Annales Mathematicae et Informaticae
***35** (2008) pp. 75–94*
http://www.ektf.hu/ami

Simplifying the propositional satisfiability problem by sub-model propagation*

Gábor Kusper, Lajos Csőke, Gergely Kovásznai

Institute of Mathematics and Informatics
Eszterházy Károly College, Eger, Hungary

Submitted 30 September 2008; Accepted 8 December 2008

Abstract

We describes cases when we can simplify a general SAT problem instance by sub-model propagation. Assume that we test our input clause set whether it is blocked or not, because we know that a blocked clause set can be solved in polynomial time. If the input clause set is not blocked, but some clauses are blocked, then what can we do? Can we use the blocked clauses to simplify the clause set? The Blocked Clear Clause Rule and the Independent Blocked Clause Rule describe cases when the answer is yes. The other two independent clause rules, the Independent Nondecisive- and Independent Strongly Nondecisive Clause Rules describe cases when we can use nondecisive and strongly nondecisive clauses to simplify a general SAT problem instance.

Keywords: SAT, blocked clause, nondecisive clause

MSC: 03-04

1. Introduction

Propositional Satisfiability is the problem of determining, for a formula of the propositional calculus, if there is an assignment of truth values to its variables for which that formula evaluates the true. By SAT we mean the problem of propositional satisfiability for formulae in conjunctive normal form (CNF).

SAT is the first, and one of the simplest, of the many problems which have been shown to be NP-complete [7]. It is dual of propositional theorem proving, and many practical NP-hard problems may be transformed efficiently to SAT. Thus, a good SAT algorithm would likely have considerable utility. It seems improbable that a polynomial time algorithm will be found for the general SAT problem but we know

*Partially supported by TéT 2006/A-16.

that there are restricted SAT problems that are solvable in polynomial time. So a "good" SAT algorithm should check first the input SAT instance whether it is an instance of such a restricted SAT problem or can be simplified by a preprocess step. In this paper we introduce some possible simplification techniques. We list some polynomial time solvable restricted SAT problems:

1. The restriction of SAT to instances where all clauses have length k is denoted by k-SAT. Of special interest are *2-SAT* and *3-SAT*: 3 is the smallest value of k for which k-SAT is **NP**-complete, while 2-SAT is solvable in linear time [10, 1].

2. *Horn SAT* is the restriction to instances where each clause has at most one positive literal. Horn SAT is solvable in linear time [9, 19], as are a number of generalizations such as *renamable Horn SAT* [2], *extended Horn SAT* [5] and *q-Horn SAT* [3, 4].

3. The hierarchy of *tractable* satisfiability problems [8], which is based on Horn SAT and 2-SAT, is solvable in polynomial time. An instance on the k-th level of the hierarchy is solvable in $O(nk + 1)$ time.

4. *Nested SAT*, in which there is a linear ordering on the variables and no two clauses overlap with respect to the interval defined by the variables they contain [12].

5. SAT in which no variable appears more than twice. All such problems are satisfiable if they contain no unit clauses [20].

6. *r,r-SAT*, where r,s-SAT is the class of problems in which every clause has exactly r literals and every variable has at most s occurrences. All r,r-SAT problems are satisfiable in polynomial time [20].

7. A formula is *SLUR* (Single Lookahead Unit Resolution) *solvable* if, for all possible sequences of selected variables, algorithm SLUR does not give up. Algorithm SLUR is a nondeterministic algorithm based on unit propagation. It eventually gives up the search if it starts with, or creates, an unsatisfiable formula with no unit clauses. The class of SLUR solvable formulae was developed as a generalization including Horn SAT, renamable Horn SAT, extended Horn SAT, and the class of CC-balanced formulae [18].

8. *Resolution-Free* SAT Problem, where every resolution results in a tautologous clause, is solvable in linear time [16].

8. *Blocked* SAT Problem, where every clause is blocked, is solvable in polynomial time [13, 14, 17].

In this paper we describes cases when we can simplify a general SAT problem instance by sub-model propagation, which means hyper-unit propagating [15, 16] a sub-model [17]. Assume that we test our input clause set whether it is blocked or not, because we know [17] that a blocked clause set can be solved in polynomial time. If the input clause set is not blocked, but some clauses are blocked, then what can we do? Can we use the blocked clauses to simplify the clause set? The Blocked Clear Clause Rule and the Independent Blocked Clause Rule describe cases when the answer is yes.

The other two independent clause rules, the Independent Nondecisive- and Independent Strongly Nondecisive Clause Rules describe cases when we can use nonde-

cisive and strongly nondecisive clauses to simplify a general SAT problem instance.

The notion of blocked [13, 14] and nondecisive clause [11] was introduced by O. Kullmann and A. V. Gelder. They showed that a blocked or nondecisive clause can be added or deleted from a clause set without changing its satisfiability.

Intuitively a blocked clause has a liter on which every resolution in the clause set is tautology. A nondecisive clause has a literal on which every resolution in the clause set is either tautology or subsumed. We also use the notion of strongly nondecisive clause, which has a liter on which every resolution in the clause set is either tautology or entailed. We also use very frequently the notion of clear clause. A clause is clear if every variable which occurs in the clause set occurs also in this clause either positively or negatively. Note, that clear clauses are called also total or full clauses in the literature.

The Blocked Clear Clause Rule describes two cases. The two cases have a common property: the input clause set contains a blocked clear clause. In the first case the input clause set is a subset of CC, in the second case the blocked clear clause is not subsumed. In both cases the sub-model generated from the blocked clear clause and from one of its blocked literals is a model for the input clause set.

In both cases we need in the worst-case $O(n^2m^3)$ time to decide whether the input clause set fulfills the requirements of the Blocked Clear Clause Rule. We need $O(n^2m^3)$ time, because we have to check blocked-ness in both two cases, which is an $O(n^2m^2)$ time method, and not subsumed-ness in the second case, which is an $O(m)$ time method.

The Independent Blocked Clause Rule is a generalization of the Blocked Clear Clause Rule. We can apply it if we have a blocked clause and it subsumes a clear clause that is it not subsumed by any other clause from the clause set, i.e., the blocked clause is independent. In this case the sub-model generated from the independent blocked clause and from one of its blocked literals is a partial model, i.e., we can simplify the input clause set by propagating this sub-model.

