

Look-Ahead Versus Look-Back for Satisfiability Problems

Chu Min Li and Anbulagan

LaRIA, Université de Picardie Jules Verne
33, Rue St. Leu, 80039 Amiens Cédex 01, France
tel: (33) 3 22 82 78 75, fax: (33) 3 22 82 75 02
e-mail: {cli@laria.u-picardie.fr, Anbulagan@utc.fr}

Abstract. CNF propositional satisfiability (SAT) is a special kind of the more general Constraint Satisfaction Problem (CSP). While look-back techniques appear to be of little use to solve hard random SAT problems, it is supposed that they are necessary to solve hard structured SAT problems. In this paper, we propose a very simple DPL procedure called *Satz* which only employs some look-ahead techniques: a variable ordering heuristic, a forward consistency checking (Unit Propagation) and a limited resolution before the search, where the heuristic is itself based on unit propagation. *Satz* is favorably compared on random 3-SAT problems with three DPL procedures among the best in the literature for these problems. Furthermore on a great number of problems in 4 well-known SAT benchmarks *Satz* reaches or outspeeds the performance of three other DPL procedures among the best in the literature for structured SAT problems. The comparative results suggest that a suitable exploitation of look-ahead techniques, while very simple and efficient for random SAT problems, may allow to do without sophisticated look-back techniques in a DPL procedure.

1 Introduction

Consider a set of Boolean variables $\{x_1, x_2, \dots, x_n\}$, a literal l is a variable x or its negated form \bar{x} , a clause c is a logical *or* of some literals such as $x_1 \vee \bar{x}_2 \vee x_3$. A propositional formula F in Conjunctive Normal Form (CNF) is a logical *and* of several clauses such as $c_1 \wedge c_2 \wedge \dots \wedge c_m$. F is often simply written as a set $\{c_1, c_2, \dots, c_m\}$ of clauses.

Given F , the CNF propositional satisfiability (SAT) problem consists in testing whether clauses in F can all be satisfied by some consistent assignment of truth values $\{true, false\}$ to the variables. If it is the case, F is said satisfiable; otherwise, F is said unsatisfiable. SAT is a specific kind of finite-domained Constraint Satisfaction Problem (CSP) in which every variable ranges over the values $\{true, false\}$ and is the first NP-complete problem [4]. When each clause in F exactly contains r literals, the restricted SAT problem is called r -SAT. 3-SAT is the smallest NP-complete subproblem of SAT. We distinguish two types of SAT problems: problems having structures such as regularities, symmetries etc... and random problems without any structure. While real world problems are often

structured, random problems represent the "core" of SAT and are independent of any particular domain.

The most effective systematic algorithms are based on the popular Davis-Putnam procedure in Loveland's form (DPL procedure) [6]. DPL procedures such as C-SAT [7], Tableau [5], and POSIT [8] usually employ a variable ordering heuristic and a forward consistency checking (Unit Propagation) known as look-ahead techniques in CSP terms. These algorithms actually have more or less difficulties to solve structured SAT problems. Recently several authors propose to embed (and emphasize) look-back techniques such as backjumping (also known as intelligent backtracking or non-chronological backtracking) and learning (also known as nogood or constraint recording) in a DPL procedure to attack structured SAT problems. GRASP [17] and relsat(4) [2] are such DPL procedures employing both look-ahead and look-back techniques, which are efficient for structured SAT problems but are not effective for random SAT problems.

In this paper we propose a very simple DPL procedure called *Satz* which only employs look-ahead techniques and a simple preprocessing of the input CNF formula to add some resolvents of length ≤ 3 into the clause database. The broad experimental comparative results of *Satz* with several state-of-the-art DPL procedures (C-SAT, Tableau, POSIT, GRASP, relsat(4)) suggest that a suitable exploitation of unit propagation and the preprocessing may be effective for both random SAT problems and a lot of structured ones. Our experience with *Satz* also enforces the belief that if a DPL procedure is efficient for random SAT problems, it should be also efficient for a lot of structured ones.

The paper is organized as follows. Section 2 presents *Satz* by discussing its heuristic and the preprocessing of the CNF formula. Section 3 compares *Satz* on random 3-SAT problems with C-SAT, Tableau, POSIT, the three DPL procedures among the best in the literature for random 3-SAT problems. Section 4 compares *Satz* on 4 well-known SAT benchmarks with GRASP, POSIT, relsat(4), the three DPL procedures among the best in the literature for structured SAT problems. All experiments are made on a SUN Sparc 20 workstation with a 125 MHz CPU. Section 5 discusses the look-ahead and look-back techniques. Section 6 concludes.

2 About *Satz*

We roughly sketch the DPL procedure in Figure 1.

