On the Advantage of a Non-Clausal Davis-Putnam Procedure

Jens Otten *

Fachgebiet Intellektik, Fachbereich Informatik

Technische Hochschule Darmstadt

Alexanderstr. 10, 64283 Darmstadt, Germany

jeotten@intellektik.informatik.th-darmstadt.de

Abstract

We propose a non-clausal decision procedure for classical propositional logic. It is a generalization of the Davis-Putnam procedure and deals with formulae which are not necessarily in clausal form. Since there is no need to translate the investigated formula into normal form, we avoid increasing its size but preserve its structure. This will in general lead to a much smaller search space and thus to a more efficient method. We introduce a short Prolog implementation of our procedure showing that it is possible to implement a non-clausal decision procedure in a compact and adaptable way. To demonstrate the good performance of the non-clausal approach some experimental results are provided. To this end we compare the non-clausal implementation with a clausal-based Davis-Putnam prover using the standard translation into clausal form as well as the definitional translation.

1 Introduction

Decision procedures for classical propositional logic, i.e. procedures determining whether or not a given propositional formula is valid, play an important role in the field of Artificial Intelligence and mathematical logic. Automated theorem proving, solving hard combinatorical optimization problems and verifying circuits in hardware design are some of these applications.

The Davis-Putnam procedure [Davis and Putnam, 1960] or (more precisely) the Davis-Logemann-Loveland procedure [Davis *et al.*, 1962] is one of the most famous and successful decision procedures for classical propositional logic. The essential idea of this procedure consists of the following step: Assign the truth values "true" and "false" to a selected atomic formula A of the investigated formula F and simplify both resulting formulae F_1 and F_2 accordingly. The original formula F is valid if and only if both formulae F_1 and F_2 are valid. Applying this step recursively to the resulting formulae F_1 and F_2 yields a search tree of which the leaves are marked with "true" or "false". If *all* leaves are marked with "true" the formula F is valid, otherwise F is not valid.

There are many implementations of the Davis-Putnam procedure, e.g. C-SAT, LDPP [Uribe and Stickel, 1994], POSIT [Freeman, 1995], SATO [Zhang, 1993], SATX [Li, 1996], TABLEAU [Crawford and Auton, 1993]. All these implementations have in common, that they require the formulae to be proven in clausal form. Since the usual translation into clausal form is based on the application of distributivity laws, this leads to an exponential increase of the resulting clausal form in the worst-case. Using a definitional translation [Eder, 1992; Plaisted and Greenbaum, 1986] yields (at most) a quadratic increase of the resulting formula's size at the expense of introducing new atomic formulae.

The aim of this paper is *not* to present another implementation of the Davis-Putnam procedure (although an implementation is given), but to propose a *non-clausal* version of this method. By generalizing this procedure to deal with arbitrary formulae we avoid any translation steps. Thus we do not only avoid to increase the size of the formula, but we also shorten the search tree considerably. Furthermore, the application of an additional split rule is possible which also reduce the search space.

Operating on a matrix representation of a formula, we will get a short and elegant description of the non-clausal proof procedure. In the following we will also show that such a procedure can be implemented in a very compact way and that this approach indeed has a good performance compared to a clausal-based procedure. The problems we have used to test the performance stem from deciding the validity of a formula F in intuitionistic propositional logic via a translation $\mathcal{T}$ into classical propositional logic. In this context the formula F is intuitionistically valid if and only if the translated (non-clausal) formula $\mathcal{T}(F)$ is classically valid.

This paper is organized as follows. In the next section the standard Davis-Putnam procedure is introduced. The non-clausal approach is described in section 3 and an implementation in Prolog is given in section 4. The performance compared to a clausal procedure is presented in section 5. We conclude with a summary of the results and a few remarks on further investigations.

*The author is supported by the Adolf Messer Stiftung

2 The Davis-Putnam Procedure

In this section the usual Davis-Putnam procedure is introduced and its technique is explained. In the next section we will generalize this technique to obtain a non-clausal procedure.