Note that if we know the subsumed clear clause which is not subsumed by any other clause from the input clause set then we know the whole model. This applies for the other independent clause rules.

We need in the worst-case $O(2^n n^2m^3)$ time to decide whether the input clause set fulfills the requirements of the Independent Blocked Clause Rule. We need $O(2^n n^2m^3)$ time, because we have to check blocked-ness, which is an $O(n^2m^2)$ time method, and independent-ness, which is an $O(2^n m)$ time method.

The Independent Nondecisive Clause Rule is a generalization of the Independent Blocked Clause Rule. We can apply it if we have a independent nondecisive clause. In this case the sub-model generated from it and from one of its nondecisive literals is a partial model, i.e., we can simplify the input clause set by propagating this sub-model.

We need in the worst-case $O(2^n nm^4)$ time to decide whether the input clause set fulfills the requirements of the Independent Nondecisive Clause Rule. We need $O(2^n nm^4)$ time, because we have to check nondecisive-ness, which is an $Max\{O(n^2m^2), O(nm^3)\}$ time method, and independent-ness, which is an $O(2^n m)$

time method. We assume that $nm^3 > n^2m^2$.

The Independent Strongly Nondecisive Clause Rule is a generalization of the Independent Nondecisive Clause Rule. We can apply it if we have a independent strongly nondecisive clause. In this case the sub-model generated from it and from one of its strongly nondecisive literals is a partial model, i.e., we can simplify the input clause set by propagating this sub-model.

We need in the worst-case $O(2^{n+1}m)$ time to decide whether the input clause set fulfills the requirements of the Independent Strongly Nondecisive Clause Rule. We need $O(2^{n+1}m)$ time, because we have to check strongly nondecisive-ness, which is an $O(n^2)$ time method, and independent-ness, which is an $O(2^n m)$ time method.

Since the independent clause test is too expensive (it is exponential) we introduce some heuristics which can guess which clause might be independent. Furthermore, we introduce an algorithm which might find strongly nondecisive clauses in $O(n^3m^2)$ time.

2. Definitions

Set of variables, literals

Let V be a finite *set of Boolean variables*. The *negation of a variable* v is denoted by $\overline{v}$. Given a set U, we denote $\overline{U} := \{\overline{u} \mid u \in U\}$ and call the *negation of the set* U.

Literals are the members of the set $W := V \cup \overline{V}$. *Positive literals* are the members of the set V. *Negative literals* are their negations. If w denotes a negative literal $\overline{v}$, then $\overline{w}$ denotes the positive literal v.

Clause, clause set, assignment, assignment set

Clauses and *assignments* are finite sets of literals that do not contain simultaneously any literal together with its negation.

A clause is interpreted as disjunction of its literals. An assignment is interpreted as conjunction of its literals. Informally speaking, if an assignment A contains a literal v, it means that v has the value $True \in A$. A *clause set* or *formula* (formula in CNF form) is a finite set of clauses. A clause set is interpreted as conjunction of its clauses. If C is a clause, then $\overline{C}$ is an assignment. If A is an assignment, then $\overline{A}$ is a clause. The empty clause is interpreted as False. The empty assignment is interpreted as True. The empty clause set is interpreted as True.

The empty set is denoted by $\emptyset$. The *length* of a set U is its cardinality, denoted by $|U|$. The natural number n is the *number of variables*, i.e., $n := |V|$.

Cardinality, *k*-clause, clear clause, CC

If C is a clause and $|C| = k$, then we say that C is a *k-clause*. Special cases are *unit clauses* or *units* which are 1-clauses, and *clear* or *total clauses* which are

n-clauses. Note that any unit clause is at the same time a clause and an assignment.

In this paper we prefer the name clear clause instead of total or full clause. Although, total clause is used in the literature, in our point of view the name clear clause is more intuitive.

The clause set CC is the set of all clear clauses.

Subsumption, entailed-ness, independent-ness

The clause C *subsumes* the clause B iff C is a subset of B. The interpretation of the notion of subsumption is logical consequence, i.e., B is a logical consequence of C.

We say that a clause C is *subsumed by the clause set* S, denoted by $C \supseteq\in S$, iff there is a clause in S which subsumes it. We say that a clause C is *entailed by the clause set* S, denoted by $C \supseteq\in_{CC} S$, iff for any clear clause, which is subsumed by C, there is a clause in S which subsumes that clear clause.

The interpretation of the notion of subsumed and entailed is the same, logical consequence, i.e., C is a logical consequence of S. Note that if a clause is subsumed by a clause set then it is entailed, but not the other way around. Furthermore, if a clear clause is subsumed by a clause set then it is entailed and the other way around.

$$C \supseteq\in S :\Longleftrightarrow \mathit{Clause}(C) \wedge \mathit{ClauseSet}(S) \wedge \exists[B \in S]B \subseteq C.$$

$$C \supseteq\in_{CC} S :\Longleftrightarrow \mathit{Clause}(C) \wedge \mathit{ClauseSet}(S) \wedge \forall[D \in CC][C \subseteq D]\exists[B \in S]B \subseteq D.$$

We shall explain the intuition behind the notation $\supseteq\in$. If we rewrite its definition and leave out the "not interesting" parts (written in brackets) then we obtain this notation:

$$\exists[B \in S]B \subseteq C \iff (\exists[B])C \supseteq (B \wedge B) \in S \iff C \supseteq\in S.$$

We say that a clause C is *independent in clause set* S iff it is not entailed by S.

Clause difference, resolution

We introduce the notion of *clause difference*. We say that two clauses *differ in* some variables iff these variables occur in both clauses but as different literals. If A and B are clauses then the clause difference of them, denoted by $\mathrm{diff}(A, B)$, is

$$\mathrm{diff}(A, B) := A \cap \overline{B}.$$

If $\mathrm{diff}(A, B) \neq \emptyset$ then we say that A *differs from* B. Note that $\mathrm{diff}(A, B) = \overline{\mathrm{diff}(B, A)}$.

