ELSEVIER

Discrete Applied Mathematics 96–97 (1999) 127–138

DISCRETE APPLIED MATHEMATICS

The complexity of the falsifiability problem for pure implicational formulas

Peter Heusch *

Institut fur Informatik, Universität zu Köln, Pohligstr.1, D-50969 Köln, Germany

Received 1 May 1997; revised 1 June 1998; accepted 5 October 1998

Abstract

We consider Boolean formulas where logical implication ($\rightarrow$) is the only operator and all variables, except at most one (denoted z), occur at most twice. We show that the problem of determining falsifiability for formulas of this class is NP-complete but if the number of occurrences of z is restricted to be at most k then there is an $O(|F|^k)$ algorithm for certifying falsifiability. We show this hierarchy of formulas, indexed on k, is interesting because even lower levels (e.g., $k = 2$) are not subsumed by several well-known polynomial time solvable classes of formulas.

Keywords: Algorithms and data structures; Logic in computer science

1. Introduction

The satisfiability problem (SAT) for Boolean formulas in conjunctive normal form (CNF) was the first problem that was shown to be NP-complete, [1]. Its complexity has been the subject of quite a number of studies.

Unfortunately, CNF-SAT does not induce a simple natural hierarchy of polynomially solvable subproblems like, e.g. CLIQUE, where for every fixed k the question whether a CLIQUE of size k exists in some given graph is polynomially solvable. The natural subproblems of CNF-SAT instead can be grouped into two classes: those which are solvable within time $O(n)$ or $O(n^2)$, and those which are NP-complete. Examples of this behaviour are the classes 2-SAT and 3-SAT (see [3] for definition), for inputs from the first class the SAT-problem is solvable in linear time, for inputs from the second class the satisfiability problem becomes NP-complete.

There are also classes C_i of formulas where for any $F \in C_i$ the satisfiability problem is solvable in time $O(|F|^i)$, $|F|$ denoting the number of occurrences of variables in the

* Fax: (0221) 4-70-53-87.
E-mail address: heusch@informatik.uni-koeln.de (P. Heusch)

0166-218X/99/$ - see front matter
PII: S0166-218X(99)00036-0

formula, for example the classes C_i where every formula in C_i is satisfiable by setting at most i variables to true, but this classification is quite unsatisfactory in the sense that the test whether an input F belongs to some class C_k may need up to $O(|F|^k)$ steps. Important classes showing this behaviour are the classes Γ_i defined by Gallo and Scutella [2].

We will present a new hierarchy $S_1 \subseteq S_2 \subseteq \cdots$ with the property that for every $F \in S_i$ the falsifiability problem, i.e. the question whether an assignment to the variables exist such that the formula evaluates to *false* can be solved in time $O(i|F|^{i-1})$, while the test whether $F \in S_i$ can be solved in linear time. The proof of the runtime bound is constructive; we analyze an algorithm that outputs a set Z of variables with the property that assigning *false* to all variables in Z and *true* to all other variables yields a solution. The algorithm uses backtracking to compute its result, contrary to other backtracking algorithms it always performs one-sided decisions by never requiring explicitly that *true* be assigned to some variable.

Furthermore, we will prove that every SAT-problem is polynomially reducible to some problem in $\bigcup_i S_i$, hence the falsifiability problem for $\bigcup_i S_i$ is NP-complete.

Lukasiewicz [6] showed the existence of a single short axiom such that all true implicational statements are derivable from this axiom. Nevertheless, the class also contains formulas that represent hard inputs for combinatorial optimization problems, and also the "easy" subclasses are unrelated to some commonly known classes of "easy" inputs for the satisfiability problem. This is shown in the last section by proving that the commonly known "easy" subclasses of SAT are neither sub- nor superclasses of S_2.

The remaining part of this paper is organized in the following way: the next section contains the definitions needed, in Section 3 we prepare our main result which is presented in Section 4. In the last section we give a relationship between the class of formulas solvable in linear time by our algorithm and other classes for which satisfiability is solvable in linear time.