DPL procedure essentially constructs a binary search tree through the space of possible truth assignments until it either finds a satisfying truth assignment or concludes that no such assignment exists, each recursive call constituting a node of the tree. Recall that all leaves (except eventually one for a satisfiable problem) of a search tree represent a dead end where an empty clause is found. Look-ahead techniques such as variable ordering heuristics play a determinant role to reach the dead end early to minimize the length of the current path in the search tree.

```

procedure DPL(F)
Begin
  if  $F$  is empty, return "satisfiable";
  while  $F$  contains a pure literal, satisfy the literal and simplify  $F$ .
   $F := \text{UnitPropagation}(F)$ ; If  $F$  contains an empty clause, return
  "unsatisfiable".

  /* branching rule */
  select a variable  $x$  in  $F$  according to a heuristic H, if the calling
  of  $\text{DPL}(F \cup \{x\})$  returns "satisfiable" then return "satisfiable", otherwise
  return the result of calling  $\text{DPL}(F \cup \{\bar{x}\})$ .
End.

procedure UnitPropagation(F)
Begin
  While there is no empty clause and a unit clause  $l$  exists in  $F$ , assign a
  truth value to the variable contained in  $l$  to satisfy  $l$  and simplify  $F$ .
  Return  $F$ .
End.

```

Fig. 1. The DPL Procedure

2.1 Heuristics Based on Unit Propagation: a Simple Look-Ahead Technique

The most popular SAT heuristic actually is Mom's heuristic, which involves branching next on the variable having Maximum Occurrences in clauses of Minimum Size [7, 5, 8, 16, 11, 10]. Intuitively these variables allow to well exploit the power of unit propagation and to augment the chance to reach an empty clause. However Mom's heuristic may not maximize the effectiveness of unit propagation, because it only takes clauses of minimum size into account to weigh a variable, although some extensions try to also take longer clauses into account with exponentially smaller weights (e.g. 5 ternary clauses are counted as 1 binary clause).

Recently another heuristic based on Unit Propagation (UP heuristic) has proven useful and allows to exploit yet more the power of unit propagation [8, 5, 13, 14]. Given a variable x , a UP heuristic examines x by respectively adding the unit clause x and \bar{x} to F and independently makes two unit propagations. A UP heuristic allows then to take all clauses containing a variable and their relations into account in a very effective way to weigh the variable. As a secondary effect, it allows to detect the so-called *failed literals* in F which when satisfied falsify F in a single unit propagation. However since examining a variable by two unit propagations is time consuming, it is natural to try to restrict the variables to be examined.

The success of Mom's heuristic suggests that the larger the number of binary occurrences of a variable is, the higher its probability of being a good branching variable is, implying that one should restrict UP heuristics to those variables having a sufficient number of binary occurrences.

In [14], we have studied the behaviours of different restrictions of UP heuristics on hard random 3-SAT problems. We found that UP heuristic is substantially better than Mom's one even in its pure form where all free variables are examined at all nodes. Furthermore, the more variables are examined, the smaller the search tree is, confirming the advantages of UP heuristic, but too many unit propagations slow the execution. Based on the experimental evaluations of different alternatives we put forward a dynamic restriction of UP heuristic ensuring that at least T variables selected by a Mom's heuristic are examined by unit propagations. The resulted UP heuristic is realized by the unary predicate $PROP_z$:

Definition: Let $PROP$ be a binary predicate such that $PROP(x, i)$ is true iff x occurs both positively and negatively in binary clauses and having at least i binary occurrences in F , T be an integer, then $PROP_z(x)$ is defined to be the first of the three predicates $PROP(x, 4)$, $PROP(x, 3)$, $true$ (in this order) whose denotational semantics contains more than T variables.

$PROP_z$ is optimal for hard random 3-SAT problems. As we will see, it is also very powerful for structured ones.

Satz is a DPL procedure with the UP heuristic $PROP_z$ with T being empirically fixed to 10. Precisely let $diff(F_1, F_2)$ be a function which gives the number of clauses of minimum size in F_1 but not in F_2 , *Satz*'s branching rule is sketched in Figure 2, where the equation defining $H(x)$ is suggested by Freeman [8] in POSIT.

```

For each free variable  $x$  such that  $PROP_z(x)$  is true do
let  $F'$  and  $F''$  be two copies of  $F$ 
Begin
 $F' := \text{UnitPropagation}(F' \cup \{x\})$ ;  $F'' := \text{UnitPropagation}(F'' \cup \{\bar{x}\})$ ;
If both  $F'$  and  $F''$  contain an empty clause then
return " $F$  is unsatisfiable".
If  $F'$  contains an empty clause then  $x := 0$ ,  $F := F''$ 
else if  $F''$  contains an empty clause then  $x := 1$ ,  $F := F'$ ;
If neither  $F'$  nor  $F''$  contains an empty clause then
let  $w(x)$  denote the weight of  $x$ 
 $w(x) := diff(F', F)$  and  $w(\bar{x}) := diff(F'', F)$ ;
End;

For each variable  $x$  do  $H(x) := w(\bar{x}) * w(x) * 1024 + w(\bar{x}) + w(x)$ ;

Branching on the free variable  $x$  such that  $H(x)$  is the greatest.

```

Fig. 2. The Branching Rule of *Satz*

Satz allows a very simple and very natural implementation exactly corresponding to the above algorithm description, except that the backtracking is

not recursive and its iterative implementation is originally inspired from U-Log [9], a Prolog language interpreter. There is no other technique in *Satz* apart from the two improvements described in section 2.3. To reproduce the performance of *Satz*, one only uses arrays instead of complex data structures in the program (*Satz* uses linked lists to input the clauses then copies all data into arrays before the searching).