We say that *resolution can be performed* on two clauses iff they differ only in one variable. Note that this is not the usual notion of resolution, because we allow resolution only if it results in a non-tautologous resolvent. For example resolution cannot be performed on $\{v, w\}$ and $\{\overline{v}, \overline{w}\}$ but can be performed on $\{v, w\}$ and

$\{\overline{v}, z\}$. If resolution can be performed on two clauses, say A and B, then the *resolvent*, denoted by $\mathrm{Res}(A, B)$, is their union excluding the variable they differ in:

$$\mathrm{Res}(A, B) := (A \cup B) \setminus (\mathrm{diff}(A, B) \cup \mathrm{diff}(B, A)).$$

Note that if we interpret $\mathrm{Res}(A, B)$ as a logical formula then it is a logical consequence of the clauses A and B.

Pure literal, blocked- literal, clause, clause set

We say that a literal $c \in C$ is *blocked in* the clause C and in the clause set S iff for each clause B *in*S which contains $\overline{c}$ we have that there is a literal $b \in B$ such that $b \neq \overline{c}$ and $\overline{b} \in C$. A *clause is blocked in a clause set* iff it contains a blocked literal. A *clause set is blocked* iff all clauses are blocked in it. We denote these notions by $Blck(c, C, S)$, $Blck(C, S)$ and $Blck(S)$, respectively.

Note that if literal $c \in C$ is blocked in C, S then for all $B \in S, \overline{c} \in B$ we have that resolution cannot be performed on C and B. This means that this clause is "blocked" against resolution.

We say that a literal is *pure* in a clause set if its negation does not occur in the clause set. Note that pure literals are blocked.

(Weakly/strongly) nondecisive- literal, clause, clause set

We define formally the notion of *weakly nondecisive* literal, clause and clause set. We denote these notions by $WnD(c, C, S)$, $WnD(C, S)$ and $WnD(S)$, respectively.

$$WnD(c, C, S) : \Longleftrightarrow \forall[B \in S][\overline{c} \in B](\exists[b \in B][b \neq \overline{c}]\overline{b} \in C \vee \mathrm{Res}(C, B) \supseteq \in S).$$

$$WnD(C, S) : \Longleftrightarrow \exists[c \in C]WnD(c, C, S).$$

$$WnD(S) : \Longleftrightarrow \forall[C \in S]WnD(C, S).$$

We define formally the notion of*nondecisive* literal, clause and clause set. We denote these notions by $NonD(c, C, S)$, $NonD(C, S)$ and $NonD(S)$, respectively.

$$NonD(c, C, S) : \Longleftrightarrow$$

$$\forall[B \in S][\overline{c} \in B](\exists[b \in B][b \neq \overline{c}]\overline{b} \in C \vee \mathrm{Res}(C, B) \cup \{c\} \supseteq \in S \setminus \{C\}).$$

$$NonD(C, S) : \Longleftrightarrow \exists[c \in C]NonD(c, C, S).$$

$$NonD(S) : \Longleftrightarrow \forall[C \in S]NonD(C, S).$$

We define formally the notion of *strongly nondecisive* literal, clause and clause set. We denote these notions by $SND(c, C, S)$, $SND(C, S)$ and $SND(S)$, respectively.

$$SND(c, C, S) : \Longleftrightarrow$$

$$\forall[B \in S][\overline{c} \in B](\exists[b \in B][b \neq \overline{c}]\overline{b} \in C \vee \mathrm{Res}(C, B) \cup \{c\} \supseteq \in_{CC} S \setminus \{C\}).$$

$$SND(C, S) : \Longleftrightarrow \exists[c \in C]SND(c, C, S).$$

$$SND(S) : \Longleftrightarrow \forall[C \in S]SND(C, S).$$

Resolution-mate, sub-model

If C is a clause and c is a literal in C then the *resolution-mate* of clause C by literal c, denoted by $\mathrm{rm}(C, c)$, is

$$\mathrm{rm}(C, c) := (C \cup \{\overline{c}\}) \setminus \{c\}.$$

Note that resolution can be always performed on C and $\mathrm{rm}(C, c)$, and

$$\mathrm{Res}(C, \mathrm{rm}(C, c)) = C \setminus \{c\}.$$

This means that we obtain a shorter clause.

The *sub-model* generated from the clause C and from the literal c, denoted by $\mathrm{sm}(C, c)$, is

$$\mathrm{sm}(C, c) := \overline{\mathrm{rm}(C, c)}.$$

We say that C and c are the *generator* of $\mathrm{sm}(C, c)$. The name "sub-model" comes from the observation that in a resolution-free clause set an assignment created from one of the shortest clauses in this way is a part of a model [16], i.e., a sub-model.

Note that $\mathrm{rm}(C, c)$ is a clause but $\mathrm{sm}(C, c)$ is an assignment.

The sub-model $\mathrm{sm}(C, c)$ is a special assignment which always satisfies clause C, since it sets literal c to be True.

Model, (un)satisfiable

An assignment M is a *model for a clause set* S iff for all $C \in S$ we have $M \cap C \neq \emptyset$.

A clause set is *satisfiable* iff there is a model for it. A clause set is *unsatisfiable* iff it is not satisfiable. A clause set is *trivially satisfiable* iff it is empty and it is *trivially unsatisfiable* if it contains the empty clause.

3. The Blocked Clear Clause Rule

In this section we introduce the Blocked Clear Clause Rule, a generalization of the Clear Clause Rule. This rule is introduced by the author.

Assume we test our input clause set whether it is blocked or not, because we know [17] that a blocked clause set can be solved in polynomial time. If the input clause set is not blocked, but some clauses are blocked, then what can we do? Can we use the blocked clauses to simplify the clause set? If it contains a not subsumed blocked clear clause, we can. This is what the Blocked Clear Clause Rule states.

It has two variants. The first one states that if a clause set contains only clear clauses and one of them is blocked then the sub-model generated from this blocked clause and from one of its blocked literal is a model. This is a very rare case, but since we can construct for each clause set the equivalent clear clause set, this rule plays an important role.

The second one states that if a clause set contains a not subsumed blocked clear clause then the sub-model generated from it and from one of its blocked literals is a model. This case is still a very rare one, but might occur more frequently as the first variant.

