Evaluating search heuristics and optimization techniques in propositional satisfiability

E. Giunchiglia, M. Maratea, A. Tacchella, and D. Zambonin

DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy

Abstract. This paper is devoted to the experimental evaluation of several state-of-the-art search heuristics and optimization techniques in propositional satisfiability (SAT). The test set consists of random 3CNF formulas as well as real world instances from planning, scheduling, circuit analysis, bounded model checking, and security protocols. All the heuristics and techniques have been implemented in a new library for SAT, called SIM. The comparison is fair because in SIM the selected heuristics and techniques are realized on a common platform. The comparison is significative because SIM as a solver performs very well when compared to other state-of-the-art solvers.

1 Introduction

The problem of propositional satisfiability (SAT) is fundamental in many areas of computer science such as formal verification, planning and theorem proving. Despite the exponential worst-case complexity of all known algorithms for solving SAT [1], recent implementations of the Davis-Logemann-Loveland (DLL) algorithm are able to solve problems having thousands of propositions in a few seconds. Much of the success of these solvers is due to clever search heuristics and various optimization techniques that are implemented on top of the basic DLL algorithm. In general, it is difficult to assess the effectiveness of such heuristics and techniques because (i) the quality of the implementations varies greatly and, (ii) each solver is hard-coded with specific search strategies.

In this paper we present an experimental comparison of several state-of-the-art search heuristics and optimization techniques. The test set consists of random 3CNF formulas as well as real world instances from planning, scheduling, circuit analysis, bounded model checking, and security protocols. All the heuristics and techniques have been implemented in a new library for SAT, called SIM. The comparison is fair because in SIM the selected heuristics and techniques are realized on a common platform. Thus, our experimental evaluation is not biased by the differences due to the quality of the implementation, and provides a clear picture about the relative effectiveness of different search strategies. The comparison is significative because SIM itself performs very well when compared to other state-of-the-art solvers.

The paper is structured as follows. In Section 2 we characterize the test set that we have been using. We also experiment with various state-of-the-art SAT solvers on our test set with the purpose of showing that the chosen problems are indeed significative, and of getting an indication of the relative strength of the various SAT solvers. Section 3 is devoted to "basic SIM", i.e., to the presentation of SIM corresponding to the basic DLL algorithm. In Sections 4 and 5, we present the procedures implementing respectively backjumping and learning. The architecture, the branching heuristics, the backjumping and learning procedures are described in as many details as possible (given the space requirements). These detailed presentations are necessary in order to fully understand the procedures, but also the computational cost associated to each of them. As another added value of the paper, we describe all the procedures (except for the ones corresponding to the branching heuristics) by presenting the corresponding pseudo-code. We believe that our presentation is detailed enough to provide a good starting point for implementation. The presentation of the procedures in each Section is complemented with the experimental analysis, showing the relative efficiency of the different heuristics if implemented in basic SIM (Section 3), or combined with backjumping (Section 4) and learning (Section 5). In the final remarks (Section 6) we summarize the results obtained, and point out that SIM, used as a stand-alone solver, behaves very well.

Formal Preliminaries: We use the term *atom* as a shorthand for propositional letter. A *literal* is an atom or its negation. If l is a literal, (i) $|l|$ is the atom in l, and (ii) $\overline{l}$ is $\neg l$ if l is an atom, and is $|l|$ otherwise. A *clause* C is an n-ary disjunction of literals such that, for each pair of literals l, l' in C it is not the case that $|l|{=}|l'|$. A clause is *Horn* if it contains at most one atom not preceded by the negation symbol. A *formula* is an m-ary conjunction of clauses. As customary, we think of clauses as sets of literals, and formulas as sets of clauses. An *assignment* is a function mapping each atom into $\{\text{T,F}\}$. An assignment can be extended to formulas according to the standard truth tables of propositional logic. An assignment μ *satisfies* a formula φ if $\mu(\varphi) = \text{T}$. A formula φ is *satisfiable* if there exists an assignment which satisfies φ. The problem we are interested in is: "Given a formula, is it satisfiable?".

2 Experimenting with DLL solvers

2.1 Designing the test set

Traditionally, SAT solvers are compared on instances corresponding to real world problems and/or randomly generated samples. It is well known that problems belonging to the two categories have different characteristics. Roughly speaking, real world instances present some "structure", corresponding somehow to the "structure" of the original problem coded in SAT. For example, in the case of a a SAT-instance corresponding to the verification problem of a circuit C, typically the SAT-instance consists of different parts, each part corresponding to a sub-circuit of C. Randomly generated tests became popular after [2], in which it is showed that, using the Fixed Clause Length model (FCL) [3], it is possible to generate very hard instances. Since then, they have been widely used to test SAT solvers' performances. However, they lack the structure of real world instances. Given their different characteristics, we decided to use both a test set consisting of instances corresponding to real world problems and a test set of randomly generated samples.

Our test set of real world problems consists of 200 instances, 97 satisfiable and 103 unsatisfiable, yielding an approximate 50% chance of a satisfiable (resp. unsatisfiable) instance. In selecting these instances, we wanted to have benchmarks (i) which are well known in the literature, and (ii) –if possible– that have already been used for comparing SAT-solvers. In particular, we chose:

– the "famous" 30 parity problems, see, e.g., [4];
– the 16 instances of the Bejing Competition held in 1996: 7 circuit equivalences, 6 scheduling problems, 3 planning problems;
– the 32 Data Encryption Standard (DES) problems, see [5, 6];
– 34 formal equivalence verification problems, see [7];

– 31 instances of bounded model checking (BMC), see [8, 6];
– 7 instances of formal verification properties of pipelined circuits from Velev's web page `http://www.ece.cmu.edu/~mvelev`;
– 11 quasigroup problems, see [9];
– 39 planning problems, including 2 `hanoi*` instances from SATLIB (`www.satlib.org`), and blocks world, logistics and rockets problems: most of these problems have been already used to test SAT solvers (see, e.g., [10]) and are available from SATLIB as well.