2. Definitions

A Boolean formula $F = C_1 \wedge C_2 \wedge \cdots \wedge C_r$ in conjunctive normal form (CNF) over n variables $v_1, \ldots, v_n$ is a conjunction of clauses $C_1, \ldots, C_r$, where each clause C_l is a disjunction of literals $x_{i_1}, \ldots, x_{i_l}$, a literal either stands for a variable (positive literal) or its complement (negative literal). A Boolean formula is in pure implicational form (PIF), iff it contains only positive literals and the only connective being used is the logical implication. For any implication $A \rightarrow B$ we call A the implicant and B the consequence of the implication. Since the implication is a nonassociative connective, we define $A \rightarrow B \rightarrow C$ to be read as $A \rightarrow (B \rightarrow C)$. An assignment $t : \{v_1, \ldots, v_n\} \mapsto \{true, false\}$ satisfies a Boolean formula F, iff F evaluates to *true* when every variable v is replaced by $t(v)$ and the usual evaluation rules for Boolean operators are applied, t falsifies F iff F evaluates to *false*. A partial assignment is a function $t : \{v_1, \ldots, v_n\} \mapsto$

$\{true, false, undef\}$, a (partial) assignment t' extends a partial assignment t, iff

$$t(v) \neq undef \Rightarrow t'(v) = t(v).$$

An assignment t' is called 1-extension of a partial assignment t, if t' extends t and $t(v) = undef$ implies $t'(v) = true$.

For any Boolean formula $F = F_1 \rightarrow F_2$ and for any occurrence of a subformula F' of F we define

$$\mathscr{D}_l(F', F) = 0 \quad \text{if } F = F',$$
$$\mathscr{D}_l(F', F) = 1 + \mathscr{D}_l(F', F_1) \quad \text{if } F' \text{ lies in the implicant of } F,$$
$$\mathscr{D}_l(F', F) = \mathscr{D}_l(F', F_2) \quad \text{if } F' \text{ lies in the consequence of } F.$$

If F is represented by a tree then $D_l(F', F)$ denotes the number of left edges we have to pass on the path from the root of F to the root of F'. A literal v in F is positive, iff $D_l(v, F)$ is even, else v is called a negative literal. A variable v is called pure, iff either all its corresponding literals are positive or all its corresponding literals are negative. The set

$$\mathscr{B}(F) = \{F' \mid F' \text{ is subformula of } F,\ D_l(F', F) = 0\}$$

is called the backbone of F, those subformulas F' of F that have $D_l(F', F)=1$ are called the backbone implicants of F. The backbone of F contains exactly one subformula that is a variable, this variable is the rightmost variable $V_r(F)$ of F. The set of Boolean formulas in PIF where every variable except the rightmost variable occurs at most twice is called 2-PIF. We call a formula in 2-PIF normalized, if no variable occurs more than once as rightmost literal of a subformula F' with $D_l(F', F)$ even. A backbone implicant F' of F is called a critical subformula, w.r.t a partial assignment t, iff $t(V_r(F')) = false$; the reason behind this is that all other backbone implicants are satisfied by the 1-extension of t and hence do not touch the falsifiability of F. We will see that the number of subformulas that are possibly or actually critical plays an important role in the analysis of the falsifying algorithm. If a subformula F' of F is critical and F'' is a backbone implicant of F', we call F'' compensating (w.r.t a partial assignment t), if t falsifies F''. This is due to the fact that a formula F in PIF can be satisfied by setting $V_r(F)$ to *true* or by falsifying at least one of the backbone implicants, hence to falsify F, $V_r(F)$ must be set to *false* and all backbone implicants have to be satisfied.

3. A hierarchy for pure implicational formulas

We will now define the formula subsets that subdivide 2-PIF and prove some results about them as well as about 2-PIF itself. We define S_i to contain all those formulas F in normalized 2-PIF, such that $V_r(F)$ occurs at most i times in F, i.e. $i-1$ times with odd left distance.