Lemma 3.1 (Blocked Clear Clause Rule). *Let S be a clause set. Let $C \in S$ be a blocked and clear clause. Let $a \in C$ be a blocked literal C, S.*
(a) *If S is a subset of CC, then $sm(C, a)$ is a model for S.*
(b) *If C is not subsumed by $S \setminus \{C\}$, then $sm(C, a)$ is a model for S.*

Proof. (a) To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap \mathrm{sm}(C, a)$ is not empty. Since S is a subset of CC we know that B is a clear clause. Hence, there are two cases, either $a \in B$ or $\overline{a} \in B$.

In case $a \in B$ we have, by definition of sub-model, that $a \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

In case $\overline{a} \in B$, since $a \in C$ is blocked in C, S we know, by definition of blocked literal, that for some $b \in B$ we have $b \neq \overline{a}$ and $\overline{b} \in C$. From this, by definition of sub-model, we know that $b \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

Hence, if S is a subset of CC, then $\mathrm{sm}(C, a)$ is a model for S.

(b) To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap \mathrm{sm}(C, a)$ is not empty. Since C is not subsumed by $S \setminus \{C\}$ we know, by definition of subsumption, that $B \not\subseteq C$. From this, since C is a clear clause we know that for some $b \in B$ we have $\overline{b} \in C$. There are two cases, either $\overline{b} = a$ or $\overline{b} \neq a$.

In the first case we have $\overline{b} = a$, i.e., $\overline{a} \in B$. From this since $a \in C$ is blocked in C, S we know, by definition of blocked literal, that for some $d \in B$ we have that $d \neq \overline{a}$ and $\overline{d} \in C$. From this, by definition of sub-model, we know that $d \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

In the second case we have $\overline{b} \neq a$. From this and from $\overline{b} \in C$ we know, by definition of sub-model, that $b \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

Hence, If C is not subsumed by $S \setminus \{C\}$, then $\mathrm{sm}(C, a)$ is a model for S. □

An alternative proof idea is that we say that it suffices to show that the resolution-mate of C ($\mathrm{rm}(C, a)$) is not subsumed by S. Then we know, by Clear Clause Rule, that its negation ($\mathrm{sm}(C, a)$) is a model.

This alternative proof idea shows in which sense say we that the Blocked Clear Clause Rule is a generalization of the Clear Clause Rule.

This rule is the base of the independent clause rules. Therefore, it is very important for us.

4. The Independent Blocked Clause Rule

In this section we introduce the Independent Blocked Clause Rule, a generalization of the Blocked Clear Clause Rule. This rule is introduced by the author.

The Independent Blocked Clause Rule states that if a clause set contains an independent blocked clause, then it is satisfiable and a sub-model generated from this clause and from one of its blocked literals is a partial model, i.e., we can simplify the clause set by propagating this sub-model. These requirements are fulfilled quite often by real or benchmark problems, but checking independent-ness is expensive.

We know that a clause $A \in S$ is independent in the clause set $S \setminus \{A\}$ if it is not entailed by $S \setminus \{A\}$. The formal definition is the following:

$$A \text{ independent in } S : \Longleftrightarrow \exists[C \in CC][A \subseteq C]\forall[B \in S][B \neq A]B \not\subseteq C.$$

The following algorithm checks whether the input clause is independent or not in the input clause. If it is independent, then it returns a clear clause subsumed by the input clause but not subsumed by any other clause from the input clause set. Otherwise, it returns the empty clause. In the worst-case it uses $O(2^n m)$ time, because it follows the definition of independent, and there we have two quantifiers, one on CC which has 2^n elements, the other on the input clause set, which has m elements.

Independent clause test

```
1  function IsIndependent(S : clause set, A : clause) : clause
2  begin
3    for each C ∈ CC, A ⊆ C do
4      B_notsubsumes_C := True;
5      for each B ∈ S, B ≠ A while B_notsubsumes_C is True do
6        if (B ⊆ C) then B_notsubsumes_C := False;
7      od
8      if (B_notsubsumes_C) then return C;
9        // In this case we found a suitable C, we return it.
10   od
11   return ∅;
12     // In this case we found no suitable clause.
13     // Therefore, we return the empty clause.
14 end
```

One can see that the independent clause test is very expensive (exponential). We will discuss later how can we get around this problem by suitable heuristics.

Lemma 4.1 (Independent Blocked Clause Rule). *Let S be a clause set. Let $A \in S$ be blocked in S and independent in $S \setminus \{A\}$. Let $a \in A$ be a blocked literal in A, S. Then there is a model M for S such that $sm(A, a) \subseteq M$.*

Proof. We know that A is independent in $S \setminus \{A\}$. Hence, by definition of independent, we know that there is a clear clause C that is subsumed by A and not subsumed by any other clause in S. Since $A \subseteq C$ we know that $\text{sm}(A, a) \subseteq \text{sm}(C, a)$. Hence, it suffices to show that $\text{sm}(C, a)$ is a model for S. To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap \text{sm}(C, a)$ is not empty. The remaining part of the proof is the same as the proof of the (b) variant of the Blocked Clear Clause Rule.

Hence, $B \cap \text{sm}(C, a)$ is not empty. Hence, there is a model M for S such that $\text{sm}(A, a) \subseteq M$. □

This proof is traced back to the proof of Blocked Clear Clause Rule. We can do this because we know that there is a clear clause which is blocked and not entailed by $S \setminus \{A\}$. We know that for clear clauses the notion of subsumed and entailed are the same.

The proof of this lemma shows that if we perform an independent clause check and we find a clear clause which is subsumed by only one clause, then we know the whole model ($\text{sm}(C, a)$) and not only a part of the model ($\text{sm}(A, a)$). But usually we do not want to perform expensive independent-ness checks. How can we get around this problem? The solution is a heuristic which tells us which blocked clause could be independent.

Such a heuristic could be for instance the selection of the shortest blocked clause. The shortest clause subsumes the largest number of clear clauses. Therefore, it has a good chance to be independent, but there is no guarantee for it. We give more details about heuristics after the discussion of the simplifying rules.

5. The Independent Nondecisive Clause Rule

In this section we introduce the Independent Nondecisive Clause Rule, a generalization of the Independent Blocked Clause Rule. This rule is introduced by the author.