The source code in C of *Satz* is available from the first author.

2.2 Discussion on UP Heuristics

Hooker and Vinay [10] study satisfaction hypothesis and simplification hypothesis which are often used to motivate or explain the branching rule of a DPL procedure. Other things being equal, satisfaction hypothesis assumes that a branching rule performs better when it creates subproblems that are more likely to be satisfiable, while simplification hypothesis assumes that a branching rule works better when it creates subproblems with fewer and shorter clauses. One of their conclusions is that simplification hypothesis is better than satisfaction hypothesis.

Satz uses another hypothesis called *constraint hypothesis* which assumes that a branching rule works better when it creates subproblems with more and stronger constraints so that a contradiction can be found earlier. Since the clauses of minimum size are strongest constraints in a CNF formula, $w(x)$ and $w(\bar{x})$ in Figure 2 represent respectively the new constraints of the two generated subproblems if x is selected as the next branching variable. According to constraint hypothesis a DPL procedure should branch next to x such that $w(x)$ and $w(\bar{x})$ are the greatest. The equation defining $H(x)$ and linking $w(x)$ and $w(\bar{x})$ in Figure 2 to favor x such that $w(x)$ and $w(\bar{x})$ are roughly equal and to balance the two subtrees is experimentally better than the following one:

$$w(x) + w(\bar{x}) + \alpha * \min(w(x), w(\bar{x})) \quad (*)$$

where α is a constant.

It seems that C-SAT uses simplification hypothesis and the equation (*) to define $H(x)$, POSIT tries to combine simplification and constraint hypotheses, Tableau uses constraint hypothesis and the same equation as POSIT to define $H(x)$.

C-SAT [7] examines a variable near the leaves of a search tree by two unit propagations (called local processing) to rapidly detect failed literals. Pretolani also uses a similar approach (called pruning method) based on hypergraphs in H2R [16]. But the local processing and the pruning method as are respectively presented in [7] and [16] do not contribute to the branching variable heuristic. We find the first effective exploitation of UP heuristic in POSIT [8] and Tableau [5]. POSIT and Tableau use similar idea as C-SAT to determine the variables to be examined by unit propagation at a search tree node: x is to be examined by unit propagation iff x is among the k most weighted variables.

The main difference of *Satz* with Tableau and POSIT is that *Satz* does not specify a upper bound k of the number of variables to be examined by unit

propagation at a node. Instead, *Satz* specifies a lower bound T . In fact, *Satz* examines many more variables at a node.

Given the depth of a node, Table 1 illustrates the mean number of free variables ($\#free_vars$) and the mean number of variables examined ($\#examined_vars$) by *Satz* at the node, with the depth of the root being 0. In order to compare with C-SAT, Tableau and POSIT we also give the theoretical value of k_C (for C-SAT), k_T (for Tableau) and k_P (for POSIT) at the node, respectively according to the definitions of k in [7, 5, 8].

Table 1. Average number of variables examined by *Satz* for 300 variable and 1275 clause random 3-SAT problems (500 problems are solved).

depth	$\#free_vars$	$\#examined_vars$	k_C	k_T	k_P
1	298.24	298.24	0	263	265
2	296.52	296.52	0	227	230
3	294.92	293.89	0	193	198
4	292.44	292.21	0	141	149
5	288.60	282.04	0	61	72
6	285.36	252.14	0	0	10 or 3
7	281.68	192.82	0	0	10 or 3
8	277.54	125.13	0	0	10 or 3
9	273.17	71.51	0	0	10 or 3
10	268.76	40.65	0	0	10 or 3
11	264.55	26.81	0	0	10 or 3
12	260.53	21.55	0	0	10 or 3
13	256.79	19.80	0	0	10 or 3
14	253.28	19.24	0	0	10 or 3
15	249.96	19.16	0	0	10 or 3
16	246.77	19.28	0	0	10 or 3
17	243.68	19.57	0	0	10 or 3
18	240.68	19.97	0	0	10 or 3
19	237.73	20.46	0	0	10 or 3
20	234.82	20.97	0	0	10 or 3

It is clear from Table 1 that *Satz* examines many more variables at each node than any of C-SAT, Tableau or POSIT. Near the root, *Satz* examines all free variables. Elsewhere *Satz* examines a sufficient number of variables.

2.3 Resolvents-Driven Improvements to *Satz*

Always under constraint hypothesis, we make two resolvents-driven improvements in *Satz*. The first improvement is the preprocessing of the input formula by adding some resolvents of length ≤ 3 , inspired from [3].

For example, if F contains two clauses

$$x_1 \vee x_2 \vee x_3, \bar{x}_1 \vee x_2 \vee \bar{x}_4$$

then we add a clause $x_2 \vee x_3 \vee \bar{x}_4$ to F . The new clauses can in turn be used to produce other resolvents of length ≤ 3 . The process is performed until saturation. In practice, we impose two constraints: (1) a resolvent resulted from two binary clauses should be unary to be added into the clause database, (2) a resolvent resulted from a binary clause and a ternary clause should be binary to be added into the clause database. The preprocessing without the two constraints is provided as an option.