The Davis Putnam procedure requires the input formula to be in disjunctive normal form.[1]

Definition: A formula F is in *disjunctive normal form* (or *clausal form*), iff it is a disjunction of clause-formulae, i.e. of the form $F \equiv c_1 \vee c_2 \vee \ldots \vee c_n$. A *clause-formula* c_i is a conjunction of literals, i.e. of the form $c_i \equiv L_{i_1} \wedge L_{i_2} \wedge .. \wedge L_{i_{m_i}}$. A *literal* is either an atomic formula (positive literal) or its negation (negative literal).

To simplify the proof procedure as well as the notation of formulae, we use a matrix representation for formulae.

Definition: The *matrix* M of a formula F is the set of clauses of F, i.e. $M = \{C_1, C_2, \ldots, C_n\}$, where each *clause* C_i is a set of literals of the clause-formula c_i, i.e. $C_i = \{L_{i_1}, L_{i_2}, \ldots, L_{i_{m_i}}\}$. The *negation* $\bar{L}$ of a literal L is defined as $\bar{L} \equiv A$, if $L \equiv \neg A$ (for some atomic formula A), otherwise $\bar{L} \equiv \neg L$.

A clause is interpreted as the conjunction of its literals and a matrix as the disjunction of its clauses. Therefore a clause is *true*, iff *all* its literals are *true*. A matrix is *true*, iff *at least one* of its clauses is *true*. A clause/matrix which is not *true* is *false*. A matrix M is *valid*, iff the corresponding formula F is valid.

Lemma: The empty clause is *true*; a matrix containing the empty clause is *true*; the empty matrix is *false*.

```
input: matrix M representing a formula F in clausal
       form
output: true, if F is valid; false otherwise

 begin DP(M)
  if M = {} then return false;
  if {} ∈ M then return true;
  if {L} ∈ M
    then return DP(REDUCE_L̄(M));          /* UNIT */
  for all L ∈ lit_M with L̄ ∉ lit_M
    do M := REDUCE_L̄(M);                  /* PURE */
  select L ∈ lit_M;
  if DP(REDUCE_L(M))=true and             /* splitting */
     DP(REDUCE_L̄(M))=true
    then return true else return false;
 end DP.
```

Figure 1: The Davis-Putnam procedure DP

We can represent a matrix M as a graphical "matrix", if we place its clauses $C_1, \ldots, C_n$ side by side and the literals $L_{i_1}, \ldots, L_{i_{m_i}}$ of each clause C_i one upon the other (see figure 3).

The Davis-Putnam procedure is defined in figure 1 (where L denotes a literal and lit_M is the set containing all literals of M). Calling $\text{DP}(M)$ returns *true*, if the matrix M is valid, otherwise it returns *false*.

If the matrix M is an empty set or contains an empty clause, *false* and *true* are returned, respectively. Otherwise an literal occuring in M is selected and *true*/*false* is assigned to it respectively ("splitting"). Only if both resulting matrices are valid, the matrix M is valid.

```
begin REDUCE_L(M)
  M1:={C|C ∈ M and L̄ ∉ C};  /* clause elimination */
  M2:={C \ {L}|C ∈ M1};      /* literal deletion */
  return M2;
end REDUCE.
```

Figure 2: The Procedure REDUCE

Assigning *true* to all literals L (and *false* to $\bar{L}$) in the matrix M and returning the resulting simplified matrix is performed by the procedure $\text{REDUCE}_L(M)$ (see figure 2). This is done by the following two steps:

1. All clauses containing the negation $\bar{L}$ of L (and thus the truth value *false*) are deleted from M, since a clause (conjunction) containing a *false* literal is *false* and can be removed.
2. The literal L is deleted from all remaining clauses, since a *true* literal in a conjunction can be removed.

According to [2] we call the first step "*clause elimination*" and the second step "*literal deletion*". Both steps are illustrated in figure 3 (where L is *true* and $\bar{L}$ is *false*).

