CH-SAT: A Complete Heuristic Procedure for Satisfiability Problems

Jin-Kao Hao and Laurent Tétart [1]

Abstract. This paper presents CH-SAT, a Complete Heuristic procedure for the SATisfiability problem (SAT) based on local search techniques. CH-SAT aims to combine the efficiency of local search and the completeness of memorizing-backtracking. CH-SAT extends successively a consistent partial assignment using a complete assignment as guidance for variable-ordering. To extend the partial assignment, CH-SAT uses a two-step selection strategy to determine the next variable: the selection of an unsatisfiable clause followed by the selection of a variable in this clause. If the partial assignment can no longer be extended, it is memorized in the form of a new clause before the search re-starts. Experiments on Dimacs benchmarks show the interest of CH-SAT for solving some classes of hard instances.

1 INTRODUCTION

The satisfiability problem (SAT) [5] is of great importance both in theory and in practice. The statement of the problem is very simple. Given a well-formed boolean expression E, is there a truth assignment that satisfies it? In theory, SAT is one of the six basic core NP-complete problems. In practice, many applications such as VLSI test & verification, consistency maintenance, fault diagnosis, planning and so on can be formulated with SAT. SAT is originally stated as a decision problem. However, there are many interesting related problems. We can mention, among others, 1) model-finding: find satisfying truth assignments; 2) MAX-SAT: if a clausal form formula is unsatisfiable, to find a truth assignment which satisfies the maximum number of clauses, 3) model-counting: find the number of all the satisfying truth assignments.

Existing methods for SAT can be roughly classified into two categories: *complete* and *incomplete* methods. As examples of the first category, we can mention the logic-based approach (Davis-Putnam procedure, Binary decision diagram), the constraint-network-based approach such as CSP techniques, and 0/1 linear programming. The second category includes such methods as local search (simulated annealing, Tabu search, Hill-climbing) and evolutionary or genetic algorithms.

During the last few years, a lot of empirical work has been carried out and much progress has been made. Efficient new procedures based on some surprisingly simple ideas were developed. These procedures are able to solve much larger and harder instances than previous methods. Among the incomplete methods, a good example is the local search procedure GSAT and its variants [14, 15]. Other incomplete procedures based on evolutionary algorithms, simulated annealing, or Tabu search were also reported [2, 7, 16, 8, 4]. At the same time, efforts are being made as regards complete methods. Some efficient implementations based on the combination of the Davis-Putnam procedure and local search techniques have been developed, see for example [3, 11]. Finally, studies on the phase transition phenomena of random SAT instances give a deeper insight into the hardness of this kind of instance [13, 6].

In this paper, we present CH-SAT, a *Complete Heuristic* procedure which tries to combine the advantages of both incomplete and complete methods, i.e., the efficiency of local search and the completeness of memorizing-backtracking. CH-SAT has been prompted by the work of [12, 17, 14] and may be considered as a procedure of the weak-commitment search specialized for the satisfiability problem. The main difference is that CH-SAT uses various heuristics for choosing clauses and variables.

CH-SAT is best viewed as a *weak* form of the informed-backtracking algorithm (IFA) [12]. In essence, it tries to extend successively a consistent partial assignment whenever possible in an informed manner. To do this, CH-SAT uses an (inconsistent) complete assignment and heuristics working on it to make its choice of the next variable. Two differences distinguish CH-SAT from a standard IFA. First, when the partial assignment can no longer be extended, the current partial assignment is first memorized in the form of a new clause (nogood) and then abandoned. The search re-starts with a new complete assignment which incorporates the last consistent partial assignment. This memorizing prevents the recreation of a former abandoned partial assignment. Note that the way in which CH-SAT backtracks is the same as in the weak-commitment search [17], i.e. CH-SAT is not committed to its partial assignment, in contrast to IFA which never gives up completely a partial assignment until all possible backtracks are carried out. Second, to extend the partial assignment, CH-SAT uses a two-step selection strategy to determine the next variable: the selection of an *unsatisfiable* clause followed by the selection of a variable in this selected clause. For each type of selection, many heuristics are possible.