The Independent Nondecisive Clause Rule states that if a clause set contains an independent nondecisive clause, then it is satisfiable and a sub-model generated from this clause and from one of its nondecisive literals is a partial model, i.e., we can simplify the clause set by propagating this sub-model. These requirements are fulfilled quite often by real or benchmark problems, but checking independent-ness is expensive.

Lemma 5.1 (Independent Nondecisive Clause Rule). *Let S be a clause set. Let $A \in S$ be nondecisive in S and independent in $S \setminus \{A\}$. Let $a \in A$ be a nondecisive literal in A, S. Then there is a model M for S such that $sm(A, a) \subseteq M$.*

Proof. We know that A is independent in $S \setminus \{A\}$. Hence, by definition of independent, we know that there is a clear clause C that is subsumed by A and not subsumed by any other clause in S. Since $A \subseteq C$ we know that $\text{sm}(A, a) \subseteq \text{sm}(C, a)$. Hence, it suffices to show that $\text{sm}(C, a)$ is a model for S. To show this, by definition

of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap \mathrm{sm}(C, a)$ is not empty. There are three cases: either (a) $a \in B$ or (b) $\overline{a} \in B$ or (c) $a \notin B$ and $\overline{a} \notin B$.

In case (a) we have $a \in B$. From this and from the definition of sub-model we know that $a \in B \cap \mathrm{sm}(C, a)$.

In case (b) we have $\overline{a} \in B$. From this and from $a \in A$ is nondecisive in A, S, by definition of nondecisive literal, we know that either there is a literal $b \in B$ which has $b \neq \overline{a}$ and $\overline{b} \in A$ or there is a clause $D \in S, D \neq A$ which has $D \subseteq A \cup B\{\overline{a}\}$.

In the first case we know, by definition of sub-model, that $b \in \mathrm{sm}(A, a)$.

In the second case since C is independent in $S \setminus \{A\}$, by definition of independent, we know that D does not subsume C, i.e., for some $d \in D$ we have $d \notin C$. From this and from $A \subseteq C$ and from $D \subseteq A \cup B\{\overline{a}\}$ we can show that $d \notin A$, $d \in B$ and $d \neq \overline{a}$. From $d \notin C$ we know, by definition of clear clause, that $\overline{d} \in C$. Hence, by definition of sub-model, $d \in B \cap \mathrm{sm}(C, a)$.

In case (c) we have $a \notin B$ and $\overline{a} \notin B$. Since C is not subsumed by $S \setminus \{A\}$ we know, by definition of subsumption, that $B \not\subseteq C$. From this, since C is a clear clause we know that for some $b \in B$ we have $\overline{b} \in C$. There are two cases, either $\overline{b} = a$ or $\overline{b} \neq a$.

In the first case we have $\overline{b} = a$, i.e., $\overline{a} \in B$. But we already know that $\overline{a} \notin B$. Hence, this is a contradiction.

In the second case we have $\overline{b} \neq a$. From this and from $\overline{b} \in C$ we know, by definition of sub-model, that $b \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

Hence, there is a model M for S such that $\mathrm{sm}(A, a) \subseteq M$. □

This lemma is more powerful then the Independent Blocked Clause Rule, because each blocked clause is nondecisive but not the other way around.

6. The Independent Strongly Nondecisive Clause Rule

In this section we introduce the Independent Strongly Nondecisive Clause Rule, a generalization of the Independent Nondecisive Clause Rule. This rule is introduced by the author.

The Independent Strongly Nondecisive Clause Rule states that if a clause set contains an independent strongly nondecisive clause, then it is satisfiable and a sub-model generated from this clause and from one of its strongly nondecisive literals is a partial model, i.e., we can simplify the clause set by propagating this sub-model. These requirements are fulfilled very often by 3-SAT benchmark problems, but checking independent-ness and strongly nondecisive-ness is expensive.

We will see from our test result that the Independent Blocked Clause Rule can be applied only on few 3-SAT instances. The Independent Nondecisive Rule is better, but still can be applied only on every tenth benchmark problem. Therefore, we tried to find an even more powerful simplification rule. Finally, we found the Independent Strongly Nondecisive Clause Rule.

The idea is the following: We know that a nondecisive clause is either blocked or a special construction ($\mathrm{Res}(A, B) \cup \{a\}$) is subsumed. This rings a bell. If we would use the notion of entailed instead of subsumed then the rule would be more powerful. Let us check whether this idea works or not.

Lemma 6.1 (Independent Strongly Nondecisive Clause Rule). *Let S be a clause set. Let $A \in S$ be strongly nondecisive in S and independent in $S \setminus \{A\}$. Let $a \in A$ be a strongly nondecisive literal in A, S. Then there is a model M for S such that $sm(A, a) \subseteq M$.*

Proof. We know that A is independent in $S \setminus \{A\}$. Hence, by definition of independent, we know that there is a clear clause C that is subsumed by A and not subsumed by any other clause in S. Since $A \subseteq C$ we know that $\mathrm{sm}(A, a) \subseteq \mathrm{sm}(C, a)$. Hence, it suffices to show that $\mathrm{sm}(C, a)$ is a model for S. To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap \mathrm{sm}(C, a)$ is not empty. There are three cases, either (a) $a \in B$ or (b) $\overline{a} \in B$ or (c) $a \notin B$ and $\overline{a} \notin B$.

In case (a) we have $a \in B$. From this and from the definition of sub-model we know that $a \in B \cap \mathrm{sm}(C, a)$.

In case (b) we have $\overline{a} \in B$. From this and from $a \in A$ is nondecisive in A, S, by definition of nondecisive literal, we know that either there is a literal $b \in B$ which has $b \neq \overline{a}$ and $\overline{b} \in A$ or $\mathrm{Res}(A, B) \cup \{a\})$ is entailed in $S \setminus \{A\}$.

In the first case we know, by definition of sub-model, that $b \in \mathrm{sm}(A, a)$.

