View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CWI's Institutional Repository

Centrum voor Wiskunde en Informatica
REPORT RAPPORT

The Propositional Formula Checker HeerHugo

Jan Friso Groote, Joost P. Warners

Software Engineering (SEN)

SEN-R9905 January 31, 1999

Report SEN-R9905
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

The Propositional Formula Checker HeerHugo

Jan Friso Groote[a,b] Joost P. Warners[a,c]

[a] CWI
P.O. Box 94079, 1090 GB, Amsterdam, The Netherlands
e–mail: JanFriso.Groote@cwi.nl, Joost.Warners@cwi.nl

[b] Computing Science Department, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

[c] Department of Technical Mathematics and Informatics, Faculty of Information Technology and Systems
Delft University of Technology, P.O. Box 5031, 2600 GA Delft, The Netherlands

ABSTRACT
HeerHugo is a propositional formula checker that determines whether a given formula is satisfiable or not. Its main ingredient is the branch/merge rule, that is inspired by an algorithm proposed by Stållmarck, which is protected by a software patent. The algorithm can be interpreted as a breadth first search algorithm. HeerHugo differs substantially from Stållmarck's algorithm, as it operates on formulas in conjunctive normal form and it is enhanced with many logical rules including unit resolution, 2–satisfiability tests and additional systematic reasoning techniques. In this paper, the main elements of the algorithm are discussed, and its remarkable effectiveness is illustrated with some examples and computational results.

1991 Mathematics Subject Classification: 03B05, 68Q20, 68T15
1991 Computing Reviews Classification System: F.0, F.2, F.4
Keywords and Phrases: Satisfiability, Propositional logic, Theorem proving.
Note: The second author is supported by the Dutch Organization for Scientific Research (NWO) under grant SION 612-33-001.

1. Introduction

The satisfiability problem of propositional logic (SAT) is considered important in many disciplines, such as mathematical logic, electrical engineering, computer science, operations research etc. It was the first problem shown to be NP–complete [3], which means that it is considered unlikely that an algorithm exists that can solve any instance of SAT in polynomial time. It may be noted though that several subclasses of SAT exist, which allow a polynomial time algorithm, such as 2SAT and HornSAT [1, 7]. Over the years, many algorithms for solving satisfiability problems have been developed; for a comprehensive overview the reader is referred to [15]. The algorithms can basically be subdivided into two categories. Firstly, there are the *complete* algorithms; such algorithms find one or all satisfying solutions or prove that the formula under consideration is a contradiction. Examples of such algorithms are (variants of) the Davis–Putnam procedure (which we refer to as DPLL algorithms) [6, 5] and binary decision diagrams (BDDs) [2]. Secondly, there are *incomplete* algorithms. These search for a satisfying assignment; if one is found, the formula is solved. Unfortunately, such methods are not able to prove a formula contradictory. These methods include various kinds of local search [25, 14, 13].

In this paper, an implementation of a complete algorithm, called HeerHugo, is described. The initial purpose of this implementation was to validate correct operation of the Vital Processor Interlocking unit guarding the safety of all operations at the Dutch railway stations in Hoorn–Kersenboogerd and Heerhugowaard (see also [9, 12, 21]); hence the name of the solver. For this practical application HeerHugo turned out to be very effective and so it was decided to test HeerHugo's strength on other kinds of SAT problems as well.

HeerHugo is inspired by an algorithm proposed by Stållmarck[1], which can be interpreted as a *breadth first* search algorithm, with several enhancements (see also [26, 16, 33]). The actual algorithms implemented in HeerHugo differ substantially from Stållmarck's algorithm, as it operates on formulas in conjunctive normal form (CNF) and incorporates many logical rules that are more or less common in theorem proving. It seems to us that the breadth first search type algorithm is largely overlooked in the literature on practical satisfiability solving. The DPLL method [5], which seems to be the most widely used, is in fact a *depth first* search algorithm. It appears that for many classes of formulas breadth first search is substantially more effective than depth first search, especially for instances stemming from real–life applications; on the popular (at least for research purposes) difficult random 3SAT instances [22] depth–first search seems more effective. The effectiveness of HeerHugo is mainly due to its reasoning strength. Based on (nested) assumptions conclusions are drawn that are subsequently added to the formula; such conclusions include not only truth values of propositions, but also the equivalence of sets of variables. Note that implementations of DPLL algorithms that are enhanced with 'intelligent backjumping' techniques (such as SATO [36]), appear to have a comparable reasoning strength.

Besides being a complete algorithm for determining satisfiability, breadth first search can be applied as *preprocessing* technique to substantially reduce the size of the formula under consideration by restricting the number of nested assumptions. If during this process the empty clause is not derived, one could decide to switch to a DPLL or other algorithm.

This paper is organized as follows. In the next section we deal with the preliminaries and notation. HeerHugo is able to handle propositional formulas with all the well known connectives. It first translates a formula to a formula in conjunctive normal form, maintaining satisfiability, by introducing auxiliary variables. This transformation is briefly described in Section 3.1. Subsequently various simple inference rules are applied to simplify the formula under consideration as much as possible; these are specified in Section 3.2. The way the breadth–first search is carried out is discussed in Section 3.3. In the subsequent subsections some complexity issues and possible enhancements are mentioned. Section 4 and 5 are concerned with the description of HeerHugo's actual performance. Some specific effects of the simple inference rules are discussed, and a number of computational results are given. We conclude with a discussion in Section 6. In Appendix A it is explained how HeerHugo 2.0 can be obtained and used.

Acknowledgements Thanks go to Marc Bezem for helping to track some references, and to Bert Lisser and John Tromp for their comments on an earlier version of this manuscript.

[1]Stållmarck's prover is protected by a software patent [27].

