kcnfs: An Efficient Solver for Random *k*-SAT Formulae

Gilles Dequen[1⋆] and Olivier Dubois[2]

[1] LaRIA, Université de Picardie, Jules Verne, CURI, 5 rue du moulin neuf, 80000 Amiens, France,
dequen@laria.u-picardie.fr
[2] LIP6, CNRS-Université Paris 6, 4 place Jussieu, 75252 Paris cedex 05, France,
Olivier.Dubois@lip6.fr

Abstract. In this paper we generalize a heuristic that we have introduced previously for solving efficiently random 3-SAT formulae. Our heuristic is based on the notion of backbone, searching variables belonging to local backbones of a formula. This heuristic was limited to 3-SAT formulae. In this paper we generalize this heuristic by introducing a sub-heuristic called a re-normalization heuristic in order to handle formulae with various clause lengths and particularly hard random k-SAT formulae with $k \geq 3$. We implemented this new general heuristic in our previous program *cnfs*, a classical DPLL algorithm, renamed *kcnfs*. We give experimental results which show that *kcnfs* outperforms by far the best current complete solvers on any random k-SAT formula for $k \geq 3$.

1 Introduction

In [1] we presented a new branching variable selection heuristic for solving random 3-SAT formulae using a DPLL-type procedure. This heuristic relied on the notion of *local backbones* of a SAT formula, derived from a notion introduced in a statistical physics study [2]. The heuristic described in [1] was designed and works efficiently only on 3-SAT formulae. It cannot work on formulae with different clause lengths. In this paper we present a generalization of our previous heuristic covering formulae with clauses of any length and in particular for solving the classical random k-SAT formulae with $k \geq 3$. The generalization consists in inserting in our previous "backbone-search heuristic" (BSH for short), a sub-heuristic, which we call a *renormalization* heuristic, taking account of the various lengths of clauses. We describe in this paper such a renormalization heuristic and integrating it into the backbone-search heuristic, we get a general heuristic. We show that this general heuristic quite markedly improves the best current performances of solving of random k-SAT formulae as far as values of k can be handled in reasonable time.
The paper is organized as follows. In section 2 we first recall the principle of the backbone-search heuristic, we illustrate how it works differently from the

⋆ Contract grant sponsor: The Région Picardie under HTSC project

E. Giunchiglia and A. Tacchella (Eds.): SAT 2003, LNCS 2919, pp. 486–501, 2004.

classical heuristics and we resume and update the experimental results of performances of this heuristic implemented in the program *cnfs* [1], a DPLL-type procedure. Then, in section 3 we describe our renormalization heuristic. Having implemented it in a general program which we renamed *kcnfs*, we give the performance results obtained on solving hard 3-SAT, 4-SAT, 5-SAT ... random formulae as compared with those obtained with the best known solvers.

2 The Backbone-Search Heuristic for Solving Random 3-SAT Formulae

2.1 The Principle of the Computation of the Backbone-Search Heuristic (BSH)

Let $\mathcal{F}$ be a 3-SAT formula and consider a DPLL-type algorithm ([3], [4]), for solving $\mathcal{F}$. Such an algorithm develops a solving tree until all clauses of $\mathcal{F}$ are satisfied by values 0 or 1 assigned to the variables of $\mathcal{F}$ at nodes of the tree or until no satisfying truth assignment has been found. This type of algorithm is called a complete algorithm since it gives an answer for unsatisfiable as well as satisfiable formulae. To solve efficiently satisfiable formulae, right truth values to be assigned to the variables are to be found. To solve efficiently unsatisfiable formulae, an implicit solving tree with a size as small as possible is to be developed. It appears that these two goals are hard and even somewhat incompatible to reach by a same heuristic. However developing a solving tree leads automatically to a solution of a formula if it is satisfiable. Trying to find right truth values for satisfying a formula does not lead automatically to be able to answer that the formula is unsatisfiable if it cannot be satisfied. Moreover developing a solving tree cannot be less efficient in the worst case (and in fact in any case for random formulae) for a satisfiable formula than for an unsatisfiable formula with the same size. For these reasons, designing an efficient heuristic for a complete DPLL-type algorithm amounts to supposing that one tries to solve only unsatisfiable formulae. This is exactly the way we designed the heuristic BSH which we describe below.

Let x be one of variables of the 3-SAT formula $\mathcal{F}$. The idea behind BSH is to measure the correlations of the literal x on the one hand and the literal $\bar{x}$ on the other hand with all the other variables of the formula through the clauses where x and $\bar{x}$ appear. More precisely let us take for example a clause where the literal x appears : $(x \vee u \vee v)$. If both literals u and v can be TRUE in most truth assignments satisfying many (possibly all) clauses, then x is a little correlated with the other variables. If one of the literals u and v must be FALSE if the other one is TRUE in most truth assignments satisfying many clauses, then x is a little more correlated with the other variables than in the previous case. Finally if both literals u and v are FALSE in most truth assignments satisfying many clauses, then x is strongly correlated to the other variables. Our heuristic BSH is intended to identified the variables for which both associated literals are strongly correlated with the others, by means of the function $h(t)$ we gave in [1]

Set $i \leftarrow 0$, let t be a literal and let MAX be an integer.
Integer $h(t)$
begin
$i \leftarrow i + 1$
compute $\mathcal{I}(t)$
IF $i < MAX$
return $\sum_{(u \vee v) \in \mathcal{I}(t)} h(\bar{u}) \times h(\bar{v})$
ELSE
return $\sum_{(u \vee v) \in \mathcal{I}(t)} (2p_2(\bar{u}) + p_3(\bar{u})) \times (2p_2(\bar{v}) + p_3(\bar{v}))$ (1)
END IF
end

Fig. 1. Function $h(t)$ of the 3-SAT backbone search heuristic BSH

and described in Fig. 1, where t represents the literal x or $\bar{x}$. BSH chooses as a branching variable to be assigned successively 0 and 1 at a current node of the solving tree, one of the variables x having the highest score $h(x) \times h(\bar{x})$.