Notice that some of these instances are known to be very hard to solve for currently available SAT solvers. This is, e.g., the case for the 8 DES instances `cnf-r4-*`. We decided to have them on purpose, given that we plan to use this test set also for future experimental evaluations. All the 200 instances are available on SIM's web page.

For the random problems, we use the FCL model. Formulas are thus generated according to the number of propositions N, the number of literals per clause K, and the number of clauses L. Usually, K is fixed to 3 (see, e.g., [11]), N is fixed to a certain value, while L is varied in order to cover the "100% satisfiable – 100% unsatisfiable" transition. In our experiments, we considered $N = 300$. For $N = 300$, none of the solvers that we consider consistently exceeds the time limit, and the samples in the phase transition (i.e., for the value of L where the probability of a formula being satisfiable is 50%) are not trivial.

2.2 Experimental results

There are many SAT solvers available. In our analysis, we restricted our attention to the DLL-based publicly available procedures. Among these, we considered some of the most effective. In particular: SATZ213, a new version of SATZ [11], POSIT ver 1.0 [12], EQSATZ [6], RELSAT ver. 2.00 [10], SATO ver 3.2 [13]. We do not describe these systems, for which we refer to the corresponding citation. Here, we only point out that these systems rely on very different data-structures and/or search heuristics and/or techniques. Indeed, some of them have been designed to solve specific classes of problems. This is the case, e.g., of EQSATZ, which has been designed to solve formulas with a lot of equivalences, like the parity problems. Of course, we decided to test these solvers (and not others) in a completely arbitrary way. However, the above solvers are state-of-the art, and present interesting features. SATZ and SATZ213 are well known for their dramatic performances on randomly generated tests. POSIT is interesting because of its differences from SATZ (and SATZ213). As reported in [11], the main difference is in the branching heuristic the two solvers use: both examine k propositions before choosing the one to assign, but while POSIT specifies an upper bound on k, SATZ does not. Almost quoting [11], SATZ examines many more propositions at each node than POSIT. Given the fundamental role played by the branching heuristic, we wanted to have a direct comparison between these two solvers on our tests. RELSAT and SATO are well known for their effectiveness on real world problems. EQSATZ is very recent, but has already showed impressive performances on problems with lots of equivalences [6].

Before discussing the experimental results, some more information. All the tests have been run on several identical Pentium III, 600MHz, 128MBRAM. The execution of a system on an instance (be it random or real world) is stopped after 1200s of CPU time.

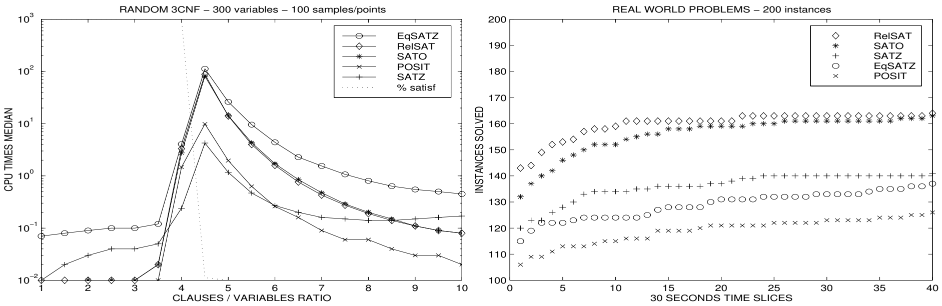

Fig. 1. POSIT, SATZ213, EQSATZ, RELSAT, and SATO. Left: random samples. Right: real world instances.

All the state-of-the-art solvers have been run in their default configuration on all the instances, with the only exception of SATO on randomly generated problems. As a matter of fact, SATO implements a form of "size learning" in which all the "reasons" of length ≤ 20 are indefinitely added to the set of input clauses. Given that learning does not help on randomly generated tests, and that RELSAT implements a form of "relevance learning" in which all the "reasons" of length ≤ 3 are indefinitely retained, we decided to change SATO's default when tested on random samples. In particular, we let SATO keep only the clauses with length ≤ 3, like RELSAT. Learning clauses of ≤ 3 on random problems does not slow down SATO in a significative way as it happens with its default value. In the case of SIM, options are changed in order to see the corresponding effects. For the plots showing systems' performances on random tests, the x-axis is the ratio L/N; the y-axis –in logarithmic scale– is the median CPU time on 100 samples per point for each solver. For the real-word tests, the y-axis is the number of instances solved by each solver within the CPU time specified on the x-axis.

Figure 1 shows the performances of POSIT, SATZ213, EQSATZ, RELSAT, and SATO on our test sets. As it can be seen, POSIT and SATZ213 are very effective on random tests, while RELSAT (with 164 instances solved) and SATO (with 163) are very effective on the real world ones. The bad performances of POSIT with respect to SATZ213 on the real world instances, point out that SATZ213's philosophy, i.e., exploring many propositions before choosing one, seems to pay off. Remarkably, EQSATZ is worse than SATZ213 on both random and real world tests. These data (with the exception of EQSATZ, about which not much is known yet) could have been expected. Indeed, POSIT and SATZ213 (resp. RELSAT and SATO) have been designed to be effective on random (resp. real world) problems. As we will see, the good performances of RELSAT and SATO on the real world instances are because of their learning mechanisms. About EQSATZ, its performances are somehow surprising: evidently, the data-structures and procedures that are at the basis of the big wins for problems with lots of equivalences (EQSATZ is the only system able to solve some of the `par32-*` parity instances in 1200s) have some overhead causing big losses for problems with (almost) no equivalences.