2. Preliminaries and Notation

Let us start with introducing some notation. Let $\mathcal{P} = \{p, p_0, p_1, p_2, \dots, q, r, \dots\}$ be the set of *atomic propositions* or *variables*. Each proposition can be either *true* or *false*. We have a number of *connectives*: $\neg$ ('not' or 'negation'), $\vee$ ('or' or 'disjunction'), $\wedge$ ('and' or 'conjunction'), $\rightarrow$ ('implication') and $\leftrightarrow$ ('equivalence' or 'bi-implication'). Furthermore, a literal is an atomic proposition or its negation. Let $\mathcal{L} = \{l, l_0, l_1, l_2, \dots\}$ be the set of *literals*. Propositional (sub)formulas are denoted by Φ and ψ. An example of a propositional formula Φ is

$$\Phi = (p_0 \leftrightarrow \neg p_1) \vee ((\neg p_0 \rightarrow p_2) \wedge (p_1)).$$

The satisfiability problem of propositional logic is to assign truth values to the propositions, such that the formula Φ evaluates to *true*, or to prove that no such assignment exists. If the formula is *true* for each assignment to the variables, it is called a *tautology*, while if it is *false* for each assignment it is said to be a *contradiction*. Otherwise it is said to be *satisfiable*.

In this paper, the satisfiability problem is studied in conjunctive normal form (CNF). A CNF formula consists of a conjunction of disjunctions, where each of the disjunctions is called a *clause*, in notation:

$$\Phi = \bigwedge_{k=1}^{n} C_k,$$

where each clause C_k is of the form

$$\bigvee_{i \in I_k} l_i,$$

with l_i a literal. The length $\ell(C_k)$ of a clause is defined as the number of distinct literals occurring in its minimal representation, i.e. tautological clauses have length zero, and doubled occurrences of identical literals in the same clause are reduced to a single occurrence. A CNF formula in which the maximum clause length is equal to ℓ is referred to as an $\leq \ell$CNF formula. In particular, $\leq$2CNF formulas are solvable in linear time [1], while $\leq$3CNF formulas are in general NP-complete [3].

Note that the $\leftrightarrow$ and $\rightarrow$ operators can be eliminated using the well known De Morgan's laws [4] to obtain a CNF formula. In general, this cannot be done efficiently; i.e. transforming an arbitrary formula to CNF using these laws requires a number of operations that is exponential in the length of the formula. However, using auxiliary variables, any formula can be put into the ($\leq$3)CNF format in linear time, maintaining satisfiability. This transformation is used by HeerHugo. It is described in the next section.

3. Algorithms and rules

We describe how HeerHugo is working in four parts. The first part describes how an arbitrary formula can efficiently be transformed to a conjunctive normal form, where clauses contain at most 3 literals. In the second section we describe efficient (mainly linear) operations that can be carried out on the conjunctive normal form to reduce the number of distinct propositions that occur in it. In the third section we describe Stålmarck's branch and merge rule, which is required to make the system complete. Subsequently, the way new clauses are derived is discussed, some complexity issues are addressed and finally the inclusion of additional satisfiability tests is mentioned.

3.1 Transformation to ≤3CNF

We describe a linear transformation of any formula Φ to $\leq$3CNF form, i.e. formulas in conjunctive normal form with at most 3 literals per clause. This transformation is believed to be first described by Tseitin [29] and therefore we refer to it as the Tseitin translation. The original formula allows a satisfiable assignment if and only if its transformed $\leq$3CNF counterpart allows a satisfiable assignment. More precisely, any satisfying assignment of the original formula can be extended to a satisfying assignment of the associated $\leq$3CNF formula. Note that if the original formula is a tautology, the resulting CNF formula is merely satisfiable; it is easy to see that (unless P=NP) no polynomial time transformation to CNF preserving tautology exists, since the tautology problem on CNFs is easy. Note that proving a formula to be a tautology is equivalent to proving the negation of its transformed counterpart to be a contradiction.

Take an arbitrary formula Φ. Introduce for any subformula ψ of Φ a new proposition p_ψ, except if ψ is itself an atomic proposition. First construct the following formula:

$$p_\Phi \wedge \bigwedge_{\substack{\psi \in Sub(\Phi) \\ \psi \equiv \psi_1 \oplus \psi_2}} (p_\psi \leftrightarrow (p_{\psi_1} \oplus p_{\psi_2})) \wedge \bigwedge_{\substack{\psi \in Sub(\Phi) \\ \psi \equiv \neg\psi_1}} (p_\psi \leftrightarrow \neg p_{\psi_1}).$$

Here $Sub(\Phi)$ is the set of subformulas of Φ, and $\oplus$ denotes one of the binary connectives. Obviously, the number of subformulas is linear in $|\Phi|$, and since each of the logical expressions involved can be expressed in at most 4 clauses the transformation is linear in the size of the formula.

Note that to obtain a slightly more concise formulation, the equivalence operators may be replaced by implications [34]. Thus satisfiability is maintained, but in case the original formula is satisfiable, the resulting formula allows more satisfiable solutions.

For completeness, we give an example of the result of transforming a 'triple' $(p_\psi \leftrightarrow p_{\psi_1} \wedge p_{\psi_2})$ and a formula $(p_\psi \leftrightarrow \neg p_{\psi_1})$ to $\leq$3CNF form. We obtain $(\neg p_\psi \vee p_{\psi_1}) \wedge (\neg p_\psi \vee p_{\psi_2}) \wedge (p_\psi \vee \neg p_{\psi_1} \vee \neg p_{\psi_2})$ and $(p_\psi \vee p_{\psi_1}) \wedge (\neg p_\psi \vee \neg p_{\psi_1})$, respectively. One of the differences between HeerHugo and Stålmarck's satisfiability checker is that the latter works with so-called 'triples', formulas of the form $p \leftrightarrow q \oplus r$, whereas HeerHugo operates on the larger $\leq$3CNF formulas, which allows for more logical rules to be applied.

3.2 Simple rules

From now it is tacitly assumed that we consider $\leq$3CNF formulas. In order to simplify a formula, a number of transformations is simultaneously applied. By simplification of a formula we mean in general the removal of propositions and clauses, or the addition of short clauses (i.e. clauses that are likely to contain useful information). The transformations are described in the following paragraphs.

