# Davis and Putnam 40 years later: a first experimentation.

Philippe Chatalic and Laurent Simon
Laboratoire de Recherche en Informatique,
U.M.R. CNRS 8623,
Université Paris-Sud, 91405 Orsay Cedex, France
{*chatalic, simon*}*@lri.fr*

**Abstract**

This paper presents an experimental study of the very popular Davis and Putnam procedure, as stated in its original version of 1960. Though widely acknowledged in the literature as inefficient, it seems that no real experimental study has been conducted to test this procedure in a practical way and to evaluate its limits. This study shows that despite the fact that it adresses a much more complex problem than the backtrack search procedure with the same name, applying this procedure on several benchmark examples of the DIMACS base gives surprising results, sometimes surpassing many specialized algorithms for SAT. Several interesting characteristics of this approach are also pointed out, thanks to an extensive experimental study of this algorithm, on random and on structured instances.

**Keywords:** Satisfiability testing, Davis and Putnam algorithm, Experimental study

**Résumé**

Cet article présente une étude expérimentale sur la procédure de Davis et Putnam dans sa version initiale de 1960. Un consensus semble s'être installé dans la littérature sur son inefficacité. Pourtant, personne ne l'a semble-t-il véritablement testée de manière pratique, ni cherché à évaluer ses limites. Cette étude montre que, bien que répondant à un problème bien plus complexe que la procédure du même nom avec retour arrière, elle donne des résultats surprenants sur certains exemples de la base de DIMACS, dépassant les performances de beaucoup d'algorithmes spécialisés pour SAT. Certaines caractéristiques propres à cette approche sont également mises en évidence par le biais d'une étude approfondie de son comportement sur des instances aléatoires.

**Mots clés :** Satisfiabilité, Algorithme de Davis et Putnam, Étude expérimentale

# 1 Introduction

In 1960, Davis and Putnam [3] proposed an algorithm for proving the unsatisfiability of propositional formulas. Two years later, a modified version of this algorithm was presented by Davis, Logemann and Loveland [9]. The last decade has seen a number of important works about the SAT problem, from a theoretical perspective [14] as well as from an experimental point of view [11, 6]. Stimulated by hardware progress, many works have led to continuous improvements in problem sizes. It is striking that most complete algorithms are based on the 1962 procedure [9]. The two versions ([3, 9]) are however based on very different principles. Although they are both based on resolution, the 1960 version consists in a sequence of saturations in order to eliminate the propositional variables one by one, while the second one corresponds to a backtrack search algorithm associated with a unit propagation mechanism. The latter being far less expensive in memory, the power of computers available at that time probably explains why the former method was rapidly left aside. Moreover, this process has been accentuated by a frequent confusion in the literature, where the procedure of Davis, Logemann and Loveland is often called "David and Putnam" procedure. In the following, we refer to the initial version of the algorithm by DP and to the second version by DLL.

If the initial version seems to be more expensive in solving the SAT problem, it is not for all that uninteresting. As a matter of fact, the data managed by the procedure may be used to reconstruct *all* models of a formula. It is also possible to limit the number of variables eliminated by DP to infer constraints on the remaining variables. This proves useful for the validation or compilation of knowledge bases ([10, 13]).

However, except the work of Dechter and Rish [5] and that of Hery and Laleuf [10], it seems that nobody has really studied this procedure and that no serious experimental study has been conducted to evaluate the characteristics and the limits of this approach.

This paper is an account for such an experiment. The next section recalls the basic principles of DP and DLL algorithms and describes the major choices for CalcRes, our implementation of the DP algorithm. The third part presents the results of our tests on random instances as well as on standard benchmarks of the SAT literature. Some surprising results show that, although much more complex than DLL, the DP algorithm is worth more consideration.

# 2 DP based algorithms

In the following, we use usual notations for representing propositional formulas built up a set of *propositional variables* $P$, $Q$, $\dots$ , with the usual

$DP(\varphi, \Omega)$:
**if** $\varphi = \Box$
**then return** $\varphi$
**else if** $\Omega = \emptyset$
  **then return** $\varphi$
  **else let** $P$ *be a variable of* $\Omega$
    **in return** $DP((\varphi \wedge \neg P) \vee (\varphi \wedge P), \Omega \setminus \{P\})$

Figure 1: Davis and Putnam initial algorithm (DP)

connectors $\neg$, $\wedge$ and $\vee$. Recall that *clauses* correspond to disjunctions of *literals* ($P$ or $\neg P$). They will be represented by Greek letters $\alpha, \beta, \ldots$, or by $\Box$ for the empty clause. A clause $\alpha$ is said to be *subsumed* by another clause $\beta$ if all literals of $\beta$ belong to $\alpha$. A formula is said to be in *conjunctive normal form* (cnf) if it is a conjunction of clauses. For convenience, we sometimes do not make any distinction between conjunctions and sets of clauses. All algorithms presented in this section are supposed to work on a cnf formula. *Resolution* is the operation which, to two clauses $(\alpha \vee P)$ and $(\beta \vee \neg P)$, associates the clause $(\alpha \vee \beta)$. We call *cut* the operation which consists in performing all possible resolutions on $P$. When this operation is followed by the elimination of all clauses containing the variable $P$, we call it *cut elimination*. A *unit clause* is a clause reduced to a single literal. Performing a cut elimination on a variable $P$ occurring in a unit clause of $\varphi$ is called *unit propagation* and the resulting formula is denoted by $UnitProp(\varphi, P)$.

## 2.1 DP and DLL Algorithms

In its initial version of 1960 [3], the DP algorithm allows us to determine whether a formula $\varphi$ is satisfiable or not. Let $\Omega$ be the set of propositional variables of $\varphi$. DP's principle amounts to a succession of cut eliminations, of all variables of $\Omega$.

Note that when all variables have been eliminated, the returned value is either the empty clause (which means that $\varphi$ is unsatisfiable) either the tautology (i.e. an empty set of clauses). A first difficulty, when implementing this algorithm, is that performing the cut on the selected variable may involve the creation of a large number of clauses. In addition, the formula $(\varphi \wedge \neg P) \vee (\varphi \wedge P)$ has to be turned into cnf prior the recursive call, which may be a very expensive operation (this point will be more precise in section 2.3).

As illustrated by figure 2, DLL algorithm avoids this problem by means of a slight modification with respect to DP, which amounts to transforming the logical connector $\vee$ into an alternative.

$DLL(\varphi, \Omega)$:
  **if** $\varphi = \Box$
  **then return** $UNSATISFIABLE$
  **else if** $\Omega = \emptyset$
    **then return** $SATISFIABLE$
    **else if** there is some variable $P$ in some unit clause
      **then return** $DLL(UnitProp(\varphi, P), \Omega \setminus \{P\})$
      **else let** $p$ be a literal of one variable $P$ of $\Omega$
        **in if** $DLL(\varphi \wedge p, \Omega \setminus \{P\}) = SATISFIABLE$
          **then return** $SATISFIABLE$
          **else return** $DLL((\varphi \wedge \neg p), \Omega \setminus \{P\})$

Figure 2: DLL algorithm (Davis, Logeman, Loveland)

## 2.2 Directional Resolution

In [5], Dechter and Rish have noticed that, in some cases, DLL may prove particularly inadequate for solving SAT. A resolution based algorithm called *Directional Resolution* (DR for short) is proposed. Given a fixed[1] ordering $d = P_1, \ldots, P_n$ on $\Omega$, DR transforms an initial formula $\varphi$ into an equivalent formula $E_d(\varphi)$, called *directional extension* of $\varphi$ with respect to $d$. It proceeds by successive cut eliminations, according to the order $d$. This ordering is supposed to be inferred from the structure of the initial formula. The scheme of the algorithm DR is described by figure 3.

Dechter and Rish have proved that *2-cnf* and *horn* classes are solved by DR in polynomial time. They have also identified a particular class of problems, called *k-chains*, which prove to be difficult for their DLL algorithm but that are easily solved with their DR algorithm. Moreover, they have established that the formula $E_d(\varphi)$ produced by DR may be used to obtain any model of $\varphi$ in $O(|E_d(\varphi)|)$ time.

## 2.3 CalcRes

### 2.3.1 Principle

The procedure presented here is called CalcRes [13] and may be considered as a variation on DP in which not all, but only a subset of the variables occurring in the initial formula, are eliminated. Let $\varphi$ be a propositional formula, $\Omega$ be the set of variables occurring in $\varphi$ and $\Psi$, a subset of $\Omega$. CalcRes may be easily described by substituting $\Omega$ by $\Psi$ in the figure 1. As a consequence, DP may be seen as a particular case of CalcRes corresponding to $\Psi = \Omega$.

[1] The authors suggest that dynamic ordering is also possible.