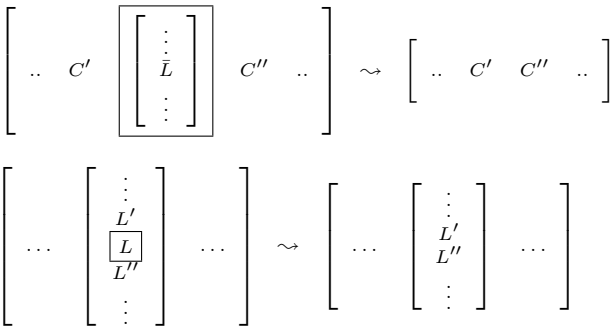

Figure 3: Clause Elimination and Literal Deletion

In the given procedure the *unit clause rule* (UNIT) and the *pure literal rule* (PURE) are additionally performed (see figure 1). The unit rule avoids the splitting of the matrix M, if there is a unit clause in M, i.e. a clause containing only one literal. The pure rule does delete all clauses containing the literal L from M, if the literal $\bar{L}$ does not occur in M. Both reductions are not necessary to obtain a correct and complete proof procedure, but they are good optimization techniques.

[1] Since we want to decide the *validity* of the given formula we use the "positive representation". Of course using conjunctive normal form and checking the satisfiability is also possible, since a formula F is valid iff $\neg F$ is *not* satisfiable.

3 A Non-Clausal Davis-Putnam Procedure

To apply the ideas of the last section to arbitrary formulae we have to generalize the corresponding definitions and concepts.

The syntax of (arbitrary) formulae is defined inductively by the following definition.

Definition: 1. An atomic formula A (i.e. a propositional variable) is a formula. 2. If F and G are formulae then $(\neg F)$, $(F \wedge G)$, $(F \vee G)$ and $(F \Rightarrow G)$ are also formulae, with the logical connectives "$\neg$" (negation), "$\wedge$" (conjunction), "$\vee$" (disjunction) and "$\Rightarrow$" (implication).[2]

Within the clausal-based Davis-Putnam procedure a formula in clausal form is represented by a matrix, which is a set of clauses. To get a similar representation for arbitrary formulae, we have to generalize the definition of a matrix, i.e. to consider *nested* matrices. For this reason we introduce the notion of *signed formulae*.

Definition: A *signed formula* is a tuple (F, p) consisting of a formula F and a *polarity* p, where $p \in \{0, 1\}$.

The matrix of a signed formula is defined inductively.

Definition: The *matrix* of a signed formula (F, p) is defined by the following table. $\{M_G\}$, $\{M_H\}$ and $\{M_G, M_H\}$ are called *clauses*.

(F, p)	matrix of (F, p)	M_G/M_H is the matrix of
$(A, 0)$, A is atomic	$\{\{A\}\}$	–/–
$(A, 1)$, A is atomic	$\{\{\neg A\}\}$	–/–
$((\neg G), p)$	M_G	$(G, 1-p)$ / –
$((G \wedge H), 1)$	$\{\{M_G\}, \{M_H\}\}$	$(G, 1)$ / $(H, 1)$
$((G \vee H), 0)$	$\{\{M_G\}, \{M_H\}\}$	$(G, 0)$ / $(H, 0)$
$((G \Rightarrow H), 0)$	$\{\{M_G\}, \{M_H\}\}$	$(G, 1)$ / $(H, 0)$
$((G \wedge H), 0)$	$\{\{M_G, M_H\}\}$	$(G, 0)$ / $(H, 0)$
$((G \vee H), 1)$	$\{\{M_G, M_H\}\}$	$(G, 1)$ / $(H, 1)$
$((G \Rightarrow H), 1)$	$\{\{M_G, M_H\}\}$	$(G, 0)$ / $(H, 1)$

Using the previous definition we are now able to define the matrix of (arbitrary) formulae.

Definition: The *matrix* of a formula F is the matrix of the signed formula $(F, 0)$.

Remark: Matrices of the form $M = \{\ldots, \{\{C_1, ..., C_n\}\}, \ldots\}$ can be simplified to $M' = \{\ldots, C_1, \ldots, C_n, \ldots\}$ where $C_1, .., C_n$ are clauses. Clauses of the form $C = \{\ldots, \{\{M_1, \ldots, M_m\}\}, \ldots\}$ can be simplified to $C' = \{\ldots, M_1, \ldots, M_m, \ldots\}$ where $M_1, .., M_m$ are matrices.