The definition of these sets immediately implies the following lemma, whose proof is obvious:

Lemma 1. *For any Boolean formula F in 2-PIF, the membership problem whether F belongs to S_i can be determined in linear time.*

Another interesting point that a hierarchy must fulfill to be interesting is that it must also be a real hierarchy, i.e. that it must not collapse beyond a certain class, as in the case of CNF-SAT, where an increase of the number k of literals allowed in one clause does not change the complexity of the problem substantially for $k \geqslant 3$, it remains NP-complete. The following theorem based on a theorem by Kleine Büning et al. given in [4], however, gives a strong hint that this is indeed the case with the hierarchy induced by the S_i:

Theorem 1. *The falsifiability problem for formulas in 2-PIF is* NP*-complete.*

Proof. We reduce the well-known NP-complete SAT-problem for Boolean formulas in CNF where every variable occurs at most 3 times to the falsifiability problem for Boolean formulas in PIF. Let F be such a formula in CNF. W.l.o.g. we may assume that every variable with three occurrences occurs exactly once positive and twice negative in F. Let a be such a variable and let C_1, C_2 be the clauses such that $C_1 = \neg a \vee C_1'$ and $C_2 = \neg a \vee C_2'$. We then introduce new variables a', a'' and replace C_1, C_2 by $\neg a \vee (a' \wedge a'')$, $\neg a' \vee C_1'$ and $\neg a'' \vee C_2'$. By repeating this process for every variable occurring three times in F we get a new formula F' s.t. every variable is contained at most twice in F'.

The next step is to eliminate the logical operations $\wedge$, $\vee$ and $\neg$. Without changing the number of variables this can be achieved by substitution of $a \rightarrow \mathit{false}$ for $\neg a$, $(a \rightarrow \mathit{false}) \rightarrow b$ for $a \vee b$ and $(a \rightarrow (b \rightarrow \mathit{false})) \rightarrow \mathit{false}$ for $a \wedge b$. At this point we may apply some simplification rules, e.g. substituting a for $a \rightarrow \mathit{false} \rightarrow \mathit{false}$. To eliminate the logical constant *false*, we replace every occurrence of *false* by a new variable z, which will be forced to be set to *false* later on. Since every single transformation increases the size of the formula at most by a constant factor, the size of the resulting formula F'' is bounded polynomially by the size of F.

Clearly, F'' contains every variable at most twice and is satisfiable by every assignment that satisfies the original formula F and sets z to *false*, thereby setting those "variables" to *false*, where z was replaced for the constant value *false*. This immediately results in the formula $F'' \rightarrow z$ being falsifiable iff F was satisfiable, hence the falsifiability problem for Boolean formulas in 2-PIF is NP-complete. □

4. Main theorem

Theorem 1 showed that every instance of an NP-complete problem must be contained in one of the set S_i. In the next step we show that our hierarchy is indeed a polynomial hierarchy, i.e. that the falsifiability problem is polynomially solvable for every fixed S_i.

We will prove this by the analysis of an algorithm called `PIF_solve`, this algorithm uses backtracking to determine a set Z of variables s.t. setting the variables from Z to *false* and all other variables to *true* falsifies the formula, if such a Z exists. To show the correctness of `PIF_solve`, we need some preparatory lemmas about Boolean formulas in PIF.

Lemma 2. *Let F be a formula in PIF. Then F is falsifiable iff an assignment t with $t(V_r(F))=$ false exists such that for every subformula F' of F that is critical w.r.t. t there is a compensating subformula F'' of F'.*

Proof. To falsify F an assignment t must exist such that $t(V_r(F)) = false$ and every backbone implicant of F is satisfied under t. If a backbone implicant F' is not critical, it is satisfied by t since $t(V_r(F'))=true$. Otherwise it is satisfied, iff w.r.t. t a compensating subformlua F'' of F' exists, i.e. F'' is falsified under t, for in this case F' evaluates to *true*, too.

If, on the other hand, no such assignment exists, then for every assignment t there is at least one critical backbone implicant F' such that F' does not contain any compensating subformula F'', hence F' evaluates to *false* under t, thereby satisfying F. It follows that F is a tautology. □

By the last lemma it is sufficient to find an assignment t such that every critical backbone implicant has a compensating subformula. The lemma is, however, not very helpful in finding such an assignment, testing all possible assignments leads to a run time bound of 2^n, n the number of variables in F. The following lemma shows how to reduce an instance of a given falsifiability problem to a set of smaller ones.