$DR(\varphi, \Omega, d)$:
  **let** $c_1, \ldots, c_n$ a partition of the set of clauses of $\varphi$ such that
    $c_i$ contains all the clauses involving either the literal $P_i$
    or the literal $\neg P_i$ and no literal corresponding to a
    variable of higher index for $d$
  **in for** $i = n$ **downto** 1 **do**
      **for** each pair $\{(\alpha \vee P_i), (\beta \vee \neg P_i)\} \subseteq c_i$ **do**
        **let** $\gamma = \alpha \vee \beta$
        **in if** $\gamma = \square$
          **then return** $E_d(\varphi) = \emptyset$
          **else** determine the highest literal index $j$ in $\gamma$ and
              add $\gamma$ to $c_j$.
      **done**
    **done**
    **return** $E_d(\varphi) = \bigcup_i c_i$.

Figure 3: Directional Resolution algorithm (DR)

However, while DP's result is always either *true* or *false*, the output of CalcRes corresponds to a more general cnf formula expressed on the set of variables $\Omega \setminus \Psi$. This formula is called the *resolvent* of $\varphi$ with respect to $\Psi$ and will be denoted by $\varphi_\Psi$ in the following. Of course, the resolvent is a logical consequence of $\varphi$. Thus, it makes explicit logical constraints on variables of $\Omega \setminus \Psi$, which are implicit in $\varphi$. An important result concerning the resolvent is the following:

Theorem: *Let $\varphi$ a propositional formula, $\Omega$ the set of variables occurring in $\varphi$ and $\Psi \subseteq \Omega$, $\varphi$ and $\varphi_\Psi$ are equivalent with respect to satisfaction.*

The proof of this theorem may be established by showing that for any model $m$ of $\varphi_\Psi$ there exists at least one model $m'$ of $\varphi$ which coincides with $m$ on $\Omega \setminus \Psi$ and thus *extends* $m$ onto $\Omega$.

This result is particularly significant when considering potential applications of resolvent calculus in the framework of knowledge base systems. Producing resolvents may be considered as a form of knowledge base compilation, when focusing on some variables of interest. It may also be useful for validation techniques [10].

### 2.3.2 Algorithm

CalcRes amounts to a succession of cut elimination steps. At each step, a variable $P$ is first chosen, the formula resulting from the cut elimination

CalcRes($\varphi, \Psi$):
  **if** $\varphi = \Box$
  **then return** $\varphi$
  **else if** $\Psi = \emptyset$
    **then return** $\varphi$
    **else let** $P$ be **some** variable of $\Psi$
      **in return** CalcRes($(\varphi^{P\prime} \vee \varphi^{\neg P\prime}) \wedge \overline{\varphi^P},\ \Psi \setminus \{P\}$)

Figure 4: CalcRes: an algorithm for calculating $\varphi_\Psi$

of $P$ is then computed and, if necessary, the result is rewritten as a cnf. To perform the cut operation, the whole set of clauses may be partitioned in three subsets: the set $\varphi^P$ of clauses containing the literal $P$, the set $\varphi^{\neg P}$ of clauses containing the literal $\neg P$ and the set $\overline{\varphi^P}$ of clauses containing neither $P$ nor $\neg P$. The cut amounts to perform all possible resolutions between clauses of $\varphi^P$ and of $\varphi^{\neg P}$, let us denote by $\varphi^{P\prime}$ (resp. $\varphi^{\neg P\prime}$) the set of clauses obtained from $\varphi^P$ (resp. $\varphi^{\neg P}$) by suppressing the literal $P$ (resp. $\neg P$). It may be shown that the formula resulting from the elimination of $P$ corresponds to $(\varphi^{P\prime} \vee \varphi^{\neg P\prime}) \wedge \overline{\varphi^P}$. CalcRes algorithm may thus be described by figure 4.

In order to apply recursively the procedure, it is necessary to rewrite the resolvent as cnf formula, which implies to distribute the set of clauses of $\varphi^{P\prime}$ over $\varphi^{\neg P\prime}$. Note that the number of clauses produced by the cut operation is thus in $O(|\varphi^{P\prime}| * |\varphi^{\neg P\prime}|)$ which is practically prohibitive. However, some of the produced clauses may subsume (or conversely be subsumed by) other existing clauses, which may reduce the total number of clauses. Our experiments have brought out the importance of such subsumptions which are definitely worth to be checked for.

### 2.3.3 Using tries

Systematic testing of all possible subsumptions may be very expensive. However, this cost may be reduced in a significant way by using a suitable data structure for handling sets of clauses. Our implementation is based on the use of *tries*. According to a given ordering on the set of variables, *tries* allow clause factorization (i.e. an efficient representation of clauses beginning with the same sequence of literals).

A set of clauses is then represented by a tree structure. This data structure is detailed in [13] and may also be found in the literature about prime implicates computation [4] as well as in recent DLL implementations [16]. The literal ordering used in CalcRes is the same as the one used by DeKleer [4], i.e. *most frequent variables in the initial formula first.* However, when such an ordering has been established, our current implementation doesn't allow us for dynamic reordering.

### 2.3.4 Heuristics

The number of clauses resulting from successive cuts may grow very rapidly, as well as the length of these clauses. For this reason it seems essential to *control* as much as possible such a growth rate by choosing in the most appropriate way the variable on which to perform the cut. In CalcRes this choice is performed dynamically. For instance, one may attempt to minimize the number of clauses produced by a cut elimination (an estimate of which is given by $H(P) = |\varphi^{P\prime}| * |\varphi^{\neg P\prime}| - |\varphi^{P\prime}| - |\varphi^{\neg P\prime}|$). Of course other choices are possible. One may attempt to minimize the average (or the maximal) length of the derived clauses. Also, one may try to minimize the sum or their length. Our experiments have clearly shown that, while the use of such heuristics is essential to limit the combinatorial explosion, characterizing the merit of some heuristic or other is not easy. Our results show that, on structured instances, each one seems to perform better than others on some instances but no one really surpass the others. On random examples, all heuristics give more or less the same order of elimination.

In the following, we restrict to the case where the heuristic function tends to minimize the estimate of additional clauses produced by the cut elimination. We use the above $H$ function for that. Let us note that with such a criterion, the *most interesting* variable for CalcRes, appears to be the *less interesting* for DLL (when using a standard heuristic [7]). As a matter of fact, this corresponds to the variable which after a positive or negative instantiation, involves the least number of simplifications in the formula. The two approaches yet coincide if there exists some *pure variable* $P$ (i.e. occurring only positively or negatively in $\varphi$). In such a case, $H(P)$ has a negative value and this variable is chosen prior to the other non pure variables. Also, let us note that if a variable $P$ occurs in a unit clause, the subsumption mechanism imply that $H(P) = 0$ and thus $P$ is also preferred to other non pure variables.

# 3 Experimental study

The implementation of CalcRes[2] has been done in Objective Caml, a functional language from the ML family, which can generate native machine code to improve performances. In our experimental study, we first consider CalcRes as an implementation of DP (i.e. in the case where all variables are eliminated). We try to compare CalcRes with DR which, to the best of

[2]Downloadable from http://www.lri.fr/~simon/research/calcres

our knowledge, remains the only other existing implementation of DP. Then we try to evaluate the intrinsic performances of DP as well as its limits. Although much more complex than DLL, since it may be used to produce all models of a formula, it may also be of some interest to situate DP with respect to DLL. The latter has been extensively studied and tested during the last years. As in [11, 15], we have tested CalcRes both on random examples and on classes of structured examples from the DIMACS benchmarks base [6]. Testing on random instances is useful for evaluating the limit in size of unstructured problems that DP is able to solve. It is also interesting to compare the respective behavior of both algorithms. We also investigate CalcRes' behavior when only some of the variables are eliminated. Testing on structured examples is also very interesting since, for many examples the obtained results prove much better that what could be expected.

DLL implementations used for our comparisons are essentially Asat from Olivier Dubois [7] and Sato 3.0 from Hantao Zhang [15]. Sato uses a backjumping mechanism and keeps track of existing conflicts.This avoids to test again incompatible assignments of variables. It is currently considered as one of the best complete SAT prover for structured problems, as those of the DIMACS benchmark base. Because of its simplicity and efficiency, Asat surpasses yet Sato on random examples and may be considered as a prototype of a simple but very good DLL implementation. We also give the execution times of two other implementations of DLL called Satz [8] and RelSat 1.1.2 [2], which are also considered as some of the best current ones, although their execution times are in general much longer than those of Sato 3.0.

## 3.1 Comparison of CalcRes and DR

We may observe on figure 5 that there is an *exponential difference* of execution time between Dechter and Rish's implementation of DR and our implementation of CalcRes on *3-cnf* examples. Even if some of this difference may be explained by hardware progress, the exponential gain is real. This may likely be explained by the use of *tries* and of dynamic heuristics in our approach. Dechter and Rish [5] have identified a class of problems called *k-chains* for which they give a theoretical justification to the fact that DP is much more efficient than DLL. It seems that this class of problem has been specifically designed to stress out DLL's weakness. Our tests have been performed on 300 instances of *k-chains* corresponding to a set of 25 connected bases. Each base is built over 5 variables and contain from 13 to 18 clauses[3].