In the clausal-based Davis-Putnam procedure we regard a matrix as a set of clauses, where each clause is a set of literals. In our non-clausal approach a clause is a set of matrices, where each matrix is either a literal or a set of clauses. Again we can represent a matrix M as a "graphical" matrix, if we place its clauses side by side and the matrices of each clause one upon the other.

[2]Equivalence "$\Leftrightarrow$" can be defined as follows: $F \Leftrightarrow G$ iff $(F \Rightarrow G) \wedge (G \Rightarrow F)$.

Example: Consider the formula $F_a \equiv (((A \wedge (A \Rightarrow (B \wedge \neg B))) \vee (C \wedge D)) \Rightarrow (C \wedge D))$. The (simplified) matrix of F_a is given in figure 4.

$$\left[\begin{array}{cc} \left[\begin{array}{cc} \left[\begin{array}{c} [\neg A] \\ \left[\begin{array}{cc} [\neg C] & [\neg D] \end{array}\right] \end{array}\right] & \left[\begin{array}{c} A \\ \left[\begin{array}{cc} [\neg B] & [B] \end{array}\right] \end{array}\right] \end{array}\right] & \left[\begin{array}{c} C \\ D \end{array}\right] \end{array}\right]$$

Figure 4: The Matrix of F_a

Within the so-called *negational normal form*, a clause is interpreted as the *conjunction* of its matrices and a non-atomic matrix is interpreted as the *disjunction* of its clauses. Again a clause is *true*, iff *all* its elements are *true*. A matrix is *true*, iff *at least one* of its clauses is *true*. A clause/matrix which is not *true* is *false*. A matrix is *valid*, iff the corresponding formula F is valid.

Lemma: The empty matrix $\{\}$ is *false*, the empty clause $\{\}$ is *true*. A matrix $\{\ldots, \{\}, \ldots\}$ containing the empty clause is *true*. A clause $\{\ldots, \{\}, \ldots\}$ containing the empty matrix is *false*.

The non-clausal Davis-Putnam procedure is defined in figure 5 (where L denotes a literal and lit_M is the set containing all literals of M). Calling NCDP(M) returns *true*, if the matrix M is valid, otherwise it returns *false*.

```
input: matrix M representing an arbitrary formula F
output: true, if F is valid; false otherwise
 begin NCDP(M)
  if M = {} then return false;
  if {} ∈ M then return true;
  if {L} ∈ M
    then return NCDP(MREDUCE_L̄(M)); /* UNIT */
  for all L ∈ lit_M with L̄ ∉ lit_M
    do M :=MREDUCE_L̄(M);              /* PURE */
  if M={{M_1, .., M_n}, C_1, .., C_m} and n≥2
    then for all i ∈ {1, .., n}       /* beta- */
         do NCDP({{M_i}, C_1, .., C_m}); /* splitting */
  select L ∈ lit_M;
  if NCDP(MREDUCE_L(M))=true and /* splitting */
     NCDP(MREDUCE_L̄(M))=true
    then return true else return false;
 end NCDP.
```

Figure 5: The Non-Clausal Davis-Putnam Procedure

If the matrix M is an empty set or contains an empty clause, *false* and *true* are returned, respectively. Otherwise a literal L occuring in M is selected and *true*/*false* is assigned to it respectively("splitting"). Only if both resulting matrices are valid, the matrix M is valid.

