

# Automatic extraction of functional dependencies

Éric Grégoire, Richard Ostrowski, Bertrand Mazure, and Lakhdar Saïs

CRIL CNRS – Université d'Artois  
rue Jean Souvraz SP-18  
F-62307 Lens Cedex France  
{gregoire,ostrowski,mazure,sais}@cril.univ-artois.fr

**Abstract.** In this paper, a new polynomial time technique for extracting functional dependencies in Boolean formulas is proposed. It makes an original use of the well-known Boolean constraint propagation technique (BCP) in a new preprocessing approach that extracts more hidden Boolean functions and dependent variables than previously published approaches on many classes of instances.

**Keywords:** SAT, Boolean function, propositional reasoning and search.

## 1 Introduction

Recent impressive progress in the practical resolution of hard and large SAT instances allows real-world problems that are encoded in propositional clausal normal form (CNF) to be addressed (see e.g. [11, 7, 18]). While there remains a strong competition about building more efficient provers dedicated to hard random  $k$ -SAT instances [6], there is also a real surge of interest in implementing powerful systems that solve difficult large real-world SAT problems. Many benchmarks have been proposed and regular competitions (e.g. [4, 1, 14, 15]) are organized around these specific SAT instances, which are expected to encode structural knowledge, at least to some extent.

Clearly, encoding knowledge under the form of a conjunction of propositional clauses can flatten some structural knowledge that would be more apparent in more expressive propositional logic representation formalisms, and that could prove useful in the resolution step [13, 8].

In this paper, a new pre-processing step is proposed in the resolution of SAT instances, that extracts and exploits some structural knowledge that is hidden in the CNF. The technique makes an original use of the well-known Boolean constraint propagation (BCP) process. Whereas BCP is traditionally used to produce implied and/or equivalent literals, in this paper it is shown how it can be extended so that it delivers an hybrid formula made of clauses together with a set of equations of the form  $y = f(x_1, \dots, x_n)$  where  $f$  is a standard connective operator among  $\{\vee, \wedge\}$  and where  $y$  and  $x_i$  are Boolean variables of the initial SAT instance. These Boolean functions allow us to detect a subset of dependent variables, that can be exploited by SAT solvers.

This paper extends in a significant way the preliminary results that were published in [12] in that it describes a technique that allows more dependent variables and hidden functional dependencies to be detected in several classes of instances. We shall see that the set of functional dependencies can underlie cycles. Unfortunately, highlighting actual dependent variables taking part in these cycles can be time-consuming since it coincides to the problem of finding a minimal cycle cutset of variables in a graph, which is a well-known NP-hard problem. Accordingly, efficient heuristics are explored to cut these cycles and deliver the so-called dependent variables.

The paper is organized as follows. After some preliminary definitions, Boolean gates and their properties are presented. It is then shown how more functional dependencies than [12] can be deduced from the CNF, using Boolean constraint propagation. Then, a technique allowing us to deliver a set of dependent variables is presented, allowing the search space to be reduced in an exponential way. Experimental results showing the interest of the proposed approach are provided. Finally, promising paths for future research are discussed in the conclusion.



## 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*  $\Sigma$  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. We note  $\mathcal{V}(\Sigma)$  (resp.  $\mathcal{L}(\Sigma)$ ) the set of variables (resp. literals) occurring in  $\Sigma$ . A *unit clause* is a clause formed with one unique literal. A *unit literal* is the unique literal of a unit clause.

In addition to these usual set-based notations, we define the negation of a set of literals ( $\neg\{l_1, \dots, l_n\}$ ) as the set of the corresponding opposite literals ( $\{\neg l_1, \dots, \neg l_n\}$ ).

An *interpretation* of a Boolean formula is an assignment of truth values  $\{true, false\}$  to its variables. A *model* of a formula is an interpretation that satisfies the formula. 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  $\neg a$ . A resolvent is called *tautological* when it contains opposite literals.

Let us recall here that any Boolean formula can be translated thanks to a linear time algorithm into CNF, equivalent with respect to SAT (but that can use additional propositional variables). Most satisfiability checking algorithms operate on clauses, where the structural knowledge of the initial formulas is thus flattened. In the following, CNF formulas will be represented as Boolean gates.