As it can be seen on table 1, our results confirm those of Dechter and Rish. Asat's execution times are up to 2000 times longer than those of CalcRes on such instances. However this is not true for Sato, which solves such

[3]DR results are based only on 20 experiments for each class.

| Nb of clauses. | ASAT | CALCRES | DR |
|---|---|---|---|
| 299 | 65.8 | 0.05 | 1.4 |
| 349 | 82.96 | 0.05 | 2.2 |
| 399 | 11.69 | 0.05 | 2.8 |
| 449 | 45.82 | 0.05 | 3.7 |
| 499 | 0.44 | 0.05 | 3.8 |
| 549 | 0.14 | 0.05 | 4.0 |

Table 1: CPU time for *k-chains* problems

problems instantaneously (likely due to its backjumping mechanism). When compared with DR, experimental results show that CALCRES is hundred times faster.

## 3.2 Tests on uniform random examples

Our tests have been performed on *3-cnf* instances, generated using the fixed clause-length model [11] which is often used as a standard benchmark class in the literature. All results correspond to an average of 1000 test instances, for a given number of variables (denoted in the following by *nbv*) and of clauses (denoted by *nbc*).

Experimental studies concerning DLL have shown the existence of a peak of difficulty when the value of the ratio *nbc/nbv* is around 4.25 [11]. Below 3.5, nearly all instances are satisfiable, and above 5.0, nearly all are unsatisfiable. When restricting to the value 4.25, our experiments show that CALCRES reaches its time limits with formulas of about fifty variables. This is far less than what current DLL implementations are able to solve (by a factor 10) but is coherent with the fact that DP addresses a much more complex problem than DLL.

### 3.2.1 Behavior around the threshold

The first thing that may be observed (also on figure 5) is that there does not exist a peak of difficulty w.r.t. *nbc/nbv* for DP, neither around the threshold, nor elsewhere. One may clearly notice that, *the more constraints there are at the beginning, the more difficult it is to solve the problem.* This observation has been confirmed by other tests that have been performed on larger instances (30 variables, 1000 tests, for *nbc* varying from 30 to 280). This may be explained by the fact that DP never tries to prove that the initial formula is inconsistent. Moreover, at each step, DP adds new produced clauses. The more clauses there are at the beginning, the more clauses are produced. This can be observed by keeping track of the variation of the total number of clauses during the elimination process (i.e. according to the cut

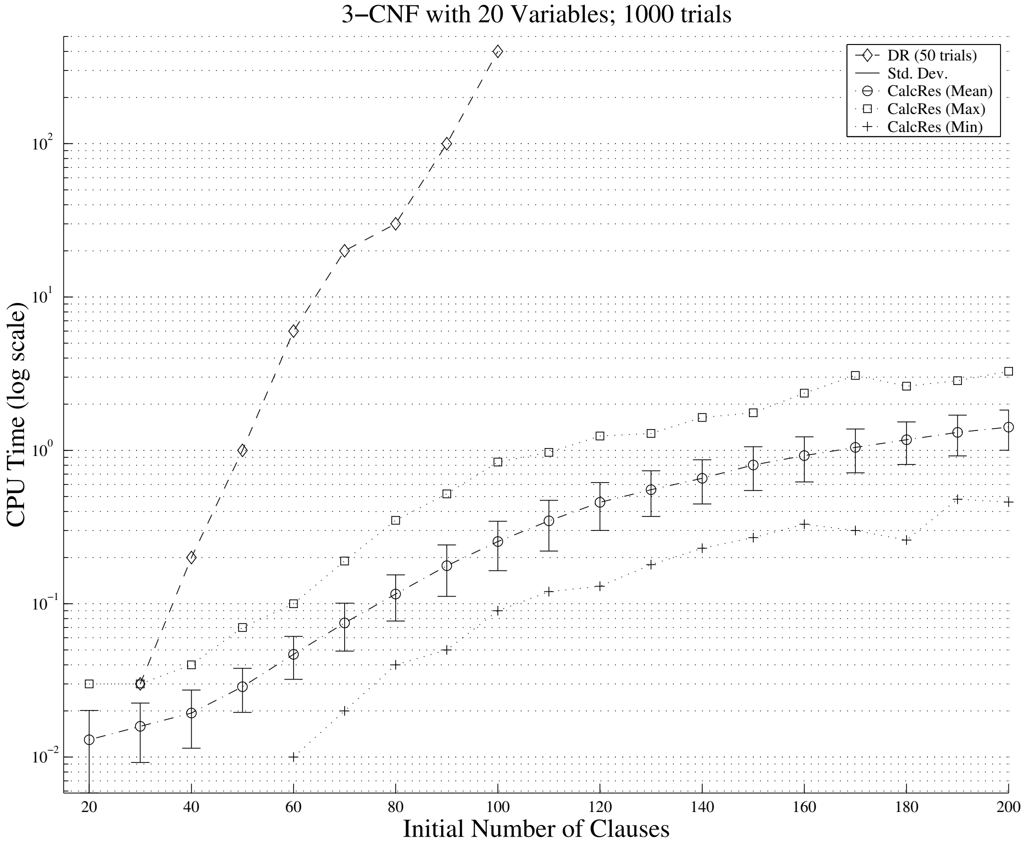


Figure 5: Comparison of CalcRes and DR w.r.t. *nbc*, and absence of peak around the treshold

number), and of its evolution with respect to *nbc* as described by figure 6. Each of the plain lines corresponds to the evolution of the average (on 1000 instances) of the total number of clauses, during the successive cuts.

However, the average value is not always sufficient to summarize all the characteristics of this evolution. On figure 7 we also give, for two particular values of *nbc* (130 and 280) the standard deviation, as well as the maximum and minimum value of the 1000 trials, for each cut. Note that the points corresponding to minimum and maximum curves do not necessarily correspond to the same instances (this explains the singularity of the max curve for the $16^{th}$ cut in the first case).

Results show that the number of clauses goes through a peak and that the level of the peak goes higher when *nbc* grows (there is a ridge). When looking at the level lines on figure 6, we may also observe that there is a deviation of the ridge. When the number of initial clauses rises, the peak is reached earlier. In fact, figure 7 shows that the top of the bell curve is reached earlier, because the decreasing of the total number of clauses, happens sooner. We can see that, with $nbc = 280$, almost all the calculus have found the empty

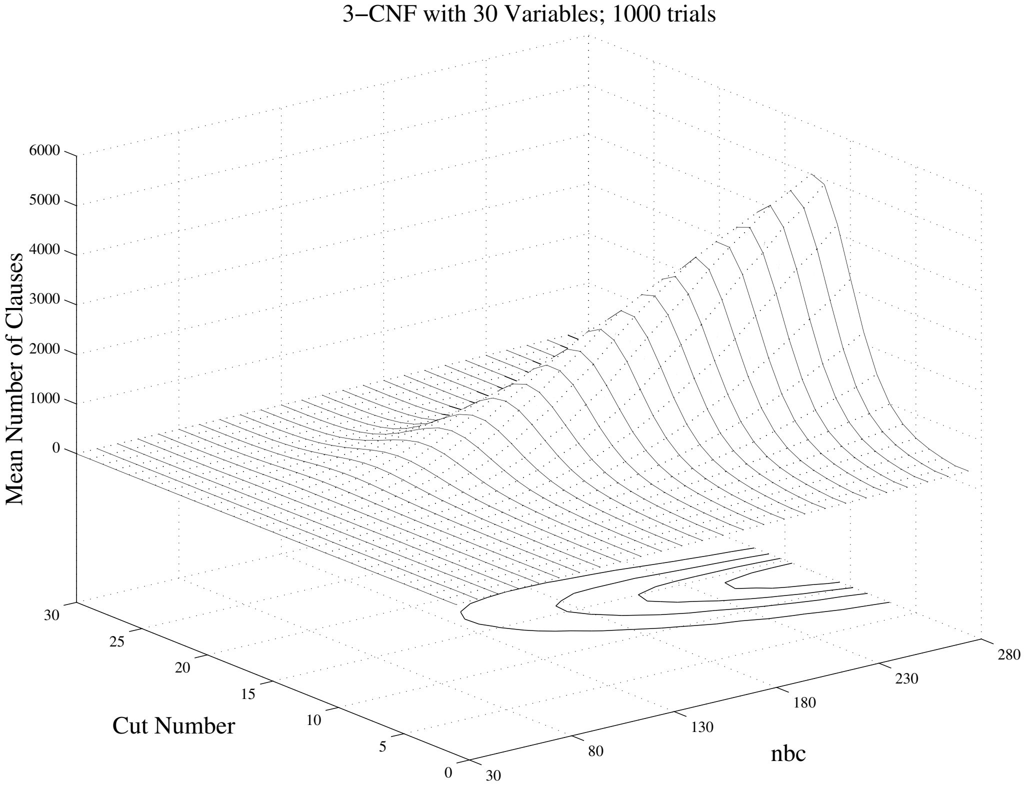


Figure 6: Variation of the total number of clauses during the calculus with respect to *nbc* and cut number

clause (i.e. that the formula is unsatisfiable) at the $21^{st}$ cut (see the maximum curve on figure $7(B)$). In comparison, the minimum curve on figure $7(A)$ tells us that none of the 1000 examples we tried leads to the end of the calculus before the $26^{th}$ cut. Despite this, the calculus remains easier for DP in this last case. *As a conclusion we may note that the difficulty induced by the addition of initial constraints grows quicker than the facility to derive the empty clause.*

### 3.2.2 Peak of difficulty during the elimination

For a fixed *nbc*, the number of clauses during the successive cuts varies along a bell-shaped curve (figure 6 and 7). This may be partly explained by the large number of clauses produced by cuts at the beginning of the calculus and by the fact that the number of variables is continuously decreasing.