Assigning *true* to all literals L (and *false* to $\bar{L}$) in the matrix M and returning the resulting simplified matrix is performed by the procedures $\text{MREDUCE}_L(M)$ and $\text{CREDUCE}_L(C)$ (see figure 6).[3]

$\text{MREDUCE}_L(M)$ performs this assignment for a matrix M. If the regarded matrix is the literal L or $\bar{L}$,

[3]Notice the similarity between these two procedures.

begin MREDUCE$_L(M)$
 if M=$\underline{L}$ then return $\{\{\}\}$; /* assign *true* */
 if M=$\bar{L}$ then return $\{\}$; /* assign *false* */
 if M is a literal then return M;
 $M_1 := \{C' | C' =$CREDUCE$_L(C)$ and $C \in M\}$;
 if $\{\} \in M_1$ then return $\{\{\}\}$; /* matrix elimination */
 return $\{C | C \in M_1$ and $C \neq \{\{\}\}\}$; /* simplify */
end MREDUCE.

begin CREDUCE$_L(C)$
 $C_1 := \{M' | M' =$MREDUCE$_L(M)$ and $M \in C\}$;
 if $\{\} \in C_1$ then return $\{\{\}\}$; /* clause elimination */
 return $\{M | M \in C_1$ and $M \neq \{\{\}\}\}$; /* simplify */
end CREDUCE.

Figure 6: The Procedures MREDUCE and CREDUCE

the truth-values *true* (i.e. $\{\{\}\}$) and *false* (i.e. $\{\}$) are returned respectively. Otherwise the assignments of its clauses are evaluated. If there is a *true* clause (i.e. $\{\}$), the whole matrix is deleted and *true* (i.e. $\{\{\}\}$) is returned. We call this step "*matrix elimination*". Notice that the *literal deletion* in the clausal-based procedure is a special case of the matrix elimination described here.

CREDUCE$_L(C)$ performs the assignment of *true* to the literal L for a clause C, i.e. the assignments of its matrices are evaluated. If there is a *false* matrix (i.e. $\{\}$), the whole clause is deleted and *false* (i.e. $\{\{\}\}$) is returned. Like in the clausal-based procedure we call this step "*clause elimination*".

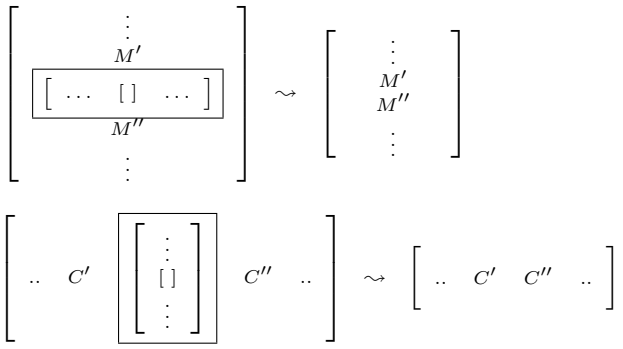

Figure 7: Matrix Elimination and Clause Elimination

Beside the usual unit clause rule and pure literal rule (see previous section) another splitting rule is applied. The *beta splitting rule* (see figure 8) is justified by the following corollary, which is not difficult to prove.

Lemma: A matrix $\{\{M_1, .., M_m\}, C_1, .., C_n\}$ is valid, iff $\{\{M_i\}, C_1, .., C_n\}$ is valid for all $i \in \{1, .., , m\}$.

4 An Implementation

The implementation of our non-clausal proof procedure is written in Prolog. This makes it possible to get a compact code which can easily be modified.

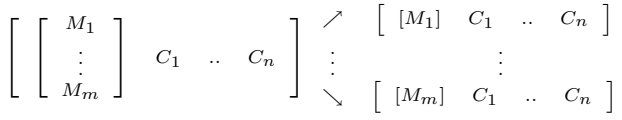

Figure 8: The Beta Splitting Rule

The sets describing matrices and clauses are represented as Prolog lists, where literals are represented as (possibly negated) atoms. For example the matrix $\{\{\neg A\}, \{A, \{\{\neg B\}, \{B\}\}\}\}$ is represented as the list `[[-A],[A,[[-B],[B]]]]`.