## 3 Boolean gates

A (*Boolean*) *gate* is an expression of the form  $y = f(x_1, \dots, x_n)$ , where  $f$  is a standard connective among  $\{\vee, \wedge, \Leftrightarrow\}$  and where  $y$  and  $x_i$  are propositional literals, that is defined as follows :

- $y = \wedge(x_1, \dots, x_n)$  represents the set of clauses  $\{y \vee \neg x_1 \vee \dots \vee \neg x_n, \neg y \vee x_1, \dots, \neg y \vee x_n\}$ , translating the requirement that the truth value of  $y$  is determined by the conjunction of the truth values of  $x_i$  s.t.  $i \in [1..n]$ ;
- $y = \vee(x_1, \dots, x_n)$  represents the set of clauses  $\{\neg y \vee x_1 \vee \dots \vee x_n, y \vee \neg x_1, \dots, y \vee \neg x_n\}$ ;
- $y = \Leftrightarrow(x_1, \dots, x_n)$  represents the following *equivalence chain* (also called *biconditional formula*)  $y \Leftrightarrow x_1 \Leftrightarrow \dots \Leftrightarrow x_n$ , which is equivalent to the set of clauses  $\{y \vee x_1 \vee \dots \vee x_n, y \vee \neg x_1 \vee \dots \vee \neg x_n, \neg y \vee x_1 \vee \neg x_2 \vee \dots \vee \neg x_n, \dots, \neg y \vee \neg x_1 \vee \dots \vee \neg x_{n-1} \vee x_n\}$ .

In the following, we consider gates of the form  $y = f(x_1, \dots, x_n)$  where  $y$  is a variable or the Boolean constant *true*, only.

Indeed, any clause can be represented as a gate of the form  $true = \vee(x_1, \dots, x_n)$ . Moreover, a gate  $\neg y = \wedge(x_1, \dots, x_n)$  (resp.  $\neg y = \vee(x_1, \dots, x_n)$ ) is equivalent to  $y = \vee(\neg x_1, \dots, \neg x_n)$  (resp.  $y = \wedge(\neg x_1, \dots, \neg x_n)$ ). According to the well-known property of equivalence chain asserting that every equivalence chain with an odd (resp. even) number of negative literals is equivalent to the chain formed with the same literals, but all in positive (resp. except one) form, every gate of the form  $y = \Leftrightarrow(x_1, \dots, x_n)$  can always be rewritten into a gate where  $y$  is a positive literal. For example,  $\neg y = \Leftrightarrow(\neg x_1, x_2, x_3)$  is equivalent to  $y = \Leftrightarrow(x_1, x_2, x_3)$  and  $\neg y = \Leftrightarrow(\neg x_1, x_2, \neg x_3)$  is equivalent to e.g.  $y = \Leftrightarrow(x_1, x_2, \neg x_3)$ .

A propositional variable  $y$  (resp.  $x_1, \dots, x_n$ ) is an *output variable* (resp. are *input variables*) of a gate of the form  $y = f(x'_1, \dots, x'_n)$ , where  $x'_i \in \{x_i, \neg x_i\}$ .

A propositional variable  $z$  is an *output (dependent) variable of a set of gates* iff  $z$  is an output variable of at least one gate in the set. An *input (independent) variable of a set of gates* is an input variable of a gate which is not an output variable of the set of gates.

A gate is satisfied under a given Boolean interpretation iff the left and right hand sides of the gate are simultaneously *true* or *false* under this interpretation. An interpretation satisfies a set of gates iff each gate is satisfied under this interpretation. Such an interpretation is called a model of this set of gates.

#### 4 From CNF to gates

Practically, we want to find a representation of a CNF  $\Sigma$  using gates that highlights a *maximal* number of dependent variables, in order to decrease the actual computational complexity of checking the satisfiability of  $\Sigma$ . Actually, we shall describe a technique that extracts gates that can be deduced from  $\Sigma$ , and that thus *cover* a subset of clauses of  $\Sigma$ . Remaining clauses of  $\Sigma$  will be represented as or-gates of the form  $true = \vee(x_1, \dots, x_n)$ , in order to get a uniform representation.

More formally, assume that a set  $G$  of gates whose corresponding clauses  $Cl(G)$  are logical consequences of a CNF  $\Sigma$ , the set  $\Sigma_{uncovered(G)}$  of uncovered clauses of  $\Sigma$  w.r.t.  $G$  is the set of clauses of  $\Sigma \setminus Cl(G)$ .

Accordingly,  $\Sigma \equiv \Sigma_{uncovered(G)} \cup Cl(G)$ .