In the second case we know that $\mathrm{Res}(A, B) \cup \{a\})$ is entailed in $S \setminus \{A\}$. From this we know, by definition of entailed, that

$$\forall [D \in CC][A \cup B \setminus \{\overline{a}\} \subseteq D] \exists [E \in S][E \neq A] E \subseteq D.$$

From this we know that there is a literal $b \in B, b \neq a$ such that $b \notin C$ because otherwise we would have that $A \cup B \setminus \{\overline{a}\} \subseteq C$, which would mean that C is subsumed in $S \setminus \{A\}$, which would be a contradiction. From $b \notin C$ we know, by definition of clear clause, that $\overline{b} \in C$. From $b \neq a$ we know, by definition of sub-model, that $b \in \mathrm{sm}(C, a)$. Hence, $b \in B \cap \mathrm{sm}(C, a)$.

In case (c) we have $a \notin B$ and $\overline{a} \notin B$. Since C is not subsumed by $S \setminus \{A\}$ we know, by definition of subsumption, that $B \not\subseteq C$. From this, since C is a clear clause we know that for some $b \in B$ we have $\overline{b} \in C$. There are two cases, either $\overline{b} = a$ or $\overline{b} \neq a$.

In the first case we have $\overline{b} = a$, i.e., $\overline{a} \in B$. But we already know that $\overline{a} \notin B$. Hence, this is a contradiction.

In the second case we have $\overline{b} \neq a$. From this and from $\overline{b} \in C$ we know, by definition of sub-model, that $b \in \mathrm{sm}(C, a)$. Hence, $B \cap \mathrm{sm}(C, a)$ is not empty.

Hence, there is a model M for S such that $\mathrm{sm}(A, a) \subseteq M$. □

Note that $\mathrm{Res}(A, B) \cup \{a\} = A \cup B \setminus \{\overline{a}\}$.

We see that this proof is almost the same as the proof of the Independent Nondecisive Clause Rule except for the second part of case (b). Here we use the

following idea: C is subsumed by A but not by $A \cup B \setminus \{\overline{a}\}$, hence there is a literal $b \in B$ which has $b \neq a$ and $b \notin C$.

So the Independent Strongly Nondecisive Clause Rule works. But to decide whether we can apply it or not we have to perform an entailed-ness check, which is an exponential time method.

What can we do? There are some special cases when it is easy to check entailed-ness. For example the clause E is entailed in the clause set S if we have $E \in S$ or there is a clause $B \in S$ which simply subsumes E. This cases are very rare. The case we are going to describe occurs very often in 3-SAT problem instances.

Assume that we want to check whether the clause E is entailed in the clause set S. Assume we found a clause $D \in S$ which has the following two properties: (a) $\mathrm{diff}(E, D) = \emptyset$ and (b) $D \setminus E$ is a singleton. The first property is needed otherwise D could not subsume any clear clause subsumed by E. The second property says that D subsumes the "half" of E.

Assume that $D \setminus E = \{d\}$. Then D subsumes all clear clauses which are the superset of $E \cup \{d\}$. If E subsumes $2k$ clear clauses and $d \notin E$ then $E \cup \{d\}$ subsumes k clear clauses and $E \cup \{\overline{d}\}$ subsumes the remaining k clear clauses. Hence, we can say that D subsumes the "half" of E. So we can reduce the problem to whether $E \cup \{\overline{d}\}$, the remaining "half", is entailed in S or not. We call this step to cut E in half.

This situation occurs very often in 3-SAT problem instances, because our $E = A \cup B \setminus \{\overline{a}\}$ has a length of 5, clauses in the input clause set have a length of 3, and usually we have $n \gg 5$, where n is the number of variables. This means that it is very likely that we can use this step at least once.

The following algorithm uses this step to find strongly nondecisive clauses. In the worst-case it is a $O(n^3m^2)$ time method, but there is no guarantee that it finds any strongly nondecisive clauses.

GetSNDClauses

```
1  function GetSNDClauses(S : clauseset) : array of ⟨clause, literal⟩
2  begin
3    i := 0;
4    // We need i to index the array SND.
5    for each A ∈ S do
6      a_is_snd := False;
7      for each a ∈ A while a_is_snd is False do
8        B_snds_a := True;
9        for each B ∈ S, ā ∈ B while B_snds_a is True do
10         b_blocks_a := False;
11         D_subsumes_E := False;
12         B := B \ {ā};
```

```
13          if (diff(B, A) ≠ ∅) then b_blocks_a := True;
14          else
15            E := A ∪ B;
16            for each D ∈ S, D ≠ A while D_subsumes_E is False do
17              if (D ⊆ E) then D_subsumes_E := True;
18              if (diff(D, E) = ∅ ∧ |D \ E| = 1) then
19                E := E ∪ (D \ E);
20                Restart the last loop ;
21                // We have to restart the loop on clauses D,
22                // because the remaining half could be subsumed
23                // by a clause, which was already considered.
24              fi
25            od
26          fi
27          if (¬b_blocks_a ∧ ¬D_subsumes_E) then B_snds_a := False;
28        od
29        if (B_snds_a) then a_is_snd := True;
30      od
31      if (a_is_nond) then (SND[i], i) := (⟨A, a⟩, i + 1);
32    od
33    return SND;
34  end
```

The new rows are the ones from 14 till 26. We use in the 20th row a very interesting solution, we restart the innermost loop. We discuss this issue a bit later.

One can see that this algorithm returns an array of ordered pairs. An ordered pair contains a strongly nondecisive clause C and a strongly nondecisive literal $c \in C$.

Note that this algorithm might not find all strongly nondecisive clauses, because it does not use entailed-ness check, but the "cut E in half" step, described above.

This algorithm is an $O(n^3m^2)$ time method in the worst-case, where n is the number of variables and m is the number of clauses of the input clause set. It is an $O(n^3m^2)$ time method, because we have two loops on clauses and two loops on literals, but the innermost loop might be restarted n times in the worst-case.

One might ask, why do we need to restart the innermost loop? Assume we have the situation that we can cut E in half, i.e., we have found a clause $D \in S, D \neq A$ which has $\text{diff}(D, E) = \emptyset$ and $D \setminus E$ is a singleton. Then there is no D' clause among the ones we already considered such that D' subsumes $E \cup (D \setminus E)$, because D' fulfills the same requirements as D, i.e., it would be already used to cut E in

half. Then why should we restart?

That is true, but there might be clauses among the ones we already considered which can cut the new E in half and in the rest of the clause set there is no suitable clause which subsumes E or can cut it in half. Therefore, we have to restart the innermost loop.