The predicate `dp(M)` actually implements our non-clausal proof procedure NCDP(M). It succeeds, if the matrix M is valid, otherwise it fails.

```
%%% dp (implements the non-clausal proof procedure)

dp([]) :- !, fail.              % matrix is NOT valid
dp(M)  :- member([],M), !.      % matrix is valid

dp(M)  :-                               % unit rule
   member([L],M), (atom(L),N= -L ; -N=L), !,
   reduce(M,N,L,M1), dp(M1).

dp([[[C1|M1],[C2|M2]|C]|M]) :- !,  % beta-splitting
   dp([[[C1|M1]]|M]), dp([[[C2|M2]]|M]), dp([C|M]).

dp(M)  :-                               % splitting
   selLit(M,L,N),
   reduce(M,L,N,M1), dp(M1),            % true -> L
   reduce(M,N,L,M2), dp(M2).            % false -> L
```

The following predicate `reduce(M,L,N,M1)` implements the procedure MREDUCE$_L(M)$ as well as the procedure CREDUCE$_L(C)$. Thus `M` can be a matrix as well as a clause. This predicate returns the simplified matrix/clause `M1` obtained by assigning *true* (i.e. `[[]]`) to each literal L (`L`) and *false* (i.e. `[]`) to the negation of L (`N`) in the matrix/clause `M`.[4]

```
%%% reduce (implements MReduce/CReduce procedure)

reduce(L,L,_,[[]])   :- !.          % assign true
reduce(N,_,N,[])     :- !.          % assign false

reduce([C|M],L,N,M1) :- !,
   reduce(C,L,N,C1),        % eval. clause/matrix
   ( C1=[] -> M1=[[]];      % mat./cl. elimination
     C1=[[]] -> reduce(M,L,N,M1);  % simplify
     reduce(M,L,N,M2),      % eval. clauses/matrices
   ( M2=[[]] -> M1=[[]];    % mat./cl. elimination
     M2=[],C1=[M3] -> M1=M3; M1=[C1|M2] )).

reduce(M,_,_,M).                    % M is a literal
```

The predicate `selLit(M,L,N)` performs the selection of a literal in the matrix `M`. The present implementation just returns the first literal `L` in `M` together with its negation `N`.

[4] At first it might be a bit confusing to apply the predicate `reduce` to matrices as well as to clauses. A look at the definitions of the corresponding procedures should make this technique clear, however.

```
%%% selLit (select a literal)

selLit([M|_],L,N) :- !, selLit(M,L,N).
selLit(-N,-N,N)   :- !.          % negative literal
selLit(L,L,-L).                  % positive literal
```

Notice that the described Prolog implementation is of course not as fast as an optimized C or Lisp implementation. As already mentioned our aim was not to implement a "state-of-the-art" prover, but to write a compact and readable implementation of our non-clausal proof procedure to allow further investigations (e.g. the comparison to a clausal-based Davis Putnam prover presented in the next section). It should be a minor problem to translate the given Prolog code into a functional or an imperative programming language.

5 Comparing Clausal and Non-Clausal Proof Procedure

In the following we will compare the performance of our non-clausal proof procedure to a clausal-based Davis-Putnam prover.

For this purpose we use the Prolog implementation of the previous section[5] and the Davis-Putnam prover of the KoMeT system [Bibel *et al.*, 1994]. KoMeT is also implemented in Prolog thus making the comparison fair. Moreover it is one of the few theorem provers providing not only the usual translation into clausal form, but also a definitional (or structure-preserving) translation [Eder, 1992; Plaisted and Greenbaum, 1986].

In the following tables the first column contains the name of the problem, the next two columns contain the times used by the Davis-Putnam Prover of KoMeT with the standard translation ("DPnormal") as well as the definitional translation ("DPdefini"), and the last column contains the time used by our own prover "ncDP". Times are measured on a SUN SPARC10 with ECLiPSe Prolog and are given in seconds, where ">600" means that no proof was found within 600 seconds.

We start with formulae which are in clausal form, namely the "complete formulae" com n (containing n distinct atomic formulae) and the "pigeonhole" formulae pigeon n (n+1 pigeon into n holes).

name	DPnormal	DPdefini	ncDP
com8	3.30	59.45	1.38
com9	7.81	244.67	4.05
com10	19.10	>600	11.73
pigeon4	0.75	1.68	0.53
pigeon5	5.80	5.58	4.00
pigeon6	80.10	26.28	33.21

The following formulae ft n are of the form $\neg\neg(p_1 \Leftrightarrow (p_2 \Leftrightarrow \ldots(p_{n-1} \Leftrightarrow p_n)..) \Leftrightarrow (..(p_1 \Leftrightarrow p_2) \Leftrightarrow p_3)\ldots \Leftrightarrow p_n)$, the formulae samp n are of the form $((p_1 \Rightarrow p_1) \wedge(p_2 \Rightarrow p_2)\wedge \ldots \wedge(p_n \Rightarrow p_n))$.