Not trivially, we shall see that the additional clauses  $Cl(G) \setminus \Sigma$  can play an important role in further steps of deduction or satisfiability checking.

Knowing output variables can play an important role in solving the consistency status of a CNF formula. Indeed, the truth-value of an  $y$  output variable of a gate depends on the truth value of the corresponding  $x_i$  input variables. The truth value of such output variables can be obtained by propagation, and they can be omitted by selection heuristics of DPLL-like algorithms [3]. In the general case, knowing  $n'$  output variables of a gate-oriented representation of a CNF formula using  $n$  variables allows the size of the set of interpretations to be investigated to decrease from  $2^n$  to  $2^{n-n'}$ . Obviously, the reduction in the search space increases with the number of detected dependent variables.

Unfortunately, to obtain such a reduction in the search space, one might need to address the following problems:

- Extracting gates from a CNF formula can be a time-consuming process in the general case, unless some depth-limited search resources or heuristic criteria are provided. Indeed, showing that  $y = f(x_1, \dots, x_i)$  (where  $y, x_1, \dots, x_i$  belong to  $\Sigma$ ) follows from a given CNF  $\Sigma$ , is coNP-complete.
- when the set of detected gates contains recursive definitions (like  $y = f(x, t)$  and  $x = g(y, z)$ ), assigning truth values to the set of independent variables is not sufficient to determine the truth values of all the dependent ones. Handling such recursive definitions coincides to the well-known NP-hard problem of finding a minimal cycle cutset in a graph.

In this paper, these two computationally-heavy problems are addressed. The first one by restricting deduction to Boolean constraint propagation, only. The second one by using graph-oriented heuristics.

Let us first recall some necessary definitions about Boolean constraint propagation.

#### 5 Boolean constraint propagation (BCP)

Boolean constraint propagation or unit resolution, is one of the most used and useful lookahead algorithm for SAT.

Let  $\Sigma$  be a CNF formula,  $BCP(\Sigma)$  is the CNF formula obtained by *propagating* all unit literals of  $\Sigma$ . Propagating a unit literal  $l$  of  $\Sigma$  consists in suppressing all clauses  $c$  of  $\Sigma$  such that  $l \in c$  and replacing all clauses  $c'$  of  $\Sigma$  such that  $\neg l \in c'$  by  $c' \setminus \{\neg l\}$ . The CNF obtained in such a way is equivalent to  $\Sigma$  with respect to satisfiability.

The *set of propagated unit literals* of  $\Sigma$  using BCP is noted  $UP(\Sigma)$ . Obviously, we have that  $\Sigma \models UP(\Sigma)$ . BCP is a restricted form of resolution, and can be performed in linear time.

It is also complete for Horn formulas. In addition to its use in DPLL procedures, BCP is used in many SAT solvers as a processing step to deduce further interesting information such as implied [5] and equivalent literals [2][9]. Local processing based-BCP is also used to deliver promising branching variables (heuristic UP [10]).

In the sequel, it is shown that BCP can be further extended, allowing more general functional dependencies to be extracted.

## 6 BCP and functional dependencies

Actually, BCP can be used to detect hidden functional dependencies. The main result of the paper is the practical exploitation of the following original property: gates can be computed using BCP only, while checking whether a gate is a logical consequence of a CNF is coNP-complete in the general case.

*Property 1.* Let  $\Sigma$  be a CNF formula,  $l \in \mathcal{L}(\Sigma)$ , and  $c \in \Sigma$  s.t.  $l \in c$ . If  $c \setminus \{l\} \subset \neg UP(\Sigma \wedge l)$  then  $\Sigma \models l = \wedge(\neg\{c \setminus \{l\}\})$ .