In the following sections, we present the CH-SAT procedure and heuristics for selecting clauses/variables. Some combinations of these heuristics are empirically evaluated using benchmarks of the second Dimacs implementation challenge. The performance of CH-SAT is also contrasted with GSAT (for satisfiable instances) and C-SAT (for unsatisfiable instances), two representatives of incomplete and complete procedures.

[1] LGI2P, EMA-EERIE, Parc Scientifique G. Besse, F-30000 Nîmes, France. Email: {hao,tetart}@eerie.fr

ECAI-96 Workshop on Advances in Propositional Deduction
Budapest, Hungary from August 12 - 16, 1996

Future extensions and improvements are discussed in the last section.

2 THE CH-SAT PROCEDURE

2.1 Notations and Definitions

The following notations are used to simplify the presentation:

- a SAT instance is defined by a set of clauses $\{C_1, C_2 ... C_m\}$,[2] linked by the logic *and*; $X = \{V_1, V_2 ... V_n\}$ is the set of all the variables in E;
- a (complete) assignment I is defined by $I = \{< V_i, v_i > \mid$ for *each* $V_i \in X$ and $v_i \in \{0,1\}\}$, I is also noted as $I = (V_1 = v_1 \wedge ... \wedge V_n = v_n)$;
- for a partial assignment $PS = \{< V, v > \mid$ for *some* $V \in X$ and $v \in \{0,1\}\}$, PS is also noted as $PS = (V_{i_1} = v \wedge ... \wedge V_{i_k} = v_{i_k})$; $X(PS)$ is used to note the set of the variables in PS;
- for a variable V in an assignment I, $T(V)$ gives the truth value of V in I.

Definition 2.1 *A partial assignment PS is said to be consistent if all the clauses instantiated by PS are satisfied. Otherwise, PS is said to be inconsistent.*

Definition 2.2 *A (consistent) partial assignment PS is said to be inextensible if there exists a variable* $V_i \in (X - X(PS))$ *such that both* $< V_i, 0 >$ *and* $< V_i, 1 >$ *make PS inconsistent.*

Definition 2.3 *A variable* V *of* $< V, v > \in PS$ *is said to be forced if PS is consistent and* $PS - \{< V, v >\} \cup \{< V, \neg v >\}$ *is inconsistent. In other words, a forced variable V has the unique consistent value v for the partial assignment.*

2.2 The CH-SAT Procedure

The CH-SAT procedure is complete: for satisfiable formulae, it gives satisfying truth assignments, and for unsatisfiable formulae, it confirms the non-satisfiability. Basically, CH-SAT is a weak form of a backtracking algorithm which is informed by a complete assignment and a set of heuristics. During each iteration, CH-SAT tries to extend successively the current partial assignment (empty at the beginning). To determine the next variable, it uses various heuristics, based on the complete assignment, to select first an unsatisfied clause and then a variable in the clause. If during the search it is no longer possible to extend the consistent partial assignment, the partial assignment is abandoned and its negation is added to the initial formula as a new clause. The search then re-starts with an empty partial assignment and a new complete assignment which incorporates the last consistent partial assignment. The general procedure is given in Fig. 1 below.

We will now give some explanations about the procedure. First, *cost(I)* is an evaluation function used to measure the quality of an assignment I with respect to the given instance E. This function is defined as the total number of clauses (including added ones) that are not satisfied by I. Therefore, a solution is found when the cost has the value 0.

[2] A clause is a disjunction of literals, a literal being a variable or the negation of a variable.

Second, the complete assignment I can be generated randomly or constructed using, for example, a greedy method. Each time the partial assignment PS is extended to include a new pair $< V, v >$ such that $< V, \neg v > \in I$, I is modified to have the new value of V. Therefore, at any moment, I is a superset of PS. In this way, the procedure can terminate when the condition $cost(I) = 0$ becomes true even if PS has not been extended to a complete assignment yet.

