Applying the Davis-Putnam procedure to non-clausal formulas *

Enrico Giunchiglia[1] and Roberto Sebastiani[2]

[1] DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
[2] DISA, Università di Trento, Via Inama, 5 – 38100 Trento, Italy

Abstract. Traditionally, the satisfiability problem for propositional logics deals with formulas in Conjunctive Normal Form (CNF). A typical way to deal with non-CNF formulas requires (i) converting them into CNF, and (ii) applying solvers usually based on the Davis-Putnam (DP) procedure. A well known problem of this solution is that the CNF conversion may introduce many new variables, thus greatly widening the space of assignments in which the DP procedure has to search in order to find solutions.
In this paper we present two variants of the DP procedure which overcome the problem outlined above. The idea underlying these variants is that splitting should occur only for the variables in the original formula. The CNF conversion methods employed ensure their correctness and completeness. As a consequence, we get two decision procedures for non-CNF formulas (i) which can exploit all the present and future sophisticated technology of current DP implementations, and (ii) whose space of assignments they have to search in, is limited in size by the number of variables in the original input formula.
In [11], it is showed that limiting the splitting step to a subset of the set of variables (the truth values of the others being consequentially determined) can lead to significant speeds up.

1 Introduction

In the last few years the satisfiability problem for CNF propositional formulas (SAT) has been widely investigated both theoretically and empirically and very efficient solvers have been developed (see e.g. [2]). Among the correct and complete solvers, the most popular are variants of the DP procedure [7, 6]. Highly sophisticated DP implementations include smart splitting heuristics and efficient data structures for manipulating CNF formulas (see, e.g., [9, 4, 22]).

* The ideas underlying this work have benefited from some discussions with Alessandro Armando, Alessandro Massarotto, David McAllester, Bart Selman, Armando Tacchella and Toby Walsh. Fausto Giunchiglia provided very useful feedback. Piergiorgio Bertoli and Luciano Serafini provided valuable help with the proofs of the theorems. All people mentioned above are thanked for their help.

On the other hand, many applications require decision procedures for *non-CNF* propositional formulas. For instance, the decision procedures for modal and terminological logics presented in [12, 13] —which proved to be empirically and theoretically superior to standard tableau systems [13, 10]— are built on top of non-CNF propositional procedures. In the field of digital hardware design, the correctness of a boolean circuit is showed by proving the equivalence between the propositional encodings of the circuit and its specification. In order to apply the DP procedure also in these applications, the standard solution is to pre-convert the formula to a CNF form using methods based on [15] (lately optimized in [8]).[1] The problem with this solution is that these conversions produce CNF formulas possibly containing many new variables. These new variables greatly widen the space of assignments in which the DP procedure has to search in order to find solutions.

In this paper we present two variants of the DP procedure which (when applied to CNF conversions of non-CNF formulas) overcome the problem outlined above. The idea underlying these variants —called DP* and DP**— is that splitting should occur only for the variables in the original formula. The CNF conversion methods employed ensure that the newly introduced variables get assigned either by unit propagation or by the pure literal rule. As a consequence, we get two correct and complete decision procedures for non-CNF formulas

1. which can exploit all the present and future sophisticated technology of current DP implementations, and
2. whose space of assignments they have to search in, is limited in size by the number of variables in the original input formula.

In [11], it is showed that limiting the splitting step to a subset of the set of variables (the truth values of the others being consequentially determined) can lead to significant speeds up (up to 4 orders of magnitude when using a time limit of 2700 seconds).

2 Preliminaries

We assume all the standard definitions and results of propositional logic. To simplify the presentation, we consider only formulas built from propositional literals – i.e., atoms and their negations – through finitely many applications of the connectives $\wedge, \vee, \equiv$. As customary in implementations, we allow conjunctions and disjunctions of an arbitrary number of formulas. Notationally, we use the capital letters $A_1, A_2, \ldots$ for propositional atoms; $l, l_1, \ldots$ for literals; $\Phi, \Phi_1, \ldots$ for CNF formulas; $\varphi, \varphi_1, \ldots$ for generic propositional formulas; and $\mu, \mu', \mu_1, \ldots$ for (partial) truth assignments. We use the symbols T and F as the logical constants for truth and falsity, and the same symbols denote also their respective truth values. If $l_i = \neg A_i$, $\neg l_i$ stands for A_i. We represent truth assignments as

[1] Actually, the methods in [15, 8] were proposed in the setting of first-order logic. We are simply considering their restriction to propositional logic.