The computation carried out by $h(t)$ is as follows. t is any literal not assigned a value at a current node of the solving tree and $h(t)$ is the score of t. The evaluation $h(t)$ is based on the set $\mathcal{I}(t)$ of binary clauses such that for each of them if it is FALSE then it forces t to be TRUE unless a contradiction occurs. There are two cases to be considered. The first case is one of the occurrences of t appears in a ternary clause such as $(t \vee u \vee v)$. Then, if the binary clause $(u \vee v)$ is FALSE, as a result t is necessary TRUE so that $(t \vee u \vee v)$ is satisfied. The binary clause $(u \vee v)$ is therefore put in the set $\mathcal{I}(t)$. The second case is one of the occurrences of t appears in a binary clause such as $(t \vee u)$. Then, one searches recursively for the binary clauses which can force $\bar{u}$ to be TRUE. Such binary clauses force consequently t to be TRUE through the unary clause u or a chain of unary clauses. These binary clauses are also put in the set $\mathcal{I}(t)$. In this way, $\mathcal{I}(t)$ is the union of all binary clauses which can force t to be TRUE directly or indirectly (through unary clauses) if they were FALSE.
$h(t)$ is said to be evaluated at the level 1, 2, 3 ... if the integer MAX is set to 1, 2, 3, ... At the level 1, $h(t)$ is given directly by the sum of products (1) in Fig. 1. $p_2(\bar{u})$ and $p_3(\bar{u})$ are the numbers of binary and ternary clauses, respectively, where $\bar{u}$ appears in the sub-formula at the current node where $h(t)$ is evaluated. It is the same for $p_2(\bar{v})$ and $p_3(\bar{v})$. Each term of the sum (1) represents the weighted number of possibilities that a binary clause $(u \vee v)$ of $\mathcal{I}(t)$ is FALSE if one assumes that every occurrence of $\bar{u}$ and every occurrence of $\bar{v}$ can be forced to be TRUE in the clauses where $\bar{u}$ and $\bar{v}$ appear. The weighting of p_2 by 2 is due to the fact that as a rough estimate (for a uniform distribution of assignments) $\bar{u}$ has a double probability to be forced to be TRUE in a binary clause with respect to a ternary clause. On the whole, $h(t)$ can be interpreted as a measure of some correlation between t and all the other variables which can force t to be true, in

the sub-formula at the current node where $h(t)$ is evaluated. This measure can be computed at level 1, 2, 3 ... recursively by setting MAX at 1, 2, 3 .... The measure is considered all the more accurate as the level of recursive computation is high. However it must be noticed that in the computation of $h(t)$, it is not taken into account that the same variable can be met several times with the same sign or opposite signs. One can imagine that this event occurs all the more often as the evaluation level is high. That is the reason why the level of evaluation must be limited in order that $h(t)$ as a measure of some correlation is not distorted (see next section). Finally if t should be TRUE in all assignments satisfying the maximum number of clauses (possibly all) of the sub-formula at the current node where $h(t)$ is evaluated, the correlation that $h(t)$ is intended to measure must be the strongest and then $h(t)$ should have one of the highest values among all values which it can take at the considered node. In this sense $h(t)$ tends to find out the variables belonging to the local backbone of the sub-formula at the considered node.

2.2 Limitations in the Implementation of the Heuristic BSH

There are two limitations which must restrict the use of the heuristic BSH for an efficient implementation. First it can be noticed that if at a current node of the solving tree the sub-formula associated with the considered node has many binary clauses, the computation of $h(t)$ can be too time-consuming with respect to simpler heuristics essentially based on the binary clauses. In *kcnfs* a limitation has been defined empirically as a function of the proportion of binary clauses in a sub-formula. The second limitation concerns the level of evaluation of $h(t)$ defined by the constant MAX. It is due both to the formal argument mentioned in the previous section and again to the cost of the computation. A good limitation must be a compromise between the accuracy of the evaluation which must remain significant so that it is not distorted by variables taken into account more than one time and the cost of the computation. In *kcnfs* this limitation has been defined empirically as a function of the number of variables and the maximum length of the clauses in the formula.

2.3 Effect of BSH Compared with Basic Heuristics

One can give a small example in order to illustrate the quite different effect of the backbone-search heuristic BSH compared with usual rules of choosing of a branching variable which heuristics implemented in many SAT solvers use. In Table 1, an unsatisfiable 3-SAT formula with 7 variables and 13 clauses is given.