Third, the *next variable* to be added to the partial solution PS is chosen by a two-step strategy: choose first an *unsatisfiable* clause and then a variable in this clause. Therefore, the first part of the decision is made *globally* among all the unsatisfied clauses while the second one is made *locally* among one clause. Note that this strategy was first used in WSAT [15], a member of the GSAT family.

```
CH-SAT Procedure

Input: E: a set of clauses
Output: a model if E is satisfiable; Unsatisfiable otherwise
begin
  I ← complete truth assignment
  PS ← empty
  satisfiable ← true
  while (cost(I)>0 and satisfiable=true) do
    choose a clause C that is not satisfied by I
    choose a variable V in C to be added to PS
    (let T(V) be V's current truth value in I)
    if adding < V, ¬T(V) > to PS does not lead to unsat. clause then
      add < V, ¬T(V) > to PS
      replace < V, T(V) > by < V, ¬T(V) > in I
    else
      /*see if < V, T(V) > can be added to PS */
      if adding < V, T(V) > to PS does not lead to unsat. clause then
        add < V, T(V) > to PS
        mark V as being forced;
      else
        /* neither ¬ T(V) nor T(V) can be included in PS
        ⇒ dead-end ⇒ backtrack */
        if PS is not empty then
          construct a clause C from ¬ PS and add C to E
          abandon PS;
          undo the marks of the forced variables
        else
          satisfiable ← false
  if cost(I)=0 and satisfiable=true then
    solution found
  else
    E is unsatisfiable
end
```

Fig. 1: The CH-SAT Procedure

Many heuristics are possible for both choices. Different heuristics lead to different variable-ordering strategies. In the following, some of them are studied.

1. **Heuristics for selecting an *unsatisfied* clause:**
   - **c.1 random A**: take randomly one unsatisfied clause;
   - **c.2 shortest A**: take the shortest unsatisfied clause;
   - **c.3 random B**: take randomly one unsatisfied clause from among added clauses;
   - **c.4 shortest B**: take the shortest unsatisfied clause, with priority for added clauses.
2. **Heuristics for selecting a variable in the chosen clause:**
   - **v.1 random**: take randomly a variable from the chosen clause;
   - **v.2 best-one**: take a variable which gives the greatest decrease in the cost function. If no such variable exists, take the one which leads to the smallest deterioration in the cost function (break ties randomly);
   - **v.3 first-improvement**: take the first variable which improves the cost function. If no such variable exists, take the one which lead to the smallest deterioration in the cost function (break ties randomly);

- **v.4 random-walk**: with probability p, apply the **random** heuristic, with $1-p$, apply the **best-one** or **first-improvement** heuristic.

Evidently, any combination of a **c.x** heuristic and a **v.y** heuristic gives a different selection strategy (there are more than 16 possibilities in our case). It would be interesting to investigate the performance of these combinations. Limited empirical evaluations for some of these combinations are reported later in the paper.

2.3 Construct a New Clause from PS_{inex}

In CH-SAT, when a partial assignment cannot be extended further, a new clause is created from the *negation* of the partial assignment and added to E. The principle is the following. A partial assignment represents a subtree or a specific area in the search space. The fact that a partial assignment can no longer be extended implies that there is no solution in this area. This fact can be memorized as a no-good to prevent the search process from re-visiting this area. One natural way to do this is to memorize the negation of the inextensible partial assignment as a clause. Let PS_{inex} be such an inextensible partial assignment, then the associated clause $C(\neg PS_{inex})$ can be constructed as follows.

Since $\neg PS_{inex} \Leftrightarrow \neg(V_{i_1} = v_{i_1} \wedge ... \wedge V_{i_k} = v_{i_k}) \Leftrightarrow (\neg V_{i_1} = v_{i_1} \vee ... \vee \neg V_{i_k} = v_{i_k}) \Leftrightarrow (V_{i_1} = \neg v_{i_1} \vee ... \vee V_{i_k} = \neg v_{i_k})$, therefore, $C(\neg PS_{inex}) = L_{i_1} \vee ... \vee L_{i_k}$ is defined as follows: $L_{i_j} = V_{i_j}$ $(1 \geq j \geq k)$ if $v_{i_j} = 0$; $L_{i_j} = \neg V_{i_j}$ if $v_{i_j} = 0$. By definition, $C(\neg PS_{inex})$ is not satisfied by the current PS_{inex}. It is easy to see that any assignment which satisfies E $\wedge C(\neg PS_{inex})$ must satisfy E.