	$\varphi = (\bigvee_{i=1}^{m} \varphi_i)$	$\varphi = (\bigwedge_{i=1}^{m} \varphi_i)$	$\varphi = (\varphi_1 \equiv \varphi_2)$
$ren_{\equiv}(\varphi)$	$(\neg B \vee \bigvee_{i=1}^{m} B_i) \wedge$ $(B \vee \neg B_1) \wedge$ $\ldots \wedge$ $(B \vee \neg B_m)$	$(B \vee \bigvee_{i=1}^{m} \neg B_i) \wedge$ $(\neg B \vee B_1) \wedge$ $\ldots \wedge$ $(\neg B \vee B_m)$	$(\neg B \vee B_1 \vee \neg B_2) \wedge$ $(\neg B \vee \neg B_1 \vee B_2) \wedge$ $(B \vee B_1 \vee B_2) \wedge$ $(B \vee \neg B_1 \vee \neg B_2)$
$ren_{\supset}(\varphi)$	$(\neg B \vee \bigvee_{i=1}^{m} B_i)$	$(\neg B \vee B_1) \wedge$ $\ldots \wedge$ $(\neg B \vee B_m)$	$(\neg B \vee B_1 \vee \neg B_2) \wedge$ $(\neg B \vee \neg B_1 \vee B_2)$

Table 1. Definition of $ren_{\equiv}(\varphi)$ and $ren_{\supset}(\varphi)$.

sets of literals like $\{A_1, \neg A_2, \ldots\}$, with the intended meaning $\{A_1 := \mathsf{T}, A_2 := \mathsf{F}, \ldots\}$.

By $assign(l, \varphi)$ we denote the formula obtained

1. substituting every occurrence of l $[\neg l]$ in φ with T $[\mathsf{F}]$, and
2. recursively evaluating the result according to the standard propositional simplification rules ($\varphi \vee \mathsf{T} \Rightarrow \mathsf{T}$, $\varphi \wedge \mathsf{T} \Rightarrow \varphi$, $\varphi \equiv \mathsf{T} \Rightarrow \varphi$, etc.).

If $\mu = \{l_1, \ldots, l_k\}$, then $assign(\mu, \varphi)$ denotes the formula

$$assign(l_k, assign(\ldots, assign(l_1, \varphi) \ldots)). \tag{1}$$

It is easy to check that for any two literals l_1 and l_2 with $l_1 \neq l_2$, and for any formula φ,

$$assign(l_1, assign(l_2, \varphi)) = assign(l_2, assign(l_1, \varphi))$$

so that the specific ordering used in (1) does not matter.

For any assignment μ and formula φ, we say that

- μ *evaluates* φ *to* T $[\mathsf{F}]$ if $assign(\mu, \varphi)$ returns T $[\mathsf{F}]$, and
- μ *evaluates* φ if μ evaluates φ either to T or to F.

Finally, the *size* $|\varphi|$ of a formula φ is the number of atoms occurring in it.

2.1 CNF conversions

Consider a propositional formula φ. φ may be converted into CNF by the "classical" method: First, φ is converted into negative normal form. Second, the rule

$$(\bigwedge_i \bigvee_j \varphi_{ij}) \vee (\bigwedge_k \bigvee_m \varphi_{km}) \Longrightarrow \bigwedge_{i,k} (\bigvee_j \varphi_{ij} \vee \bigvee_m \varphi_{km}).$$

is recursively applied to distribute $\wedge$'s over $\vee$'s. Notationally, we will call cnf the function implementing such a conversion. As result of this conversion, we get a formula Φ which is logically equivalent to φ. However, this transformation may lead to a considerable increase in the size of the formula (in the worst case, $|\Phi|$ is $O(2^{|\varphi|})$), which makes the method of no practical utility in many cases.