Lemma 3. *Let $F = F_1 \to F_2 \to \cdots \to F_k \to z$ be a Boolean formula in PIF with backbone implicants $F_1,\ldots,F_k$ and let t be a partial assignment to the variables of F with $t(z) = false$. Furthermore let $F_j = G_1 \to G_2 \to \cdots \to G_h \to z'$ be a critical backbone implicant w.r.t. t. Then F can be falsified by an extension t' of t iff for at least one $1 \leqslant l \leqslant h$ the formula $\tilde{F} = F_1 \to F_2 \to \cdots \to F_{j-1} \to F_{j+1} \to \cdots \to F_k \to G_l$ is falsified by t'.*

Proof. Suppose that F is falsifiable by an extension t' of t. Since F_j is critical, t' must falsify at least one (e.g. G_l) of the subformulas $G_1,\ldots,G_h$ to satisfy F_j; at the same time all other backbone implicants $F_1,\ldots,F_{j-1},F_{j+1},\ldots,F_k$ must also be satisfied by t.

Hence $\tilde{F} = F_1 \to F_2 \to \cdots \to F_{j-1} \to F_{j+1} \to \cdots \to F_k \to G_l$ is falsified by t'. On the other hand, if $\tilde{F}$ is falsified by an extension t' of t, G_l is a compensating subformula for F_j, hence t' satisfies all backbone implicants of F and falsifies z, so F is falsified, too. □

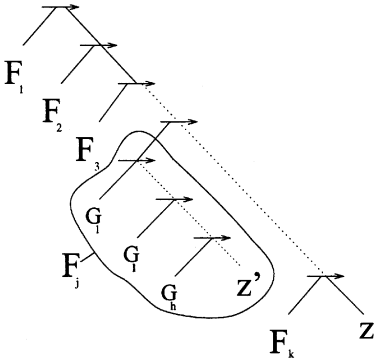

Fig. 1.

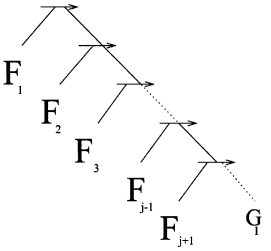

Fig. 2.

This can also be seen as a graph manipulation process: if the formula is interpreted as a tree where the inner nodes correspond to operators and the outer nodes correspond to variables, then we can falsify the formula from Fig. 1 iff for at least one l the formula given in Fig. 2 is falsifiable.

Most algorithms solving the SAT-Problem check whether there are variables in the input, such that all corresponding literals are positive or such that all corresponding literals are negative. The following lemma shows that these variables are precisely the pure variables in the sense of Section 2.

Lemma 4. *Let F be a Boolean formula in 2-PIF and let v be a pure variable in F. Iff a falsifying assignment t to the variables of F exists, it can be transformed into a falsifying assignment t' such that*

$$t'(v) = \textit{true} \quad \textit{if all occurrences of } v \textit{ have odd } D_1,$$
$$t'(v) = \textit{false} \quad \textit{if all occurrences of } v \textit{ have even } D_1.$$

Proof. The Boolean function that is defined by F is either positive or negative in v, depending on whether all occurrences of v in F have even or odd D_1. This is easily seen by transforming our 2-PIF formula into an equivalent with connectives $\wedge$ and $\vee$,

since literals with even D_l become positive literals and literals with odd D_l become negative literals.

Since our aim is to falsify F, we may safely set $t'(v) = \mathit{false}$ if all occurrences of v have even D_l without satisfying F, the same holds vice versa if all occurrences of v have odd D_l. □

Corollary 1. *Let F be a Boolean formula in 2-PIF. Then F can be transformed into a new formula F' such that F' is normalized and not longer than F; furthermore F' is falsifiable iff F is.*

Proof. With the exception of $V_r(F)$ every variable occurs at most twice in F, hence if some variable $v \neq V_r(F)$ has two occurrences with even left distance, we simply replace one of them by a new variable v'. If $V_r(F)$ has other occurrences with even left distance, we replace these other occurrences by new variables $v', v'', \ldots$ In either cases, F' has the same length as F. Because of the preceeding lemma, iff F is falsifiable there exists a falsifying solution such that all replaced occurrences evaluate to false. Hence the same assignment extended by assigning *false* to all newly introduced variables will also falsify F'. □