Unit resolution Unit resolution is motivated by the occurrence of clauses of length one in the formula. Such clauses are called *unit clauses*. It relies on the obvious observation that a literal occurring in a unit clause must be *true* in any satisfying assignment (if any). In the unit resolution phase, all unit clauses are found, and the corresponding literals are set to *true*.

Subsequently, the truth values are propagated through the formula. During this process new unit clauses can emerge, or a contradiction can be found. The procedure is repeated until either a contradiction is found or no unit clauses remain. All unit resolution steps can be applied in linear time [7]. It is interesting to note that unit resolution is complete for Horn formulas, or propositional Prolog programs. These are formulas in conjunctive normal form (without a restriction on the clause length) where each clause contains at most one positive literal.

Removal of implication cycles The second simplification procedure is motivated by the occurrence of clauses of length two. If the maximum clause length is equal to two the formula under consideration can be solved in linear time [1]. Moreover, by considering a 2CNF *sub*formula of a larger formula, it is possible to reduce the problem size, using this algorithm. Clauses of length two can be viewed as implications, i.e. $p \vee q$ is equivalent to $\neg p \rightarrow q$ and $\neg q \rightarrow p$. Such implications can form a cycle, for example $p \rightarrow q$, $q \rightarrow \neg r$ and $\neg r \rightarrow p$. Clearly, all literals in such a cycle are equivalent, and henceforth can be given the same name. This observation yields a complete linear time algorithm for 2CNF formulas [1]. Obviously, if the 2CNF subformula is found to be contradictory, the full formula is a contradiction as well.

Classical Davis–Putnam rule One of the earliest rules in propositional reasoning is proposed in [6]. We refer to this rule as the *classical Davis–Putnam rule*. It can be formulated for a proposition p as follows. Given a CNF of the form

$$\Phi = (p \vee \psi) \wedge \psi',$$

where p does not occur in ψ' (and ψ). (Note that ψ and ψ' both denote $\leq$3CNF formulas.) Basically, for every occurrence of $\neg p$ in ψ' we can substitute ψ, maintaining all satisfying assignments. When carrying out this substitution it is easy to obtain a CNF again, but auxiliary propositions must be introduced to maintain $\leq$3CNF. Moreover, if both p and $\neg p$ occur, the resulting $\leq$3CNF is considerably larger than the original. This is basically the motivation for Davis, Logemann and Loveland's modification of the original Davis–Putnam algorithm [5]. Their (depth–first search) algorithm is currently one of the most widely used algorithms for solving SAT problems.

In HeerHugo the classical Davis–Putnam rule is carried out only when the formula reduces in size, or when there is a limited growth. Just before applying the branch/merge rule (see below) to a proposition p, it is investigated whether it can be fruitfully applied. In this respect we mention two special cases:

- When $\neg p$ does not occur in ψ', then the Davis–Putnam rule implies that $(p \vee \psi)$ can be discarded. This is called *monotonic variable fixing* (also known as the *pure literal rule*).
- Another interesting case is when p occurs exactly once, in a clause of length 2, while $\neg p$ occurs arbitrarily often. For example, this clause has the form $p \vee q$. According to the Classical Davis–Putnam rule all occurrences of $\neg p$ may be substituted by q, reducing the number of propositions and thus the size of the formula.

HeerHugo is facilitated to immediately (i.e. while applying unit resolution) detect whether one of these special cases of the Classical Davis–Putnam rule can be applied.

Subsumption and ad hoc resolution HeerHugo constantly changes its set of clauses by removing and adding clauses (see also the next section). Whenever a clause is added, it is checked whether a similar clause exists. For example, if a clause $C_1 \equiv p \vee q \vee r$ is added, then the set of clauses is checked for the occurrence of a clause C_2 of any of the forms listed in Table 1, and the corresponding action is taken. The actions are motivated by *subsumption*, i.e. $C_2 \rightarrow C_1$, or *resolution*, i.e. $(C_1 \wedge C_2) \leftrightarrow C_3$ for some clause C_3 which is subsequently added. If no such clause C_2 is present, C_1 is simply added. Obviously, if p, q and/or r appear with negations, the obvious dual actions are taken. Similarly, in case a clause of the form

C_2	Action
$p \vee q \vee r$	do not add C_1
$p \vee q \vee \neg r$	do not add C_1, remove C_2 and add $p \vee q$
$p \vee \neg q \vee r$	do not add C_1, remove C_2 and add $p \vee r$
$\neg p \vee q \vee r$	do not add C_1, remove C_2 and add $q \vee r$
$p \vee q$	do not add C_1
$p \vee \neg q$	do not add C_1, add $p \vee r$
$\neg p \vee q$	do not add C_1, add $q \vee r$
$p \vee r$	do not add C_1
$p \vee \neg r$	do not add C_1, add $p \vee q$
$\neg p \vee r$	do not add C_1, add $q \vee r$
$q \vee r$	do not add C_1
$q \vee \neg r$	do not add C_1, add $p \vee q$
$\neg q \vee r$	do not add C_1, add $p \vee r$

Table 1: Actions following the derivation of a 'new' clause $C_1 \equiv p \vee q \vee r$.

$C_1 \equiv p \vee q$ is added and a clause C_2 of any of the forms listed in Table 2 is present, the corresponding actions are taken. Again, if no such C_2 is present, C_1 is simply added. The dual cases, where p and/or q are negated, are left to the reader.

C_2	Actions
$p \vee q$	do not add C_1
$p \vee \neg q$	do not add C_1, remove C_2 and add p
$\neg p \vee q$	do not add C_1, remove C_2 and add q
$\neg p \vee \neg q$	do not add C_1, remove C_2 and set $p \equiv q$
$p \vee q \vee r$	remove C_2
$p \vee q \vee \neg r$	remove C_2
$p \vee \neg q \vee r$	remove C_2, add $p \vee r$
$p \vee \neg q \vee \neg r$	remove C_2, add $p \vee \neg r$
$\neg p \vee q \vee r$	remove C_2, add $q \vee r$
$\neg p \vee q \vee \neg r$	remove C_2, add $q \vee \neg r$

Table 2: Actions following the derivation of a 'new' clause $C_1 \equiv p \vee q$.

