. Integrating equivalency reasoning into davis-putnam procedure. In *proceedings of Conference on Artificial Intelligence AAAI*, pages 291–296, Austin, USA, 2000. AAAI Press.
18. C. Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Principles and Practice of Constraint Programming (CP97)*, volume 1330 of LNCS, pages 341–355. Springer Verlag, 1997.
19. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of DAC*, 2001.
20. B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of AAAI*, pages 440–446, 1994.
21. C. Sinz and E. Dieringer. Dpvis - a tool to visualize the structure of sat instances. In *proceedings of SAT*, pages 257–268, 2005.
22. A. Urquhart. Hard examples for resolution. *JACM*, 34(1):209–219, 1987.