The stronger decrease, when *nbc* rises may be related to the appearance of binary clauses. We may note that for $nbv = 130$ the first binary clause appears at the $23^{th}$. For $nbv = 280$, it appears at the $12^{th}$ and their number increases rapidly. Since binary clauses generally induce a lot of subsumptions

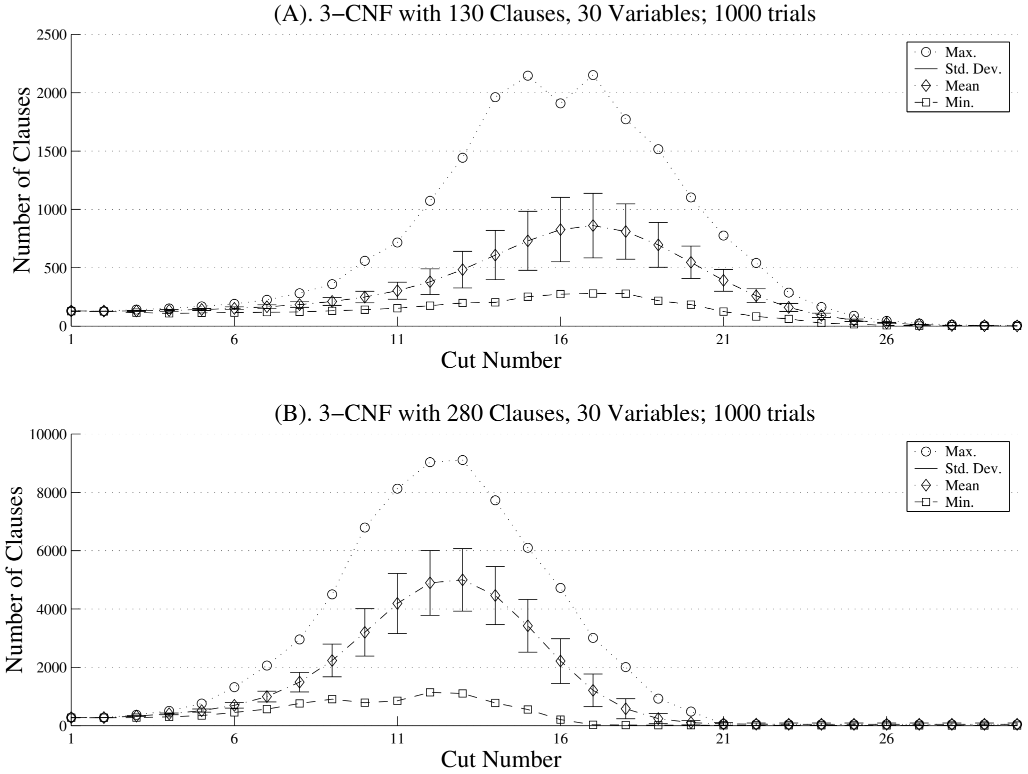


Figure 7: Variation of the total number of clauses for $|\Psi| = 30$, $nbc = 130$ and $nbc = 280$

this may explain the sooner decrease of the total number of clauses, leading to the ridge deviation observed on figure 6.

### 3.2.3 Incidence of the size of $\Psi$ for CalcRes

The specificity of CalcRes with respect to DP is that all the variables are not necessarily eliminated. Here, we study the incidence of the size of the set of eliminated variables $\Psi$ on time, memory and on the structure of the produced resolvents. Intuitively, one could reasonably expect that the smaller the number of variables to eliminate, the easier it should be to compute the resolvent. One may observe on figure 8 that this is not true. Maximal difficulty seems to correspond to the case where only half of the variables are eliminated.

This may be explained by the fact that, reducing the size of $\Psi$ also reduces the possible choices for the heuristic function. In such case the number of produced clauses at each step is more important (see figure 9). When $|\Psi| = 18$, the maximum number of clauses is reached at the $18^{th}$ cut and corresponds to 5000. In comparison, when $|\Psi| = 36$ the maximum is only 2000. But on the other hand, one must not forget that the calculus is finished

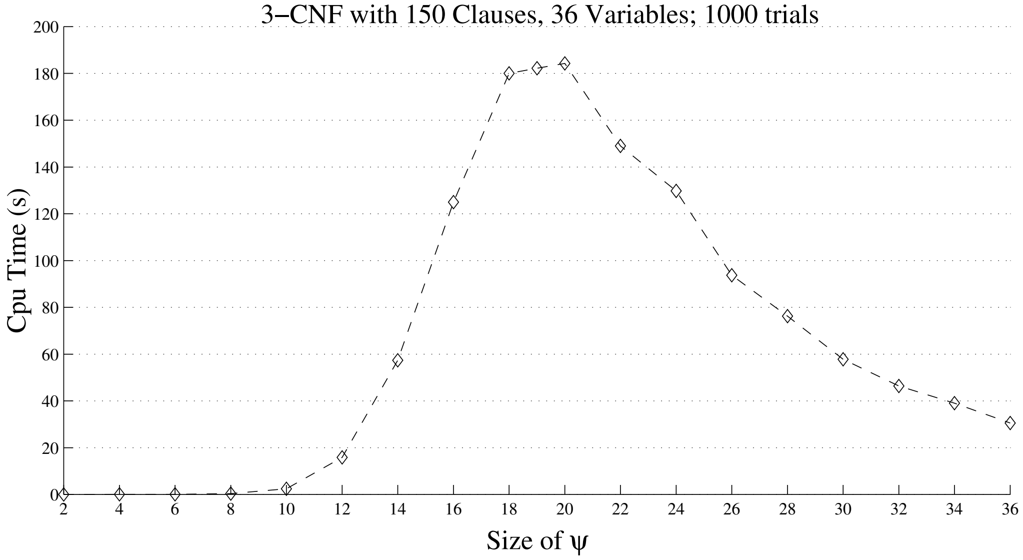


Figure 8: CalcRes cpu time according to the size of $\Psi$

earlier in the first case than in the second one. Thus, this phenomenon is clearly not sufficient to explain why the cpu time may be multiplied by more than 6.5 (figure 8).

A more satisfying answer may be obtained by observing that such a diminution of the size of $\Psi$ also induces a more important variation of the number of clauses eliminated by subsumption. The figure 10 shows that for 18 variables to be eliminated over 36, the $17^{th}$ cut leads to the elimination by subsumption of more than 924000 clauses, while only 820 are added to the set of remaining clauses. On the other hand, when all variables are eliminated, the $17^{th}$ cut causes only 7800 clauses to be subsumed. In fact, in the first case, regarding the large number of clauses eliminated by subsumption, it is rather satisfying to have such a small increase of the total number of clauses between the two consecutive steps.