Consider the choice of the first variable at the root of the tree to be developed by a DPLL-type algorithm for solving this formula. A basic heuristic consists in choosing a variable having the maximum number of occurrences in minimum size clauses (MOMS for short). All clauses of the formula being of the same length at the root, all variables are regarded equally. In Table 2, the first four highest numbers of occurrences per variable and the corresponding variables are given. An

Table 1. Unsatisfiable 3-SAT formula with 7 variables and 13 clauses

$$(\bar{a} \vee b \vee \bar{f}) \wedge (\bar{a} \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee \bar{b} \vee \bar{g}) \wedge (\bar{a} \vee \bar{c} \vee d) \wedge$$
$$(a \vee \bar{c} \vee d) \wedge (a \vee \bar{c} \vee \bar{d}) \wedge (a \vee b \vee f) \wedge (a \vee b \vee g) \wedge$$
$$(b \vee d \vee \bar{e}) \wedge (b \vee d \vee e) \wedge (b \vee c \vee \bar{d}) \wedge$$
$$(\bar{b} \vee c \vee e) \wedge (\bar{b} \vee c \vee \bar{e})$$

additional criterion, with respect to the previous one for the choice of a branching variable, is also quite commonly used in the heuristics. It is the respective proportions of the positive (uncomplemented) and negative (complemented) occurrences of each variable. A classical heuristic choice tends to maximize both the number of occurrences and the balance of the proportions of opposite literals of the chosen variable. A simple function which tries to combine these two criterions is for example the product of the numbers of occurrences of opposite literals of the variables. The Table 2 lists, according to this product heuristic, the highest four scores which correspond to the same variables as the first four ones according to the MOMS heuristic. Finally as comparison, the highest four scores obtained with BSH applied to all variables of the formula are also listed in Table 2. These highest four scores correspond again to the same variables.

Table 2. comparison of the highest four scores obtained for the variables in the formula of Table 1, between a basic heuristic and BSH

variable	#occ (*pos ; neg*)	Product	BSH
a	8 (*4+ ; 4−*)	16	810
b	9 (*6+ ; 3−*)	18	800
c	7 (*3+ ; 4−*)	12	2016
d	7 (*4+ ; 3−*)	12	1296

As noted, on the whole, for the three heuristics the highest four scores correspond to the same variables. However with the scores of BSH the variables are put in a different order from the one corresponding to the two other basic heuristics and above all, BSH is able to differentiate strongly the four variables between them, while they look alike for the two other heuristics. The latter point is the most important effect of BSH which helps to distinguish better a good branching variable with respect to the size of the solving tree. In our example this is particularly clear because as usual at the root of a solving tree, many variables look very similar for heuristics and this is the case for the four variables in Table 2. However BSH distinguishes the variable c as a much better variable than the three other ones. This is essentially due to the fact that the nature of the evaluation of the variables by BSH is quite different from the classical heuristics. BSH evaluates the correlation of a variable with the other variables, in other words

the message sent by the variables of a formula, through the clauses, to a specific variable to be set at some value.

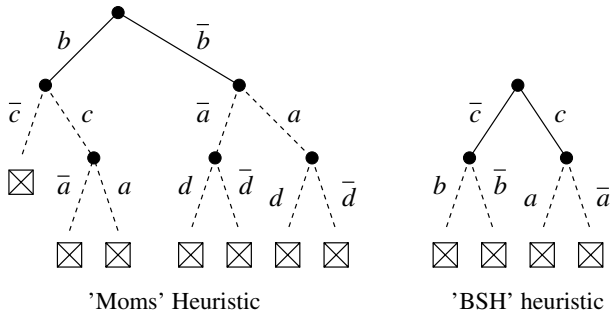

Fig. 2. Solving trees of a DPLL-type procedure developed with MOMS and BSH heuristics on the formula given in Table 1.

Practically the benefit brought by BSH can be seen for our example on the solving trees Fig. 2. At the left hand, one of the possible solving trees developed with the MOMS heuristic is drawn and at the right hand, the root of the tree is chosen as being the variable c selected with BSH and then for the subsequent nodes, each branching variable is chosen also with the MOMS heuristic. As it can be seen for the examples drawn on the Fig. 2, one can check that the size of the tree with the root variable c, chosen with BSH, is smaller than any other tree developed with MOMS. The more sophisticated heuristics implemented in the best current solvers tend to improve the evaluation of the possible branching variables with respect to MOMS but not as BSH can do it. Let us take for example the heuristic of the solver *satz* which is one of the best current solvers for random formulae [5]. The evaluation by this heuristic of the variables of our example Table 1, is given Table 3 in comparison with the Product heuristic and BSH. With the *satz* heuristic the variable c has the best score as with BSH. But the differences of score with the other variables are not as definite as with BSH. For this reason one can easily suppose that on large formulae and not a toy as our illustration example, the *satz* heuristic distinguishes less accurately than BSH the best variables to choose as branching variables. This is clearly confirmed by the comparison of solving performances of hard random formulae related in Section 3. Thus what is seen on the Fig. 2, illustrates what happens on large size formulae even with heuristics more sophisticated than MOMS.

2.4 Local Treatments: Picking and Pickback, in Addition to BSH

As said above, the heuristic BSH is intended to measure in some way the correlation of a variables with the other variables through clauses of a formula. But

Table 3. Comparison of the highest four scores obtained for the variables in the formula of Table 1, between the Product heuristic, *satz* heuristic and BSH

variable	Product	*satz* heuristic	BSH
a	16	504	810
b	18	561	800
c	12	690	2016
d	12	256	1296

this measurement does not take into account two things, the size of the neighborhood containing the correlated variables and the variables for which only one associated literal is strongly correlated with the other variables. If the size of the neighborhood is small a contradiction might be detected with a small tree and then to lead to a backtrack. If one literal associated with a variable is strongly correlated in a small neighborhood then the variable might be set to a fixed truth value by developing also a small tree. Picking and pickback are local treatments to detect such contradictions or such variables to be fixed. These local treatments are only applied on variables having the highest scores of correlation. They consist for the former in a look ahead and for the latter in a look back.