A more convenient way to convert φ into CNF is based on the idea of renaming the subformulas of φ (see e.g. [20, 15]). In this method, a newly introduced atom B_{φ_i} is associated to each non-literal subformula φ_i of φ. Then, each B_{φ_i} substitutes every occurrence of φ_i inside φ, and the expression $(B_{\varphi_i} \equiv \varphi_i)$ is added to the result. In the rest of the paper, given a formula φ we will use B_φ to represent

- a new atom (the atom associated to φ), if φ is not a literal; and
- φ, otherwise.

For sake of readability we will abbreviate B_φ and B_{φ_i} with B and B_i respectively. One straightforward implementation of this method is given by the function ct^*, which is described by the recursive equation:

$$ct^*(\varphi) := \begin{cases} \top & \text{if } \varphi \text{ is a literal,} \\ (\bigwedge_{i=1}^m ct^*(\varphi_i) \ \wedge \ ren_\equiv(\varphi)) & \text{if } \varphi = (\bigvee_{i=1}^m \varphi_i) \text{ or } \varphi = (\bigwedge_{i=1}^m \varphi_i), \\ (\bigwedge_{i=1}^2 ct^*(\varphi_i) \ \wedge \ ren_\equiv(\varphi)) & \text{if } \varphi = (\varphi_1 \equiv \varphi_2), \end{cases}$$

where $ren_\equiv(\varphi)$ is defined in Table 1. It is easy to show that the formula $(B \wedge ct^*(\varphi))$ is a CNF formula which is satisfiable if and only if φ is satisfiable.

A variant of the latter method has been proposed in [15] and lately optimized in [8]. If a formula φ_i is not a literal, and is not a subformula of an equivalence in φ, then we can substitute B_i for φ_i in φ and add the expression $(\neg B_i \vee \varphi_i)$ to the result. A straightforward implementation of this method is given by the function ct^{**} described by the following recursive equation:

$$ct^{**}(\varphi) := \begin{cases} \top & \text{if } \varphi \text{ is a literal,} \\ (\bigwedge_{i=1}^m ct^{**}(\varphi_i) \ \wedge \ ren_\supset(\varphi)) & \text{if } \varphi = (\bigvee_{i=1}^m \varphi_i) \text{ or } \varphi = (\bigwedge_{i=1}^m \varphi_i), \\ (\bigwedge_{i=1}^2 ct^*(\varphi_i) \ \wedge \ ren_\supset(\varphi)) & \text{if } \varphi = (\varphi_1 \equiv \varphi_2), \end{cases}$$

where $ren_\supset(\varphi)$ is defined in Table 1. The above transformation is equivalent to that described in [15]. In that paper, the authors show that $(B \wedge ct^{**}(\varphi))$ is a CNF formula which is satisfiable if and only if φ is satisfiable.

Notice that both $|ct^*(\varphi))|$ and $|ct^{**}(\varphi))|$ are $O(|\varphi|)$. However, ct^* generally returns bigger formulas than ct^{**}. For instance, if $\varphi = (\bigvee_{i=1}^m \varphi_i)$, then $ren(B \equiv \varphi)$ contains one $m+1$-ary clause plus m binary clauses, while $ren(\neg B \vee \varphi)$ contains only one $m+1$-ary clause (see Table 1).