Clearly, the added resolvents become explicit constraints in F as other clauses. Moreover a variable constrained both by a literal and its negated form has an occurrence more to be favored by the branching rule, which is the case for x_2 in the above example which is constrained both by x_1 and \bar{x}_1 .

The standard preprocessing makes *Satz* about 10% faster for hard random 3-SAT problems and allows to instantaneously solve the problems of *aim* class in DIMACS benchmark. It also makes *Satz* 2 times faster when solving the problems of *dubois* class. The preprocessing without the two constraints allows to instantaneously solve all *dubois* problems.

The second improvement consists in weighting more precisely the variables at the nodes where $PROP_z$ is *true* for all variables, because these nodes are often near the root of the search tree and their branching variable has more important effect to reduce the tree size. We define $w(x)$ as the number of resolvents the newly produced binary clauses would result in in F' by a single step of resolution. $w(\bar{x})$ is similarly defined.

Refer to Figure 2, after executing $F' := UnitPropagation(F' \cup \{x\})$, $w(x)$ is defined to be

$$\sum_{\substack{l \vee l' \text{ is in } F' \text{ but not in } F}} [f(\bar{l}) + f(\bar{l}')]$$

where $f(\bar{l})$ is the number of weighed occurrences of \bar{l} in F , an occurrence \bar{l} in a binary clause being counted as 5 in ternary clauses, an occurrence \bar{l} in a ternary clause being counted as 5 in clauses of length 4, etc... The exponential factor 5 is empirically fixed and is better in our experimentation than 2, the factor used in Jeroslow-Wang rule [11].

Clearly the refined weight of a variable looks further forward by measuring the impact of the newly produced binary clauses. The improvement allows to accelerate *Satz* by more than 10% for hard random 3-SAT problems.

3 Experimental Comparative Results on Hard Random 3-SAT Problems

We compare *Satz* with C-SAT, Tableau, and POSIT, the three other DPL procedures among the best in the literature for hard random 3-SAT problems. The 3-SAT problems are generated by using the method of Mitchel et al. [15] from 4 sets of n variables and m clauses at the ratio $m/n = 4.25$, n stepping from 250 variables to 400 variables by 50. Empirically the random 3-SAT problems generated at the ratio $m/n = 4.25$ are the most difficult to solve.

We use an executable of C-SAT dated July 1996. The version of Tableau used here is called *3tab* and is the same used for the experimentation presented in [5]. POSIT is compiled using the provided *make* commande on the SUN Sparc-20 workstation from the sources named *posit - 1.0.tar.gz*¹. Tables 2, 3 show the performances of the 4 DPL procedures on hard random 3-SAT problems of 250, 300, 350, and 400 variables, where *time* standing for the real mean run time is reported by the unix command `/usr/bin/time` and *t_size* standing for search tree size is reported or computed from the number of branches reported by the DPL procedures (note that none of C-SAT, POSIT, *3tab*, and *Satz* uses backjumping here).

Table 2. Mean run time (in second) and mean search tree size of C-SAT, Tableau, POSIT and *Satz* at the ratio $m/n=4.25$ for satisfiable problems.

	250 vars 159 problems		300 vars 170 problems		350 vars 144 problems		400 vars 51 problems	
System	<i>time</i>	<i>t_size</i>	<i>time</i>	<i>t_size</i>	<i>time</i>	<i>t_size</i>	<i>time</i>	<i>t_size</i>
C-SAT	5.3	4610.1	39	23980	240	123198	1813	747478
Tableau	5.3	4013.8	41	22531	272	123106	1836	616693
POSIT	4.9	6393.6	38	40262	270	246715	2362	1814814
<i>Satz</i>	3.8	3846.0	19	18083	106	90071	614	461991

Table 3. Mean run time (in second) and mean search tree size of C-SAT, Tableau, POSIT and *Satz* on ratio $m/n=4.25$ for unsatisfiable problems.

	250 vars 141 problems		300 vars 130 problems		350 vars 106 problems		400 vars 49 problems	
System	<i>time</i>	<i>t_size</i>	<i>time</i>	<i>t_size</i>	<i>time</i>	<i>t_size</i>	<i>time</i>	<i>t_size</i>
C-SAT	17.0	16142.0	128	83235	882	481936	5905	2538072
Tableau	16.1	11736.4	128	69862	947	430323	7362	2469466
POSIT	10.8	14455.5	83	89959	665	609623	4872	3726644
<i>Satz</i>	9.6	9864.0	54	51998	335	288812	1825	1389700

Tables 2, 3 show that on hard random 3-SAT problems, *Satz* is faster than the above cited versions of C-SAT, Tableau and POSIT, *Satz*'s search tree size is the smallest, and *Satz*'s run time and search tree size grow more slowly. Table 4 shows the gain of *Satz* compared with the cited version of C-SAT, Tableau and POSIT at the ratio $m/n=4.25$. Each item is computed from Tables 2, 3 (average of all problems at a point) using the following equation:

$$gain = (value(system)/value(Satz) - 1) * 100\%$$

¹ available via anonymous ftp to ftp.cis.upenn.edu in pub/freeman/

where *value* is real mean run time or real mean search tree size and *system* is C-SAT, Tableau or POSIT. From Table 4, it is clear that the gain of *Satz* grows with the size of the input formula.