2.5 Performance Comparison Results on Random 3-SAT Formulae

The Tables 4 and 5 list performances comparative results of *cnfs* against several of the best current solvers : *posit* [6], *csat* [7], *satz* [8]and *OkSolver* [9] which was the winner of the SAT 2002 competition[3] in the category of random formulae. For information, we give also the performances of known solvers as *zchaff (v. 2003.7.1)* [10] and *sato* [11] not being specifically devoted to solving random formulae. The performance results are given in terms of mean sizes of trees (Table 4) and mean solving times (Table 5) on sample of 1000 random 3-SAT formulae with 200, 300 and 400 variables at a ratio, clauses to variables, equal to 4.25 which corresponds to the SAT-UNSAT transition area where the formulae are on average the hardest. These results show that *cnfs* outperforms the other solvers. Moreover it is interesting to note that the detailed results showed that *cnfs* was systematically faster than the other solvers on any solved unsatisfiable formula. On hard formulae larger than 400 variables the gap of performance between *cnfs* and the other solvers increases as a function of size of the formula. Finally *cnfs* was able to solve formulae up to 700 variables which were until now out of reach with any other solver.

[3] `http://www.satlive.org/SATCompetition/2002/`

Table 4. Mean tree sizes on hard random 3-SAT formulae having 200, 300 and 400 variables with *sato*, *csat*, *posit*, *satz*, *OkSolver* and *cnfs* solvers

Solvers	SAT unsat (N) sat (Y)	200 V 850 C (482 N) (518 Y) mean #nodes (*std dev.*)	300 V 1275 C (456 N) (534 Y) mean #nodes (*std dev.*)	400 V 1700 C (245 N) (255 Y) mean #nodes in millions (*std dev.*)
zchaff	unsat	57666 (*23708*)	1.7 10^6 (0.9 10^6)	-
	sat	9365 (*11435*)	512687 (*405756*)	-
sato	unsat	14076 (*5225*)	443313 (*109694*)	-
	sat	4229 (*4754*)	151728 (*157102*)	-
csat	unsat	2553 (*997*)	90616 (*37729*)	3.6 (*1.7*)
	sat	733 (*763*)	26439 (*31224*)	1.8 (*2.4*)
OkSolver	unsat	1346 (*430*)	36137 (*12181*)	1.1 (*0.5*)
	sat	314 (*381*)	8454 (*8594*)	0.3 (*0.4*)
posit	unsat	1992 (*754*)	82572 (*35364*)	3.4 (*1.7*)
	sat	789 (*694*)	34016 (*31669*)	1.5 (*1.4*)
satz	unsat	623 (*206*)	18480 (*7050*)	0.5 (*0.2*)
	sat	237 (*216*)	6304 (*6541*)	0.2 (*0.2*)
cnfs	unsat	470 (*156*)	12739 (*4753*)	0.3 (*0.1*)
	sat	149 (*154*)	3607 (*4089*)	0.1 (*0.1*)

Table 5. Mean solving times on hard random 3-SAT formulae having 200, 300, 400 variables with *sato*, *csat*, *posit*, *satz* and *cnfs* solvers

Solvers	SAT unsat (N) sat (Y)	200 V 850 C (482 N) (518 Y) mean time (*std dev.*)	300 V 1275 C (456 N) (534 Y) mean time (*std dev.*)	400 V 1700 C (245 N) (255 Y) mean time (*std dev.*)
zchaff	unsat	18.9s (*21.5s*)	2h 54m (*2h*)	-
	sat	1.4s (*2.1s*)	20m (*22m*)	-
sato	unsat	89.2s (*84.6s*)	18h 27m (*7h*)	-
	sat	13.2s (*38.0s*)	4h (*6h 40m*)	-
csat	unsat	0.5s (*0.2s*)	29.6s (*13.3s*)	29m 01s (*14m 36s*)
	sat	0.1s (*0.1s*)	9.0s (*10.7s*)	7m 47s (*10m 01s*)
OkSolver	unsat	0.6s (*0.2s*)	32.1s (*11.2s*)	26m 03s (*12m 14s*)
	sat	0.1s (*0.2s*)	7.6s (*7.7s*)	5m 58s (*9m 23s*)
posit	unsat	0.3s (*0.1s*)	15.1s (*6.6s*)	13m 07s (*6m 35s*)
	sat	0.1s (*0.1s*)	6.3s (*5.9s*)	5m 56s (*5m 36s*)
satz	unsat	0.2s (*0.1s*)	4.9s (*1.8s*)	2m 44s (*1m 09s*)
	sat	0.1s (*0.1s*)	1.7s (*1.7s*)	1m 05s (*1m 03s*)
cnfs	unsat	0.1s (*0.0s*)	3.7s (*1.3s*)	1m 50s (*0m 43s*)
	sat	0.0s (*0.0s*)	1.1s (*1.2s*)	0m 37s (*0m 40s*)

