The propositional formula checker HeerHugo

J.F. Groote

CWI, Department of software technology

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

Tel: +31-20-5924232 Fax: +31-20-5924199

E-mail: `jfg@cwi.nl`

Abstract

HeerHugo is a propositional formula checker[1], that determines whether a given formula is satisfiable or not. Its name comes from the dutch railway station Heerhugowaard, as it was developed to validate correct operation of the Vital Processor Interlocking unit guarding the safety of all railway traffic at this station.

We describe how HeerHugo attempts to prove or disprove formulas, and we end with some theoretical considerations regarding the (lack of) progress in the area of automatic theorem proving.

1 Introduction

In 1994 we[2] contacted the Dutch Railway Company (Nederlandse Spoorwegen) in order to find test beds in which the theory we had developed could be applied. We ran into the problem of checking correctness properties of Vital Processor Interlockings that are being used to guard safety at smaller Dutch railway yards [10, 11]. After some preprocessing, the question could be reduced to checking whether large propositional formulas ($\leq$ 120K) were satisfiable. In order to prove these formulas we started off by applying much praised Binary Decision Diagrams (BDDs [2]) only to find out that BDDs were in no way up to this task [9]. By accident we came into contact with Gunnar Stålmarck who had founded a company called Logikonsult based on a particularly efficient propositional formula checker. Their checker solved the formulas ($\leq$ 6K) expressing correctness of the VPI at the railway station Hoorn-Kersenboogerd. Gunnar gladly explained me some principles behind their prover [20, 13]. Their prover is protected by a software patent [18].

In order to understand the stunning effectiveness of the Stålmarck propositional formula checker, we decided to build a checker along the same principles ourselves, although for a while we could use the Stålmarck checker for experiments. When starting with verifying the large formulas belonging to Heerhugowaard a sudden change of operating systems made it impossible to use Stålmarck's checker (which was not available for the newer platform). At that moment the experimental checker became instrumental for our verifications and we decided to baptize the new checker HeerHugo.

We used a somewhat different fundamental setup in our checker than in Stålmarck's. This allowed us to experiment with a wide range of inference and complexity reducing rules. Typically, all rules that we use (and sometimes invented) are documented somewhere in the vast literature related to the satisfiability problem in proposition logic. However, there does not seem to be any understanding as to which rules are more and which rules are less effective.

[1]For experimental purposes the checker can be obtained by contacting the author.

[2]Bas van Vlijmen took initiative and established the first contacts.

Acknowledgements.

Thanks go to Joost Warners and Marc Bezem for helping to track some references, and to Bert Lisser, John Tromp and Joost Warners for their comments on this text.

2 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.