Table 4. The gain of *Satz* vs. C-SAT, Tableau and POSIT in terms of run time and search tree size on the ratio $m/n=4.25$ computed from Tables 2 and 3.

System	250 vars 300 problems		300 vars 300 problems		350 vars 250 problems		400 vars 100 problems	
	<i>time</i>	<i>t_size</i>	<i>time</i>	<i>t_size</i>	<i>time</i>	<i>t_size</i>	<i>time</i>	<i>t_size</i>
C-SAT	66%	50%	126%	51%	152%	58%	216%	77%
Tableau	60%	15%	132%	31%	175%	45%	276%	66%
POSIT	18%	53%	68%	89%	133%	130%	198%	200%

4 Experimental Comparative Results on Structured SAT Problems

We compare *Satz* with three other DPL procedures (POSIT, GRASP and relsat(4)) among the best in the literature for structured SAT problems, where GRASP and POSIT are the two best in [17] compared with C-SAT, Tableau, H2R [16], SATO [12], TEGUS [18] on DIMACS and UCSC benchmarks and relsat(4) is the best procedure tested in [2]. We use 4 well-known benchmarks of structured SAT problems: DIMACS², UCSC³, Beijing challenging problems⁴ and planning problems proposed by Kautz and Selman⁵. The version of POSIT is the same as in the last section. we use an executable of GRASP system (version May 1996) available from Joao M. Silva⁶. The relsat(4) system v1.00 is received from Roberto J. Bayardo Jr.⁷.

Following Hooker & Vinay [10] and Silva & Sakallah [17], we use the cutoff time (two hours) as a surrogate for the real run time and partition the DIMACS and UCSC benchmarks into classes, e.g. class *aim-100* includes all problems with the name *aim-100-**. In each class, $\#M$ denotes the total number of class members, $\#S$ denotes the number of problems effectively solved by the corresponding DPL procedure in less than two hours. *Time* denotes the total CPU time in seconds taken to process all members of a class (a problem that can not be solved in less than 2 hours contributes 7200 seconds to the total time). We do not include the classes F, G, PAR32 and Hanoi5 that none of *Satz*, GRASP,

² available from <ftp://dimacs.rutgers.edu/pub/challenge/satisfiability>

³ available from <ftp://dimacs.rutgers.edu/pub/challenge/sat/contributed/UCSC>

⁴ available from <http://www.cirl.uoregon.edu/crawford/beijing>

⁵ available from <ftp://ftp.research.att.com/dist/ai/logistics.tar.Z>

⁶ e-mail: jpms@inesc.pt

⁷ e-mail: bayardo@cs.utexas.edu

POSIT and *relnat(4)* solve in less than 2 hours. The Beijing challenging problems and planning problems are individually listed and a problem that can not be solved in less than 2 hours is marked by " > 7200 ". The obtained results are shown in Tables 5, 6, 7, 8. A first observation is that *Satz*, GRASP, *relnat(4)* are all significantly better than POSIT on these benchmarks. So in the following we only analyse the performances of *Satz*, GRASP and *relnat(4)*.

We find that *Satz* is comparable with *relnat(4)* and GRASP on most DIMACS and UCSC problems (a total of 22 classes) except *pret* class where there is no resolvent of length ≤ 3 and most variables are symmetric so that UP heuristics fail to distinguish them. *Satz* is significantly better than both GRASP and *relnat(4)* in 3 classes (*hole*, *ii16*, *par16*) and for 12 classes *aim-50*, *aim-100*, *aim-200*, *bf*, *dubois*, *ii8*, *jnh*, *par8*, *bf0432*, *ssa0432*, *ssa6288* and *ssa7552*, it has equivalent performance. *Satz* is also very efficient on most *ssa* and *bf* problems except a small number among them (8 *ssa* over 102 and 3 *bf* over 223) which augments the total time for *Satz* to solve the corresponding classes. For example, while most problems in the class *bf1355* can be solved within 3 seconds, the problem *bf1355-243* takes 1395 seconds to be solved. The hardest problem for *Satz* in the *ssa* and *bf* classes is *bf2670-244* which takes 3 hours and 47 minutes to be solved.

In Beijing challenging benchmark, *Satz* solves the same number of problems as *relnat(4)* and one more than GRASP. On Kautz and Selman's planning problems, *Satz* is also comparable with GRASP and *relnat(4)* except 4 problems over 31. Note that *Satz* contains a preprocessing of the input CNF formula to delete duplicate clauses, tautologies, and duplicate literals in clauses, which is time consuming for large problems such as some Kautz and Selman's planning problems or some Beijing challenging problems containing more than 100000 clauses and is not negligible especially when the search itself takes little time.