A more detailed data analysis shows that some instances which are solved by some solvers are not solved by the others, and the aother way around. We take this fact as an indication that our test set of real world problems is a good one. Indeed, the `cnf-r4-*` instances mentioned in section 2.1, are not solved by any solver.

3 Basic DLL in SIM

In this section we give a rather detailed description of the data structure, the search control, and the heuristics of SIM. These parts together, define a basic implementation of the DLL algorithm [14]. The conventions we use to present the data types and algorithms are those of [15], described at pages. 4,5. In particular, array elements are accessed by specifying the array name followed by the index in square brackets: for example, $A[i]$ indexes the i-th element of the array A. Compound data are organized into *objects* which are comprised of several *attributes* (or *fields*). A particular field is accessed using the field name followed by an instance of its object in square brackets: for example, $f[x]$ accesses the field f in the object x. Instances, i.e., variables representing arrays or objects, are treated as pointers to the data representing the arrays or objects. If a pointer does not refer to any object, we give it the special value NIL . Finally, stacks are considered a primitive data type and are accessed with the usual primitives PUSH, POP, TOP, etc.. We also assume to have a primitive FLUSH which flushes the stack.

3.1 Data structures and basic primitives of SIM

We now introduce *propositions*, *literals*, *clauses*, and *states* by defining the corresponding data types. Assuming that these data types have been already defined (as we do below), a *proposition* for us is an instance of the proposition data type, and analogously for the others. In the following, we assume that, T, F, U, UN, LS, and RS are six pairwise distinct constants, each one being distinct from NIL . A *proposition data type* is comprised of the following attributes (in the following, p is a proposition):

- *value* is either T, F, or U: intuitively, $value[p]$ is the value assigned to p,
- *mode* is either UN, LS, or RS: intuitively, $mode[p]$ is UN (resp. LS, resp. RS) if p is assigned by unit-propagation (resp. left split, resp. right split),
- *Pos* and *Neg* are arrays of clauses: intuitively, they are the clauses in which p occurs positively and negatively.

For each proposition p, we say that p is *open* if $value[p]$ = U and *valued* otherwise. A *literal data type* has the attributes:

- *prop* is the proposition *associated to* (or *corresponding to*) the literal;
- v is the *sign* of the literal, either T or F.

For each literal l, we say that l is *open* or *valued* if the corresponding proposition $prop[l]$ is open or valued, respectively; l is *positive* (resp. *negative*) if $v[l]$ = T (resp. $v[l]$ = F). A *clause data type* is comprised of the attributes:

- *open* is a non negative integer, representing the number of open literals in the clause;
- *sub* is the clause *subsumer* represented by a proposition;
- *Lits* is an array of literals.

For each clause cl, we say that cl is (i) *empty* if $open[cl] = 0$; (ii) *unary* (resp. *binary*) if $open[cl] = 1$ (resp. $open[cl] = 2$); (iii) *open* if $sub[cl]$ = NIL , and *valued* (or *subsumed*) otherwise. A proposition p *occurs* in a clause cl if $p = prop[l]$ for some literal l in $Lits[cl]$. The proposition occurs *positively* if l is positive and *negatively* otherwise. Finally, a *state data type* is comprised of the attributes:

– *Props*, an array of propositions;
– *Clauses*, an array of clauses;
– *Stack* and *Unit* are the *search* stack of propositions and the *unit propagation* stack of clauses, respectively;
– *open* is a non negative integer, representing the number of open clauses.

Each state s, in order to represent a valid state of the computation, must satisfy the following properties:

1. *Props*[s] stores precisely the propositions that occur in the clauses of *Clauses*[s];
2. for each proposition p in *Props*[s], *Pos*[p] stores precisely the clauses contained in *Clauses*[s] where p occurs positively, and *Neg*[p] stores precisely the clauses contained in *Clauses*[s] where p occurs negatively;
3. *open*[s] amounts to the number of open clauses in *Clauses*[s];
4. *Stack*[s] stores precisely the valued propositions in *Props*[s];
5. *Unit*[s] contains all the open unary clauses in *Clauses*[s].

If s represents an initial state of the computation, then for s it also true that:

6. *Clauses*[s] is not empty; each clause cl in *Clauses*[s] is open; its number of open literals is > 0 and is precisely the length of *Lits*[cl];
7. each proposition in *Props*[s] is open.

Intuitively, it is easy to see that propositions, literals and clauses data types are faithful representations of the corresponding propositional logic objects as defined in Section 1. The arrays *Clauses*[s] and *Props*[s], together with their contents of propositions and clauses, are a faithful representation of formulas. Notice that we explicitly disallow empty input formulas, i.e. $\{\}$, and input formulas containing empty clauses, e.g. $\{\{\}\}$, since the satisfiability problem for these formulas can be simply solved by inspection.