3 Renormalization in the Search-Backbone Heuristic BSH for Formulae with Clauses Having Various Lengths

3.1 The Principle of the Renormalization Computation

Recall that the function $h(t)$ of BSH described in Section 2.1 is intended to measure some correlation of the literal t with the other variables in the considered sub-formula. This correlation of one occurrence of t depends intuitively on the length of the clause where the considered t appears and on the correlation of the other literals themselves associated with t in this clause. For example in the following clause of length 6 : $(t \vee u_1 \vee u_2 \vee u_3 \vee u_4 \vee u_5)$, t is likely (in a random formula) much less correlated through $(u_1 \vee u_2 \vee u_3 \vee u_4 \vee u_5)$ with the rest of the variables than in the following clause of length 3 : $(t \vee v_1 \vee v_2)$ through $(v_1 \vee v_2)$. In contrast, if in this latter clause v_1 and v_2 are weakly correlated with the other variables of the formula, for example (which is extreme) if they are pure literals, then through a clause of any length, as large as it can be, where t occurs, t is more correlated with the rest of the variables than through $(v_1 \vee v_2)$. We have just identified two factors of correlation of an occurrence of t, the length of the clause where t appears and the correlation of the literals associated with t. This latter factor is naturally taken into account by the level of evaluation defined for $h(t)$ in section 2.1, whose principle extends directly to clauses with various lengths. The first factor is linked to the computation of $h(t)$ itself. We have seen in section 2.1 that this computation is based on the set $\mathcal{I}(t)$ only consisting in clauses of length 2. The number of possibilities that each of the clauses of $\mathcal{I}(t)$ is FALSE, is estimated as the product of two terms associated with each literal of a binary clause. Considering a formula with clauses of various lengths and applying the same principle described in Section 2.1, the set $\mathcal{I}(t)$ contains also clauses of various lengths. This can brings about disparities in estimating the correlations of different occurrences of t with the rest of the variables. For example if two occurrences of t can be forced to be TRUE through two clauses of different lengths, say z_1 and z_2, with $z_2 \geq z_1$, a direct generalization of the computation of $h(t)$ in Section 2.1 could yield a larger number of possibilities that t is forced to be TRUE by the clause having the largest length z_2 than by the clause with the length z_1. This seeming incoherence does not mean that the principle of computation described in Section 2.1 is not sound. The reason is merely that the correlations are not estimated at the same scale and therefore must be normalized. First, we point out that the reason the lengths of clauses in $\mathcal{I}(t)$ are homogeneous and equal to 2 for 3-SAT formulae is that if an occurrence of t occurs in a binary clause associated with only one literal, then binary clauses which could forced t to be TRUE are searched recursively through only a chain of unary clauses preserving thereby the number of possibilities that t is forced to be TRUE. In presence of clauses with various lengths this property no longer holds. More sophisticate structures could be considered such as trees, but the computation of $h(t)$ would turn out probably prohibitive. To overcome the difficulty, we introduce a "fictitious" literal with which we fill the clauses in the set $\mathcal{I}(t)$ such that they have all the same length equal to the largest possible

length of clauses in the sub-formula with which $h(t)$ is evaluated. Then it must be associated a reasonable value with this fictitious literal so that the evaluation of the correlation of t makes sense. We choose to assign to it the mean value of all values computed for the literals in the clauses where t appears. Finally since the literal is fictitious we can choose adequately its truth value as FALSE. Hence there is no uncertainty about its truth value, the probability of being FALSE is 1, whereas the probability for "real" literals to be FALSE or TRUE can be roughly estimated (in a random assignment) at 1/2. Consequently each time the fictitious literal is used its value is weighted by a factor 2. Thus we obtain a complete normalization in the computation of the correlation of t with the rest of variables. The normalization computation is to be done at every node of the solving tree and is specific for each sub-formula associated with the considered node. That is the reason why we say that we have to renormalize in the course of the development of the solving tree.

3.2 The Backbone-Search Renormalized Heuristic BSRH

From a practical viewpoint the backbone-search renormalized heuristic is described by the function $hr(t)$ in Fig. 3.

This presentation of BSRH should not be considered as an implementation (it would be very inefficient), it is just a formal description for the sake of clarity. The computation carried out by $hr(t)$ is as follows. t is any literal not assigned a value at a current node of the solving tree and $hr(t)$ is the score of t. k is the maximum length of the clauses in the sub-formula associated with the considered node.

The evaluation $hr(t)$ is based on the sets $\mathcal{I}_j(t)$ of clauses of length j such that for each of them if it is FALSE then it forces t to be TRUE unless a contradiction occurs. j varies from 2 (using the principle of the chains of unary clauses described in section 2.1) up to at most $k-1$ (the clauses containing at least t). As previously for $h(t)$, $hr(t)$ can be evaluated at the level 1, 2, 3 ... according to a recursive computation, by setting the integer MAX to 1, 2, 3, ... The measure is considered all the more accurate as the level of recursive computation is high. $hr(t)$ is the result of a computation based on every set $\mathcal{I}_j(t)$. At level 1, this computation is given directly by the sum of products (1) in Fig. 3. The literals $\bar{l}_i$ are evaluated by the function $eval(t)$, described in Fig. 4, which is a generalization of the computation in the corresponding sum (1) in Fig. 1. $p_j(t)$ is the number of occurrences of t in the clauses of length j. Each term of the sum (1) represents the weighted number of possibilities that a clause of length j of $\mathcal{I}(t)$ is FALSE if one assumes that every occurrence of its negated literals can be forced to be TRUE in the clauses where they appear. The sum (2) adds all the values associated with every evaluated literal of the clauses in all the sets $\mathcal{I}_j(t)$ and (3) compute the renormalization coefficient. Finally (4) gives the evaluation of $hr(t)$ according to the renormalization computation principle described in the preceding section. it must be noticed that, as for $h(t)$, in the computation of $hr(t)$, it is not taken into account that a same variable can be met several times with the same sign or opposite signs. As said previously in section 2.1, one can

Set $i \leftarrow 0$, let t be a literal, let k be the maximum clause length of the considered formula and let MAX, Sum_{eval} and $Card_{eval}$ be integers.