We are now able to formulate the algorithm `PIF_solve` that computes the set Z of literals such that an assignment evaluating every variable in Z to *false* and every other variable to *true* falsifies the formula. If no such Z exists, `PIF` does not produce any output.

```
(1)  program PIF;
(2)  var F:PIF;
(3)  procedure PIF_solve(F:PIF,Z:set);
(4)  begin
(5)    let F = F_1 → ··· → F_j → ··· → F_k → z;
(6)    if Z ∩ ⋃_i V_r(F_i) = ∅ then begin
(7)      print solution Z; exit;
(8)    end
(9)    find the smallest j such that V_r(F_j) ∈ Z;
(10)   let F_j = G_1 → ··· → G_h → z';
(11)   for l = 1 to h do begin
(12)     F' = F_1 → ··· F_{j-1} → F_{j+1} → ··· → G_l;  Z' = Z ∪ {V_r(G_l)};
(13)     PIF_solve(F', Z');
(14)   end
(15) end
(16) begin {Main program}
(17)   Read(F);
(18)   PIF_solve(F, {V_r(F)});
(19) end.
```

To analyze the runtime behaviour of `PIF_solve`, we count the number of sets Z that are possibly constructed during the computation performed by `PIF_solve`. It will turn out that only a small fraction of all possible 2^n sets is ever constructed by the different instances of `PIF_solve`.

Furthermore, we will show that no two different instances of `PIF_solve` share the same set Z, hence the runtime of `PIF_solve` is bounded by the number of sets Z times the number of elementary steps that are executed by every instance of `PIF_solve`.

The important idea to reduce the number of different sets Z is the observation that there is a predecessor–successor relationship $\prec$ among the variables in F such that whenever a variable v is inserted into Z, then all of its predecessors are already contained in Z. We say that $v \prec v'$, iff an occurrence of v is rightmost literal of a subformula F' of F, $D_l(F', F)$ is odd and an occurrence of v' is rightmost literal of a backbone implicant of F'. We say that v is predecessor of v' w.r.t. F, or equivalently that v' is successor of v w.r.t. F, iff $v \prec v'$.

It is evident that for F in normalized 2-PIF every variable v has at most one predecessor, if no variable occurs more than once at even left distance, since there is a uniquely defined subformula F such that v is rightmost literal of some backbone implicant of F. However, there are variables without predecessors like $V_r(F)$ and all variables that occur at odd left distance only.

We define a variable $v \in Z$ that occurs l times in F with odd left distance as active, if there are less than l successors of v in Z; we define $v \in Z$ as blocking, if it does not have any successor w.r.t. F. The fact that reduces the number of possible sets Z is that Z can be fully described by a constant-sized subset of itself, the set of all variables in Z that are either active or blocking. Furthermore, we assign a weight w to every variable $v \in Z$ as follows:

$$
\begin{aligned}
&w(v) = i - l \quad \text{if } v \text{ is active, occurs } i \text{ times with odd left distance in } F \text{ and} \\
&\qquad\quad l \text{ of its successors are contained in } Z, \\
&w(v) = 1 \quad \text{if } v \text{ is blocked,} \\
&w(v) = 0 \quad \text{else.}
\end{aligned}
$$

Lemma 5. *Let F be a Boolean formula in normalized 2-PIF, with $F \in S_{i+1}$. Then every recursive call of* `PIF_solve` *with input F preserves the following invariant*:

$$\sum_{v \in Z} w(v) = i.$$

Proof. First note that performing a recursive call of `PIF_solve` may result in some literals disappearing from the formula. A literal that remains in the formula, however, cannot change its left distance from odd to even or vice versa, since its left distance remains either unchanged or decreases by 2. Hence in line (12) we have $z' \prec V_r(G_l)$, since z' must have had an odd left distance w.r.t. the original F.

To prove the lemma, we use an induction on the depth of the calling stack:

- `PIF_solve` is called from line 18: $V_r(F)$ is the only variable in Z and since $w(v)=i$ by definition of w the lemma holds in this case.
- `PIF_solve` is called recursively from `PIF_solve`. Then by induction hypothesis the equation $\sum_{v\in Z} w(v) = i$ was satisfied when the calling instance of `PIF_solve` was entered. Since the variable $V_r(G_l)$ is a successor of z', $w(z')$ decreases by 1. The variable $V_r(G_l)$ may be either blocking or active in F. Since it occurs at most once with odd left distance by assumption, $w(V_r(G_l)) = 1$ in both cases. Therefore $\sum_{v\in Z} w(v) = i$ also holds immediately before entering the next recursive call.