3.3 A branching scheme: The branch/merge rule

After applying all rules from the previous section, it might be that no contradiction has been derived. In this case a branch/merge rule is adopted. This rule stems from [26] where it is called the dilemma rule, and appears to be surprisingly effective in proving formulas. We actually believe that the success of the Stålmarck prover should be solely attributed to this rule, since its other aspect (namely the handling of triples), is a slightly enhanced variant

of unit resolution, which is completely subsumed by all rules mentioned previously. The branch/merge rule is depicted in Figure 1. It must be interpreted as follows. Starting with

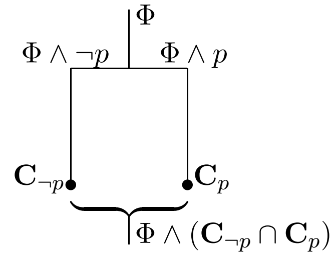

Figure 1: Illustration of the branch/merge rule.

a $\leq$3CNF formula Φ, the proof is split in two parts. In the first part, the assumption $\neg p$ is added to Φ. Using the simple rules a set of conclusions $\mathbf{C}_{\neg p}$ is drawn. A conclusion can have any of the following forms:

- A certain proposition does or does not hold; q or $\neg q$.
- Two propositions are, or are not equivalent; $q \leftrightarrow r$ or $q \leftrightarrow \neg r$.

After exhaustively applying the simple rules, $\mathbf{C}_{\neg p}$ contains all such conclusions. Subsequently, $\mathbf{C}_{\neg p}$ is made *closed* in the following way. It is checked whether $\mathbf{C}_{\neg p}$ contains the conclusion that both q and $\neg q$ must hold, for some proposition q. If this is the case an inconsistency has been derived; then $\mathbf{C}_{\neg p}$ is made to contain the set of all literals, as well as all (in)equivalences between literals (since out of a contradiction anything can be concluded). Otherwise, for all conclusions of the form q, an equivalence $\neg p \leftrightarrow q$ is added to $\mathbf{C}_{\neg p}$. Subsequently, the transitive closure of bi–implications is determined. I.e. if $\neg p \leftrightarrow q$ and $q \leftrightarrow r$ are in $\mathbf{C}_{\neg p}$, $\neg p \leftrightarrow r$ is added. In the same way, $\mathbf{C}_p$ is constructed by assuming p and adding it to Φ.

After exploring both branches, the intersection of $\mathbf{C}_{\neg p}$ and $\mathbf{C}_p$ is calculated, and each fact in the intersection is added to Φ. Calculating this intersection as well as the transitive closure can be done in linear time in the number of conclusions. When the intersection is not empty, the simple rules are applied to simplify $\Phi \wedge (\mathbf{C}_{\neg p} \cap \mathbf{C}_p)$. Note that if one of the branches lead to a contradiction, the conclusions of the other branch are all in the intersection.

HeerHugo applies the branch/merge rule for every proposition in turn, leading to a diagram of shape (b), shown in Figure 2. The branch/merge is repeatedly applied to all propositions in sequence until the stage is reached where for all propositions the intersection of $\mathbf{C}_{\neg p}$ and $\mathbf{C}_p$ is empty. This means that applying the rule to any proposition does not lead to any new facts. In this case the branch/merge rule is applied in a nested way (see (c), Figure 2). First, at level one, a single proposition is set to *true* and *false*, and under this assumption, the branch/merge rule is applied to all other propositions, until no new facts can be derived. Then a next proposition at level one is chosen.

It may be noted that a less powerful variant of the branch/merge rule is known as *single lookahead unit resolution*. This is a polynomial–time algorithm for solving extended Horn

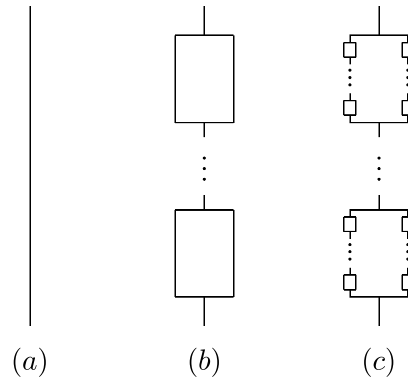

Figure 2: Schematic illustration of the application of simple rules (a), the branch/merge rule (b) and the nested branch/merge rule (c).

formulas [24]. It is also incorporated in several implementations of general DPLL–based branching algorithms (e.g. [8]) for detecting possible variable fixings and early closing of branches; it does not take into account equivalencies of variables.

A note on adding 'expensive' clauses It can happen that after assuming a literal p to be *true* , assuming the validity of a literal q leads to a contradiction. In this case we have derived $p \wedge q \rightarrow false$, which is equivalent to the clause $\neg p \vee \neg q$. We consider such a clause expensive, because we may have searched for quite a while before finding it. Furthermore, it is considered useful, since it is a concise fact. Therefore, such a clause is added to the formula. This is also done when the nested assumption of three propositions leads to a contradiction. We have also experimented with adding a clause $\neg p \vee q$ ($\neg p_0 \vee \neg p_1 \vee q$) if we derive q under the assumption p (assumptions p_0 and p_1). We found a degradation of performance, due to the fact that many redundant clauses were generated in this way.

3.4 Including satisfiability tests

As may be clear from the description of the algorithm so far, it is basically a form of applying resolution [23], and thus it is particularly suited for proving contradictory formulas. In order to make it effective for satisfiable formulas as well, it is necessary to include satisfiability tests. These look, guided by some heuristic, for a satisfying assignment of the formula. If such an assignment is found, the formula is proven satisfiable and the search can be cut short. Typically, a satisfiability test should be applied at each level of the search. We have experimented with two types of satisfiability tests:

- The first is based on the gradient of polynomial transforms of CNF formulas [20]. Essentially, a proposition is chosen and set to *true* if its number of unnegated occurrences is higher than its number of negated occurrences, otherwise it is set to *false*. Subsequently the simple rules are applied, and a next proposition is chosen, until either a contradiction is found, or a satisfiable assignment is constructed.
- The other is a weighted variant of a greedy local search algorithm [25]. Starting from a random assignment, variables are 'flipped' (set from *true* to *false* or vice versa), such that

the weight of satisfied clauses increases. If no improving 'flip' is identified, the weights of the unsatisfied clauses are increased until an improving flip comes into existence. In this way, 'difficult' clauses are getting large weights and thus are more likely to be satisfied in the end. This process is repeated until either a satisfiable assignment is constructed, or the maximum number of flips is exceeded.