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 )
```

3 Algorithms and rules

We describe how the prover is working in three 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 proposition letters that occur in it. In the third section we describe Stålmarck's branch and merge rule, which is necessary to make the system complete.

3.1 Transformation to $\leq$3CNF

We describe a linear transformation to $\leq$3CNF form, i.e. formulas in conjunctive normal form with at most 3 literals per clause, which maintains satisfiability. This transformation is believed to be first described by Tseitin [17]. It has been shown that it is impossible to linearly transform a formula to CNF while maintaining validity.

Take an arbitrary formula ϕ. For the explanation we restrict ourselves to the connectives $\wedge$ and $\neg$, but it is easy to see that the transformation described here applies to any unary or binary connective. Introduce for any subformula ψ of ϕ a new proposition letter p_ψ, except if ψ is itself a proposition letter, say p, in which case we take $p_\psi \equiv p$.

First construct the following formula:

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

Here $Sub(\phi)$ is the set of subformulas of ϕ. Note that the size of this set is linear in ϕ. Note that if subformula ψ occurs positively in ϕ i.e. within the scope of an even number of negations, then, maintaining satisfiability, the $\leftrightarrow$ in the formula above can be replaced by $\rightarrow$. If ψ occurs negatively, i.e. in the scope of an odd number of negations, the $\leftrightarrow$ can be replaced by $\leftarrow$ (implication from right to left). It appears that this latter improvement (sometimes) leads to a considerable efficiency gain of HeerHugo, especially when the formula is satisfiable.

The next step is to transform each 'triple' $(p_\psi \leftrightarrow p_{\psi_1} \wedge p_{\psi_2})$ and each formula $p_\psi \leftrightarrow \neg p_{\psi_1}$ to $\leq$3CNF form. This is easily accomplished by replacing these formulas respectively by

$$\begin{array}{lll} (\neg p_\psi \vee p_{\psi_1})\wedge & \qquad & (\neg p_\psi \vee p_{\psi_1})\wedge \\ (\neg p_\psi \vee p_{\psi_2})\wedge & & (p_\psi \vee \neg p_{\psi_1}) \\ (p_\psi \vee \neg p_{\psi_1} \vee \neg p_{\psi_2}) & & \end{array}$$

It is common to use the word literal for formulas of the form p and $\neg p$. We typically use the letter l to denote a literal. A clause is a maximal formula of the form $(l_1 \vee \cdots \vee l_n)$ in a CNF.

We have the following property that justifies our transformation. The $\leq$3CNF formula that is obtained in the way described above has exactly the same satisfying assignments as the original formula ϕ, provided these are restricted to the proposition letters occurring in ϕ.

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 we are working with the larger $\leq$3CNF formulas.

3.2 Simple rules

From now on we work with $\leq$3CNF formulas. In order to simplify a formula, we simultaneously apply a number of transformations that we describe below.

Unit resolution

An effective step to reduce the size of a formula is the application of unit resolution. Basically, the idea is as follows. Suppose there is a unit literal in a $\leq$3CNF formula ϕ, i.e. a clause of the form p or $\neg p$. If the clause has the form p, then in order to satisfy ϕ, p must be true. So, assume p is true, and remove all clauses that contain p, as such clauses are also true. If there is a clause containing $\neg p$, e.g. $\neg p \vee q \vee r$, then the clause is satisfiable iff it is satisfiable after removal of $\neg p$ (yielding $q \vee r$ in the example). So, remove all occurrences of $\neg p$ in clauses. Note that after this step all occurrences of p are removed from ϕ. The case with unit literal $\neg p$ is dual to the case just described.

Note also that during such a unit resolution step, new unit clauses can be created. We use an algorithm that applies all unit resolution steps in linear time [6]. Furthermore, note that during a unit resolution steps unit clauses p and $\neg p$ can come into existence. In this case ϕ is clearly contradictory. It is interesting to know that unit resolution is complete for Horn clauses, 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

It happens sometimes that there are clauses of length two, e.g. $\neg p \vee q$, that viewed as implications, h.l. $p \rightarrow q$ form a cycle. E.g. $p \rightarrow q$, $q \rightarrow \neg r$ and $\neg r \rightarrow p$. Clearly, all proposition letters on such a cycle are equivalent, and henceforth can be given the same name. If the clauses above occur in a formula ϕ above all occurrences of p in ϕ can be replaced by q, and every occurrence of $\neg r$ in ϕ can be replaced by $\neg q$. Detecting such implicational cycles is linear in the size of ϕ. Make a graph with as vertices literals p and $\neg p$ for every p. Add for each clause $l \vee l'$ an edge from $\neg l$ to l' and one from $\neg l'$ to l. Use the standard linear algorithm for detecting strongly connected components to find implicational cycles. In [1] it is shown that this method yields a linear algorithm to determine whether a 2CNF (a CNF with clauses of length 2) is satisfiable.