This completes the proof. □

The following corollary now states what is really important for our proof, namely that Z can be described by a subset of itself with constant size, and that therefore the number of different sets is polynomially bounded in the size of the formula:

Corollary 2. *Let F be a Boolean formula in normalized 2-PIF, with $F \in S_{i+1}$. Then for every instance of* `PIF_solve` *which is (recursively) called the corresponding parameter Z is completely determined by those variables whose weight is strictly positive and the number of different sets Z is limited by n^i, n the number of variables in F.*

Proof. As we have seen in the proof of Lemma 5, all variables $v \in Z$ either have $w(v) > 0$ or there exists some other variable $v' \in Z$ with $v \prec v'$. It follows that Z is completely described by those variables in Z with $w(v) > 0$. The number of different sets Z is now limited by the number of different possible weights w. Since $w(\cdot)\geqslant 0$ this number is $\sum_{l=0}^{i} \binom{n-1}{l}$, for sufficiently large n (compared to i), this is less than n^i. □

Unfortunately, the number of sets Z by itself is not sufficient to limit the runtime of `PIF_solve`, since a lot of different instances of `PIF_solve` might possibly share the same set Z. This, however, is not the case as the following lemma shows:

Lemma 6. *Let F be a Boolean formula in normalized 2-PIF. Then no two instances of* `PIF_solve` *share the same set Z.*

Proof. Assume that two different instances I_A and I_B of `PIF_solve` share the same set Z' as second parameter. I_A and I_B must have a least common ancestor instance C of `PIF_solve` with parameters F_C, Z_C. Let us denote the two immediately called instances that are ancestors of I_A and I_B by C_A and C_B furthermore let F_A, F_B, Z_A and Z_B, be their resp. parameters.

Concerning the way `PIF_solve` works we must have $Z_A = Z_C \cup \{V_r(G_a)\}$ and $Z_B = Z_C \cup \{V_r(G_b)\}$; since the input was in normalized 2-PIF, we have $V_r(G_a) \neq V_r(G_b)$. Moreover, F_A does not contain any occurrence of $V_r(G_b)$ with even left distance and

vice versa F_B does not contain any occurrence of $V_r(G_a)$ with even left distance, since the recursive call eliminates the resp. subformula from F_C. Hence no descendant of C_A can ever insert $V_r(G_b)$ into Z_A and no descendant of C_B can ever insert $V_r(G_a)$ into Z_B.

During recursive calls the set Z is only enlarged, so Z' must be a superset of both Z_A and Z_B. Since by the argument given above no set can contain both $V_r(G_a)$ and $V_r(G_b)$ at the same time, the assumption that I_A and I_B can share the same set Z' must be false, thereby proving the lemma. □

Now we state our main result, whose correctness, after the preceeding lemmas is almost obvious:

Theorem 2. *Let F be a Boolean formula in normalized 2-PIF with n variables and $i+1$ occurrences of $V_r(F)$, then* `PIF_solve` *finds a falsifying assignment for F iff one exists, furthermore the runtime of* `PIF_solve` *is bounded by* $O(i * n^i)$.

Proof. Since F is in normalized 2-PIF, at most n^i different instances of `PIF_solve` must be called to find a falsifying assignment to the variables of F. If suitable data structures are applied, the only time consuming operation in `PIF_solve` is the check whether a critical backbone implicant exists. At most i variables must be checked out to find such a critical backbone implicant. Since the check for one variable can be done in constant time, we get the runtime bound $O(i * n^i)$. □

5. Relationships to other input classes

Since CNF-SAT is NP-complete, several input restrictions have been developed that permit to test a formula for satisfiability in polynomial time. The most common of these restrictions are restrictions for formulas in CNF, they are defined as follows:

- 2-*SAT*: Clauses contain at most two literals.
- *Horn formulas*: Clauses may contain at most one positive literal.
- *Nested SAT*: An ordering of the clauses must exist with the property that if a clause C preceeds another clause C' then no variable from C except from the first and the last (w.r.t. an ordering of the variables) may be contained in C'.
- *READ*-2: No variable may occur more than twice in a formula.