5 Look-Ahead Versus Look-Back

In CSP terms, *Satz* essentially employs some simple look-ahead techniques to reach a dead end as early as possible: a variable ordering heuristic (UP heuristic), a forward consistency checking (Unit Propagation) and a simple preprocessing, the heuristic being itself based on forward consistency checking. However *Satz* does not include look-back techniques such as backjumping and learning, another class of techniques for CSP problems.

The heuristic of GRASP may be explained by the satisfaction hypothesis which intends to directly satisfy the largest number of clauses and that of *relnat(4)* by simplification hypothesis which intends to value the largest number of variables when branching. Their variable ordering heuristic is simpler than *Satz*. However they exploit sophisticated backjumping and learning to attack structured SAT problems. The experimental results presented in the last section suggests that the good performance of GRASP and *relnat(4)* on a lot of structured SAT problems can be reached simply by a limited resolution at the top

Table 5. Total run time (in seconds) of DIMACS problems.

Problem Class	#M	Satz		GRASP		POSIT		relnsat(4)	
		#S	Time	#S	Time	#S	Time	#S	Time
aim-50	24	24	13.8	24	0.5	24	0.3	24	8.1
aim-100	24	24	3.6	24	1.2	24	352	24	8.4
aim-200	24	24	4.3	24	8.4	13	82792	24	8.2
bf	4	4	25.6	4	5.8	2	14415	4	5.9
dubois	13	13	1.7 ^a	13	6.0	8	50599	13	4.6
hanoi4	1	1	677	1	3910	1	42.8	1	38.0
hole	5	5	483	4	9838	5	429	5	3671
ii8	14	14	5.6	14	17.1	14	1.0	14	7.5
ii16	10	10	106	9	7515	8	14454	10	331
ii32	17	16	7624	17	6.0	16	7551	17	2158
jnh	50	50	7.0	50	10.8	50	0.3	50	19.4
par8	10	10	0.8	10	0.2	10	0.04	10	3.6
par16	10	10	251	10	11349	10	27.0	10	463
pret	8	4	29612	8	13.0	4	29156	8	5.6
ssa	8	8	1748	8	3.9	8	35.1	8	39.1

^a the dubois problems are solved by using an option of *Satz* where all resolvents of length ≤ 3 are added into the clause database. For dubois100.cnf, we add the missing 0 at the end of some clauses.

Table 6. Total run time (in seconds) of UCSC problems.

Problem Class	#M	Satz		GRASP		POSIT		relnsat(4)	
		#S	Time	#S	Time	#S	Time	#S	Time
bf0432	21	21	35.7	21	35.3	21	20.6	21	28.0
bf1355	149	149	3576	149	109	68	648849	149	133
bf2670	53	51	26445	53	50.7	53	1149	53	359
ssa0432	7	7	1.7	7	0.6	7	0.1	7	3.6
ssa2670	12	7	40062	12	35.4	12	1139	12	257
ssa6288	3	3	7.2	3	0.2	3	7.7	3	4.4
ssa7552	80	80	60.5	80	15.4	80	1722	76	43.6

of the search tree and an optimal UP heuristic. The latter approach has the advantages of being simpler and much more efficient for random 3-SAT problems.

5.1 Heuristics Versus Backjumping

A better variable ordering heuristic allows to avoid many useless backtracking. In fact, let $x_{i_1}, x_{i_2}, \dots, x_{i_b}, \dots, x_{i_{d-1}}, x_{i_d}$ be a path from the root to a dead end in a DPL search tree, the standard chronological backtracking backtracks to $x_{i_{d-1}}$ while a backjumping may jump to x_{i_b} in the case where the variables $x_{i_{b+1}}, \dots, x_{i_{d-1}}$ do not contribute to the conflict discovered at x_{i_d} , avoiding in this

Table 7. Run time (in seconds) of Beijing challenging problems.

Problem	Type	#vars	#clauses	Satz	GRASP	POSIT	relsat(4)
2bitadd_10	synthesis	590	1422	> 7200	> 7200	> 7200	> 7200
2bitadd_11	synthesis	649	1562	201	6.6	0.3	0.5
2bitadd_12	synthesis	708	1702	0.4	6.0	0.05	0.5
2bitcomp_5	synthesis	125	310	0.03	0.03	0.01	0.5
2bitmax_6	synthesis	252	766	0.07	0.1	0.01	0.4
3bitadd_31	synthesis	8432	31310	> 7200	> 7200	> 7200	> 7200
3bitadd_32	synthesis	8704	32316	4512	> 7200	> 7200	> 7200
3blocks	planning	370	13732	2.0	28.6	1.8	2.9
4blocksb	planning	540	34199	8.2	473	49.3	6.0
4blocks	planning	900	59285	1542	> 7200	> 7200	296
e0ddr2-10-by-5-1	scheduling	19500	108887	215	143	> 7200	299
e0ddr2-10-by-5-4	scheduling	19500	104527	232	88.5	3508	30.8
enddr2-10-by-5-1	scheduling	20700	111567	> 7200	95.3	> 7200	33.2
enddr2-10-by-5-8	scheduling	21000	113729	229	89.6	> 7200	33.7
ewddr2-10-by-5-1	scheduling	21800	118607	339	101	283	33.4
ewddr2-10-by-5-8	scheduling	22500	123329	279	102	> 7200	41.7

way the amount of time exploring a useless region of the search space. However if we look at the case more carefully, we find that if the backjumping allows to avoid the useless exploring, it is because the branching variables $x_{i_{b+1}}, \dots, x_{i_{d-1}}$ are not good. If the variable ordering heuristic chose x_{i_d} as the branching variable immediately after x_{i_b} , the backjumping would be simply chronological backtracking.