*Proof.* Let  $c = \{l, \neg l_1, \neg l_2, \dots, \neg l_m\} \in \Sigma$  s.t.  $c \setminus \{l\} = \{\neg l_1, \neg l_2, \dots, \neg l_m\} \subset \neg UP(\Sigma \wedge l)$ . The Boolean function  $l = \wedge(\neg\{c \setminus \{l\}\})$  can be written as  $l = \wedge(l_1, l_2, \dots, l_m)$ . To prove that  $\Sigma \models l = \wedge(l_1, l_2, \dots, l_m)$ , we need to show that every model of  $\Sigma$ , is also a model of  $l = \wedge(l_1, l_2, \dots, l_m)$ . Let  $I$  be a model of  $\Sigma$ , then

1.  $l$  is either *true* in  $I$ :  $I$  is also a model of  $\Sigma \wedge l$ . As  $\{\neg l_1, \neg l_2, \dots, \neg l_m\} \subset \neg UP(\Sigma \wedge l)$ , we have  $\{l_1, l_2, \dots, l_m\} \subset UP(\Sigma \wedge l)$ , then  $\{l_1, l_2, \dots, l_m\}$  are *true* in  $I$ . Consequently,  $I$  is also a model of  $l = \wedge(l_1, l_2, \dots, l_m)$ ;
2. or  $l$  is *false* in  $I$ : as  $c = \{l, \neg l_1, \neg l_2, \dots, \neg l_m\} \in \Sigma$  then  $I$  satisfies  $c = \{\neg l_1, \neg l_2, \dots, \neg l_m\} \in \Sigma$ . So, at least one the literals  $l_i, i \in \{1, \dots, m\}$  is *true* in  $I$ . Consequently,  $I$  is also a model of  $l = \wedge(l_1, l_2, \dots, l_m)$

Clearly, depending on the sign of the literal  $l$ , and-gates or or-gates can be detected. For example, the and-gate  $\neg l = \wedge(l_1, l_2, \dots, l_n)$  is equivalent to the or-gate  $l = \vee(\neg l_1, \neg l_2, \dots, \neg l_n)$ . Let us also note that this property covers binary equivalence since  $a = \wedge(b)$  is equivalent to  $a \Leftrightarrow b$ .

Actually, this property allows gates to be detected, which were not in the scope the technique described in [12]. Let us illustrate this by means of an example.

*Example 1.* Let  $\Sigma_1 \supseteq \{y \vee \neg x_1 \vee \neg x_2 \vee \neg x_3, \neg y \vee x_1, \neg y \vee x_2, \neg y \vee x_3\}$ .

According to [12],  $\Sigma_1$  can be represented by a graph where each vertex represents a clause and where each edge corresponds to the existence of tautological resolvent between the two corresponding clauses. Each connected component might be a gate. As we can see the first four clauses belong to a same connected component. This is a necessary condition for such a subset of clauses to represent a gate. Such a restricted subset of clauses (namely, those appearing in the same connected component) is then checked syntactically to determine if it represents an and/or gate. Such a property can be checked in polynomial time. In the above example, we thus have  $y = \wedge(x_1, x_2, x_3)$ .

Now, let us consider, the following example,

*Example 2.*  $\Sigma_2 \supseteq \{y \vee \neg x_1 \vee \neg x_2 \vee \neg x_3, \neg y \vee x_1, \neg x_1 \vee x_4, \neg x_4 \vee x_2, \neg x_2 \vee x_5, \neg x_4 \vee \neg x_5 \vee x_3\}$ .

Clearly, the graphical representation of this later example is different and the above technique does not help us in discovering the  $y = \wedge(x_1, x_2, x_3)$  gate. Indeed, the above necessary but not sufficient condition is not satisfied.

Now, according to Property 1, both the and-gates behind Example 1 and Example 2 can be detected. Indeed,  $UP(\Sigma_1 \wedge y) = \{x_1, x_2, x_3\}$  (resp.  $UP(\Sigma_2 \wedge y) = \{x_1, x_4, x_2, x_5, x_3\}$ ) and

$\exists c \in \Sigma_1$ , (resp.  $c' \in \Sigma_2$ ),  $c = (y \vee \neg x_1 \vee \neg x_2 \vee \neg x_3)$  (resp.  $c' = (y \vee \neg x_1 \vee \neg x_2 \vee \neg x_3)$ ) such that  $c \setminus \{y\} \subset \neg UP(\Sigma_1 \wedge y)$  (resp.  $c' \setminus \{y\} \subset \neg UP(\Sigma_2 \wedge y)$ ).

Accordingly, a preprocessing technique to discover gates consists in checking the Property 1 for any literal occurring in  $\Sigma$ . A further step consists in finding dependent variables of the original formulas, as they can be recognised in the discovered gates. A gate clearly exhibits one dependent literal with respect to the inputs which are considered independent, as far a single gate is considered. Now, when several gates share literals, such a characterisation of dependent variables does not apply anymore. Indeed, forms of cycle can occur as shown in the following example.