It is well known that for inputs from these classes the satisfiability problem is solvable in linear time, see [4,5]. The following theorem relates S_2 with the four classes mentioned above.

Theorem 3. *For every one of the classes* 2-*SAT*, *HORN*, *nested SAT and READ*-2 *there is a Boolean function function that can be expressed in S_2 but that cannot be expressed in* 2-*SAT*, *HORN*, *etc*, *and vice versa.*

Proof. "⇒" It is possible to encode unsatisfiable formulas as well as tautologies in 2-SAT, HORN, nested SAT and READ-2. Since every formula $F \in S_2$ is satisfiable by assigning *true* to $V_r(F)$, contradictions cannot be represented by those formulas, therefore S_2 does not contain 2-SAT, HORN, nested SAT and READ-2.

"⇐" To prove our claim, we present two Boolean functions that can be expressed by formulas from S_2 but not by CNF-formulas from 2-SAT, HORN, etc. W.l.o.g. we consider only formulas without logical constants since clauses containing them can either be shortened or eliminated without changing the function. We use Karnaugh-Maps to show that every possible CNF-formula realizing the given function must violate at least one of the given input restrictions.

Karnaugh-Maps are used to find shortest implicants for Boolean functions in DNF, but they can also be used to construct shortest anti-implicants for Boolean functions in CNF (i.e. clauses s.t. the function evaluates to *false*, whenever the clause evaluates to *false* for some input vector). To achieve this, we must, for any input vector whose output is *false*, insert the disjunction of the negated values. E.g. if the output shall be *false* for the input $a = true$, $b = false$ and $c = true$, we must insert the clause $\neg a \vee b \vee \neg c$ into the formula. In the same way as for the DNF-algorithm, we can construct a consensus from two clauses, thereby shortening the anti-implicants. Note that we use the equivalences $false = 0$ and $true = 1$ in our Karnaugh-Maps in accordance with the usual representation.

We use the Boolean functions represented by the following formulas from S_2:

(i) $(a \to b) \to (b \to a) \to z$ has the following Karnaugh-Map:

z\ab	00	01	11	10
0	0	1	0	1
1	1	1	1	1

Any CNF-formula equivalent to $(a \to b) \to (b \to a) \to z$ must contain the clauses $\neg a \vee \neg b \vee z$ and $a \vee b \vee z$. Both clauses together violate the input restriction for nested SAT, furthermore the second clause by itself violates the input restrictions for 2-SAT and HORN.

(ii) $(a \to b \to c \to z) \to z$ has the following Karnaugh-Map:

cz\ab	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	1	1	1
10	0	0	1	0

From the Karnaugh-Map we see that any CNF-formula equivalent to $(a \to b \to c \to z) \to z$ must contain at least three anti-implicants. Since setting $z = true$ satisfies the formula, any anti-implicant must contain z, thereby violating the input restriction for READ-2.

Since all clauses are shortest anti-implicants and for every clause there is an input vector that is covered by this clause exclusively, all CNF-formulas are shortest possible. Because the input restrictions have the property that the restriction cannot hold for a formula F if it is violated for some subformula F', it follows that every CNF-formula representing one of the given Boolean functions must also violate the corresponding input restrictions.

Since all given Boolean functions can be represented by formulas from S_2, the direction "$\Leftarrow$" follows, so the theorem is proved. □

Acknowledgements

Thanks to the unknown referees for their helpful hints.

References

[1] S. Cook, The complexity of theorem proving procedures, Proceedings of the third Annual ACM Symposium on Theory of Computing, 1971, pp. 151–158.
[2] G. Gallo, M.G. Scutella, Polynomially solvable satisfiability problems, Inform. Process. Lett. 29 (5) (1988) 221–227.
[3] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman and Company, San Francisco, 1979.
[4] H. Kleine Büning, T. Lettman, Aussagenlogik: Deduktion und Algorithmen, B.G. Teubner, Stuttgart, 1994.
[5] D.E. Knuth, Nested satisfiability, Acta Inform. 28 (1990) 1–6.
[6] J. Lukasiewicz, The shortest axiom of the implicational calculus of propositions, Proc. Irish Acad. 52 (1948) 25–33.