Considering the length of the resolvent clauses may contribute to a better understanding of what is happening. The figure 11 describes, for two different sizes of $\Psi$ (18 and 36), the repartition, according to their length, of resolvent clauses after the $17^{th}$ cut and after the $18^{th}$ cut. We may observe that, despite the fact that, after the $17^{th}$ cut, there are many clauses of length 6 and over, there are very few clauses of length 10 after the next cut. This means that most of them are eliminated by subsumption. For $|\Psi| = 36$, there remains almost no clauses of length 10 after the $17^{th}$ cut, but the number of clauses of length 6 is much less important and thus, the number of subsumptions is also less important. Theoretically, clauses longer than 10 could be produced from initial clauses of length 3 in 5 cuts, but it is

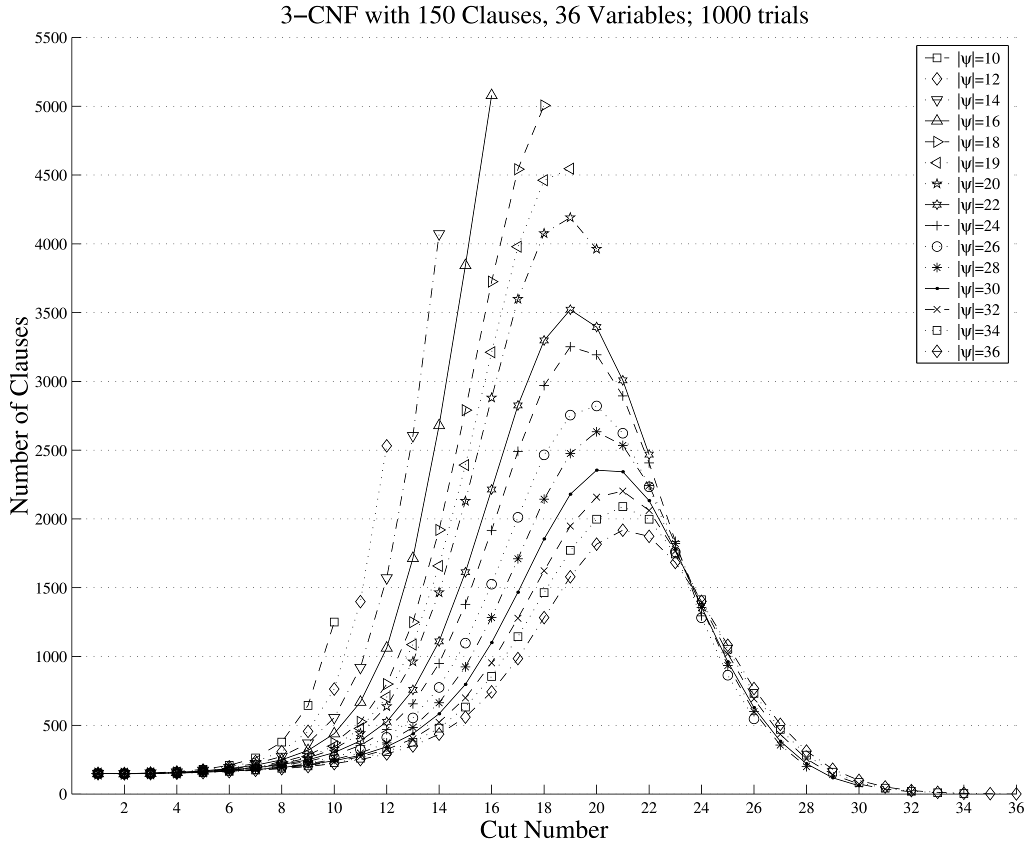


Figure 9: Number of clauses after each cut for different sizes of $\Psi$

### 3.2.4 Resolvent characteristics

Since the computation of a resolvent with CalcRes remains expensive, one may ask whether other approaches couldn't be used. A possible alternative could be to consider all possible clauses that could be constructed using only literals corresponding to the remaining variables (i.e. of $\Omega \setminus \Psi$). If such a clause is logically implied by the initial formula, then it is also implied by the resolvent and an efficient DLL algorithm could be used to prove this by refutation. Schrag and Crawford [12] use this kind of technique to find all prime implicates of length 2 of a formula, by a sequence of inconsistency tests.

To evaluate the pertinence of such an approach, it is interesting to analyze in a more detailed manner the profile of produced resolvents. The figure 12 characterizes the distribution of resolvent clauses, according to their length, in function of the number of eliminated variables. We may observe that this idea would be impractical on random generated instances, since the number of clauses of the resolvent may be very important. For instance, a resolvent with respect to 18 variables contains an average of 1400 clauses of length 6

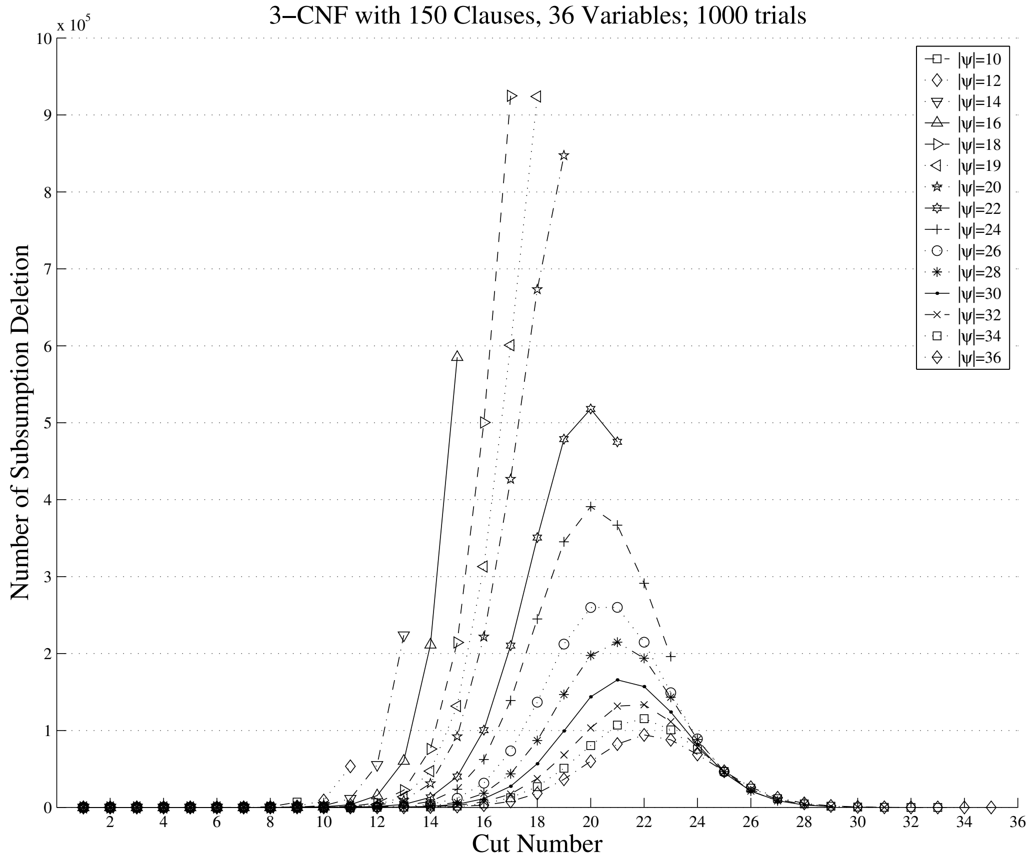


Figure 10: Nb. of subsumptions for each cut for different sizes of $\Psi$

among more that $2.3 * 10^8$ possible ones.

## 3.3 DIMACS' benchmarks

The DIMACS' SAT database gathers a collection of reference instances for the SAT problem. In contrast with *k-cnf* instances, most of them correspond to problems with a particular structure. While intensive testing of SAT solvers on random examples has brought out significant improvements in performance and helped to compare the different existing solvers, random instances are not all. It is not clear at all, that the way these different solvers compare with each other remains identical when considering structured instances.

### 3.3.1 CPU time

Table 2 accounts for the CPU time required to solve problems of the DIMACS base on a *Pentium II-400* with 256Mo memory under Linux. On the DIMACS machine scale benchmark, out tests have granted this machine

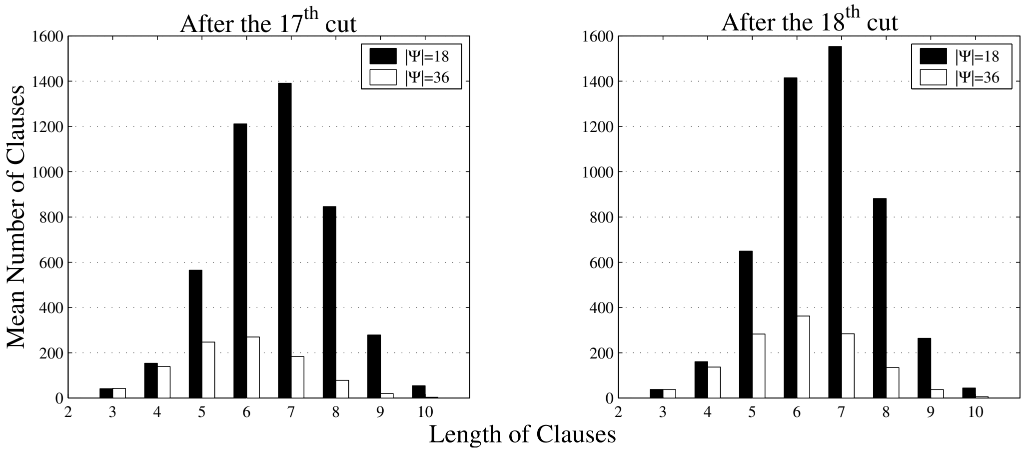


Figure 11: Distribution of clauses according to their length, after the $17^{th}$ and $18^{th}$ cut

a user time saving of 305%, in comparison with the Sparc10.41, given as a reference.