Classical Davis/Putnam rule

One of the oldest rules in propositional reasoning is proposed in [4]. It can be formulated for positive p as follows. For $\neg p$ the dual of this rule can be chosen. Given a CNF of the form

$$\begin{array}{c}(p \vee L_1)\wedge \\ \vdots \\ (p \vee L_n)\wedge \\ \phi\end{array}$$

where $L_i \equiv l_{i1} \vee \cdots \vee l_{ik_i}$ and p does not occur positively in ϕ. We can replace this CNF by

$$\phi[\neg p := \textstyle\bigwedge_{i=1}^{n} L_i]$$

maintaining all satisfying assignments on proposition letters in ϕ, except for p. If p was the only letter that occurred, the resulting formula is true iff the original was satisfiable.

When carrying out the substitution $\phi[\neg p := \bigwedge_{i=1}^{n} L_i]$ it is easy to obtain a CNF again, but auxiliary proposition letters must be introduced to maintain $\leq$3CNF. E.g. in case some L_i contains two literals and $\neg p$ occurs in a clause with 3 literals. Moreover, if p occurs often both positively and negatively, the resulting $\leq$3CNF is considerably larger than the original. In HeerHugo the classical Davis Putnam rule is carried out only when the formula reduces in size, or there is a limited growth. Just before applying the branch/merge rule (see below) to a proposition letter p it is investigated whether the Classical Davis Putnam rule can be fruitfully applied.

There are two special cases that deserve to be mentioned. When p only occurs positively or negatively in a formula, then the rule above says that all clauses in which p occurs may be thrown away. This is called monotonic variable fixing, as it is assumed that if p occurs positively, it is assigned true, and if it occurs negatively, it is assigned the value false. Another case happens when p occurs only once in a clause of length 2 (either positively or negatively), and occurs arbitrary often with the other sign. Say, this clause has the form $p \vee q$ (skipping all dual cases). According to the Classical Davis Putnam rule we may substitute all occurrences of $\neg p$ by q reducing the number of proposition letters by one, and reducing the length of the formula. HeerHugo has facilities to immediately detect whether one of these special cases of the Classical Davis Putnam rule can be applied, and tries to apply these as soon as possible.

The classical Davis Putnam rule must not be confused with the method of Loveland, Davis and Putnam [16] which is currently far more popular, but is essentially a mixture of unit resolution, monotone variable fixing and an elementary branching rule.

Subsumption and ad hoc resolution

HeerHugo constantly changes its set of clauses by removing and adding clauses. Whenever a clause is added, it is checked whether a similar clause exists. More precisely if a clause $C_1 \equiv p \vee q \vee r$ is added, and a clause C_2 is present, then an action is taken conforming to the following table. If no such clause C_2 is present, C_1 is simply added. If p, q or r appear with negations, dual actions are taken.

C_2	
$p \vee q \vee r$	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 $p \vee r$

In case a clause of the form $C_1 \equiv p \vee q$ is added and a clause C_2 is present then actions are undertaken according to the table below. Again, if no such C_2 is present, C_1 is simply added. Again, the dual cases, where p or q can be negated are left to the reader.

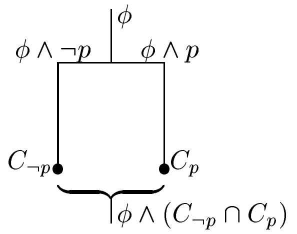
C_2	
$p \vee q$	do not add C_1
$p \vee \neg q$	do not add C_1, remove C_2 and add p
$\neg p \vee q$	do not add C_1, remove C_2 and add q
$\neg p \vee \neg q$	do not add C_1, remove C_2 and set $p \equiv q$
$p \vee q \vee r$	remove C_2
$p \vee q \vee \neg r$	remove C_2
$p \vee \neg q \vee r$	remove C_2,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

3.3 A branching scheme

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 [19] where it is called the dilemma rule.

The branch/merge rule

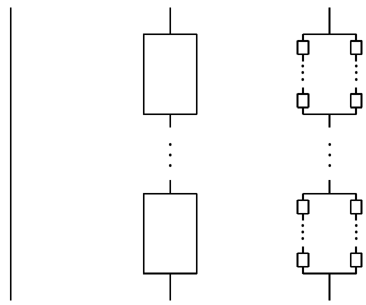
The branch/merge rule stems from [19] where it is called the dilemma rule, and appears to be remarkably effective in proving formulas. We actually believe that the success of the Stålmarck prover can be solely contributed to this rule, as its other aspect, namely the handling of tuples, is a slightly enhanced variant of unit resolution, and is completely subsumed by all rules mentioned above. The branch/merge rule can be depicted as follows:



It must be interpreted as follows. Starting with a $\leq$3CNF ϕ, the proof is split in two parts. In the first part, $\neg p$ is added to ϕ and using the simple rules, a set of conclusions $C_{\neg p}$ is drawn. These conclusions have the form q, or $\neg q$, i.e. a certain proposition letter must or must not hold, or they have the form $q \leftrightarrow r$ or $q \leftrightarrow \neg r$, i.e. q and r must, or must not be equivalent. So, $C_{\neg p}$ contains all conclusions using simple rules from $\neg p$. Moreover, $C_{\neg p}$ is closed in the following way. In case $\phi \wedge \neg p$ contains q and $\neg q$ for some proposition letter q, an inconsistency has been derived and $C_{\neg p}$ is set equal to the set of all literals as well as all (in)equivalences between proposition letters. Otherwise, in case q is derived for some proposition letter q, also an inequivalence $p \leftrightarrow \neg q$ is added to $C_{\neg p}$, and similarly, if $\neg q$ is derived, $p \leftrightarrow q$ is added to $C_{\neg p}$. Then, the transitive closure of biimplications is taken. I.e. if $p \leftrightarrow \neg q$ and $\neg q \leftrightarrow r$ are in $C_{\neg p}$, $p \leftrightarrow r$ is added. In the same way, C_p is constructed by adding p to ϕ.

After both branches have been explored, the intersection between $C_{\neg p}$ and C_p is calculated, and each fact in the intersection is added to ϕ. Calculating this intersection can be done efficiently. If the intersection is not empty, then the simple rules are being used to simplify $\phi \wedge (C_{\neg p} \cap C_p)$.

Heerhugo applies the branch/merge rule for every proposition letter in turn. Leading to a diagram as in (b) below.



(a) (b) (c)

The branch/merge is iteratively applied to all proposition letters until there is a full round where for all proposition letters the intersection between $C_{\neg p}$ and C_p is empty. This means that applying the branch rule to any proposition letter does not lead to any new facts. In this case the branch rule is applied in a nested way. See (c) above. First, at level one, one proposition letter is put to true and false, and within this assumption, the branch/merge rule is applied to all proposition letters at level one, until no new facts can be derived. Then a next proposition letter at level 1 is chosen.

If a formula can be proven to be a contradiction using simple rules only, as in (a) above, 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) above, 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 class i [19]. If application of simple rules is linear (which is quite the case in HeerHugo, but which is claimed to be the case in Stålmarck's prover) we can derive the following bounds for a contradictory formula ϕ in hardness class h:

$O(\lvert\phi\rvert^{h+1})$	Length of a proof to refute ϕ
$O(\lvert\phi\rvert^{2h+1})$	Length of the proof search to refute ϕ
2^h	Lower bound on the proof length of ϕ

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

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 proposition letters, whereas the refutability of a formula with 100 proposition letters 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. E.g.

hardware adders are in 2, safety formulas of Hoorn-Kersenboogerd are in 0, the safety formulas of Heerhugowaard are in 1. But there also formulas of which the hardness class increases with size. E.g. prime number tests and the pigeon hole formulas appear to belong to this class.

Note also 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 have never been investigated.

Addition of expensive clauses

It happens that after assuming that a proposition letter, say p, is e.g. true in the branch/merge rule, assuming the validity of a proposition letter q leads to a contradiction. In this case we have derived $p \wedge q \rightarrow$ false, which as a clause looks like $\neg p \vee \neg q$. We consider such a clause expensive, because we may have searched for quite a while before finding it. Furthermore, we consider it useful, as it is a concise fact. Therefore, we add such a clause to the formula. This is also done when the nested assumption of three proposition letters lead to a contradiction.

We have also experimented with adding a clause $\neg p \vee q$ ($\neg p_1 \vee \neg p_2 \vee q$) if we derive q under the assumption p (assumptions p_1 and p_2). We found a degradation of performance, due to the fact that many redundant clauses were generated in this way.

4 Examples

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 proposition logic. We also provide benchmarks that allow for a completely different way of comparing of the strengths of different tools.

4.1 Removal of identical subterms