*Example 3.*  $\Sigma_3 \supseteq \{x = \wedge(y, z), y = \vee(x, \neg t)\}$ .

Clearly,  $\Sigma_3$  contain a cycle. Indeed,  $x$  depends on the variables  $y$  and  $z$ , whereas  $y$  depends on the variables  $x$  and  $t$ . When a single gate is considered, assigning truth values to input variables determines the truth value of the output, dependent, variable. As in Example 3, assigning truth values to input variables that are not output variables for other gates is not enough to determine the truth value of all involved variables. In the example, assigning truth values to  $z$  and  $t$  is not sufficient to determine the truth value of  $x$  and  $y$ . However, in the example, when we assign a truth value to an additional variable ( $x$ , which is called a *cycle cutset variable*) in the cycle, the truth value of  $y$  is determined. Accordingly, we need to cut such a form of cycle in order to determinate a sufficient subset of variables that determines the values of all variables. Such a set is called a *strong backdoor* in [17]. In Example 3, the strong backdoor corresponds to the set of  $\{x\} \cup \{z, t\}$ . In this context, a strong backdoor is the union of the set of independent variables and of the variables of the cycle cutset. Finding the minimal set of variables that cuts all the cycles in the set of gates is an NP-hard problem. This issue is investigated in the next section.

## 7 Searching for dependent variables

In the following, a graph representation of the interaction of gates is considered. More formally,

A set of gates can be represented by a bipartite graph  $G = (O \cup I, E)$  as follows:

- for each gate we associate two vertices, the first one  $o \in O$  represents the output of the gate, and the second one  $i \in I$  represents the set of its input variables. So the number of vertex is less than  $2 \times \#gates$ , where  $\#gates$  is the number of gates;
- For each gate, an edge  $(o, i)$  between the two vertices  $o$  and  $i$  representing the left and the right hand sides of a gate is created. Additional edges are created between  $o \in O$  and  $i \in I$  if one of the literals of the output variable associated to the vertex  $o$  belongs to the set of input literals associated to the vertex  $i$ .

Finding a *smallest* subset  $V'$  of  $O$  s.t. the subgraph  $G' = (V' \cup O, E')$  is acyclic is a well-known NP-hard problem.

Actually, any subset  $V'$  that makes the graph acyclic is the representation of the set of variables, which together with all the independent ones, allows all variables to be determined. When  $V'$  is of size  $c$ , and the set of dependent variables is of size  $d$ , then the search space is reduced from  $2^n$  to  $2^{n-(d-c)}$ , where  $n$  is the number of variable occurring in the original CNF formula.

We thus need to find a trade-off between the size of  $V'$ , which influences the computational cost to find it, and the expected time gain in the subsequent SAT checking step.

In the following, two heuristics are investigated in order to find a cycle-cut set  $V'$ . The first-one is called *Maxdegree*. It consists in building  $V'$  incrementally by selecting vertices with the highest degree first, until the remaining subgraph becomes acyclic.

The second one is called *MaxdegreeCycle*. It consists in building  $V'$  incrementally by selecting first a vertex with the highest degree among the vertices that belong to a cycle. This heuristic guarantees that each time a vertex is selected, then at least one cycle is cut.

In the next section, extensive experimental results are presented and discussed, involving the preprocessing technique described above. It computes gates and cuts cycles when necessary in order to deliver a set of dependent variables. Two strategies are explored: in the first one, each time a gate is discovered, the covered clauses of  $\Sigma$  are suppressed; in the second one, covered clauses are eliminated at the end of the generation of gates, only. While the first one depends on the considered order of propagated literals, the second one is order-independent. These two strategies will be compared in terms of number of discovered gates, of the size of the cycle cutsets, of dependent variables and of the final uncovered clauses.

## 8 Experimental results

Our preprocessing software is written in C under Linux Redhat 7.1 (available at : <http://www.cril.univ-artois.fr/~ostrowski/Binaries/llsatpreproc>). All experiments have been conducted on Pentium IV, 2.4 Ghz. Description of the benchmarks can be found on SATLib (<http://www.satlib.org>).

We have applied both [12] and our proposed technique on all benchmarks from the last SAT competition [15, 16], covering e.g. model-checking, VLSI and planning instances. Complete results are available at :

<http://www.cril.univ-artois.fr/~ostrowski/result-llsatpreproc.ps>. In the following, we illustrate some typical ones. On each class of instances, average and standard deviation results are provided with respect to the corresponding available instances.