About the primitives of our propositional data structures, let s be a state, p a proposition in *Props*[s], v a *boolean* value (i.e., T or F) and m a mode (i.e., one of UN, LS, RS). EXTEND-PROP(s, p, v, m) is the function that extends the valuation v with mode m to the whole state s for the proposition p: it returns F if an empty clause is found in *Clauses*[s], and U otherwise. More precisely, EXTEND-PROP(s, p, v, m) does the following:

– set *value*[p] to v and *mode*[p] to m; push p in *Stack*[s];
– for each clause cl in *Pos*[p] (resp. *Neg*[p]), if *sub*[cl] = NIL and v=T (resp. v=F) then set *sub*[cl] = p and decrement *open*[s] (*unit subsumption*);
– for each clause cl in *Neg*[p] (resp. *Pos*[p]), if *sub*[cl] = NIL and v=T (resp. v=F) then decrement *open*[cl] (*unit resolution*). If the clause becomes unary, push it in *Unit*[s];
– return F if any clause became empty in the above step, and U otherwise.

Notice the use of *Unit*[s] to detect unary clauses, and of *open*[s] to keep track of subsumed clauses: both operations come for free as a result of performing EXTEND-PROP, while their corresponding simple-minded implementations are linear in the size of *Clauses*[s]. Since EXTEND-PROP destructively updates the state, we have a primitive, RETRACT-PROP, to revert the effects of EXTEND-PROP. For lack of space, we do not describe RETRACT-PROP here, but SIM features both primitives, thus avoiding the burden of copying and saving the state each time EXTEND-PROP is performed.

```
LOOK-AHEAD(s)
r ← U
while r = U and
      length[Unit[s]] > 0 do
  cl ← POP(Unit[s])
  if sub[cl] = NIL then
    i ← 1
    repeat
      l = Lits[cl][i]
      i ← i + 1
    until value[prop[l]] = U
    r ← EXTEND-PROP(s, prop[l], v[l], UN)
if r = U and open[s] = 0 then
  return T
else
  return r

DLL-SOLVE(s)
repeat
  case LOOK-AHEAD(s) of
    T : return T
    F : p ← LOOK-BACK(s, v, m)
    U : p ← HEURISTIC(s, v, m)
  if p ≠ NIL then
    EXTEND-PROP(s, p, v, m)
until p = NIL
return CHECK-SAT(s)

LOOK-BACK(s, var v, var m)
FLUSH(Unit[s])
repeat
  p ← TOP(Stack[s])
  if mode[p] ≠ LS then
    RETRACTPROP(s, p)
until length[Stack[s]] = 0 or
      mode[p] = LS
if length[Stack[s]] > 0 then
  if value[p] = T then
    v ← F
  else
    v ← T
  m ← RS
  RETRACT-PROP(s, p)
else
  p ← NIL
return p
```

Fig. 2. Implementation of the DLL algorithm in SIM.

3.2 DLL-SOLVE, LOOK-AHEAD and LOOK-BACK in SIM

In Figure 2 we present our implementation of DLL algorithm. In the figure, CHECK-SAT(s) returns T if $open[s] = 0$, and F otherwise; s in DLL-SOLVE has to be the initial state corresponding to the input formula. DLL-SOLVE returns T if the input formula is satisfiable and F otherwise.

3.3 HEURISTIC in SIM

A heuristic is a function that given an open state s and an open proposition p in s, computes a score for p and a branching order determining whether p should be satisfied or falsified first. The optimal heuristic is the one which chooses an open proposition and a branching order that will cause the exploration of less nodes. Unfortunately, deciding whether a proposition p is optimal is harder than deciding the satisfiability of the formula itself [16], so we resort to approximations: a good approximate heuristic does not require a lot of computation time and gives higher scores to the propositions that, once assigned, lead to simpler problems.

We now describe in details the branching heuristics that we have tested. Our interest lies in evaluating some of the search heuristics that have been proposed. We restrict our attention to RELSAT's, SATO's, and SATZ's default branching heuristics, and we compare them with a new heuristic (called UNITIE2) that pushes "SATZ's philosophy" to the limit: it always examines every open proposition at each branching node. The heuristics of RELSAT, SATO and SATZ have been implemented by looking at the available literature, but in particular by carefully inspecting the solvers' code. SATZ's heuristics is described in detail in [11].[1] An high-level description of SATO heuristic is given in [13]. Here we give a much more detailed presentation of it, as we understood it from the code and as we implemented it in SIM. We are not aware of any detailed description of RELSAT's heuristic, so we present it according to its implementation in RELSAT and SIM.

We start describing SATO's heuristic. Then, before discussing RELSAT's heuristic, we present the simpler UNITIE2, and reuse parts of its description also for RELSAT.

SATO's heuristic: before the search begins, one of the following options is selected:

1. if the percentage of non horn clauses in the input clauses is bigger than 28.54 of the total, then the heuristic scores all the open propositions,
2. otherwise, the heuristic scores only the first 7 propositions occurring in the first 7 shortest non horn clauses.

The score of a proposition p is given by $(pos\text{-}p+1)\times(neg\text{-}p+1)$, where *pos-p* is the number of open binary clauses in which p occurs, and analogously for *neg-p*. The branching is on the proposition with the highest score, its value determined in the following way: if the percentage of non-horn input clauses is less (resp. greater) than 2.36 of the total, then the value of p is T (resp. F) if $pos\text{-}w > neg\text{-}w$ and F (resp. T) otherwise.