Each problem class is characterized by its name and the number of its instances. For each considered solver in table 2, we give the time for the solving of all instances, assuming that any instance that is not solved within 10000s counts for 10000s. We also give the number of instances of each class effectively solved in less than 10000s.

To be fair, let us notice, that we only give results for problem classes of which CalcRes was able to solve at least one instance. For example, we do not give results for *parity16*, *flat-50* or *aim-150* despite the fact that many instances of them were quickly solved by Asat or Sato.

One should remember that our aim is not to prove that DP is better than DLL for solving SAT. This is definitely false, especially when DLL is supported by an efficient backjumping mechanism as Sato 3.0. Nevertheless, it should be pointed out that results obtained using CalcRes are much better than what could be expected since *it is quicker than* Asat *on a number of instances*. We think that the major asset of DP is that it is able to take the global structure of a problem into account. On some structured examples, this proves more efficient than making choices, as done by DLL. DP's way of proceeding may be considered as a form of local constraint propagation.

Because of its conflict memorization capabilities, Sato is also able in some way to take into account some of the global problem structure. However, let us recall that Sato has been presented recently, after years of intensive work in a very active community around SAT problem solving. In contrast, this is only the first version of CalcRes and many points may still be improved to increase its efficiency, such as the data structure used for representing the set of clauses. Moreover we must keep in mind that DP and DLL differ in

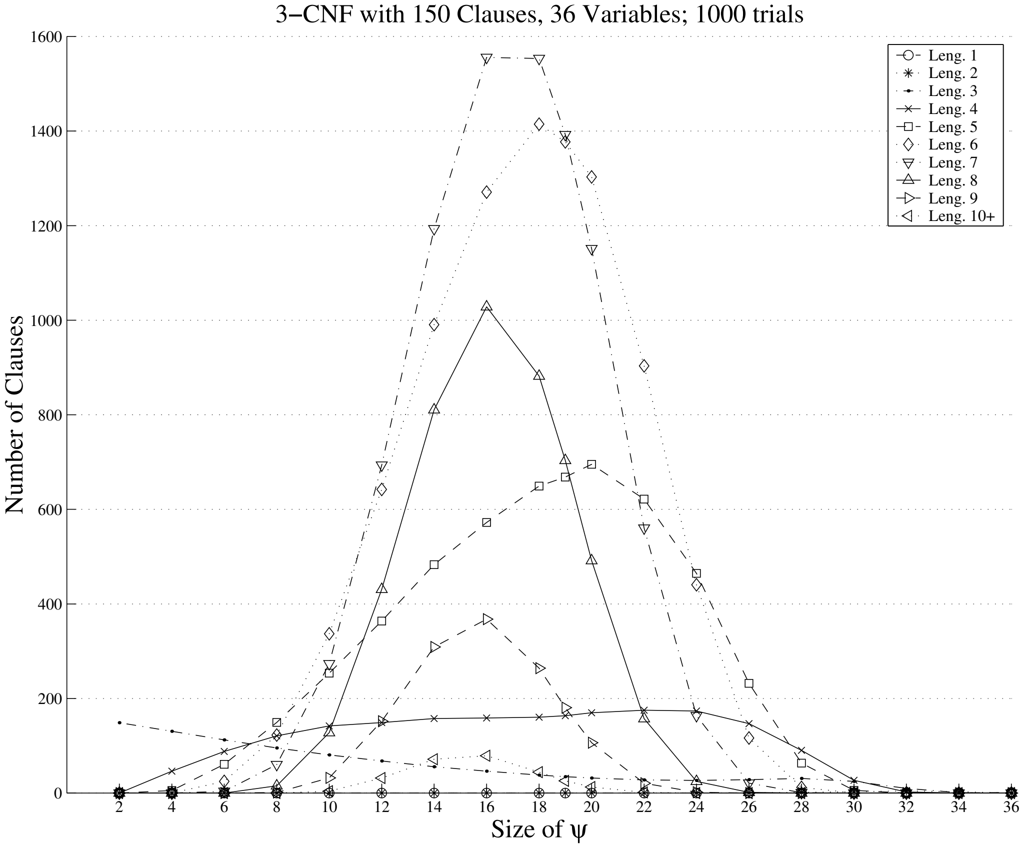


Figure 12: Repartition of resolvent clauses for different sizes of $\Psi$

complexity by an order of magnitude.

### 3.3.2 DP on several instances

As for the *3-cnf* case, we describe CalcRes behavior on several DIMACS examples. In order to assess the evolution of the size of successive resolvents, we keep track of the variation of the total number of clauses, as well as the variation of the total number of literals. As a matter of fact, since the cut operation tends to produce longer clauses, a decrease of the total number of clauses may be balanced by an increase of the average length of the clauses.

Our first observation is that the shape of the obtained curves is very different from the shape of those corresponding to random instances. But different instances of a same class generally give very similar curves (see figures 13 to 16). This clearly suggests that CalcRes is able to suit its behavior to take into account the structure shared by instances of a same class. The most striking case corresponds to the *pret60* example (figure 16), where all instance curves coincide exactly. Our experiments suggest that it is even possible to recognize the class of a given test instance just by looking

| Problems | ASAT | SATO 3 | SATZ | RELSAT | CALCRES |
|---|---|---|---|---|---|
| *Dubois* | 52770 | 0,12 | 34025 | 1,02 | 1,25 |
| 13 | 9 | 13 | 11 | 13 | 13 |
| *flat30* | 0,35 | 0,34 | 0,79 | 8,83 | 119,1 |
| 100 | 100 | 100 | 100 | 100 | 100 |
| *pret60* | 110 | 0,11 | 214,6 | 0,45 | 0,75 |
| 4 | 4 | 4 | 4 | 4 | 4 |
| *ssa* | 20368 | 1,42 | 431,1 | 11,14 | 3767 |
| 8 | 6 | 8 | 8 | 8 | 8 |
| *aim-50* | 0,58 | 0,14 | 7,33 | 0,93 | 80033 |
| 24 | 24 | 24 | 24 | 24 | 16 |
| *aim-100-1,6* | 3693 | 0,041 | 0,38 | 0,33 | 7299 |
| 8 | 8 | 8 | 8 | 8 | 8 |
| *par8* | 0,13 | 0,06 | 0,4 | 0,98 | 14,1 |
| 10 | 10 | 10 | 10 | 10 | 10 |
| *hole* | 155 | 163 | 96,32 | 10096 | 40215 |
| 5 | 5 | 5 | 5 | 4 | 1 |
| *bf* | 30101 | 1,18 | 5,79 | 1,28 | 20424 |
| 4 | 1 | 4 | 4 | 4 | 2 |

Table 2: CPU times (in seconds)

at its curve.

As opposed to the random case, we also observe that for several examples the number of clauses is first decreasing. For some of them, it decreases almost continuously, like for *ssa7552-158* (figure 13) as well as for all instances of *dubois*. For other instances, like those of *flat* and *parity* (figure 14 and 15), it first decreases and then is followed by a peak of variable sharpness. Of course there are also instances for which it increases from the beginning until reaching a peak, like for *aim* (not represented here).

Such an initial decrease, when appearing, may be explained by the fact that heuristics tend to minimize the number of produced clauses. In such structured examples it may be the case that some literals are pure or that they occur either positively or negatively in only one clause. Such situations automatically induce a reduction of the number of clauses. But if we look closer at the evolution of this parameter we may see that this is not the only reason. In fact at many steps, additional clauses are eliminated by subsumption, even if few clauses are produced by the cut.

Because cutting variables tends to produce longer clauses, it is also rather surprising to observe that, on several examples, not only the total number of clauses decreases but also the total number of literals. Again, this may be explained by the fact that heuristics choose variables generating few clauses and by the fact that some of the produced clauses do subsume longer ones.