In Table 1, for each considered class, the results of applying both [12]'s technique and the two new ones described above (in the first one, covered clauses are not suppressed as soon as they are discovered whereas they are suppressed in the second one) in terms of the mean number of discovered gates ( $\#G$ ). The results clearly shows that our approach allows one to discover more gates. Not surprisingly, removing clauses conducts the number of detected gates to decrease.

Family of Instances Name	(#Inst., #V[min-Max], #C[min-Max])	[12]'s	Our approach		
		technique #G	No cl. remov. #G	Cl. remov. #G	#C remov.
Blocks	(3,484[283-758],27423[9690-47820])	10[3]	236[134]	18[5]	271[142]
Logistics	(8,994[116-3016],12706[953-50457])	380[265]	437[417]	169[213]	630[585]
Pipe	(6,1642[834-2577],18624[6695-33270])	1312[679]	1407[697]	1240[639]	13898[9083]
Facts	(13,3178[2218-4315],48737[22539-90646])	713[147]	1601[541]	497[170]	1731[510]
Parity	(30,1044[64-3176],3614[254-10325])	568[828]	510[594]	328[455]	663[870]
Qg	(10,969[512-1331],33747[9685-64054])	310[91]	1828[652]	298[80]	1708[601]
Ca	(7,637[26-2282],1835[70-6586])	419[547]	459[592]	414[542]	1233[1615]
Dp	(11,1427[213-3193],3580[376-8308])	1117[856]	1468[1211]	915[812]	2534[2298]
Bmc2	(5,1952[316-4089],6908[1002-13531])	895[714]	1025[850]	744[623]	2082[1824]
Rand	(6,2217[2000-2500],6568[5921-7401])	2133[236]	2444[381]	2103[252]	6212[692]
Ezfact	(40,1441[193-3073],9169[1113-19785])	40[18]	268[127]	68[33]	68[33]
Med	(3,761[341-1159],20154[5556-36291])	66[32]	316[162]	14[5]	319[164]
Av-g-checker	(4,917[648-1188],28661[17087-40441])	324[105]	1098[375]	304[101]	1092[373]
nw/nc/fw	(13,3997[2756-5074],15829[10886-20123])	89[40]	468[136]	125[38]	125[38]
Am	(4,2011[433-4264],6925[1458-14751])	989[835]	772[585]	393[276]	927[625]
Cnf	(2,2424[2424-2424],14812[14812-14812])	2336[0]	3280[0]	2301[6]	13703[149]

**Table 1.** #G: Number of gates detected (average[standard deviation])

In Table 2, we took the no-remove option. We explored the above two heuristics for cutting cycles (*Maxdegree* and *MaxdegreeCycle*). For each class of instances, we provide the average number of detected dependent variables ( $\#D$ ), the size of the cycle cutsets ( $\#CS$ ) and the size of the discovered backdoor ( $\#B$ ), and the cumulated CPU time in seconds for discovering gates and computing these results. On some classes, the backdoor can be 10% of the number of variables, only.

In Table 3, the remove option was considered. The number of gates is often lower than with the no-remove option. On the other hand, the size of the cycle cutset is generally lower with the remove option.

Accordingly, no option is preferable than the other one in the general case. Indeed, finding a smaller backdoor depends both on the considered class of instances and the considered option.

Family of Instances	(#V [min-Max])	<i>Maxdegree</i>			<i>MaxdegreeCycle</i>		
		#D	#CS	#B	#D	#CS	#B
Blocks	(484[283-758])	38[13]	198[123]	353[215]	39[9]	197[124]	352[216]
Logistics	(994[116-3016])	113[158]	245[218]	441[532]	143[164]	214[194]	410[522]
Pipe	(1642[834-2577])	980[768]	265[219]	582[201]	764[449]	481[192]	798[348]
Facts	(3178[2218-4315])	738[237]	813[256]	1964[604]	487[124]	1064[362]	2216[623]
Parity	(1044[64-3176])	243[388]	84[46]	573[528]	287[410]	40[21]	528[505]
Qg	(969[512-1331])	303[202]	228[236]	228[236]	11[6]	521[194]	521[194]
Ca	(637[26-2282])	290[434]	130[142]	344[403]	265[341]	155[206]	369[481]
Dp	(1427[213-3193])	513[463]	451[485]	725[625]	551[496]	412[343]	686[498]
Bmc2	(1952[316-4089])	662[716]	27[22]	886[874]	660[696]	30[10]	888[893]
Rand	(2217[2000-2500])	1777[301]	357[339]	440[343]	1152[134]	981[111]	1064[115]
Ezfact	(1441[193-3073])	28[35]	66[45]	1370[1073]	55[27]	39[18]	1343[1060]
Med	(761[341-1159])	205[102]	110[72]	110[72]	14[4]	302[157]	302[157]
Avg-checker	(917[648-1188])	209[357]	606[283]	606[283]	276[94]	539[187]	539[187]
nw/nc/fw	(3997[2756-5074])	39[48]	151[47]	3899[854]	94[24]	96[23]	3844[855]
Am	(2011[433-4264])	327[263]	97[68]	413[241]	298[206]	126[99]	441[287]
Cnf	(2424[2424-2424])	472[564]	1801[564]	1953[564]	1170[2]	1103[2]	1255[2]