It may be noted that for many satisfiable formulas the local search algorithm is sufficient for finding a model quickly. Thus it tends to obscure HeerHugo's performance; therefore we did not make use of it in the computations in Section 5.

3.5 Some complexity issues

If a formula can be proven to be a contradiction using simple rules only (as in (a)), it is said to be in *hardness class* 0. If a formula is proven contradictory using the application of the branch/merge rule on level 1 (as in (b)), it is said to be in hardness class 1. Similarly, a formula that can be proven using i nested applications of the branch/merge rule, but cannot be proven using $i-1$ nested applications, is said to be in hardness clas i [26]. If application of simple rules is linear (which is the case in Stållmarck's prover, and almost in HeerHugo) bounds on the lengths of proof size and search for a contradictory formula Φ in hardness class h can be derived [26].

$O((3\|\Phi\|)^{h+1})$	length of a proof to refute Φ
$O((3\|\Phi\|)^{2h+1})$	length of the proof search to refute Φ
2^h	lower bound on the proof length of Φ

Here $|\Phi|$ denotes the size of the formula Φ. It should be noted that the number of propositions occurring in Φ is a better measure than its size, except in constructed cases where large formulas contain an extremely low number of propositions.

The hardness class of a formula gives a very good intuition for the provability of a formula. A formula in hardness class 0 can without problem contain 10^6 propositions, whereas the refutability of a formula with 100 propositions in hardness class 3 is doubtful. Moreover, it appears that formulas belonging to a certain kind of application, have a typical hardness class associated with them. However, there also formulas of which the hardness class increases with size. E.g. prime number tests and the pigeon hole formulas appear to range over all classes.

It is worthwhile to note that the hardness class very much depends on the nature of the simple rules. We have observed that by increasing the strength of the simple rules, formulas shifted down from higher to lower hardness classes. As far as we know the relation between hardness classes and the nature of the simple rules has never been investigated.

Finally, the hardness classes appear to be related to the notion of hierarchies of polynomially solvable SAT problems as first introduced by Gallo and Scutellá [10]. It is however a different notion, since the hardness class is not a static property of a formula (as it depends on the simple rules that are applied). Moreover, it is not defined for satisfiable instances.

4. Detecting structure in formulas

In this section we show how HeerHugo detects and employs certain kinds of structure in a formula. Maybe such observations lead to relating rules and algorithms to certain patterns in formulas, which may lead to better understanding of proof search in propositional logic.

4.1 Identification and removal of identical subterms
Suppose we start with a formula (not yet in CNF) and this formula contains identical subterms, i.e. the subformula $q \oplus r$ occurs twice. Assume that the identical subformulas are assigned auxiliary propositions p_0 and p_1. It is easy to verify that for any operator, using simple rules and the single level branch/merge rule, it is concluded that $p_0 \leftrightarrow p_1$. In this way the equivalence between all pairs of equal subterms is detected. For example, if $\oplus \equiv \wedge$, we find in the $\leq$3CNF formula 6 clauses that are equivalent to $p_0 \leftrightarrow (q \wedge r)$ and $p_1 \leftrightarrow (q \wedge r)$. Now if for example q is considered in a branch/merge step, it is derived assuming $\neg q$ that $\neg p_0$ and $\neg p_1$. Assuming q, it is derived that $p_0 \leftrightarrow r$ and $p_1 \leftrightarrow r$. So, taking the intersection of these results, we find that $p_0 \leftrightarrow p_1$.

In case the translation is used where implication arrows are used instead of bi–implications in the translation to $\leq$3CNF, a similar effect occurs, but only if the simple Classical Davis Putnam rules are being applied.

4.2 Identification and removal of similar propositions
Suppose the propositions p and q occur in a similar way; more precisely, Φ can be expressed as

$$\begin{aligned}\Phi &\equiv (p \vee \psi) \wedge (\neg p \vee \psi') \wedge \psi'' \\ &\equiv (q \vee \psi) \wedge (\neg q \vee \psi') \wedge \psi'''.\end{aligned}$$

Using the branch/merge rule, monotonic variable fixing and subsumption and ad hoc resolution, the system detects that p and q can be made equivalent without violating satisfiability. Assume using the branch/merge rule that p holds; then ψ' is derived. This means that using ad hoc resolution the subformula $(\neg q \vee \psi')$ is proven superfluous. This implies that q only occurs positively. So, using monotonic variable fixing q is set to *true*. In particular $p \leftrightarrow q$. Similarly, assuming $\neg p$, leads to the conclusion $p \leftrightarrow q$ as well. So, HeerHugo concludes that p and q may be assumed to be equivalent in this case.

5. Computational experience
The most common way to compare the efficiency of implementations of propositional checkers is by applying the tool to a set of benchmarks. In this section we provide computational results on various classes of benchmarks. All results are obtained on an Silicon Graphics PowerChallenge R10000 (195 Mhz R10000 CPU, 1MB secondary cache, Irix6.2) with sufficient main memory. Times reported are the wall clock times (in seconds) reported by HeerHugo to solve the formulas.