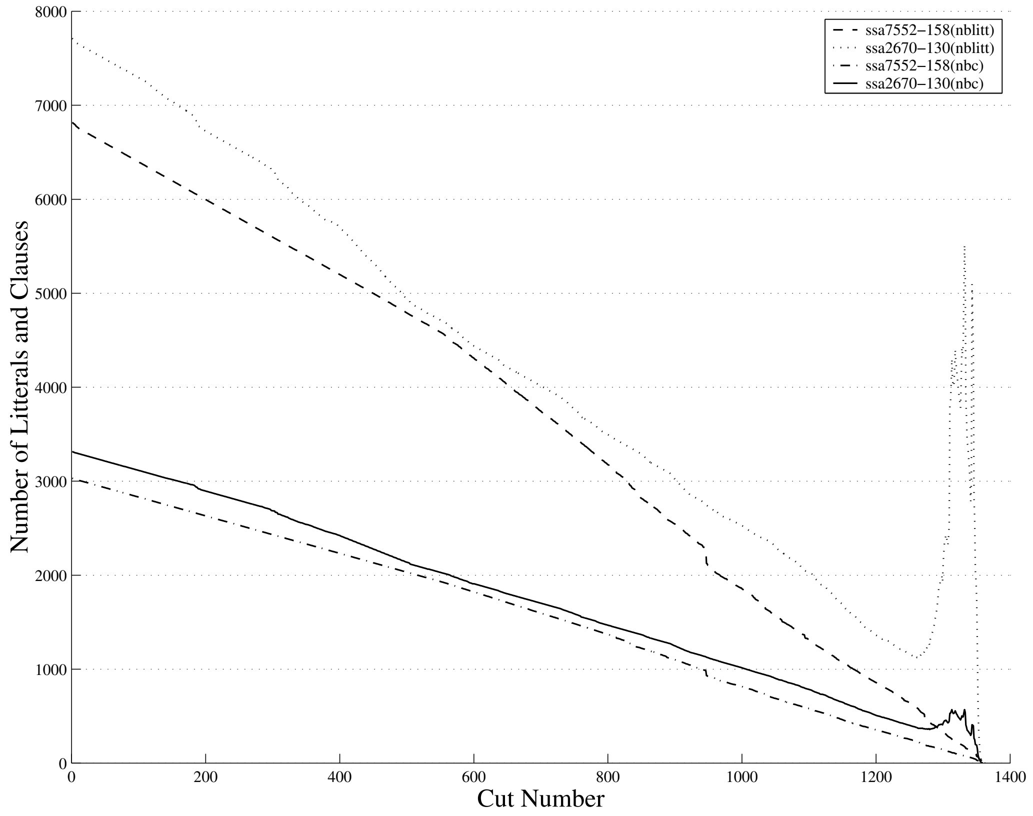


Figure 13: Variation of the number of clauses on some *ssa* instances

On such examples, successive resolvents become smaller and smaller in size. However, this size doesn't reflect the intrinsic difficulty of the problem since, after a while, the number of clauses and of literals may grow suddenly. Heuristic choices rather tend to push forward the difficulties. For instance, we may see on figure 15 that for *parity 8*, the 280 first cuts (over 350) reduce the number of clauses. In the same time the number of literals is divided by 3. But the difficult part of the calculus remains concentrated in the last 10% cuts. A detailed analysis of the structure of successive resolvents has revealed that during the first part of the calculus, no clause of size bigger than 3 is generated. In contrast, the peak corresponds to a sudden production of much longer clauses. Similar phenomenon may be observed on an instance of *parity 32*[4] for which we managed to eliminate more than 90% of the variables. The 83% first cuts lead to smaller resolvents with almost no clause longer than 4.

[4]Recall that this problem, proposed by [1] as a challenge remains unsolved with classical resolution techniques.

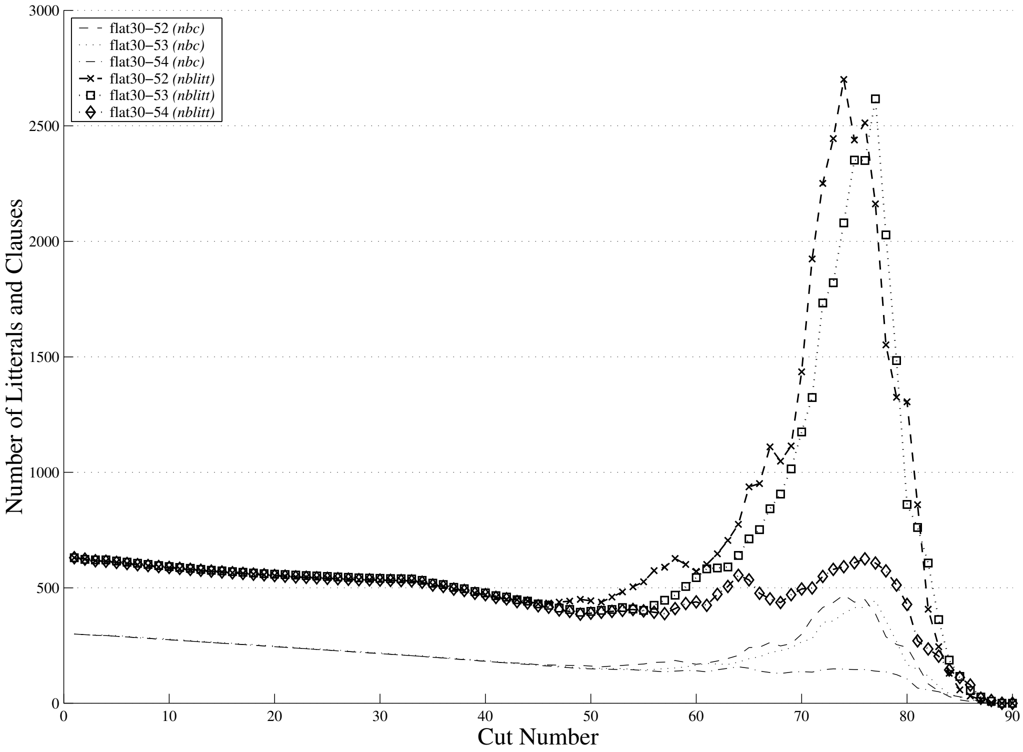


Figure 14: Variation of the number of clauses on some *flat30* instances

# 4 Perspectives: Davis and Putnam far beyond the resolution principle

An interesting question is: in the case where successive resolvents first decrease in size, are these smaller resolvents really easier to solve than the initial formula? In such a case, one could consider using CalcRes as long as it contributes to reduce the size of the resolvent and stop just before the peak area. More generally, one could consider a resolvent as a kind of *compiled formula*, that could be passed to a DLL algorithm for satisfiability checking. The few tests we have done show that, for random instances, it is not only inappropriate but it produces resolvents that are harder to solve for DLL than the original formulas, even if they contain less variables. Yet, things might differ when considering structured instances. We have performed some tests on DIMACS instances, coupling CalcRes with Asat, and in several cases the obtained results were much better that those with Asat alone. This was not the case with Sato on the tested instances, but our feeling is this idea may still pay for, with a more optimized version of CalcRes.

A possible perspective is to design some hybrid algorithm to integrate DP and DLL. This would enable to use DP as far as reasonable and to switch to DLL when coming near a peak of difficulty. Conversely, DP could be

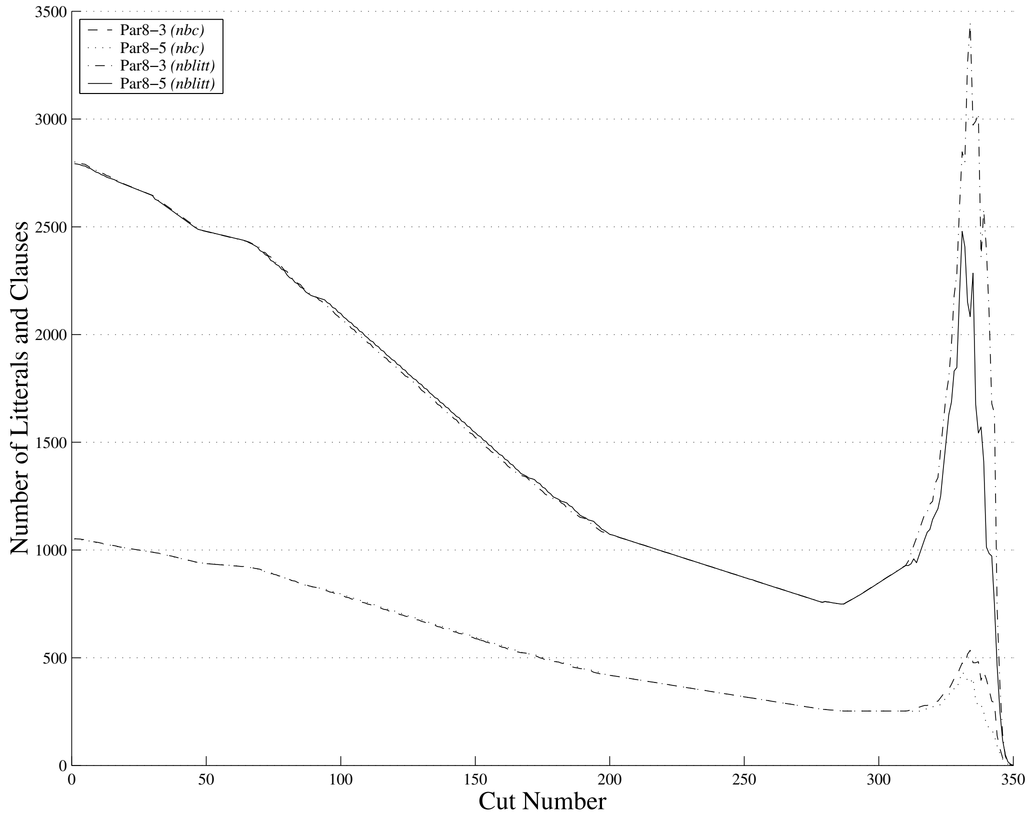


Figure 15: Variation of the number of clauses on some *par8* instances

used inside DLL, when heuristics do not enable to choose clearly between the two possible instantiations of a variable. Then, it might be sometimes more efficient to perform a cut on this variable rather than making a choice, with half of chances to be wrong.