UNITIE2 heuristic: For each open proposition p, this heuristic scores p on the basis of the effective simplifications that would be caused by assigning p on one side, and then $\neg p$ on the other. To compute the simplifications caused by assigning p, a "lean" version of EXTEND-PROP and LOOK-AHEAD is used. In particular the lean versions take care of unit resolutions only. UNITIE2 also incorporates a failed literal detection mechanism: if assigning p would cause a contradiction (in the LOOK-AHEAD), it safely assigns $\neg p$ as if it was a unit (analogously for $\neg p$). Assuming that there are no failed literals, and assuming p (resp. $\neg p$) leads to a set of clauses C^+ (resp. C^-), the score of p is given by $2 \times$ *binposnew* $\times$ *binnegnew* $+$ *unitpos* $+$ *unitneg* where *binposnew* (resp. *binnegnew*) is the number of binary clauses in C^+ (resp. C^-) and not in C, and *unitpos* (resp. *unitneg*) is the number of unit propagations in the LOOK-AHEAD after p (resp. $\neg p$) is assigned. Notice that UNITIE2 always scores all the open propositions. SATZ [11], under certain conditions, scores only a subset of the open propositions. As we said, UNITIE2 pushes SATZ's philosophy to the limit. However, UNITIE2 uses a slightly different formula for computing the score of propositions. In fact, SATZ's formula (in our notation) is $1024 \times$ *binposnew* $\times$ *binnegnew* $+$ *binposnew* $+$ *binnegnew* $+ 1$. The idea behind UNITIE2 is to use *unitpos* + *unitneg* to break ties between propositions having the same $2 \times$ *binposnew* $\times$ *binnegnew*.

RELSAT's heuristic: Also RELSAT scores an open proposition p on the basis of the effective simplifications that would be caused by assigning p on one side, and then $\neg p$ on the other. However, it does this differently from UNITIE2. For each open proposition p, RELSAT first checks whether (i) the number of binary clauses in which p occurs positively is augmented from the previous branching node, and (ii) p has not already been assigned

[1] Notice however that the heuristic in SATZ213 has changed.

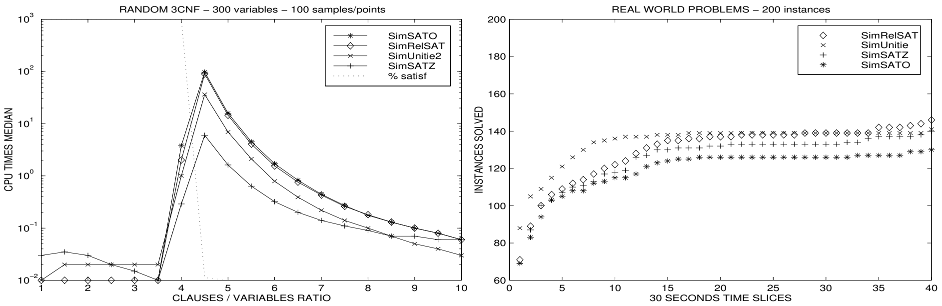

Fig. 3. Basic SIM. Left: random samples. Right: real world instances.

in the heuristic (i.e., if p has not been assigned by unit propagation while scoring another proposition). If this is the case, then RELSAT assigns $\neg p$ and does the consequent (lean) LOOK-AHEAD. If $\neg p$ fails, then $\neg p$ is immediately assigned causing the contradiction and the immediate backtrack.[2] If $\neg p$ does not fail, the number of unit propagations performed in the LOOK-AHEAD is counted as the score of p. If one of the conditions (i), (ii) is false, and if p occurs positively in some binary clauses, then the score *pos-p* of p is the number of binary clauses in which p occurs positively. All the above, has to be analogously done for computing *neg-p* corresponding to $\neg p$. The above rules produce some non-null *pos-p* or *neg-p* if there are binary clauses. If there are not, then RELSAT applies another strategy: for each proposition p, *pos-p* is given by $256 \times cl_3 + 16 \times cl_4 + 4 \times cl_5 + cl_6$ where cl_i $(3 \leq i \leq 5)$ is the number of clauses in which the number of open literals is i, and cl_6 is the number of clauses in which the number of open literals is greater or equal to 6. Analogously for *neg-p*. At this point, for some proposition p we have two scores *pos-p*, *neg-p* one of which is distinct from 0. For all these propositions p, we compute a single score given by $2 \times$ *pos-p* $\times$ *neg-p* $+$ *pos-p* $+$ *neg-p* $+ 1$. Finally, RELSAT chooses randomly among the propositions whose last computed score is ≥ 0.9 of the best score, and fixes its value to T or F randomly.

3.4 Experimental comparison with SIM

We now look at the experimental analysis. Figure 3 shows SIM's performances when using the different heuristics. In the figure, SIMSATO means SIM using SATO's heuristic, and similarly for the others. Consider Figure 3. Our plots on the random problems are similar to the ones in Section 2: SIMSATZ performs better than SIMRELSAT and SIMSATO, these last two systems performing roughly in the same way. The good performances of SIMSATZ on random problems is highlighted by Table 1, showing the minimum, median, IQ-Range and maximum of the CPU time of the solvers at the phase transition.[3] Besides having the lowest median CPU time, SIMSATZ is also the system whose performances have less variations.

[2] The contradiction arising by assigning $\neg p$ is explicitly generated in order to have the necessary interactions with backjumping and learning, see next Section.

[3] Out of 100 numbers listed in ascending order, the *Q-percentile* is the Q-th in the list. The minimum is thus the 1-percentile, the median the 50-percentile, the maximum the 100-percentile, and the IQ-range is the 75-percentile minus the 25-percentile.

Data	SIMSATO	SIMRELSAT	SIMUNITIE2	SIMSATZ
Min	31.56	29.24	7.09	1.53
Median	97.18	89.95	36.27	6.00
IQ-range	70.33	66.38	25.34	3.60
Max	256.45	348.16	121.43	14.96

Table 1. Basic SIM. Random tests. Performances at the phase transition.