5.1 Safety at railway stations
The initial purpose of HeerHugo was to solve a collection of formulas expressing the safety of operations at the Dutch railway stations in Heerhugowaard (see also [21]). A method of expressing safety criteria for railway stations as a propositional formulas is developed and discussed in [9]. Here we only give a brief outline. Given the layout of a railway station (i.e. the infrastructure of the railroads, the position of the signals and level crossings etc.), a number of requirements is formulated (in terms of propositional formulas) to protect trains from colliding or derailing. Furthermore, a set of assignments is given that correspond to situations (i.e. positions of trains, red/green signals etc.) that may occur given the current

time table. For each of these assignments all safety criteria must be satisfied *at any time*. So, denoting by $\Phi_0, \ldots, \Phi_n$ the assignments at $n+1$ sequential moments in time, and by R a particular safety criterion, it must hold that $(\Phi_0 \wedge \ldots \wedge \Phi_n) \rightarrow R$ is a tautology, or equivalently, its negation is a contradiction. Even for small stations such formulas can grow quite large. Initially, it was attempted to solve the resulting formulas using BDDs [11], but unfortunately it appeared that these could not handle the instances `heerhugo*` associated with Heerhugowaard railway station. Ultimately, it turned out that HeerHugo solves them with ease. In Table 3 the results of HeerHugo on a number of benchmarks[2] are given. Given are the size of formula after transformation to $\leq$3CNF and after the application of exhaustive search at various levels; m gives the number of propositions and n gives the number of clauses. It is indicated at which level the formulas was decided to be contradictory or satisfiable.

name	≤3CNF		level 0		level 1		time
	m	n	m	n	m	n	
h-k1	660	1197	361	788	SAT		1
h-k2	624	1140	FALSE				1
h-k3	645	1190	FALSE				1
h-k4	593	1074	FALSE				1
heerhugo2	8353	18175	5225	13942	SAT		13
heerhugo3	8351	18173	5223	13937	FALSE		5
heerhugo4	8574	18669	5339	14284	FALSE		4
heerhugo5	6286	13496	3922	10210	SAT		15

Table 3: HeerHugo results railway station instances.

5.2 Prime numbers

In this section we consider the problem of proving or disproving that certain numbers are prime. This benchmark has a few advantages over others. It is easy to generate arbitrarily large formulas that are either satisfiable, or contradictory. Moreover, it can easily be predicted whether a formula is satisfiable or not; the numbers are so small that using conventional means it is easy to check whether the numbers are prime.

The idea behind generating these formulas is the following. Given a number b represented as a binary vector. We search for a factorization of b, i.e. we search for numbers f_0 and f_1 which are strings of propositions of the same length as b such that $f_0 \cdot f_1 = b$ and $f_0, f_1 > 1$. Here $\cdot$ is the standard binary multiplication on numbers. The multiplication is given by introducing propositions s_{ij} for intermediate results and intermediate carries c_{ij}. The core of the multiplication is given by the following formulas. At the boundaries of the multiplication, these formulas are simpler, for instance because carries are known to be 0.

$$s_{i+j,j} \leftrightarrow (c_{i+j-1,j-1} \leftrightarrow (s_{i+j,j-1} \leftrightarrow (p_i \wedge q_j))),$$
$$c_{i+j,j} \leftrightarrow ((c_{i+j-1,j-1} \wedge s_{i+j,j-1}) \vee (c_{i+j-1,j-1} \wedge p_i \wedge q_j) \vee (s_{i+j,j-1} \wedge p_i \wedge q_j)).$$

In Figure 4 results are listed of applying HeerHugo to some of these formulas. We list the number to be checked for primality. Given are the size of formula after transformation to

[2]These formulas, and the prime–formulas of the next section, can be obtained by contacting one of the authors.

≤3CNF and after the application of exhaustive search at various levels; m gives the number of propositions and n gives the number of clauses. It is indicated at which level the formulas was decided to be contradictory (i.e. b is prime) or satisfiable. Considering the results, we

	≤3CNF		level 0		level 1		level 2		
b	m	n	m	n	m	n	m	n	time
112	451	1000	114	360	SAT				1
113	451	1000	94	305	FALSE				1
257	742	1673	195	636	FALSE				2
4711	1540	3535	453	1476	190	732	SAT		9
47161	2328	5384	741	2408	316	1221	FALSE		61
655379	3630	8450	1243	4046	546	2095	FALSE		1181
655381	3630	8450	1259	4097	548	2107	SAT		139
3476734	4389	10241	1585	5129	712	2727	SAT		367
3476741	4389	10241	1529	4966	701	2701	FALSE		13810
58697731	6123	14339	2125	6881	1015	3904	807	3618	*914869
58697733	6123	14339	2141	6932	1018	3930	SAT		558

Table 4: HeerHugo results on prime number benchmarks. *This instance was found a contradiction at level 3.

conclude that the hardness class of the prime–formulas increases with b. Furthermore, the table gives a good indication of HeerHugo's ability to reduce the size of the formulas. It may be stressed that the search at level 0 and 1 takes up little time; yet, the reduction in problem size is substantial.

5.3 DIMACS benchmarks

The best known set of SAT benchmarks is the DIMACS suite [28]. It contains instances stemming from practical applications, as well as constructed hard instances. We used HeerHugo to solve a subset of these instances. Most of these instances are fairly hard for DPLL algorithms (see the various results in [18]). Many of the DIMACS benchmarks are satisfiable, and thus rather easily solved by local search methods[3]; so the main interest lies in the unsatisfiable formulas. Note that all the DIMACS instances are already in CNF; yet, HeerHugo applies the transformation to ≤3CNF anyway (even if the problem is already in ≤3CNF format).

We solved the following instances:

- The `bf*` and `ssa*` formulas (which stem from practical applications); results are in Tables 5 and 6. For each problem are listed the original size of the problem (m refers to the number of variables, n to the number of clauses), the size after transforming to ≤3CNF, the sizes after the search at the levels 0 (i.e. the application of simple rules), 1 and 2. It is indicated at which level the problem was solved, and the total time it required.
- We also ran the `dubois*` and `pret*` instances, which are similar constructed 3CNF instances. These are very hard for DPLL algorithms but polynomially solvable by a special purpose algorithm [31]. HeerHugo solved each of the `dubois*` instances within

[3] This observation does not hold for the notorious `par32*` instances. These have only been solved making use of their special structure [32].

a second; they were all found to be contradictory at level 1. For the pret* instances HeerHugo requires up to 30 seconds. These were found to be contradictory at level 2. The par8* and par16* instances contain large subformulas with similar structure as the above mentioned instances (see also [32]). The smaller of these (par8*) are found to be satisfiable at level 1 or 2 within a second. For the larger ones a search at level 3 is required which takes several hours. HeerHugo did not succeed in solving the par32* formulas.