2.2 DP

There are many variants of the Davis-Putnam [7, 6] procedure, each variant differing for the set of rules implemented. Figure 1 is an high-level description of the Davis-Putnam procedure in its minimal form. In one of its most popular variants, an extra step after the unit step and before the split step is added:[2]

[2] A literal l occurs *only positively* in a CNF formula Φ if l occurs in Φ while $\neg l$ does not.

```
function DP(Φ, μ)
    if Φ = T then return μ;                              /* base      */
    if Φ = F then return False;                          /* backtrack */
    if {a unit clause (l) occurs in Φ}                   /* unit      */
        then return DP(assign(l, Φ), μ ∪ {l});
    l := choose-literal(Φ, μ);                           /* split     */
    return   DP(assign(l, Φ), μ ∪ {l})   or
             DP(assign(¬l, Φ), μ ∪ {¬l});
```

Fig. 1. Schema of the DP procedure.

```
    if {l occurs only positively in Φ}                   /* pure literal */
        then return DP(assign(l, Φ), μ ∪ {l});
```

Independently from whether the pure literal rule is implemented or not, the procedure has to be invoked by the call $DP(\Phi, \{\})$, where Φ is the CNF formula to be proved (un)satisfiable and $\{\}$ is the empty assignment. $DP(\Phi, \{\})$ returns an assignment μ (satisfying Φ) if Φ is satisfiable, and *False* otherwise. If N is the number of distinct atoms in Φ, DP searches in a space of 2^N assignments. Notice that the number of atoms N is thus much more critical than the size $|\Phi|$ of the formula. A more fine-grained analysis can be found, e.g., in [22].

3 Applying DP to non-CNF formulas

Suppose we want to check the satisfiability of a non-CNF formula φ in the variables $A_1, \ldots, A_N$. Before applying DP, we must pre-convert φ by either ct^* or ct^{**}, obtaining a CNF formula Φ in the original variables $A_1, \ldots, A_N$ and in a set of K newly added variables. Therefore standard DP will have to search in a space of 2^{N+K} assignments, for it may backtrack on the newly added variables.

In this section we propose an alternative approach. We present two ad-hoc variants of DP, called DP* and DP**, which perform no backtracking on the newly added variables, eliminating thus the 2^K factor. The underlying idea of these variants is that splitting should occur only for the variables in the original non-CNF formula.

3.1 DP*

Consider a formula φ over $A_1, \ldots, A_N$. The intuition underlying ct^* is that each new variable B_i labels a subformula φ_i of φ, so that it should be assigned the same truth value as φ_i is. As the truth value to which a subformula φ_i is evaluated is a deterministic consequence of the values assigned to $A_1, \ldots, A_N$, the value assigned to B_i should be deterministically derived from those assigned to $A_1, \ldots, A_N$ too. This is achieved by the occurrence of the conjunct $ren_{\equiv}(\varphi_i)$

in $ct^*(\varphi)$, which forces the truth value of B_i to be *deterministically derived* from that of φ_i, and thus, recursively, from the truth values of $A_1, \ldots, A_N$. This suggests that no backtracking should be performed on the newly added variables.

Following these considerations, we introduce here DP*, a variant of DP which exploits variable dependency to avoid extra search over dependent variables. In this section, given a CNF formula Φ, the *independent* variables of Φ are the variables in

- φ, if $\Phi = (B \wedge ct^*(\varphi))$ for some formula φ,
- Φ otherwise.

We will call *dependent* the variables in Φ which are not independent.

Definition 1. Let Φ be a CNF formula. DP$^*(\Phi, \mu)$ is the procedure in Figure 1 in which *choose-literal*(Φ, μ)

- returns an unassigned literal whose atom is among the *independent* variables of Φ, if there is some, and
- loops forever, otherwise.

This means that DP* cannot branch on the dependent variables in the input formula Φ, so that these can be assigned only by unit propagation. We show that DP$^*(\Phi, \{\})$ returns an assignment if Φ is satisfiable, F otherwise. To prove this fact we need to show that DP$^*(\Phi, \{\})$ terminates. In the following lemma, we say that DP$^*(\Phi, \mu)$ *returns* $\mu \cup \mu'$ *by unit propagation* meaning that DP$^*(\Phi, \mu)$ returns $\mu \cup \mu'$ after a sequence eventually empty of unit steps.