An equivalent but simpler $C(\neg PS_{inex})$ may be obtained due to forced variables. The interest of such a simplification lies in the fact that the smaller a clause is, the larger the area represented by the clause in the search space. For instance, a unit clause represents half of the search space. The simplification can be achieved thanks to *forced variables*. It should be remembered that a forced variable V in PS_{inex} has a unique consistent value v. The other value $\neg v$ is impossible for PS_{inex}. More precisely, the following theorem can be proven and be used to construct a simpler $C(\neg PS_{inex})$.

Theorem 2.1 *Let E be a SAT instance, and PS_{inex} an inextensible partial solution for E, then E is satisfiable (unsatisfiable) if and only if, for any $< V_{i_k}, v_{i_k} > \in PS_{inex}$ such that V_{i_k} is a forced variable, $E \wedge C(PS_{inex} - \{< V_{i_k}, v_{i_k} >\})$ is satisfiable (unsatisfiable).*

Using this theorem, we obtain the following procedure to construct a reduced clause $C(\neg PS_{inex})$ for PS_{inex}.

1. Delete all forced variables from PS_{inex}, leaving $PS_{inex} = (V_{i_1} = v_{i_1} \wedge ... \wedge V_{i_k} = v_{i_k})$,
2. Construct $C(\neg PS_{inex}) = L_{i_1} \vee ... \vee L_{i_k}$ where $L_{i_j} = V_{i_j}$ $(1 \geq j \geq k)$ if $v_{i_j} = 0$; $L_{i_j} = \neg V_{i_j}$ if $v_{i_j} = 1$.

2.4 Termination, Completeness, Soundness and Complexity

Termination: The proof of termination is based on three points.

- each iteration extends the partial assignment if it is possible,
- the number of partial assignments is finite,
- each inextensible partial assignment is different from those already encountered.

The first two points being evident, we will show only the third one. Let i be the current iteration number, PS_i the current *consistent* inextensible partial assignment. The fact that PS_i can no longer be extended implies that there exists a chosen V such that including either $< V, 0 >$ or $< V, 1 >$ in PS_i gives an *inconsistent* PS_{i+1}. Suppose now that there exists a $j < i$ such that $PS_i = PS_j$, then during j^{th} iteration, a clause $C(\neg PS_j)$ has been added to E. Therefore, $C(\neg PS_j)$ is not satisfied by $PS_i (= PS_j)$, i.e. PS_i is inconsistent, which is a contradiction. Consequently, PS_i must be unique.

In practice, the procedure stops either when a satisfying assignment is found or when a contradiction (a unit clause and its negation) is found[3].

Completeness: The completeness is a direct consequence of the previous proof. In fact, the proof shows that in the worst case all inextensible partial solutions are enumerated and corresponding clauses added. Therefore, if E is unsatisfiable, then sooner or later a unit clause and its negation will be found among added clauses.

Soundness: For the satisfiable part, the search stops when $cost(I) = 0$. This means that all the clauses (including added ones) of E are satisfied by I. Consequently, a satisfying truth assignment is found. For unsatisfiable instances, the stop condition implies that no non-empty partial solution is possible. In other words, both a unit clause and its negation are found. Given the completeness of the procedure, this stop condition will be reached in finite time.

Complexity: The time complexity of the procedure is evidently an exponential function of n, the number of the variables of E, because in the worst case, all the possible partial assignments will be enumerated before a contradiction is detected. Since inextensible partial assignments are memorized as nogoods, the procedure has a worst-case exponential space complexity.

3 EXPERIMENTAL RESULTS

In this section, we compare first some combinations of heuristics introduced in section 2.2 for clause/variable selection and then contrast CH-SAT[4] with GSAT and C-SAT, two representative procedures for finding models and proving unsatisfiability. Note, however, that the main goal of this work is to present CH-SAT and to study its heuristics.

Tests were carried out on a set of well-known SAT benchmarks representing different *types* of problems (planning, random, bridge-fault, induction, coloring ...), essentially from the archives of the second Dimacs Implementation Challenge[5].