Table 2. Size of backdoor with no remove option

Family of Instances	(#V [min-Max])	<i>Maxdegree</i>			<i>MaxdegreeCycle</i>		
		#D	#CS	#B	#D	#CS	#B
Blocks	(484[283-758])	18[4]	0[0]	373[219]	18[4]	0[0]	373[219]
Logistics	(994[116-3016])	135[147]	25[48]	419[539]	152[178]	7[13]	401[509]
Pipe	(1642[834-2577])	1020[735]	219[215]	543[223]	956[513]	282[124]	606[283]
Facts	(3178[2218-4315])	488[127]	0[0]	2214[621]	488[127]	0[0]	2214[621]
Parity	(1044[64-3176])	318[426]	0[0]	497[480]	318[426]	0[0]	497[480]
Qg	(969[512-1331])	122[99]	138[87]	410[189]	181[60]	80[25]	351[140]
Ca	(637[26-2282])	317[433]	94[113]	317[392]	302[388]	109[151]	332[434]
Dp	(1427[213-3193])	724[643]	149[151]	513[357]	728[641]	145[143]	509[353]
Bmc2	(1952[316-4089])	680[706]	1[1]	868[883]	680[705]	1[1]	868[884]
Rand	(2217[2000-2500])	1591[418]	495[396]	625[401]	1200[129]	886[102]	1016[111]
Ezfact	(1441[193-3073])	48[23]	10[5]	1350[1064]	49[23]	9[5]	1349[1064]
Med	(761[341-1159])	14[4]	0[0]	302[157]	14[4]	0[0]	302[157]
Avg-checker	(917[648-1188])	302[100]	0[0]	512[181]	302[100]	0[0]	512[181]
nw/nc/fw	(3997[2756-5074])	73[14]	40[22]	3864[857]	95[24]	18[10]	3842[856]
Am	(2011[433-4264])	367[254]	0[0]	373[239]	367[254]	0[0]	373[239]
Cnf	(2424[2424-2424])	1988[12]	285[12]	437[12]	2210[6]	63[6]	215[6]

Table 3. Size of backdoor with remove option

However, in most cases, the remove option and the *MaxdegreeCycle* heuristic lead to smaller backdoors.

We are currently experimenting how such a promising preprocessing step can be grafted to the most efficient SAT solvers, allowing them to focus directly on the critical variables of the instances (i.e. the backdoor). Let us stress that our preprocessing step has been implemented in a non-optimized way. However, it shows really viable thanks to good obtained computing time (**less than 1 second in most cases**), so time is omitted in different tables.

## 9 Future works

Let us here simply motivate another interesting path for future research, related to the actual expressiveness of discovered clauses. Actually, our gate-oriented representation of a Boolean formula exhibits additional information that can prove powerful with respect to further steps of deduction or satisfiability checking. To illustrate this, let us consider Example 2 again. From the CNF  $\Sigma$ , the gate  $y = \wedge(x_1, x_2, x_3)$  is extracted. The clausal representation of the gate is given by  $\{y \vee \neg x_1 \vee \neg x_2 \vee \neg x_3, \neg y \vee x_1, \neg y \vee x_2, \neg y \vee x_3\}$ .