Lemma 2. *Let φ be a formula in the variables $A_1, \ldots, A_N$. Let μ be a total assignment over $A_1, \ldots, A_N$ which evaluates φ to T [F]. DP$^*(assign(\mu, \mathrm{ct}^*(\varphi)), \mu)$ returns an assignment $\mu \cup \mu'$ by unit propagation, such that $\mu \cup \mu'$ evaluates B to T [F].*

Proof. By induction on the structure of φ. In the base case, φ is a literal, $ct^*(\varphi)$ is T and the thesis trivially follows. All the step cases are similar. We elaborate the case in which $\varphi = (\bigvee_{i=1}^{m} \varphi_i)$. In this case

$$ct^*(\varphi) = \bigwedge_{i=1}^{m} ct^*(\varphi_i) \wedge (\neg B \vee \bigvee_{i=1}^{m} B_i) \wedge \bigwedge_{i=1}^{m} (B \vee \neg B_i).$$

By induction hypothesis, for each $i \in \{1, \ldots, m\}$, DP$^*(assign(\mu, ct^*(\varphi_i)), \mu)$ returns an assignment $\mu \cup \mu_i$ by unit propagation which evaluates

- $ct^*(\varphi_i)$ to T $(1 \leq i \leq m)$ by unit propagation,
- B_i to T [F],

(notice that B does not occur in $ct^*(\varphi_i)$, and thus B does not get evaluated by μ_i). Assume that μ evaluates φ to T [F], i.e. that for some [each] $i \in \{1, \ldots, m\}$,

μ evaluates φ_i to T [F]. Then, $\mathrm{DP}^*(assign(\mu, ct^*(\varphi)), \mu)$ returns by unit propagation the assignment $\mu \cup_{i=1}^m \mu_i \cup \{B\}$ [$\mu \cup_{i=1}^m \mu_i \cup \{\neg B\}$].[3]

Theorem 3. *Let φ be an arbitrary formula. Let Φ be $(B \wedge \mathrm{ct}^*(\varphi))$. $DP^*(\Phi, \{\})$ terminates, returning an assignment if φ is satisfiable, $False$ otherwise.*

Proof. $\mathrm{DP}^*(\Phi, \{\})$, in each branch of its search tree,

- either terminates (i.e. returns an assignment or $False$) before generating a complete assignment over the independent variables of Φ,
- or generates such an assignment, after which it will terminate by Lemma 2.

Obviously, the result returned by $\mathrm{DP}^*(\Phi, \{\})$ is correct.

3.2 DP**

We introduce here DP^{**}, a further variant of DP^* which is suitable for working with CNF formulas obtained using ct^{**}. Analogously to the previous section, given a CNF formula Φ, the *independent* variables of Φ are the variables in

- φ, if $\Phi = (B \wedge ct^{**}(\varphi))$ for some formula φ,
- Φ otherwise.

We will call *dependent* the variables in Φ which are not independent.

Definition 4. Let Φ be a CNF formula. $\mathrm{DP}^{**}(\Phi, \mu)$ is the procedure in Figure 1 in which

- the pure literal rule is added,[4] and
- *choose-literal*(Φ, μ)
  - returns an unassigned literal whose atom is among the *independent* variables of Φ, if there is some, and
  - loops forever, otherwise.

Similarly to DP^*, DP^{**} exploits variable dependency to avoid extra search over dependent variables. Roughly speaking, the usage of pure literal in DP^{**} compensates for the one-way implication of $ren_{\supset}$ in ct^{**}.

Again, we need to show that $\mathrm{DP}^{**}(\Phi, \{\})$ terminates. In the following lemma, we say that $\mathrm{DP}^{**}(\Phi, \mu)$

- *assigns l by unit propagation* meaning that $\mathrm{DP}^{**}(\Phi, \mu)$ assigns l to T after a sequence (eventually empty) of unit steps.