- Finally, we solved the aim* instances. These are constructed 3CNF instances, both satisfiable and contradictory, with 50 to 200 variables. HeerHugo did not encounter any trouble in solving these; each of the instances was solved in fractions of seconds. Almost all instances are in hardness class 1 or lower. A few are in hardness class 2, but even these are solved within a second.

	original		≤3CNF		level 0		level 1		
name	m	n	m	n	m	n	m	n	time
ssa0432–003	435	1027	2798	3398	115	272	FALSE		1
ssa2670–130	1359	3321	9083	11044	546	1289	FALSE		4
ssa2670–141	986	2315	6223	7551	379	899	FALSE		3
ssa6288–047	10410	34238	97832	121659	FALSE				16
ssa7552–038	1501	3575	9748	11821	437	1112	SAT		4
ssa7552–158	1363	3034	8189	9859	241	552	SAT		2
ssa7552–038	1363	3032	8184	9852	285	661	SAT		3
ssa7552–038	1391	3126	8415	10149	421	1003	SAT		3

Table 5: Results on the ssa* instances.

	original		≤3CNF		level 0		level 1		level 2		
name	m	n	m	n	m	n	m	n	m	n	time
bf0432–007	1040	3668	10613	13240	536	1985	175	519	FALSE		35
bf1355–075	2180	6778	19289	23886	789	2593	FALSE				7
bf1355–638	2177	4768	19257	23847	887	2803	FALSE				6
bf2670–001	1393	3434	9403	11443	495	1181	FALSE				2

Table 6: Results on the bf* instances.

Considering the above results we conclude that HeerHugo is very effective in solving certain types of instances, where other algorithms (including DPLL–like algorithms) require considerable amounts of time. On the other hand, when applied to random 3SAT formulas on the threshold (especially unsatisfiable ones), HeerHugo is in general not successful when the number of nested assumptions is restricted to 2. Typically, at level one the ≤3CNF formula is reduced again to the original one, while going one level further does not yield any new conclusions. Obviously, increasing the number of nested assumptions will eventually lead to a correct answer, but this requires substantial computation times. We also observed that HeerHugo has trouble solving large satisfiable formulas like the DIMACS ii* instances. The transformation to ≤3CNF increases the size of the formulas, and HeerHugo has to do a lot of time–consuming reasoning before the formulas reduce to a more manageable size.

6. Concluding remarks

The construction of HeerHugo, and its predecessors (a tool based on normal forms of proofs and a tool using binary decision diagrams (see also [11])) as well as experience with the first order theorem prover otter [35] led to the observation that there is hardly any fundamental understanding of proof search in propositional logic. Attempts at obtaining a deeper understanding that we are aware of can be found in [17] where it is tried to clarify the effects of different branching rules, while there also exists a wealth of probabilistic results (see for an overview [15]); unfortunately, these rely on assumptions on the distribution of the formulas. This makes these results, which are very interesting from a theoretical viewpoint, difficult to apply and interpret when it comes to understanding practical satisfiability solving. The same holds for the results on proof lengths [19, 30]; the existence of a short proof in some proof system, does not at all guarantee that a reasonably short proof will be found in an automated way.

Our experiences with HeerHugo and other solvers, clearly show that the current level of understanding how proofs can be found efficiently in practice is low; it is based mainly on computational experience, rather than on any theoretical foundations. Without truly understanding why, it appeared to be possible to achieve considerable improvements in performance for formulas in higher hardness classes. Rather typically the performance doubled with almost every round of improving HeerHugo. We expect that it is possible to double performance of HeerHugo in a similar way a few more times. The reason for this is that the logical rules and search algorithms that we use are quite standard; all can be found somewhere in the literature, and other existing efficiently implementable rules are not applied yet. We did not (yet) compare the rules with each other in order to remove redundancies in search; nor do we use any heuristics for finding the best propositions to branch on.

By applying HeerHugo to a number of well known benchmarks, we have concluded that it is a powerful complementary approach to solve SAT problems, that seems to be particularly effective on formulas with a lot of structure, especially those stemming from real–life applications; see also [12]. Such formulas appear to be almost always in hardness class 2 or less, and thus efficiently solvable by HeerHugo. The efficiency with which HeerHugo solves structured instances should be attributed to its reasoning strength. Rather than just doing a lot of guessing, HeerHugo derives as many facts as possible from increasingly complex assumptions. Thus a lot of structure is recognized and subsequently exploited. As opposed to HeerHugo, standard implementations of DPLL (depth–first search) algorithms are less suited to exploit structure. When such implementations are enhanced with intelligent backjumping techniques, their reasoning strength increases [36]. It appears that for efficiently solving many classes of formulas exploiting structure is of the utmost importance.

Even when HeerHugo does not manage to solve a formula at level 2 or lower, it is able to substantially reduce the size of the formula during the search. Subsequently, the reduced formula could be solved by another solver that might be better suited for that particular type of formulas. In this sense HeerHugo can be used as a *preprocessing* tool, which is especially useful for contradictory formulas.

On the other hand, many formulas that are difficult for HeerHugo, for example random 3SAT instances and large satisfiable formulas, are often rather easily solved by a depth–first DPLL

approach or even by a greedy local search algorithm. Thus it appears that satisfiability solving has become an art of choosing the right algorithm for the right instance.