[5] We have to add a few lines of code which build up the matrix for a given formula. We also apply the pure literal rule once at the beginning of the proof process.

name	DPnormal	DPdefini	ncDP
ft6	31.69	0.72	0.43
ft8	>600	2.72	2.46
ft10	>600	13.60	13.63
samp10	64.36	2.34	<0.01
samp12	451.89	9.85	0.01
samp14	>600	46.52	0.02

In [Korn and Kreitz, 1997] the intuitionistic validity of a propositional formula F is decided by translating it into a formula $\mathcal{T}(F)$ which is classically valid, iff F is intuitionistically valid. The resulting formulae are strongly in non-clausal form. We have applied this translation to the (propositional) formulae in [Pelletier, 1986] to decide, whether they are intuitionistically valid.[6] The resulting formulae are ipell n where n is the problem's number w.r.t. [Pelletier, 1986].

The formulae dan1, dan2 and dan3 are the corresponding translations of the formulae $((a \Rightarrow b) \Rightarrow c)\wedge((d \Rightarrow e) \Rightarrow b)\wedge((g \Rightarrow h) \Rightarrow e) \Rightarrow c$, $((a \Rightarrow b) \Rightarrow c)\wedge((d \Rightarrow e) \Rightarrow b)\wedge((g \Rightarrow a) \Rightarrow e) \Rightarrow c$ and $((p \Leftrightarrow q) \Leftrightarrow r) \Leftrightarrow (p \Leftrightarrow (q \Leftrightarrow r))$, respectively.

The additional character after the name of each formula indicates if it is valid("t") or not("f").

name	DPnormal	DPdefini	ncDP
ipell1(f)	0.50	0.42	0.01
ipell4(f)	0.60	0.40	0.01
ipell10(t)	0.28	0.53	0.05
ipell12(f)	>600	81.18	2.29
ipell14(f)	1.88	0.60	<0.01
ipell17(f)	10.92	1.51	<0.01
ipell71a(t)	0.21	0.18	<0.01
ipell71b(f)	5.90	0.72	0.05
ipell72a(t)	0.28	0.77	0.11
ipell72b(t)	0.83	2.05	0.56
dan1(f)	>600	>600	0.86
dan2(t)	>600	>600	0.76
dan3(f)	>600	80.18	2.31

These experimental results show that the translation to clausal form often yields formulae which are almost impossible to prove, in particular in the case of the usual translation to clausal form.

There are three main advantages of the non-clausal proof procedure:

1. *Avoiding the translation into any clausal form.* This translation is sometimes not feasible or the resulting formula is too large to find a proof. If we use the definitional translation additional atomic formulae are introduced which increase the complexity of the problem.
2. *Application of matrix elimination steps.* This will lead to formulae which are smaller w.r.t. the corresponding clausal form. Consider, for example, the following matrix (where P, Q, R are literals and M is a matrix) and its partial clausal form:

$$\left[\left[\begin{array}{ccccc} [& [P] & [Q] & [R] &] \\ & & M & & \end{array}\right]\right] \qquad \left[\left[\begin{array}{c} P \\ M \end{array}\right] \left[\begin{array}{c} Q \\ M \end{array}\right] \left[\begin{array}{c} R \\ M \end{array}\right]\right]$$

In the clausal-based procedure the assignment of *true* to the literal R will *only* delete the literal R

[6] In the following table only the more difficult formulae are considered. For all other formulae the proof took less than 0.5 seconds for each prover.