7. Heuristics

In this subsection we introduce three heuristics. All of them are suitable more or less to guess whether a clause is independent or not.

All three heuristics are based on the following idea. A clause A is independent in the clause set $S \setminus \{A\}$ if $\overline{A}$ is a subset of a model of S, i.e., after propagating $\overline{A}$ on S, let us call the resulting clause set S', S' is satisfiable. Of course we do not want to perform expensive satisfiability checks, but we want to guess whether it is satisfiable or not. The idea is the following: the less clauses are contained in S', the more likely is that it is satisfiable.

This means that we have to count the clauses in S'. But propagation of an assignment is still to expensive for us. Therefore, we count the clauses in the following set:

$$\{B \mid B \in S \wedge \mathrm{diff}(A, B) = \emptyset\}.$$

Note that if a clause C is in this set then the clause $C' = C \setminus A$ is element of S'.

In the first version, called $IBCR$-1111, we just count each blocked clause A the clauses B that have $\mathrm{diff}(A, B) = \emptyset$ and we choose the one for which this number is the smallest.

Our test results on 3-SAT problem instances shows that this heuristic provides an independent blocked clause in 68% of the cases if there is an independent blocked clause.

In the other two versions we use weights.

In the second version, called $IBCR$-1234, we count each blocked clause A the clauses B which has $\mathrm{diff}(A, B) = \emptyset$ and we choose the one for which this number is the smallest. But we count clauses B with different weights. The weight W_B is

$$W_B := 1 + |A \cap B|.$$

For example if A is a 3-clause and $|A \cap B| = 2$ then $W_B = 3$.

Our test results on 3-SAT problem instances shows that this heuristic provides an independent blocked clause in 71% of the cases if there is an independent blocked clause.

In the third version, called $IBCR$-1248 the weight W_B is

$$W_B := 2^{|A \cap B|}.$$

For example if A is a 3-clause and $|A \cap B| = 2$ then $W_B = 4$.

Our test results on 3-SAT problem instances shows that this heuristic provides an independent blocked clause in 73% of the cases if there is an independent blocked clause.

After this short overview we give more details. First we have to explain the names of the three heuristics: $IBCR$-1111, $IBCR$-1234, and $IBCR$-1248. The word "IBCR" is just the abbreviation of Independent Blocked Clause Rule.

We have tested these heuristics on 3-SAT problem instances, where $|A \cap B|$ can be 0, 1, 2, or 3. The remaining part of the names comes from the values of weights. In the first heuristic the weight is the constant 1. Therefore, its name is $IBCR$-1111. In the second one the weight is defined by $1+|A \cap B|$, i.e., the weights are 1, 2, 3 or 4, respectively. Therefore, its name is $IBCR$-1234. In the third one the weights are 1, 2, 4, 8, respectively. Therefore, its name is $IBCR$-1248.

We present the pseudo-code of the third variant. This algorithm is an $O(n^2m^2)$ time method in the worst-case, where n is the number of variables and m is the number of clauses in the input clause set. It is an $O(n^2m^2)$ time method, because we have two loops on clauses and other two on literals.

IBCR-1248

```
1  function IBCR-1248(S : clause set) : ⟨clause, literal⟩
2  begin
3    min_Counter := Infinite;
4      // The variable min_Counter stores the minimum value of Counter.
5      // First time should be big enough.
6    for each A ∈ S do
7      a_is_blocked := False;
8      for each a ∈ A while a_is_blocked is False do
9          // Here begins the code which is relevant for the heuristic
10       Counter := 0;
11       B_blockes_a := True;
12       for each B ∈ S while B_blocks_a is True do
13         if (diff(A, B) = ∅) then Counter := Counter + 1 * (2^|A∩B|);
14           // The weight is 2^|A∩B|.
15         if (ā ∉ B) then continue ;
16           // Remember, we have to visit all B ∈ S which has ā ∈ B
17           // to decide whether a ∈ A is blocked or not.
18         b_blocks_a := False;
19         for each b ∈ B, b ≠ ā while b_blocks_a is False do
20           if (b̄ ∈ A) then b_blocks_a := True;
21         od
22         if (¬b_blocks_a) then B_blocks_a := False;
```

```
23          od
24          if (B_blocks_a) then a_is_blocked := True;
25          if (a_is_blocked and (Counter < min_Counter)) then
26             (min_Counter, min_A, min_a) := (Counter, A, a);
27          fi
28       od
29    od
30    return ⟨min_A, min_a⟩;
31 end
```

From this algorithm one can easily construct the other two or even other heuristics.

We can see that this heuristic returns a clause, say C, and a literal, say c. The clause C is a blocked clause and the literal c is a blocked literal in it. The heuristic state that C is independent. But this might be false.

If it is true, then it is fine because we can simplify our input clause set by a sub-model propagation using $\mathrm{sm}(C, c)$.

If it is false, then we still can gain something. We can add a shorter clause than C, because, by the Lucky Failing Property of Sub-Models, we know that $C \setminus \{c\}$ is entailed by the input clause set.

We do not know which case will be applied but we hope that the first one occurs more frequently.

These heuristics do not use the fact that the clause is blocked or not. Therefore, we can generalize them very easily for guessing independent-ness of (strongly) nondecisive clauses.

In the names of these heuristics we use the following acronyms: INCR for Independent Nondecisive Clause Rule; ISNCR for Independent Strongly Nondecisive Clause Rule.

8. Test results

In this section we describe shortly our java implementation of the simplification rules and we present the test results we have got on problems from the SATLIB problem library.

Our java implementation has three classes, Clause, ClauseSet and Satisfiable. The class Clause contains two BitSet objects, *positive* and *negative*. If we represent a clause where the first variable occurs positively then the first bit of the BitSet *positive* is set (1) and the first bit of BitSet *negative* is clear (0). This means that our implementation is close to the Literal Matrix View.

This implementation is not competitive with the newest SAT solvers because it does not use enhanced data structures or techniques like back jumping but it is

good enough to test whether the simplification rules can be applied on benchmark problems or not.

We have tested the heuristics on Uniform Random-3-SAT problems [6] from the SATLIB – Benchmark Problems homepage:
`http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html`

We used the smallest problem set, uf20-91.tar.gz, which contains 1000 problems, each has 91 clauses and 20 variables and is satisfiable.