Clearly the UP heuristic looks further forward and allows *Satz* to do without backjumping in most cases.

5.2 Learning

Satz does not include any classical learning techniques applied during the search. But *Satz* does include a standard "statical learning" consisting of a limited resolution before the search to add some resolvents of length ≤ 3 into the clause database. While we believe that the preprocessing is very simple and that these resolvents would be probably learned by a classical learning technique during the search, we think that the "statical learning" in *Satz* is probably insufficient for some structured problems. For example, a complete search of *all* resolvents of length ≤ 3 before the search allows to instantaneously solve all problems in *dubois* class, while the standard "statical learning" only makes *Satz* two times faster. On the other hand, many problems such as *ssa** and *ii** would explode the computer memory by a complete search of all resolvents of length ≤ 3 .

A careful integration with modest overhead of UP heuristic and a dynamic learning such as the one proposed in *relsat(4)* seems promising.

Table 8. Run time (in seconds) of Kautz & Selman’s planning problems.

Problem	sat	#vars	#clauses	Satz	GRASP	POSIT	relsat(4)
bw_large.a	Y	459	4675	0.4	0.3	0.04	0.6
bw_large.b	Y	1087	13772	1.4	3.1	0.5	1.5
bw_large.c	Y	3016	50457	7.0	225	3.4	95.1
bw_large.d	Y	6325	131973	2980	3915	?	418
f7hh.14	Y	4814	114132	1797	114	> 7200	80.8
f7hh.14.simple	Y	3269	61295	1580	59.8	> 7200	32.1
f7hh.15	Y	5315	140258	28.9	313	> 7200	253
f7hh.15.simple	Y	3759	75030	22.3	153	> 7200	96.7
f8h_10	N	2459	25290	1.6	13.6	0.6	2.6
f8h_10.simple	N	1415	14346	1.1	5.7	0.3	1.2
f8h_11	Y	2883	37388	3.0	23.9	371	3.8
f8h_11.simple	Y	1782	20895	2.2	11.4	26.7	2.9
facts7h.10	N	2218	22539	1.4	12.7	2.0	2.1
facts7h	Y	2595	32952	2.6	19.8	0.9	3.5
facts7hh.12	N	3814	68300	> 7200	784	> 7200	200
facts7hh.12.simple	N	2353	37121	> 7200	165	> 7200	66.6
facts7hh.13-simp	Y	4315	48072	12.1	46.3	> 7200	38.2
facts7hh.13-simp.simple	Y	2514	48072	9.7	36.5	1357	25.0
facts7hh.13	Y	4315	90646	15.1	79.3	> 7200	90.1
facts7hh.13.simple	Y	2809	48920	11.3	45.6	> 7200	19.5
facts7hha.12	N	2990	39618	> 7200	111	> 7200	105
facts7hha.12.simple	N	1729	21943	> 7200	58.3	> 7200	36.4
facts7hha.13	Y	3371	53824	10.0	27.6	2134	7.7
facts7hha.13.simple	Y	2069	29508	7.7	15.2	637	5.9
facts8.13	Y	3727	66735	6.5	49.9	141	14.4
facts8h.12	Y	3303	51223	5.7	27.4	1890	5.0
logistics.a	Y	828	6718	212	31.7	11.8	3.0
logistics.b	Y	843	7301	0.7	34.1	0.3	1.5
logistics.c	Y	1141	10719	6.6	116	> 7200	111
rocket_ext.a	Y	331	4446	0.2	2.6	0.2	0.8
rocket_ext.b	Y	351	2398	0.3	3.8	0.02	0.8

5.3 Random SAT Versus Structured SAT

It seems that the look-back techniques as are described in GRASP and relsat(4) are not suitable for random 3-SAT problems. For example, Table 9 shows the behaviour of GRASP et relsat(4) on hard random 3-SAT problems compared with *Satz*.

On the other hand, look-ahead techniques such as those implemented in *Satz* can be used to attack both random 3-SAT problems and many structured SAT problems. Our experiences enforce the belief: if a technique is powerful for random SAT problems, it is probably powerful for structured SAT problems.

We also believe that the essential strategy to write a DPL procedure is to try to reach a dead end as early as possible and heuristics are probably the most effective method to realize the strategy. However if heuristics fail to work well, e.g. for the problems where most variables are symmetric, one should try other

Table 9. Mean run time (in second) of *Satz*, relsat(4) and GRASP at the ratio $m/n=4.25$.