(*literal deletion*, see section 2). In the non-clausal procedure the whole matrix [[P] [Q] [R]] will be deleted (*matrix elimination*, see section 3) which yields the matrix M. This means that some additional proof steps have to be performed which are not necessary in the non-clausal procedure.

3. *Application of the beta splitting rule*. Especially in the case of formulae which represent independent problems, this is an essential technique (see samp n examples above). "Beta splitting" can shorten proofs which otherwise would increase exponentially w.r.t. the length of the input formula, so that they will become linear.

6 Conclusion

We have presented a non-clausal decision procedure which is a generalization of the usual Davis-Putnam procedure. Due to the representation of formulae by matrices we get a compact description of the corresponding procedure. A Prolog implementation is provided showing that a non-clausal proof procedure can be implemented in a short and easy way. Due to the compact code the program can easily be modified and adapted for special purposes and applications.

We have compared our non-clausal proof procedure to a clausal-based Davis-Putnam procedure. To this end we have provided some experimental results showing that the usual translation into clausal-form as well as the definitional translation can spoil the proof process. Whereas the usual translation increases the size of the formulae considerably, the definitional translation introduces new atomic formulae. The compact representation of the non-clausal matrices is a competitive alternative. Operating on such matrices makes it possible to get truly shorter proofs, due to the application of *matrix elimination* steps and the *beta splitting rule*.

It will not make much sense to apply our procedure to problems which are already formulated in clausal form. For such problems (like n-Sat for $n \geq 3$) specialized (clausal-based) proof procedures are more suitable. For problems which are formulated in non-clausal form (like our translated formulae which are used to decide the validity in intuitionistic propositional logic) the non-clausal approach often is more useful.

Of course, there is still room for further research on this co-$\mathcal{NP}$ complete [Cook, 1971] problem. To improve the performance of our procedure it has to be implemented in a more machine-oriented programming language (like C). Furthermore there are many optimization techniques for clausal provers which might also be applicable to the non-clausal case (for example the selection of the "splitting literal"). On the other hand such a procedure should be compared to other (complete) proof methods, e.g. BDDs (see [Uribe and Stickel, 1994]).

Acknowledgments

I would like to thank Daniel Korn for providing me with his translation (and many non-clausal formulae) and for helping me to put the text into a readable form. I also would like to thank Thomas Rath for providing me with the KoMeT system.

References

[Bibel *et al.*, 1994] W. Bibel, S. Brüning, U. Egly, and T. Rath. Komet. In Alan Bundy, editor, *Proceedings of the 12^{th} CADE*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 783–787. Springer Verlag, Berlin, Heidelberg, New-York, 1994.

[Cook, 1971] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of 3^{rd} Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[Crawford and Auton, 1993] J. Crawford and L. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings 11th National Conference on AI*, pages 21–27, 1993.

[Davis and Putnam, 1960] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[Davis *et al.*, 1962] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5:394–397, 1962.

[Eder, 1992] E. Eder. *Relative Complexities of First Order Calculi*. Vieweg Verlag, 1992.

[Freeman, 1995] J. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[Korn and Kreitz, 1997] D. Korn and C. Kreitz. Efficiently deciding intuitionistic propositional logic via translation into classical logic. In *Proceedings of the 14^{th} CADE*, Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, Heidelberg, New-York, 1997.

[Li, 1996] C. Li. Exploiting yet more the power of unit clause propagation to solve the 3-sat problem. In *Proceedings ECAI'96 Workshop on Advances in Propositional Deduction*, pages 11–16, 1996.

[Pelletier, 1986] F. Pelletier. Seventy-five problems for testing automatic theorem proving. *Journal of Automated Reasoning*, 2:191–216, 1986.

[Plaisted and Greenbaum, 1986] D. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[Uribe and Stickel, 1994] T. Uribe and M. Stickel. Ordered binary decision diagrams and the davis-putnam procedure. In J.-P. Jouannaud, editor, *Proceedings 1st International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 34–49. Springer Verlag, Berlin, Heidelberg, New-York, 1994.

[Zhang, 1993] H. Zhang. A decision procedure for propositional logic. *Association for Automated Reasoning Newsletter*, 22:1–3, 1993.