[3] This relies on the fact that for any disjunct φ_i of φ, any atom assigned by $\mathrm{DP}^*(assign(\mu, ct^*(\varphi_i)), \mu)$ is assigned in the same way by $\mathrm{DP}^*(assign(\mu, ct^*(\varphi)), \mu)$. Why this is indeed the case, is an easy consequence of the fact that, for any assignment μ' extending μ, the set of unit clauses in $assign(\mu', ct^*(\varphi_i))$ is a subset of the set of unit clauses in $assign(\mu', ct^*(\varphi))$.

[4] Actually, for proving the following Lemma, we may assume that the pure literal rule be applicable only to assign the newly introduced atoms to T.

– *returns* $\mu \cup \mu'$ *without splitting* meaning that $\mu \cup \mu'$ is a possible assignment returned by $\mathrm{DP}^{**}(\Phi, \mu)$ after a sequence (eventually empty) of unit and/or pure literal steps.

Lemma 5. *Let φ be a formula in the variables $A_1, \ldots, A_N$. Let μ be a total assignment over $A_1, \ldots, A_N$.*

– *If μ evaluates φ to F then $DP^{**}(assign(\mu, \mathrm{ct}^{**}(\varphi)), \mu)$ assigns $\neg B$ by unit propagation and returns an assignment $\mu \cup \mu'$ without splitting.*
– *If μ evaluates φ to T then $DP^{**}(assign(\mu, \mathrm{ct}^{**}(\varphi)), \mu)$ returns an assignment $\mu \cup \mu'$ without splitting, such that $\mu' \setminus \mu$ does not evaluate B.*

Proof. By induction on the structure of φ. In the base case, φ is a literal, $ct^{**}(\varphi)$ is T and the thesis trivially follows. As in Lemma 2, we elaborate only the case in which $\varphi = (\bigvee_{i=1}^{m} \varphi_i)$. The case $\varphi = (\bigwedge_{i=1}^{m} \varphi_i)$ is similar, while the case $\varphi = (\varphi_1 \equiv \varphi_2)$ —given Lemma 2— is simple. In this case

$$ct^{**}(\varphi) = \bigwedge_{i=1}^{m} ct^{**}(\varphi_i) \wedge (\neg B \vee \bigvee_{i=1}^{m} B_i).$$

There are two possibilities:

1. μ evaluates φ to T, i.e., μ evaluates to T a nonempty subset S of $\{\varphi_1, \ldots, \varphi_m\}$. By induction hypothesis, for each disjunct φ_i of φ and not in S, the call $\mathrm{DP}^{**}(assign(\mu, ct^{**}(\varphi_i)), \mu)$ returns without splitting an assignment $\mu \cup \mu_i$. Now consider a formula φ_i in S. $\mathrm{DP}^{**}(assign(\mu, ct^{**}(\varphi_i)), \mu)$ by induction hypothesis returns without splitting an assignment $\mu \cup \mu_i$ such that $\mu_i \setminus \mu$ does not evaluate B_i. There are two cases:
   – either B_i gets evaluated by μ. This means that $B_i = \varphi_i$ and thus B_i is evaluated to T;
   – or B_i is not evaluated by μ. This means that $\mu \cup \mu_i$ does not evaluate B_i and evaluates $ren_{\supset}(\varphi_i)$ to T. As a consequence, if φ_i is a subformula of an equivalence in φ, (i.e. if $ren_{\equiv}(\varphi_i)$ is a subformula of $ct^{**}(\varphi)$) then B_i occurs as a unit clause in $assign(\mu \cup \mu_i, ct^{**}(\varphi))$; otherwise B_i occurs only positively in $assign(\mu \cup \mu_i, ct^{**}(\varphi))$. In any case, $\mathrm{DP}^{**}(assign(\mu \cup \mu_i, ct^{**}(\varphi)), \mu \cup \mu_i)$ may assign B_i to T.