Integer $hr(t)$

begin

$i \leftarrow i+1$

$Sum_{eval} \leftarrow 0$, $Card_{eval} \leftarrow 0$

FOR $j \in \{2, \ \ldots \ , k-1\}$

compute $\mathcal{I}_j(t)$

IF $i < MAX$

$$\mathcal{S}_j \leftarrow \sum_{(l_1 \vee \ldots \vee l_j) \in \mathcal{I}_j(t)} hr(\bar{l_1}) \times \ldots \times hr(\bar{l_j})$$

$$Sum_{eval} \leftarrow Sum_{eval} + \sum_{(l_1 \vee \ldots \vee l_j) \in \mathcal{I}_j(t)} hr(\bar{l_1}) + \ldots + hr(\bar{l_j})$$

ELSE

$$\mathcal{S}_j \leftarrow \sum_{(l_1 \vee \ldots \vee l_j) \in \mathcal{I}_j(t)} eval(\bar{l_1}) \times \ldots \times eval(\bar{l_j}) \quad (1)$$

$$Sum_{eval} \leftarrow Sum_{eval} + \sum_{(l_1 \vee \ldots \vee l_j) \in \mathcal{I}_j(t)} eval(\bar{l_1}) + \ldots + eval(\bar{l_j}) \quad (2)$$

END IF

$Card_{eval} \leftarrow Card_{eval} + j \times |\mathcal{I}_j(t)|$

END FOR

$$\mathcal{E}(l_v) \leftarrow \frac{Sum_{eval}}{Card_{eval}} \quad (3)$$

return $$\sum_{j \in \{2, \ \ldots \ , k-1\}} \mathcal{S}_j \times \mathcal{E}(l_v)^{k-j-1} \times 2^{k-j} \quad (4)$$

end

Fig. 3. the function $hr(t)$ of the backbone-search renormalized heuristic

imagine that this event occurs all the more often as the evaluation level is high and therefore the level must be limited in order that $hr(t)$, as a measure of some correlation, is not distorted. For that as well as for efficiency reasons, we have applied in the implementation of the heuristic BSRH in *kcnfs* the two types of limitations described in section 2.2.

3.3 Performance Comparison Results on Random 4-SAT, 5-SAT, ... Formulae

In the same line of the comparative experiments carried out for random 3-SAT formulae, we compared the performances of *kcnfs* against most of the solvers used for 3-SAT comparisons, which are also the best known solvers on random k-SAT formulae for $k > 3$. Tables 6, 7 and Tables 8, 9 give the mean tree sizes and the mean computing times respectively on samples of 1000 random 4-SAT and 5-SAT formulae with a ratio clauses to variables around the SAT-UNSAT transition. It appears that apart from *kcnfs*, the other solvers are not in the same order according to their performances relating to the tree sizes or to the

Let t be a literal, let k be the maximum clause length of the considered formula and s be an integer.

```
Integer eval(t)
begin
    s ← 0
    FOR j ∈ {2, ..., k}
        s ← s + (p_j(t) × 2^(k−j))
    END FOR
    return s
end
```

Fig. 4. Final evaluation function of a literal t

computing times. In contrast *kcnfs* has both the best performances relating to the two criterions. Moreover as for 3-SAT formulae, the detailed results showed that *kcnfs* was systematically faster than the others on any solved unsatisfiable formula. Further experiments were carried out on formulae up to 8-SAT against *posit* which appeared to be the best among the other solvers. The Fig. 6 show the curves representing the ratio of the computing times of *cnfs* to those of *posit* as a function of the numbers of variables of formulae from 4-SAT up to 8-SAT with a ratio again around the SAT-UNSAT transition. It can be observed that the steepness of the curves increases as the length of clauses becomes larger. This confirms that the gap of performances of *kcnfs* with the other known solvers enlarges as the length of clauses in random formulae grows.

Table 6. Mean tree sizes on hard random 4-SAT formulae having 100, 130 and 150 variables with the solvers *csat*, *posit*, *satz*, *OkSolver* and *kcnfs* solvers

Solvers	SAT unsat (N) sat (Y)	100 V 988 C (106 N) (94 Y) mean #nodes (*std dev.*)	130 V 1285 C (80 N) (120 Y) mean #nodes in millions (*std dev.*)	150 V 1482 C (85 N) (115 Y) mean #nodes in millions (*std dev.*)
csat	unsat	52080 (*9298*)	0.966 (*0.151*)	6.062 (*0.967*)
	sat	15976 (*16867*)	0.342 (*0.286*)	2.361 (*2.012*)
posit	unsat	38016 (*6443*)	0.744 (*0.124*)	5.496 (*0.906*)
	sat	17396 (*9585*)	0.385 (*0.255*)	2.778 (*1.811*)
satz	unsat	30610 (*5144*)	0.484 (*0.081*)	3.173 (*0.531*)
	sat	9049 (*7076*)	0.159 (*0.128*)	1.249 (*1.048*)
OkSolver	unsat	24989 (*4543*)	0.375 (*0.064*)	2.381 (*0.386*)
	sat	7040 (*7781*)	0.119 (*0.119*)	0.913 (*0.751*)
kcnfs	unsat	10450 (*1740*)	0.174 (*0.026*)	1.111 (*1.903*)
	sat	2616 (*2957*)	0.051 (*0.047*)	0.410 (*0.381*)