From the plots on the real world problems, we see that SIMSATO (with 130 problems solved) is much worse than SIMSATZ (140), SIMUNITIE2 (141) and, above all, SIMRELSAT (146). On the other hand, SIMSATO is the fastest on the 11 quasigroup problems: this is not surprising given that this heuristic has been tuned on these problems [13]. SIMUNITIE2 gives reasonable timings on the two test sets.

4 Conflict-directed backjumping in SIM

Since the basic DLL algorithm relies on simple chronological back-tracking, it is not infrequent for DLL implementations to keep exploring a possibly large subtree whose leaves are all dead-ends. This phenomenon occurs also when the formula is satisfiable, but some choice performed way up in the search tree is responsible for the constraints to be violated. The solution, borrowed from constraint network solving [17], is to jump back over the choices that were not at the root of the conflict, whenever one is found. The corresponding technique is widely known as (conflict-directed) backjumping (CBJ).

4.1 CBJ-LOOK-BACK in SIM

The function CBJ-LOOK-BACK in Figure 4.left, implements conflict-directed backjumping in SIM. In the figure,

- *wr* is meant to store the subset of the propositions in *Stack*[*s*] whose assignment is a reason for the discovered inconsistencies. Technically, *wr* is a clause initialized by INIT-WR(*s*) which returns a clause *cl* in *Clauses*[*s*] having *open*[*cl*] = 0. At least one such clause exists, given that when CBJ-LOOK-BACK is invoked, at least one empty clause belongs to *Clauses*[*s*].
- *reason* is a new attribute of the proposition data type and it is a clause.
- UPDATE-WR(*p*, *wr*) returns a clause *cl* such that:
  - *open*[*cl*] = 0;
  - *sub*[*cl*] = NIL ;
  - *Lits*[*cl*] is the array of literals obtained by merging (without duplications) *reason*[*p*] and *Lits*[*wr*]. Then the literals $l \in$ *Lits*[*cl*] having *prop*[*l*] = *p* are eliminated from *Lits*[*cl*].

Indeed, the procedure LOOK-AHEAD in Figure 2 needs to be extended in order to set the reason of *p* when *p* is assigned by unit. The following line of code has to be written below line 9 of LOOK-AHEAD, at the same level of indentation of line 9:

$$reason[prop[l]] \leftarrow cl$$

For an high-level description of CBJ-LOOK-BACK, see [10].

```
CBJ-LOOK-BACK(s, var v, var m)
FLUSH(Unit[s])
wr ← INIT-WR(s)
repeat
  p = TOP(Stack[s])
  if p ∈ wr then
    if mode[p] ∈ {UN, RS} then
      wr ← UPDATE-WR(p, wr)
      RETRACT-PROP(s, p)
    else
      if value[p] = T then
        v ← F
      else
        v ← T
      m ← RS
      RETRACT-PROP(s, p)
      reason[p] ← wr
      return p
  else
    RETRACT-PROP(s, p)
until length[Stack[s]] = 0
return NIL
```

```
CBJ-LEARN-LOOK-BACK(s, var v, var m)
FLUSH(Unit[s])
wr ← INIT-WR(s)
repeat
  p = TOP(Stack[s])
  if p ∈ wr then
    if mode[p] ∈ {UN, RS} then
      wr ← UPDATE-WR(p, wr)
      LEARN(s, wr)
      FORGET(s, p)
      RETRACT-PROP(s, p)
    else
      if value[p] = T then
        v ← F
      else
        v ← T
      m ← RS
      RETRACT-PROP(s, p)
      case LOOK-AHEAD(s) of
        T: return NIL
        F: FLUSH(Unit[s])
           wr ← INIT-WR(s)
        U: if p = U then
             reason[p] ← wr
             return p
           else
             return HEURISTIC(s, v, m)
  else
    RETRACT-PROP(s, p)
until length[Stack[s]] = 0
return NIL
```

Fig. 4. Conflict-directed backjumping and learning in SIM.

4.2 Experimental comparison with SIM

Figure 5 shows SIM's results when incorporating backjumping. As it can be observed from the figure on the left, backjumping does not produce benefits on the random tests. Indeed, on the random problems the reason of an inconsistency is almost never localized to a small subset of the input clauses. It is nevertheless interesting to have a look at Table 2, corresponding to Table 1 in Section 3. By comparing the Min of the two tables, we see that for SIMSATO and SIMRELSAT, i.e., the two systems whose initial branching literals are selected using a heuristic not based on LOOK-AHEAD, backjumping may produce some benefits. The fact that backjumping never helps for SIMUNITIE2 and SIMSATZ, are an indication that for random instances, LOOK-AHEAD-based heuristics are a good choice. Notice however that the overhead due to backjumping is limited: comparing the Max, we see that it is roughly in the order of the 20% or less. We take this as an indication that our implementation of backjumping is good.

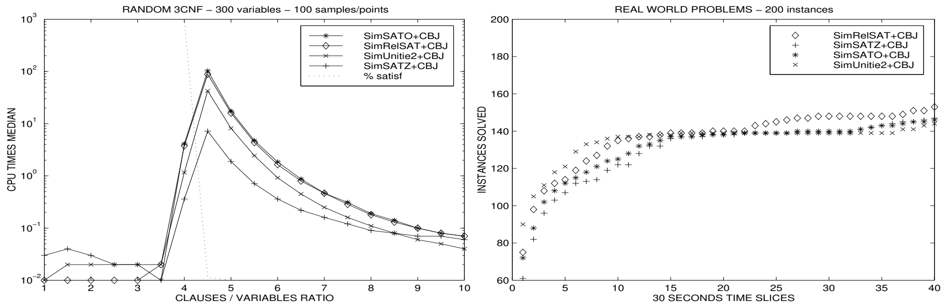

Fig. 5. SIM with backjumping. Left: random samples. Right: real world instances.