Eventually, we may argue that this is our first implementation of CalcRes, as opposed to the more and more optimized versions of DLL algorithms that have been implemented during the last ten years. Our contribution to this version of DP, w.r.t. the original DP procedure, is mainly the use of tries and of appropriate heuristics. If the choice of heuristics (provided they are dynamic and not too naive) seems to have a minor impact on the obtained results, the choice of the data structure to represent the set of clauses is central. Introduced in AI field eight years ago, the trie structure is still currently the state-of-the-art subsumption machinery (e.g. it is also used in Sato). Our current implementation of tries may still be improved and we think that significant speedup could be obtained with further optimizations.

But we are currently working on a new data structure, that generalizes the notion of tries, and seems best suited to subsumption tests and to the distribution operation, at the heart of CalcRes. Although we have for

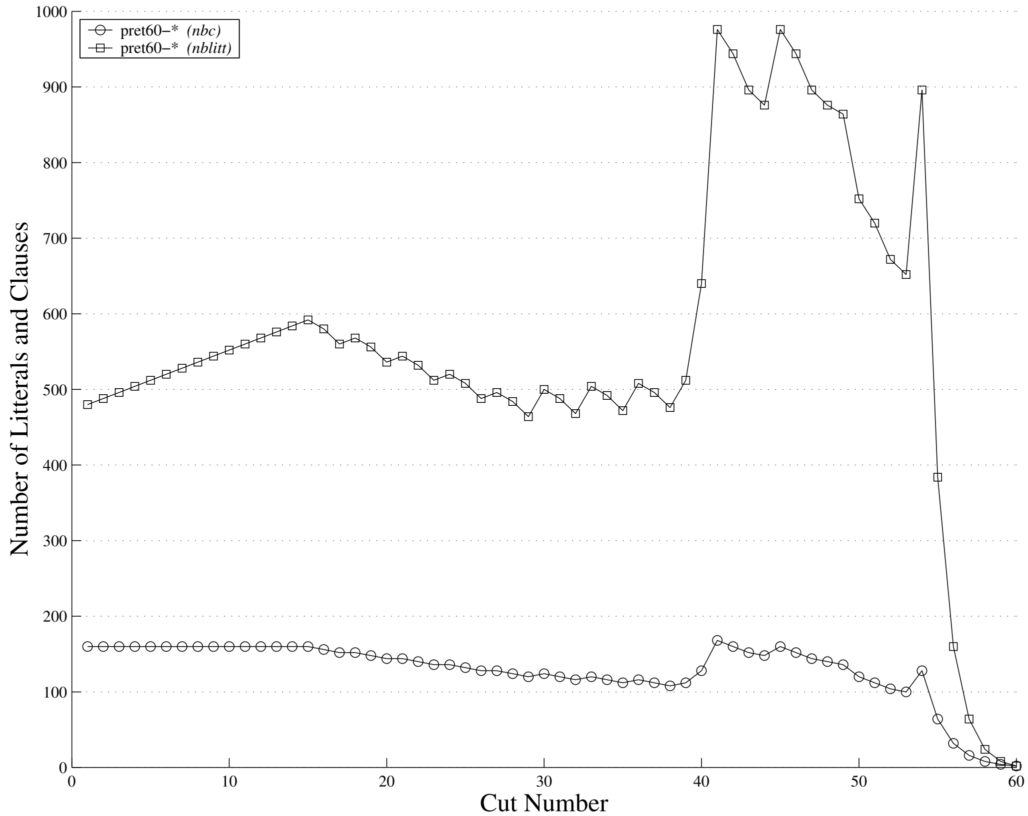


Figure 16: Variation of the number of clauses on some *pret60* instances

the moment, only few preliminary results, they seem very promising and illustrate the fact that DP can be practically and significantly improved.

As an example, with this improved data structure we were able to solve all instances of the *Hole* problem in a much shorter time than any of the other SAT provers mentioned in this paper. We have even solved new instances of this problems for much bigger sizes, such as *hole55*[5]. This is far beyond what any DLL algorithm may expect to solve (the Sato limit on this problem is around the *hole13* instance).

Although we were not able to solve any of its instances, we also tested this new version on the *Parity-32* problem. Starting from an initial formula on 3176 variables, we managed to eliminate 3041 variables, the resulting formula being composed of more than $10^{12}$ high-structured long clauses. In this calculus, distribution between formulas of more than $10^8$ of clauses were performed in a few seconds. In fact, all DLL and DP implementations, including CalcRes, have a complexity directly related to the number of resolution steps. As opposite, this new data structure enables, particularly

[5]Pigeon Hole instance for 56 pigeons and 55 holes, coded by a formula with 3080 variables and 84756 clauses.

on some high-structured instances, to compute the distribution operation regardless of the number of clauses in the set to distribute.

## 5 Conclusion

It is now clear that the original Davis et Putnam algorithm has been too quickly set aside, without real justification. Despite its higher complexity, our DP approach surpasses several DLL implementations without conflict memorization, on some structured instances of the DIMACS benchmark database. While the competition for solving the largest random formula may be arguable, this behavior of DP seems interesting to explore new methods for SAT solving, more suited to structured instances.

The work presented in this paper describes an experimental study of the original DP algorithm, which has been tested intensively on random and structured instances. The performances of the presented system surpasses significantly those of the directional resolution algorithm of Dechter and Rish. The obtained results show that, on *3-cnf* random instances, DP is not sensitive to the usual phase transition phenomenon, observed when using DLL algorithm. On structured examples, they show that taking into account the global structure of a problem may sometimes be a better strategy than making choices as does DLL.

This study also shows that, when CalcRes is used for computing resolvents, on random instances the difficulty depends on the number of variables to eliminate. The results reveal the presence of a peak of difficulty corresponding to the case where this number is around half the number of initial variables.

The results of this study give a better understanding of the limits and advantages of DP. They show that this algorithm is still worth considering, as a complementary approach to DLL. Further improvements are still possible in the implementation of DP, and prospective results even suggest that, for highly structured instances, DP may even be able to surpass DLL capabilities in a significant way.

## References

[1] H. Kautz B. Selman and D. McAllester. Ten challenges in propositional reasoning and search. In *IJCAI-97*, 1997.

[2] R.J. Bayardo and R. Schrag. Using csp look-back techniques to solve real-world sat instances. In *14th National Conference on Artificial Intelligence*, pages 203–208, 1997.

[3] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, pages 201–215, 1960.

[4] Johan de Kleer. An improved incremental algorithm for generating prime implicates. In *AAAI'92*, pages 780–785, 1992.

[5] R. Dechter and I. Rish. Directional resolution: The davis-putnam procedure, revisited. In *Proceedings of KR-94*, pages 134–145, 1994.

[6] The dimacs challenge benchmarks. available on ftp://ftp.rutgers.dimacs.edu/challenges/sat.

[7] Olivier Dubois. Can a very simple algorithm be efficient for sat? ftp://ftp.dimacs.rutgers.edu/pub/challenges/sat/contributed/dubois.

[8] C.M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of IJCAI'97*, pages 366–371, 1997.

[9] G. Logeman M. Davis and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, pages 394–397, 1962.

[10] J.-C. Laleuf M. Gondran, J.-F. Héry. *Logique et Modélisation*. Eyrolles, Collection de la D.E.R.E.F., 1995.

[11] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of sat problems. *AAAI'92*, pages 459–465, 1992.

[12] R. Schrag and M. Crawford. Implicates and prime implicates in random 3-sat. *Artificial Intelligence*, 81(1-2):199–222, 1996.

[13] Laurent Simon. Algorithmes de calcul de résolvant. Stage de DEA d'Informatique. Université Paris XI, Orsay, Septembre 1997.

[14] Colin P. Williams and Tad Hogg. The typicality of phase transitions in search. *Computational Intelligence*, 9(3):221–238, 1993.

[15] Hantao Zhang. Sato: An efficient propositional prover. In *CADE-14*, LNCS 1249, pages 272–275, 1997.

[16] Hantao Zhang and Mark E. Stickel. Implementing the davis-putnam algorithm by tries. Technical report, The University of Iowa, 1994.