   In both cases, $\mathrm{DP}^{**}(assign(\mu, ct^{**}(\varphi)), \mu)$ returns without splitting the assignment $\mu \cup_{i=1}^{m} \mu_i \cup_{\varphi_i \in S} \{B_i\}$. This assignment does not evaluate B.[5]

[5] This relies on the fact that for any disjunct φ_i of φ, any atom assigned by $\mathrm{DP}^{**}(assign(\mu, ct^{**}(\varphi_i)), \mu)$ may be assigned in the same way by $\mathrm{DP}^{**}(assign(\mu, ct^{**}(\varphi)), \mu)$. In fact, let μ' be an assignment extending μ, and let B_j be an atom occurring only positively in $assign(\mu', ct^{**}(\varphi_i))$.
– If $ren_{\equiv}(\varphi_j)$ is a subformula of $ct^{**}(\varphi)$ then B_j occurs as a unit clause in $assign(\mu', ct^{**}(\varphi))$,
– otherwise, B_j occurs only positively in $assign(\mu', ct^{**}(\varphi))$.
See also Footnote 3 in Lemma 2.

2. μ evaluates φ to F. Hence, μ evaluates to F also $\varphi_1, \ldots, \varphi_m$. By induction hypothesis, for each $i \in \{1, \ldots, m\}$, DP$^{**}(assign(\mu, ct^{**}(\varphi_i)), \mu)$ assigns $\neg B_i$ by unit propagation and returns without splitting an assignment $\mu \cup \mu_i$. Then, DP$^{**}(assign(\mu, ct^{**}(\varphi)), \mu)$
   - assigns $\neg B$ by unit propagation, and
   - returns without splitting the assignment $\mu \cup_{i=1}^{m} \mu_i \cup \{\neg B\}$.

Theorem 6. *Let φ be an arbitrary formula. Let Φ be $(B \wedge \mathrm{ct}^{**}(\varphi))$. DP$^{**}(\Phi, \{\})$ terminates, returning an assignment if φ is satisfiable, $False$ otherwise.*

Proof. As in Theorem 3, the essential fact is to guarantee that DP$^{**}(\Phi, \{\})$ terminates, which is an obvious consequence of Lemma 5.

4 Related work

Armando and Giunchiglia [1] introduced PTAUT, a variant of DP working directly with non-CNF formulas. The procedure is based on a method to detect "unit clauses" for non-CNF formulas φ (roughly speaking, l is a "unit clause" for φ if φ has the form $(l \wedge \varphi_1)$). *assign* is implemented by lazy evaluation. PTAUT has been used inside the first implementations of the procedure KSAT [12, 13, 10]. Nevertheless, PTAUT efficiency is far below the current implementations of CNF DP. For instance, so far we do not know of any non-CNF DP implementation exploiting fast unit-propagation and inexpensive backtracking techniques, like those used by TABLEAU [4] or SATO [22].

D'Agostino and Mondadori [5] introduced an evolution of the propositional tableau framework, called KE, and a decision procedure for non-CNF formulas based on KE. Unlike tableau, it branches on truth values rather than on disjunctions. Unlike DP, it assigns truth values to subformulas rather than to variables. We are not aware of any efficient implementation of this procedure available.

In the area of digital hardware design, the solvers traditionally used are based on Ordered Binary Decision Diagrams (OBDD) [3]. Though these solvers may require an exponential amount of space, they proved to very highly effective in this area. Interestingly, Uribe and Stickel [21] presented and discussed the results of an empirical comparison between an OBDD based solver and various DP implementations. While none of the two procedures was proved to be definitely superior to the other in general, OBDD outperformed in time DP over formulas encoding boolean circuits. These are the only formulas, in the test set, which are not in CNF, and thus which had to be pre-converted into CNF before applying DP procedures. According to [21], one of the causes of DP implementations poor performances is that in the CNF conversion, the difference between the original and the newly added variables is lost. They suggest that a good branching heuristic for DP implementations would probably have it distinguish them again. However, they do not push this idea any further.

Considering correct but incomplete SAT procedures, Sebastiani [16] suggested how to modify GSAT [17] to be applied to non-CNF formulas. The idea was to use a particular "score" function returning the number of clauses in $cnf(\varphi)$ which are false under a given truth assignment. This value is computed from φ in linear time, without constructing $cnf(\varphi)$. This method was not implemented. Kautz, McAllester and Selman [14] implemented an extension of WALKSAT [18] (called DAGSAT) in which search concentrates only on the independent variables. They present a comparative analysis between WALKSAT and DAGSAT, and show that on 7 out of the 8 problems they consider, DAGSAT performs better than WALKSAT.