Data	SIMSATO	SIMRELSAT	SIMUNITIE2	SIMSATZ
Min	32.95	8.44	8.35	1.86
Median	102.95	88.61	42.59	7.23
IQ-range	73.32	75.42	30.61	4.42
Max	278.29	395.23	143.35	18.15

Table 2. SIM with backjumping. Random tests. Performances at the phase transition.

Looking at the results on the real world tests (Figure 5), we see that RELSAT heuristic (153 instances solved) produces still some advantages, and that the differences with SATO heuristic (146 solved) is not as big as before. SIMSATZ and SIMUNITIE2 solve respectively 147 and 144 instances. SIMUNITIE2 has some advantage at the beginning, but at the end it looses it. Comparing the data with the corresponding ones in Section 3, we see that for each heuristic that we used, backjumping allows to solve more instances.

5 Learning in SIM

For backjumping can be very effective in "shaking" the solver from regions where no solutions can be found, but since the cause of the conflict is discarded as soon as it gets mended, the solver may get repeatedly stuck in such regions. To escape this pattern, some sort of global knowledge is needed: the negation of the causes of the conflicts may be added as an additional constraint (i.e., as a clause) that has to be satisfied. Adding all the clauses corresponding to the reasons of the discovered conflicts has the advantage that the same mistake is never repeated. However, this may cause an exponential blow up of the size of the formula. It is therefore necessary to introduce some limit to the number of stored clauses, i.e., a mechanism that enables the solver to "forget" clauses.

In *size learning*, a clause corresponding to a conflict is stored if it has no more than *Order* literals. Size learning is implemented in SATO, and is used by default with *Order*=20. In *relevance learning*, a clause corresponding to a conflict is always stored. A "forget" mechanism eliminates the learned clauses in which more than *Order* literals are open or have been assigned differently since the time they have been added to the set of input clauses. Relevance learning is implemented by RELSAT, and is used by default with *Order*=3.

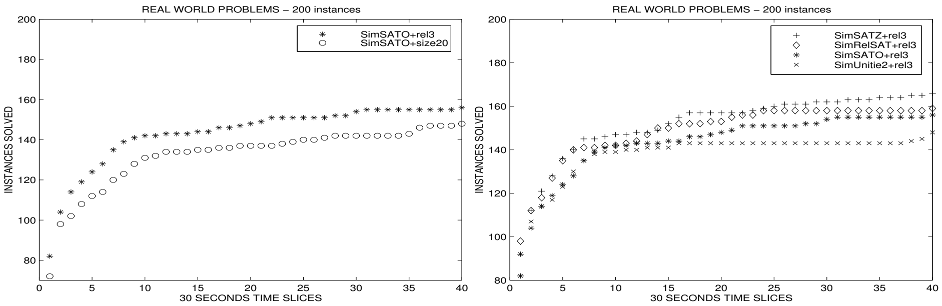

Fig. 6. SIM with learning. Real world instances. Left: size vs relevance learning. Right: different heuristics.

5.1 CBJ-LEARN-LOOK-BACK in SIM

The procedure CBJ-LEARN-LOOK-BACK in Figure 4.right, implements learning in SIM. In the figure, *Order* is yet another attribute of a state (set initially by an input parameter), while the calls LEARN(s, wr) and FORGET(s, p) have different meaning depending on whether they do "relevance learning" as RELSAT, or "size learning" as SATO.

size learning: If wr is a clause with less than *Order*[s] literals, LEARN(s, wr) adds the clause wr to *Clauses*[s], and for each literal l in *Lits*[wr], adds wr to *Pos*[*prop*[l]] (resp. *Neg*[*prop*[l]]) if l is positive (resp. negative). FORGET(s, p) is the null instruction.

relevance learning: LEARN(s, wr) performs the above operations in any case, i.e., also if wr is a clause with more than *Order*[s] literals. FORGET(s, p) deletes the learned clauses cl such that $|\{l : l \in Lits[cl], value[prop[l]] \in \{v[l], \mathrm{U}\}\}| > Order[s]$.

Notice the call to LOOK-AHEAD (line 18) and the following piece of code. Indeed, this call is necessary because of the possible presence of unary clauses among the ones that we learned.

5.2 Experimental comparison with SIM

For lack of space we do not show the plots on the random instances. They have the same qualitative behavior of the ones we already showed, and are available at SIM's web site. Here we only remark that learning does not produce benefits on random instances, but its overhead is always quite small.

On the real world instances, [10] reports that relevance learning is more effective than size learning. Here we show only one plot (Figure 6.left) in which we compare SIMSATO with relevance learning *Order*=3, and SIMSATO with size learning *Order*=20 as in SATO. As it can be seen, the plot with relevance learning dominates the one with size learning.