Suppose we start with a formula (not yet in CNF) and this formula contains identical subterms, say for simplicity that $p \wedge q$ occurs twice. If both subformulas are assigned auxiliary proposition letters r_1 and r_2, we find in the $\leq$3CNF 6 clauses that are equivalent with $r_1 \leftrightarrow (p \wedge q)$ and $r_2 \leftrightarrow (p \wedge q)$. If p is considered in a branch/merge step, we derive assuming $\neg p$ that $\neg r_1$ and $\neg r_2$. We derive assuming p that $r_1 \leftrightarrow q$ and $r_2 \leftrightarrow q$. So, taking the intersection of these results, we find that $r_1 \leftrightarrow r_2$. Note that this also works in case another operator than $\wedge$ is used. Moreover, it is easy to see that in this way the equivalence between all pairs of equal subterms is detected.

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

4.2 Removal of similar proposition letters

Suppose a proposition letter p occurs in a similar way as a proposition letter q. More precisely, for every clause $q \vee L$, there is a clause $p \vee L$, and for every clause $\neg q \vee L$ there is a clause $\neg p \vee L$. Using the branch/merge rule, monotonic variable fixing and subsumption and ad hoc resolution, the system detects that p and q are made equivalent without violating satisfiability. Assume using the branch/merge rule that p holds. Then, for any clause of the form $\neg p \vee L$, L is derived. This means that using ad hoc resolution all clauses of the form $\neg q \vee L$ are proven superfluous. But this means that q only occurs positively. So, using monotonic variable fixing q is set to true. In particular $p \leftrightarrow q$. Similarly, considering $\neg p$, leads again to the conclusion $p \leftrightarrow q$. So, HeerHugo makes p and q equivalent in this case.

4.3 Prime numbers

The most common way to compare the efficiency of implementations of propositional checkers is by applying the tool to a set of benchmarks. In order to give an impression of what we can achieve, we give below a table with times to prove or disprove 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 $\vec{n}$ represented as a binary vector. We search for numbers $\vec{p}$ and $\vec{q}$ which are vectors of proposition letters of the same length as $\vec{n}$ such that $\vec{p} \cdot \vec{q} = \vec{n}$ and $\vec{p}, \vec{q} > 1$. Here $\cdot$ is the standard binary multiplication on numbers. The multiplication is given by introducing proposition letters 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.

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

Below we list results of applying HeerHugo to some of these formulas[3]. The results are obtained on an Silicon Graphics PowerChallenge R10000 (195 Mhz R10000 CPU, 1MB secondary cache, Irix6.2) with sufficient main memory. We list the number to be checked for primality. We list whether the formula is reported satisfiable. In case the formula is a contradiction, we list its hardness class. The size in kilobytes of the formula and its number of proposition letters after transformation to $\leq$3CNF. The last column of the table below is the wall clock time reported by HeerHugo to solve the formula.

$\vec{n}$	satisfiable?	hardness	$\lvert\phi\rvert$	# $\leq$3CNF	time
112	yes	-	3K	451	1s
113	no	1	3K	451	1s
257	no	1	5K	742	2s
4711	yes	-	11K	1540	9s
47161	no	2	17K	2328	61s
655379	no	2	27K	3630	1181s
655381	yes	-	27K	3630	139s
3476734	yes	-	33K	4389	367s
3476741	no	2	33K	4389	13810s
58697731	no	3	47K	6123	914869s
58697733	yes	-	47K	6123	558s

5 Considerations

The construction of HeerHugo, and its predecessors (a tool based on normal forms of proofs and a tool using binary decision diagrams) as well as experience with the first order theorem prover otter [22] led to the observation that there is hardly any fundamental understanding of proof search in proposition logic. The only attempt of obtaining a deeper understanding that we are aware of can be found in [14] where it is attempted to clarify the effects of different branching rules.

The lack of understanding of proof search in proposition logic contrasts somewhat with the an extensive comparative theory of proof length [15, 21]. Sometimes results carry over easily, as in the case of the comparison between tableaux and truth tables [5]. In other cases the situation may not be

[3] The formulas can be obtained by contacting the author.

so clear; e.g. it is known that extended resolution may lead to an exponential improvement of proof lengths over ordinary resolution [12, 3] but it is very doubtful whether the use of extended resolution would lead to faster proof search. One observation deserves to be mentioned. When applying provers that are based on resolution, it is quickly suggested to use (ordered) hyper resolution instead of ordinary binary resolution, as this yields bigger inference steps and reduction of the use of memory. This strongly contrasts with the results in [8] from which one can easily conclude that there is a class of formulas with polynomial proofs using ordinary resolution that only have exponential proofs in hyper resolution.