A Using HeerHugo

Version 0.2 of HeerHugo is designed to have an extremely simple input/output behaviour, in order to guarantee that it can be used on any computer platform. An input formula must be in `Ain` and output will appear in a file called `Aout`. HeerHugo will also write progress information in `Aout`, which can be used for monitoring purposes. HeerHugo can be obtained by anonymous ftp from `ftp://ftp.cwi.nl/pub/jfg` for experimental purposes. Note that HeerHugo 0.2 is experimental software, and, although it appears to work correctly, may contain flaws. HeerHugo comes with a README file explaining the use of several flags, options and constants. In the file `Ain` a formula can be written using the operators `->` (implication), `&` (and), `|` (or) `~` (negation) and `<->` (bi-implication). Elementary propositions can be denoted by a sequence of letters and digits. Brackets can be used. A line starting with an `%` is taken to be a comment. The following expression is a correct input:

```
% This is an example input file for HeerHugo 2.0
% containing sufficient facts to derive that Jan and Gijs
% do not have the same birthday
( 13April <-> JanBirthday ) &
( 27September <-> GijsBirthday ) &
( ~13April | ~27September )
```

References

1. B. Aspvall, M.F. Plass, and R.E. Tarjan. A linear–time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3):121–123, 1979.
2. R.E. Bryant. Graph–based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C–35(8), 1986.
3. S.A. Cook. The complexity of theorem proving procedures. In *Proceedings of the 3rd annual ACM symposium on the Theory of Computing*, pages 151–158, 1971.
4. D. Van Dalen. *Logic and structure*. Springer–Verlag, Berlin, 3rd edition, 1994.
5. M. Davis, M. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
6. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:210–215, 1960.
7. W.F. Dowling and J.H. Gallier. Linear–time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.
8. O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. In Johnson and Trick [18], pages 415–436.
9. W.J. Fokkink. Safety criteria for the vital processor interlocking at Hoorn–Kersenboogerd. In *Proceedings 5th Conference on Computers in Railways – COMPRAIL'96, Volume I: Railway Systems and Management*, pages 101–110. Computational Mechanics Publications, 1996.
10. G. Gallo and M.G. Scutellá. Polynomially solvable satisfiability problems. *Information Processing Letters*, 29:221–227, 1988.
11. J.F. Groote. Hiding propositional constants in BDDs. *Formal Methods in System Design*, 8:91–96, 1996.
12. J.F. Groote, J.W.C. Koorn, and S.F.M. van Vlijmen. The safety guaranteeing system at station Hoorn-Kersenboogerd (extended abstract). In *Proceedings 10th IEEE Conference on Computer Assurance (COMPASS'95)*, pages 57–68, Maryland, 1995.
13. J. Gu. Local search for the satisfiability (SAT) problem. *IEEE Transactions on Systems, Man and Cybernetics*, 23(4):1108–1129, 1993.
14. J. Gu. Global optimization for satisfiability (SAT) problem. *IEEE Transactions on Knowledge and Data Engineering*, 6(3):361–381, 1994.
15. J. Gu, P.W. Purdom, J. Franco, and B.W. Wah. Algorithms for the satisfiability (SAT) problem: a survey. In D. Du, J. Gu, and P.M. Pardalos, editors, *Satisfiability problem:*

Theory and applications, volume 35 of *DIMACS series in Discrete Mathematics and Computer Science*, pages 9–151. American Mathematical Society, 1997.

16. J. Harrison. Stållmarck's algorithm as a HOL derived rule. In J. von Wright, J. Grundy, and J. Harrison, editors, *Proceedings of TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 221–234, 1996.
17. J.N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359–383, 1995.
18. D.S. Johnson and M.A. Trick, editors. *Cliques, Coloring and Satisfiability: Second DIMACS implementation challenge*, volume 26 of *DIMACS series in Discrete Mathematics and Computer Science*. American Mathematical Society, 1996.
19. J. Krajícek. *Bounded arithmetic, propositional logic, and complexity theory*, volume 60 of *Encycplodia of Mathematics and its Applications*. Cambridge University Press, 1996.
20. H. Van Maaren, J.F. Groote, and M. Rozema. Verification of propositional formulae by means of convex and concave transforms. Technical Report 95–74, Faculty of Technical Mathematics and Informatics, Delft University of Technology, Delft, The Netherlands, 1995.
21. J. Mertens. Verifying the safety guaranteeing system at railway station Heerhugowaard. Master's thesis, University of Utrecht, Utrecht, The Netherlands, 1996.
22. D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, San Jose, CA, 1992.
23. J.A. Robinson. A machine–oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.
24. J.S. Schlipf, F.S. Annexstein, J.V. Franco, and R.P. Swaminathan. On finding solutions for extended Horn formulas. *Information Processing Letters*, 54:133–137, 1995.
25. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446, San Jose, CA, 1992.
26. G. Stållmarck. A proof theoretic concept of tautological hardness. Incomplete manuscript, 1994.
27. G. Stållmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from boolean formula. United States Patent number 5,276,897; Swedish Patent 467 076; European Patent 0 403 454; Canadian Patent 2,018,828, 1994.
28. M.A. Trick. Second DIMACS challenge test problems. In Johnson and Trick [18], pages 653–657.
29. G.S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic*, part 2:115–125, 1968. Reprinted in J. Siekmann and G. Wrightson (editors), *Automation of reasoning* vol. 2, Springer–Verlag Berlin, 1983.

30. A. Urquhart. The complexity of propositional proofs. *The Bulletin of Symbolic Logic*, 1(4):425–467, 1995.

31. J.P. Warners and H. Van Maaren. Recognition of tractable satisfiability problems through balanced polynomial representations. Accepted for publication in *Discrete Applied Mathematics*.

32. J.P. Warners and H. Van Maaren. A two phase algorithm for solving a class of hard satisfiability problems. Technical Report SEN-R9802, Centre for Mathematics and Computer Science (CWI), Amsterdam, The Netherlands, 1998. Accepted for publication in *Operations Research Letters*.

33. F. Widebäck. Stållmarck's notion of n-saturation. Technical Report NP–K–FW–200, Logikkonsult NP AB, Sweden, 1996.

34. J.M. Wilson. Compact normal forms in propositional logic and integer programming formulations. *Computers and Operations Research*, 17(3):309–314, 1990.

35. L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning*. McGraw–Hill, 1992.

36. H. Zhang. SATO: An efficient propositional solver. In *Proceedings of CADE–97*, 1997.