Finally, we have already showed in [11] that limiting the splitting step in DP implementations to a subset of the set of variables (the truth values of the others being assigned by unit propagation, analogously to what happens in ct^*+DP*) can lead to significant speeds up. In particular, we considered 114 SAT problems, and on each of them we run both the TABLEAU system and TABLEAU*, a version of TABLEAU modified in order to split only over the "independent" variables. The results show that:

- The search space effectively searched by TABLEAU* (i.e. the number of nodes in the search tree) is up to 4 orders of magnitude less than the search space analyzed by TABLEAU.
- The CPU time requested by TABLEAU* to solve a problem is up to 4 orders of magnitude less than the CPU time requested by TABLEAU on the same problem.

References

1. A. Armando and E. Giunchiglia. Embedding Complex Decision Procedures inside an Interactive Theorem Prover. *Annals of Mathematics and Artificial Intelligence*, 8(3–4):475–502, 1993.
2. *Artificial Intelligence*, 81(1,2), 1996. Special Volume on Frontiers in Probelm Solving: Phase Transitions and Complexity.
3. R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
4. J. Crawford and L. Auton. Experimental results on the crossover point in 3SAT. *Artificial Intelligence*, 81, 1996.
5. M. D'Agostino and M. Mondadori. The Taming of the Cut. *Journal of Logic and Computation*, 4(3):285–319, 1994.
6. M. Davis, G. Longemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7), 1962.
7. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
8. T. Boy de la Tour. Minimizing the Number of Clauses by Renaming. In *Proc. CADE-90*, pages 558–572. Springer-Verlag, 1990.
9. DIMACS. *The Second DIMACS International Algorithm Implementation Challenge*, Rutgers University, USA, 1993.
10. E. Giunchiglia, F. Giunchiglia, R. Sebastiani, and A. Tacchella. More evaluation of decision procedures for modal logics. In *Proc. KR'98*, 1998.
11. E. Giunchiglia, A. Massarotto, and R. Sebastiani. Act, and the rest will follow: Exploiting determinism in planning as satisfiability. In *Proc. AAAI-98*, 1998.

12. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures - the case study of modal K. In *Proc. CADE-96*, Lecture Notes in Artificial Intelligence. Springer Verlag.
13. F. Giunchiglia and R. Sebastiani. A SAT-based decision procedure for ALC. In *Proc. KR'96*, Cambridge, MA, USA, November 1996.
14. Henry Kautz, David McAllester, and Bart Selman. Exploiting variable dependency in local search. In *Abstracts of the Poster Sessions of IJCAI-97*, August23–29 1997. Available at `http://www.research.att.com/~kautz/papers-ftp/index.html`.
15. D.A. Plaisted and S. Greenbaum. A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation*, 2:293–304, 1986.
16. R. Sebastiani. Applying GSAT to Non-Clausal Formulas. *Journal of Artificial Intelligence Research*, 1:309–314, 1994.
17. B. Selman, H. Levesque., and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proc. AAAI-92*, pages 440–446, 1992.
18. Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proc. AAAI-94*, pages 337–343. AAAI Press.
19. Jörg Siekmann and Graham Wrightson, editors. *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 2. Springer-Verlag, 1983.
20. G. Tseitin. On the complexity of proofs in propositional logics. *Seminars in Mathematics*, 8, 1970. Reprinted in [19].
21. T. E. Uribe and M. E. Stickel. Ordered Binary Decision Diagrams and the Davis-Putnam Procedure. In *Proc. of the 1st International Conference on Constraints in Computational Logics*, 1994.
22. H. Zhang and M. Stickel. Implementing the Davis-Putnam algorithm by tries. Technical report, University of Iowa, August 1994.