	200 vars 300 problems		250 vars 300 problems	
System	161 <i>sat</i>	139 <i>unsat</i>	159 <i>sat</i>	141 <i>unsat</i>
<i>Satz</i>	0.8	1.8	3.8	9.6
relsat(4)	10.4	35.3	129.9	571.7
GRASP	280.4	1693.8	—	—

methods such as learning or symmetry detection. In this sense, *Satz* can be still improved in the future.

6 Conclusion

We propose a very simple DPL procedure only employing some look-ahead techniques: a variable ordering heuristic, a forward consistency checking (Unit Propagation) and a limited resolution before the search, where the heuristic is itself based on unit propagation. The comparative experimental results of *Satz* on random 3-SAT problems with three DPL procedures among the best in the literature for these problems show that it is very efficient on random 3-SAT problems. While the best algorithms in the literature to solve structured SAT problems usually employ both look-ahead and look-back techniques and emphasize the latter, *Satz* reaches or outspeeds their performances on a lot of structured SAT problems. The results suggest that a suitable exploitation of look-ahead techniques, while very simple and efficient for random SAT problems, may allow to do without sophisticated look-back techniques in a DPL procedure for many structured SAT problems.

Our experiences with *Satz* also enforce the belief that if a DPL procedure is efficient for random SAT problems, it should be also efficient for many structured ones. For problems in which UP heuristic fails to distinguish the variables, e.g., problems with many symmetries, *Satz* can be still improved by learning techniques.

Acknowledgements: We are very grateful to Bart Selman for fruitful suggestions on the organization of this paper and for informing us the DPL procedures GRASP and relsat(4). We also thank Olivier Dubois, James M. Crawford, Jon W. Freeman, Joao P. Marques Silva, and Roberto J. Bayardo Jr. for providing us their DPL procedures.

References

1. Bayardo Jr. R.J., Schrag R.C., *Using CSP Look-Back Techniques to Solve Exceptionally Hard SAT Instances*, Proceedings of the Second International Conference

- on Principles and Practice of Constraint Programming (CP96), Cambridge, Massachusetts, USA, August 1996.
2. Bayardo Jr. R.J., Schrag R.C., *Using CSP Look-Back Techniques to Solve Real-World SAT Instances*, to appear in proceedings of AAAI-97, Providence, Rhode Island, July 1997.
 3. Billionnet A., Sutter A., *An efficient algorithm for 3-satisfiability problem*, Operations Research Letters, 12:29-36, July 1992.
 4. Cook S.A., *The Complexity of Theorem Proving Procedures*, Proceedings of 3rd ACM Symp. on Theory of Computing, Ohio, 1971, pp. 151-158.
 5. Crawford J.M., Auton L.D., *Experimental results on the Crossover point in Random 3-SAT*, Artificial Intelligence, No. 81, 1996.
 6. Davis M., Logemann G., Loveland D., *A machine program for theorem proving*, Commun. ACM 5, 1962, pp. 394-397.
 7. Dubois O., Andre P., Boufkhad Y., Carlier J., *SAT versus UNSAT*. Second DIMACS Implementation Challenge, D. S. Johnson and M. A. Trick (eds.), 1993.
 8. Freeman J.W., *Improvements to propositional satisfiability search algorithms*, Ph.D. Thesis, Department of computer and information science, University of Pennsylvania, Philadelphia, PA, 1995.
 9. Gloess P.Y., *U-Log, a Unified Object Logic*, Revue d'intelligence artificielle, Vol. 5, No. 3, 1991, pp. 33-66.
 10. Hooker J.N., Vinay V., *Branching rules for satisfiability*, Journal of Automated Reasoning, 15:359-383, 1995.
 11. Jeroslow R., Wang J., *Solving propositional satisfiability problems*, Annals of Mathematics and AI 1, 1990, pp. 167-187.
 12. Kim S., Zhang H., *ModGen: Theorem proving by model generation*, Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), 1994, pp. 162-167.
 13. Li C.M., *Exploiting yet more the power of unit clause propagation to solve 3-SAT problem*, Proceedings of the ECAI'96 Workshop on Advances in Propositional Deduction, Budapest, Hungary, August 1996, pp. 11-16.
 14. Li C.M., Anbulagan, *Heuristics Based on Unit Propagation for Satisfiability Problems*, to appear in Proceedings of IJCAI'97, Nagoya, Japan, August 1997.
 15. Mitchell D., Selman B., Levesque H., *Hard and Easy Distributions of SAT Problems*, Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92), San Jose, CA, July 1992, pp. 459-465.
 16. Pretolani D., *Satisfiability and hypergraphs*, Ph.D. Thesis, Dipartimento di Informatica, Università di Pisa, 1993.
 17. Silva J. P. M., Sakallah K. A., *Conflict Analysis in Search Algorithms for Propositional Satisfiability*, Proceedings of the International Conference on Tools with Artificial Intelligence, November 1996.
 18. Stephan P. R., Brayton R. K., Sangiovanni-Vincentelli A. L., *Combinational Test Generation Using Satisfiability*, Memorandum No. UCB/ERL M92/112, EECS Department, University of California at Berkeley, October 1992.