We used a Pentium 4, 2400 MHz PC machine with 1024 MB memory to perform the tests.

Here we present our test results for the problems of uf20-91.tar.gz as a table ($IBCR$: Independent Blocked Clause Rule, $INCR$: Independent Nondecisive Clause Rule, $ISNCR$: Independent Strongly Nondecisive Clause Rule):

	$IBCR$	$INCR$	$ISNCR$	from
SND clauses:	601	1128	61122	91000
Problems with SND:	256	465	1000	1000
Independent SND:	77	125	4011	91000
Prob.s with indep. SND:	60	102	951	1000
X-1111:	41 / 60	61 / 102	89 / 951	
X-1234:	43 / 60	72 / 102	142 / 951	
X-1248:	44 / 60	76 / 102	166 / 951	

By "SND clauses" we mean in the column of Independent Blocked Clause Rule blocked clauses, in the next column nondecisive clauses, and in the next column strongly nondecisive clauses. The column "from" shows how many clauses and clause sets, respectively, do we have in total.

The line X-1111: 41 / 60 61 / 102 89 / 951 means that: $IBCR$-1111 successfully guesses 41 times an independent blocked clause from the 60 cases where we checked whether we have independent blocked clauses; $INCR$-1111 is successful 61 times from 102; and $ISNCR$-1111 is successful 89 times from 951.

Now we give the same table but the results are given in percentages.

	$IBCR$	$INCR$	$ISNCR$	from
SND clauses:	0.66%	1.23%	67.16%	91000
Problems with SND:	25.6%	46.5%	100%	1000
Independent SND:	0.08%	0.13%	4.4%	91000
Prob.s with indep. SND:	6%	10.2%	95.1%	1000
X-1111:	68.334%	59.8%	9.35%	
X-1234:	71.667%	70.58%	14.93%	
X-1248:	73.334%	74.5%	17.45%	

We can see that the X-1248 is the best heuristic, but still it could guess an independent strongly nondecisive clause only in 17% of the cases where we know that there are some.

It is so because it is very hard to guess independent clauses. We have better results in the other two cases because there are a lot of instances where we have only one or two independent blocked or nondecisive clauses. One can see that only the 0.66% of clauses are blocked while 67% are strongly nondecisive.

We believe that these simplifications are very useful, because if it turns out that the selected blocked clause is not independent, after propagating a sub-model generated from it, then we can still, by the Lucky Failing Property of Sub-Models, add a shorter clause to our clause set.

References

[1] Aspvall, B., Plass, M.F., Tarjan, R.E., A linear-time algorithm for testing the truth of certain quantified boolean formulas, *Information Processing Letters*, 8(3) (1979) 121–132.

[2] Aspvall, B., Recognizing disguised NR(1) instances of the satisfiability problem, *J. of Algorithms*, 1 (1980) 97–103.

[3] Boros, E., Hammer, P.L., Sun, X., Recognition of q-Horn formulae in linear time, *Discrete Applied Mathematics*, 55 (1994) 1–13.

[4] Boros, E., Crama, Y., Hammer, P.L., Saks, M., A complexity index for satisfiability problems, *SIAM J. on Computing*, 23 (1994) 45–49.

[5] Chandru, V., Hooker, J., Extended Horn sets in propositional logic, *J. of the ACM*, 38(1) (1991) 205–221.

[6] Cheeseman, P., Kanefsky, B., Taylor, W.M., Where the really hard problems are, *Proceedings of the IJCAI-91*, (1991) 331–337.

[7] Cook, S.A., The complexity of theorem-proving procedures, *Proceedings of the 3rd ACM Symposium on Theory of Computing*, (1971) 151–158.

[8] Dalal, M., Etherington, D.W., A hierarchy of tractable satisfiability problems, *Information Processing Letters*, 44 (1992) 173–180.

[9] Dowling, W.F., Gallier, J.H., Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. of Logic Programming*, 1(3) (1984) 267–284.

[10] Even, S., Itai, A., Shamir, A., On the complexity of timetable and multicommodity flow problems, *SIAM J. on Computing*, 5(4) (1976) 691–703.

[11] Gelder, A.V., Propositional search with k-clause introduction can be polynomially simulated by resolution, *Proceedings of the 5th International Symposium on Artificial Intelligence and Mathematics*, 1998.

[12] Knuth, D.E., Nested satisfiability, *Acta Informatica*, 28 (1990) 1–6.

[13] Kullmann, O., New methods for 3-SAT decision and worst-case analysis, *Theoretical Computer Science*, 223(1-2) (1999) 1–72.

[14] Kullmann, O., On a generalization of extended resolution, *Discrete Applied Mathematics*, 96-97(1-3) (1999) 149–176.

[15] Kusper, G., Solving the SAT problem by hyper-unit propagation, *RISC Technical Report 02-02*, University Linz, Austria, (2002) 1–18.

[16] Kusper, G., Solving the resolution-free SAT problem by hyper-unit propagation in linear time. *Annals of Mathematics and Artificial Intelligence*, 43(1-4) (2005) 129–136.

[17] Kusper, G., Finding models for blocked 3-SAT problems in linear time by systematical refinement of a sub-model, *Lecture Notes in Artificial Intelligence 4314, KI 2006: Advances in Artificial Intelligence*, (2007) 128–142.

[18] Schlipf, J.S., Annexstein, F., Franco, J., Swaminathan, R.P., On finding solutions for extended Horn formulas, *Information Processing Letters*, 54 (1995) 133–137.

[19] Scutella, M.G., A note on Dowling and Gallier's top-down algorithm for propositional Horn satisfiability. *J. of Logic Programming*, 8(3) (1990) 265–273.

[20] Tovey, C.A., A simplified NP-complete satisfiability problem, *Discrete Applied Mathematics*, 8 (1984) 85–89.

Gábor Kusper
Lajos Csőke
Gergely Kovásznai
Institute of Mathematics and Informatics
Eszterházy Károly College
P.O. Box 43
H-3301 Eger
Hungary

e-mail:
gkusper@aries.ektf.hu
csoke@aries.ektf.hu
kovasz@aries.ektf.hu