Table 7. Mean tree sizes on hard random 5-SAT formulae having 60, 80, 95 variables with *csat*, *posit*, *satz*, *OkSolver* and *kcnfs* solvers

SAT unsat (N) Solvers sat (Y)		100 V 988 C (106 N) (94 Y) mean #nodes (*std dev.*)	130 V 1285 C (80 N) (120 Y) mean #nodes in millions (*std dev.*)	150 V 1482 C (85 N) (115 Y) mean #nodes in millions (*std dev.*)
csat	unsat	52080 (*9298*)	0.966 (*0.151*)	6.062 (*0.967*)
	sat	15976 (*16867*)	0.342 (*0.286*)	2.361 (*2.012*)
posit	unsat	38016 (*6443*)	0.744 (*0.124*)	5.496 (*0.906*)
	sat	17396 (*9585*)	0.385 (*0.255*)	2.778 (*1.811*)
satz	unsat	30610 (*5144*)	0.484 (*0.081*)	3.173 (*0.531*)
	sat	9049 (*7076*)	0.159 (*0.128*)	1.249 (*1.048*)
OkSolver	unsat	24989 (*4543*)	0.375 (*0.064*)	2.381 (*0.386*)
	sat	7040 (*7781*)	0.119 (*0.119*)	0.913 (*0.751*)
kcnfs	unsat	10450 (*1740*)	0.174 (*0.026*)	1.111 (*1.903*)
	sat	2616 (*2957*)	0.051 (*0.047*)	0.410 (*0.381*)

Table 8. Mean computation time on hard random 4-SAT formulae having 100, 130, 150 variables with *csat*, *posit*, *satz*, *OkSolver* and *kcnfs* solvers

SAT unsat (N) Solvers sat (Y)		60 V 1290 C (102 N) (98 Y) mean #nodes (*std dev.*)	80 V 1720 C (123 N) (77 Y) mean #nodes in millions (*std dev.*)	95 V 2042 C (131 N) (69 Y) mean #nodes in millions (*std dev.*)
csat	unsat	33511 (*2210*)	0.721 (*0.054*)	7.246 (*0.602*)
	sat	15629 (*8843*)	0.316 (*0.191*)	2.960 (*1.981*)
posit	unsat	20729 (*1419*)	0.473 (*0.035*)	4.925 (*0.437*)
	sat	11478 (*6652*)	0.236 (*0.148*)	2.989 (*1.673*)
satz	unsat	33645 (*2954*)	0.697 (*0.075*)	6.887 (*0.668*)
	sat	16132 (*9933*)	0.314 (*0.188*)	2.53 (*1.721*)
OkSolver	unsat	19537 (*1397*)	0.364 (*0.023*)	3.310 (*0.286*)
	sat	9046 (*5656*)	0.144 (*0.110*)	1.233 (*1.042*)
kcnfs	unsat	7902 (*543*)	0.147 (*0.011*)	1.456 (*0.125*)
	sat	3096 (*2263*)	0.059 (*0.044*)	0.593 (*0.414*)

3.4 SAT 2003 Competition

kcnfs has been submitted to the SAT 2003 competition in the category of random formulae. In this category, the performances of 34 solvers have been appreciated on 316 benchmarks. 222 out of the 316's have been solved by at least one solver and 30 by only one. As stressed already in the experiments given before in this paper, all solved unsatisfiable benchmarks have been solved by *kcnfs*. Moreover *kcnfs* has solved the greatest number of benchmarks with respect to its competitors. In terms of computing time *kcnfs* has been on average 35% faster than its

Table 9. Mean computation time on hard random 5-SAT formulae having 60, 80, 95 variables with *csat*, *posit*, *satz*, *OkSolver* and *kcnfs* solvers

Solvers	SAT unsat (N) sat (Y)	100 V 988 C (106 N) (94 Y) mean time (*std dev.*)	130 V 1285 C (80 N) (120 Y) mean time (*std dev.*)	150 V 1482 C (85 N) (115 Y) mean time (*std dev.*)
satz	unsat	8.8s (*1.5s*)	188.1s (*119.3s*)	23m 16s (*3m 50s*)
	sat	2.6s (*2.1s*)	62.1s (*49.9s*)	9m 05s (*7m 32s*)
OkSolver	unsat	7.7s (*1.4s*)	166.4s (*26.1s*)	21m 54s (*3m 30s*)
	sat	0.1s (*0.2s*)	7.6s (*7.7s*)	5m 58s (*9m 23s*)
csat	unsat	5.3s (*1.0s*)	119.3s (*18.9s*)	15m 22s (*2m 28s*)
	sat	1.6s (*1.7s*)	42.2s (*35.5s*)	5m 57s (*5m 13s*)
posit	unsat	3.9s (*0.6s*)	92.2s (*15.5s*)	12m 30s (*2m 07s*)
	sat	1.9s (*1.1s*)	43.2s (*30.1s*)	6m 27s (*4m 12s*)
kcnfs	unsat	2.6s (*0.4s*)	57.7s (*8.5s*)	7m 12s (*1m 12s*)
	sat	0.7s (*0.7s*)	16.7s (*15.5s*)	2m 33s (*2m 23s*)

best competitors (respectively for each formula). It has been declared the winner in the random category as a complete solver.