HeerHugo clearly shows that the current situation is deploring. 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 even more effective rules may linger around. We did not (yet) compare the rules with each other in order to remove redundancies in search. We did not use any heuristics for finding the best proposition letters to branch on.

Yet, the lack of understanding of this field offers a rather nice perspective. Already, HeerHugo (as well as Stålmarck's prover) appear quite effective when it comes to solve realistic problems. And, we expect that an improved understanding will lead to improvements of certain magnitudes (although the difficulty of obtaining such understanding can easily be underestimated). This is illustrated by special purpose provers that outperform HeerHugo when it comes to certain classes (e.g. SATO [23] and the Dubois prover [7]).

This opens the following distant perspective, namely where the area of algorithmics is reduced to translating problem instances polynomially to an instance of a satisfiability problem in proposition logic. Using a state of the art proposition solver, may solve this instance faster than the standard algorithm by detecting structure in this instance which may not be observed by the standard algorithm. Moreover, this could be a more effective approach towards generating programs, as finding and coding a transformation to proposition logic is often easier than formulating a particular algorithm. Moreover, given the chilling speed of contemporary computers, the overhead caused by such an approach will hardly be noticed.

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):677-691, 1986.

[3] S.A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT news*, 1976.

[4] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:210-215, 1960.

[5] M. D'Agostino. Are tableaux an improvement on truth-tables?, *Journal of Logic, Language and Information.* 1:235-252, 1992.

[6] William F. Dowling, Jean H. Gallier: Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. *Journal of Logic Programming*, 1(3):267-284, 1984.

[7] O. Dubois, P. Andre, Y. Boufkhad and J. Carlier, SAT versus UNSAT, Second DIMACS Challenge 1993, AMS DIMACS Series, Volume 26, Pages 415-436, 1993.

[8] A. Goerdt. Unrestricted resolution versus N–resolution. Technical Report KI–NRW 89–7. Universität Duisburg. Germany. 1989.

[9] J.F. Groote. Hiding Propositional Constants in BDDs. *Formal Methods in System Design*, Vol. 8, pages 91-96, 1996.

[10] J.F. Groote, J.W.C. Koorn and S.F.M. van Vlijmen. Formele analyse van het veiligheidssysteem op het station van Hoorn-Kersenboogerd. Informatie, Jaargang 36, nr. 6, pagina's 397-404, 1995.

[11] 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 Annual Conference on Computer Assurance (COMPASS'95), pp. 57-68, Gaithersburg, Maryland, 1995.

[12] A. Haken. The intractability of resolution. *Theoretical Computer Science* 39:297-308, 1985.

[13] J. Harrison. Stålmarck's Algorithm as a HOL derived rule. In J. von Wright, J. Grundy and J. Harrison, Editors, *Proceedings of TPHOLs'96*, Lecture Notes in Computer Science 1125, Finland, pp. 221-234, 1996.

[14] J.N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15(3):359-353, 1995.

[15] J. Krajíček. Bounded arithmetic, propositional logic, and complexity theory. Encyclopedia of mathematics and its applications 60. Cambridge University Press, 1996.

[16] D.W. Loveland. *Automated theorem proving: a logical basis.* North–Holland, Amsterdam, 1978.

[17] G.S. Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic* part 2, pp. 115-125, 1968. Reprinted in J. Siekmann and G. Wrightson (editors), *Automation of reasoning* vol. 2, pp. 466-483. Springer–Verlag Berlin, 1983.

[18] G. Stålmarck. 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; see also Swedish Patent 467 076, 1994.

[19] G. Stålmarck. A proof theoretic concept of tautological hardness. Unpublished manuscript, 1994.

[20] G. Stålmarck and M. Säflund. Modeling and verifying systems and software in propositional logic. In B.K. Daniels, editor, *Safety of Computer Control Systems, 1990 (SAFECOMP '90)*, Pergamon Press. pp. 31-36, Gatwick, UK, 1990.

[21] A. Urquhart. The complexity of propositional proofs. *The bulletin of symbolic logic*, 1(4):425-467, 1995.

[22] L. Wos, R. Overbeek, E. Lusk and J. Boyle. *Automated Reasoning.* Mc-Graw-Hill. 1992.

[23] Zhang, H., Hsiang, J.: Solving open quasigroup problems by propositional reasoning . In *Proceedings of International Computer Symposium*, Hsinchu, Taiwan, 1994.