Clearly, the additional clauses  $\{\neg y \vee x_2, \neg y \vee x_3\}$  are resolvents from  $\Sigma$ , which can only be obtained using two and six basic steps of resolution, respectively. Accordingly, the gate representation of  $\Sigma$  involves non-trivial binary resolvents, which can ease further deduction or satisfiability checking steps. Taking this feature into account either in clausal-based or gate-based deduction of satisfiability solvers should be a promising path for future research. Also, some of the discovered gates represent equivalencies ( $x \Leftrightarrow y$ ), substituting equivalent literals might lead to further reductions with respect to the number of variables.

Another interesting path for future research concerns the analysis of the obtained graph and the use of e.g. decomposition techniques. To further reduce the size of the backdoor, we also plan to study how tractable parts of the formula (e.g. horn or horn-renommable) can be exploited.

## 10 Conclusions

Clearly, our experimentation results are encouraging. Dependent variables can be detected in a preprocessing step at a very low cost. Cycles occur, and they can be cut. We are currently grafting such a preprocessing technique to efficient SAT solvers. Our preliminary experimentations show that this proves often beneficial. Moreover, we believe that the study of cycles and of dependent variables can be essential in the understanding of the difficulty of hard SAT instances.

## 11 Acknowledgements

This work has been supported in part by the CNRS, the FEDER, the *IUT de Lens* and the *Conseil Régional du Nord/Pas-de-Calais*.

## References

1. First international competition and symposium on satisfiability testing, March 1996. Beijing (China).
2. L. Brisoux, L. Sais, and E. Grégoire. Recherche locale : vers une exploitation des propriétés structurelles. In *Actes des Sixièmes Journées Nationales sur la Résolution Pratique des Problèmes NP-Complets (JNPC'00)*, pages 243–244, Marseille, 2000.
3. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Journal of the Association for Computing Machinery*, 5:394–397, 1962.
4. Second Challenge on Satisfiability Testing organized by the Center for Discrete Mathematics and Computer Science of Rutgers University, 1993. <http://dimacs.rutgers.edu/Challenges/>.
5. Olivier Dubois, Pascal André, Yacine Boufkhad, and Jacques Carlier. Sat versus unsat. In D.S. Johnson and M.A. Trick, editors, *Second DIMACS Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, pages 415–436, 1996.
6. Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-sat formulae. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, volume 1, pages 248–253, Seattle, Washington (USA), August 4–10 2001.
7. E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin. Evaluating search heuristics and optimization techniques in propositional satisfiability. In *Proceedings of International Joint Conference on Automated Reasoning (IJCAR'01)*, Siena, June 2001.
8. Henry A. Kautz, David McAllester, and Bart Selman. Exploiting variable dependency in local search. In *Abstract appears in "Abstracts of the Poster Sessions of IJCAI-97"*, Nagoya (Japan), 1997.
9. Daniel Le Berre. Exploiting the real power of unit propagation lookahead. In *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*, Boston University, Massachusetts, USA, June 14th-15th 2001.
10. Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, pages 366–371, Nagoya (Japan), August 1997.
11. Shtrichman Oler. Tuning sat checkers for bounded model checking. In *Proceedings of Computer Aided Verification (CAV'00)*, 2000.
12. Grégoire E. Mazure B. Ostrowski R. and Sais L. Recovering and exploiting structural knowledge from cnf formulas. In *Eighth International Conference on Principles and Practice of Constraint Programming (CP'2002)*, pages 185–199, Ithaca (N.Y.), 2002. LNCS 2470, Springer Verlag.
13. Antoine Rauzy, Lakhdar Sais, and Laure Brisoux. Calcul propositionnel : vers une extension du formalisme. In *Actes des Cinquièmes Journées Nationales sur la Résolution Pratique des Problèmes NP-complets (JNPC'99)*, pages 189–198, Lyon, 1999.
14. Sat 2001: Workshop on theory and applications of satisfiability testing, 2001. <http://www.cs.washington.edu/homes/kautz/sat2001/>.
15. Sat 2002 : Fifth international symposium on theory and applications of satisfiability testing, May 2002. <http://gauss.ececs.uc.edu/Conferences/SAT2002/>.
16. Sat 2003 : Sixth international symposium on theory and applications of satisfiability testing, May 2003. <http://www.mrg.dist.unige.it/events/sat03/>.
17. Ryan Williams, Carla P. Gomez, and Bart Selman. Backdoors to typical case complexity. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, pages 1173–1178, 2003.
18. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of ICCAD'2001*, pages 279–285, San Jose, CA (USA), November 2001.