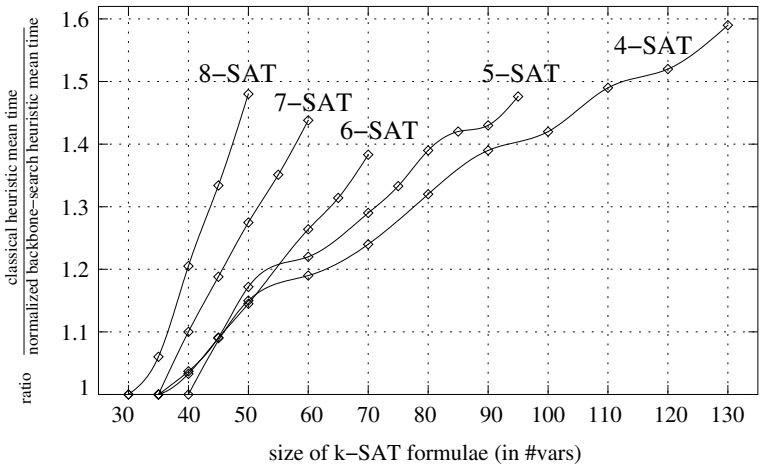

Fig. 5. ratio of mean time of *kcnfs* to mean time of *posit* on hard random k-SAT formulae from $k = 4$ up to $k = 8$

4 Conclusion

The new heuristic presented in this paper has yielded improvements in the performances of solving hard random formulae with any length of clauses, on a scale which had not been observed since a long time ago. This heuristic, as mentioned in Section 2.1, has been designed with respect of the solving of unsatisfiable formulae. Very efficient algorithms processing specifically satisfiable formulae as WALKSAT, [12], or SP, [13], for very large size of formulae, could be combined with our heuristic in a DPLL-type algorithm without damaging significantly the performances on unsatisfiable formulae reported in this paper. It would be a worthwhile operation to enhance on the whole the solving of random formulae. In a more long-term view our feeling is that, in contrast with some pessimistic conclusion having given previously in the literature as in [14], one can expect much more considerable improvements than those reported in this paper, for random as well as for structured formulae. This should come from a much more understanding both of the structure of the space of solutions and the combinatorial structure of formulae. Contributions of various communities could be helpful as progresses already achieved regarding the structure of space of solutions by statistical physics studies in [15, 16].

References

[1] Dubois, O., Dequen, G.: A Backbone Search Heuristic for Efficient Solving of Hard 3-SAT Formulae. In: Proc. of the 17th Internat. Joint Conf. on Artificial Intelligence, Seattle (2001) 248–253
[2] Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: Determining Computational Complexity from Characteristic 'Phase Transitions'. Nature 400 (1999) 133–137
[3] Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal Association for Computing Machine (1960) 201–215
[4] Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem-Proving. Journal Association for Computing Machine (1962) 394–397
[5] Li, C.M., Anbulagan: Heuristics Based on Unit Propagation for Satisfiability Problems. In: Proc. of the 15th Internat. Joint Conf. on Artificial Intelligence, Nagoya (Japan), IJCAI (1997) 366–371
[6] Freeman, J.W.: Hard Random 3-SAT Problems and the Davis-Putnam Procedure. Artificial Intelligence **81** (1996) 183–198
[7] Dubois, O., Andre, P., Boufkhad, Y., Carlier, J.: SAT versus UNSAT. DIMACS Series in Discr. Math. and Theor. Computer Science (1993) 415–436
[8] Li, C.M.: A Constraint-Based Approach to Narrow Search Trees for Satisfiability. Information Processing Letters **71** (1999) 75–80
[9] Kullman, O.: Heuristics for SAT algorithms: A systematic study. In: Extended abstract for the Second workshop on the satisfiability problem (SAT'98). (1998)
[10] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference (DAC'01). (2001)
[11] Zhang, H.: SATO. An Efficient Propositional Prover. In: Proc. of the 14th Internat. Conf. on Automated Deduction, CADE-97, LNCS (1997) 272–275

[12] Selman, B., Kautz, H.A., Cohen, B.: Noise Strategies for Improving Local Search. In: Proc. of the 12th National Conf. on Artificial Intelligence. Volume 1, Menlo Park, CA, USA, AAAI Press (1994) 337–343
[13] A. Braunstein, M. Mezard, R.Z.: Survey propagation: an algorithm for satisfiability. arXiv - cond-mat/0207194 (2002)
[14] Li, C.M., Gerard, S.: On the Limit of Branching Rules for Hard Random Unsatisfiable 3-SAT. In: Proc. of European Conf. on Artificial Intelligence, ECAI (2000) 98–102
[15] G. Biroli, R. Monasson, M.W.: A variational description of the ground state structure in random satisfiability problems. Eur. Phys. J. B 14 551 (2000)
[16] M. Mezard, G. Parisi, R.Z.: Analytic and algorithmic solutions of random satisfiability problems. Science 297 (2002)