On the basis of the above, we tested the various heuristics only with relevance learning *Order*=3. The results are shown in Figure 6.right. We see that the heuristic which allows SIM to solve more instances is the one featured by SATZ: SIMSATZ solves 166 instances, SIMRELSAT 159, SIMSATO 156, SIMUNITIE2 148. Interestingly, we have seen that UNITIE2 heuristic –on the real world tests– was second only to RELSAT's heuristic when used in basic SIM. However, if we include backjumping and/or learning, we see that its performances gets worse in comparison with the other heuristics. Analogously,

Random	POSIT	EQSATZ	SATZ213	SATO	RELSAT	SIM SATZ	SIM RELSAT	SIM SATO	SIM UNITIE2
Min	3.36	40.90	1.77	10.68	30.59	1.53	29.24	31.56	7.09
Median	9.88	112.52	4.26	81.31	88.15	6.00	89.95	97.18	36.25
IQ-range	7.39	73.76	2.46	60.74	63.99	3.60	66.38	70.33	25.34
Max	39.76	323.75	10.91	284.99	344.77	14.96	348.16	256.45	121.43
Real World #Solved	126	137	141	163	164	166	159	156	148

Table 3. Solvers performances on random and real world tests.

SIMSATZ's performances becomes better as soon as we introduce backjumping and/or learning. These are clear indications of the obvious fact that the benefits of a heuristics depend on its interactions with the other implemented techniques (see also [18]).

6 Final remarks

We conclude the paper with Table 3, comparing SIM with the various settings, and the solvers that we considered in Section 2. From this table, we see that SIMSATZ solves more real world instances than all the other solvers, including SIM with other heuristics. This last fact shows that the good performances of RELSAT and SATO on real world instances (see Section 2) are due to their learning mechanisms. As we said, SIMSATZ is not always the best on all the classes of problems: different heuristics behave better than others on different problems. Tuning the heuristic on specific classes pays off. SIMSATZ is also the best, among the SIM-systems, for solving random problems, and its performances are not far away from SATZ213's performances. Having one good settings for both random and real world instances is indeed very positive.

The good performances of SIMSATZ also indicate that the quality of our implementation is comparable to the corresponding state-of-the-art solvers. Indeed, if we compare the performances of SIMSATZ on real world instances as shown in Figure 6.right, with RELSAT's and SATO's corresponding performances (see Figure 1.right), we see that both RELSAT and SATO are able to solve more instances than SIMSATZ in the first 700 seconds. Our explanation is that RELSAT and SATO feature rather sophisticated forms of pre-processing of the input clauses, while SIM has no pre-processing at all. Indeed, pre-processing allows for smaller solution times, and we are working on its implementation.

Finally, we showed only a small part of the data that we have collected. For example, we have a figure comparing SIMRELSAT and RELSAT on the random tests: the plots overlaps and are distinguishable with difficulty. Analogously for SIMSATO and SATO.[4] These plots show that our reconstruction of their branching heuristics is accurate.

All the plots, SIM and more information about SIM are available at SIM web page:

`www.mrg.dist.unige.it/star/sim`

[4] This holds for the median time. Indeed, by looking at Table 3, we see that SIMSATO's and SATO's performances are not as similar as SIMRELSAT's and RELSAT's performances are. This is due to the particular heuristic of SATO, which in some cases chooses the first 7 propositions in the first 7 non Horn clauses. Indeed, the selected proposition (and thus the behavior) depends also on the order in which clauses are stored, and –in this– SIMSATO is different from SATO.

Acknowledgements: We would like to thank Holger H. Hoos and Thomas Stützle for the excellent work they do with SATLIB. This work is partially supported by MURST and Intel Corp.

References

1. J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the satisfiability (sat) problem: A survey. In *Satisfiability Problem: Theory and Applications*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 19–153. AMS, 1997.
2. D. G. Mitchell, B. Selman, and H. J. Levesque. Hard and Easy Distributions for SAT Problems. In *Proc. of AAAI*, pages 459–465. AAAI Press, 1992.
3. J. Franco and M. Paull. Probabilistic analysis of the Davis-Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5:77–87, 1983.
4. Bart Selman, Henry Kautz, and David McAllester. Ten Challenges in Propositional Reasoning and Search. In *Proc. of IJCAI*, pages 50–54. Morgan-Kauffmann, 1997.
5. Massacci and Marraro. Logical Cryptanalysis as a SAT Problem. *JAR: Journal of Automated Reasoning*, 24, 2000.
6. Chu Min Li. Integrating Equivalency Reasoning into Davis-Putnam Procedure. In *Proc. of AAAI*. AAAI Press, 2000.
7. 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.
8. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer Verlag, 1999.
9. H. Zhang and M. E. Stickel. Implementing the Davis-Putnam Method. In *Highlights of Satisfiability Research in the Year 2000*. IOS Press, 2000.
10. R. J. Bayardo, Jr. and R. C. Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT instances. In *Proc. of AAAI*, pages 203–208. AAAI Press, 1997.
11. Chu Min Li and Anbulagan. Heuristics Based on Unit Propagation for Satisfiability Problems. In *Proc. of IJCAI*, pages 366–371. Morgan-Kauffmann, 1997.
12. Jon W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.
13. H. Zhang. SATO: An efficient propositional prover. In *Proc. of CADE*, volume 1249 of *LNAI*, pages 272–275. Springer Verlag, 1997.
14. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the ACM*, 5(7):394–397, 1962.
15. T. H. Cormen, C. E. Leiserson, and R. R. Rivest. *Introduction to Algorithms*. MIT Press, 1998.
16. P. Liberatore. On the complexity of choosing the branching literal in DPLL. *Artificial Intelligence*, 116(1–2):315–326, 2000.
17. R. Dechter, I. Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49:61–95, 1991.
18. F. Copty, L. Fix, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Vardi. Benefits of Bounded Model Checking at an Industrial Setting. In *Proc. of CAV*, LNCS. Springer Verlag, 2001. To appear.