Results presented here are the mean of 10 independent runs, except for some vary hard instances. For each tested instance, we give the number of iterations *Iter* and user times *Time* on a Silicon PowerMachine L (Processor R8000, 75Mhz

[3] if there are no such clauses in E, they are necessarily constructed and are among the added clauses.

[4] In fact, a C++ implementation of the CH-SAT procedure.

[5] Dimacs Benchmarks can be obtained by anonymous ftp from: *ftp.dimacs.rutgers.edu*.

with 128Mo of RAM) for successful runs. If a run fails to find a solution within a fixed number of iterations (maximum fixed to be 1,500,000), *NO* is given followed by brackets showing the time spent by the run.

The performance of CH-SAT is largely controlled by the heuristics for selecting clauses/variables. As discussed in Section 2.2, many combinations are possible. In what follows, we will present some representative combinations. It should be noted that making a general assertion about the performance of heuristics is a difficult task. Consequently, the results concerning the relative performance of heuristics should be interpreted with precaution. However, we hope that empirical tests will help to disclose some intrinsic properties of the heuristics as regards different classes of problems.

3.1 Comparison of Heuristics for Selecting Clauses

We have chosen to compare the following two heuristics for selecting clauses: **c.1 random** and **c.2 shortest**. They were tested with the **v.2 best-one** heuristic for selecting variables. Tables 1 and 2 show the comparative results for the two chosen heuristics for satisfiable and unsatisfiable instances.

Instance	*Random clause*		*The shortest clause*	
Name	Iter.	Time	Iter.	Time
aim-100-2_0-1	65561	1'51"	42593	28"
aim-200-2_0-2	max	NO(5h19')	294883	8'3"
medium	796	0"20	251	0"14
hanoi	59873	53"	14627	5"
huge	83284	1'45"	15374	7"
ii16a1	741	0"68	624	0"90
ii32b3	1262	1"41	1753	1"55
mwff.300	max	NO(2h40')	max	NO(2h18')
par8-2-c	737	0"10	9907	2"14
par8-4-c	592	0"9	1842	0"24
ssa7552-038	407518	58"41	131231	37"4
ssa7552-158	351809	12'7"	34429	9"66

Table 1. Comparisons of clause heuristics for sat. instances

From the data, we see that the **c.1 shortest** heuristic works better than the **c.1 random** heuristic in terms of iterations and solving time for most of the tested instances. This is especially true for unsatisfiable instances. The main reason for this is that the **c.1 shortest** heuristic informs better the search about which direction to take than the **c.1 random** heuristic does. The following observation helps to understand this point, at least for satisfiable instances.

If we trace the cost evolution, i.e. the number of unsatisfied clauses as a function of the number of iterations, we observe that both heuristics are able to reduce rapidly the cost after a relatively small number of iterations (descending phase), followed by a long series of up-down moves (oscillation phase). The first difference between these two heuristics is that the **c.1 shortest** heuristic goes faster (requires much smaller iterations) during the descending stage. The main difference is that **c.1 shortest** focuses better its search direction during the oscillation stage. In fact, the **c.1 random** heuristic oscillates too much to be able to concentrate on any search area after the descending phase.

We have also observed that the number of added clauses of **c.1 shortest** tends to be smaller than that of **c.1 random**. In other words, **c.1 shortest** backtracks less frequently than **c.1 random** does. As to the length of added clauses, it is usually big (from 10 to 60) for both heuristics. Consequently, added clauses have little chance to be selected by the **c.1 shortest** heuristic.

Instance	*Random clause*		*The shortest clause*	
Name	Iter.	Time	Iter.	Time
aim-100-2_0-1	9122	6"	5559	2"
aim-200-2_0-2	815086	5h41'	87234	3'6"
jnh202	368601	1h10'	421106	41'26"
jnh302	43801	58"	36165	25"
dubois20	max	NO(2h)	1262335	6h14'
bf2670-001	max	NO(1h16')	max	NO(2h59')

Table 2. Comparisons of clause heuristics for unsat. instances

Note that the performance of the **c.1 shortest** heuristic in CH-SAT is consistent with its performance found in other complete procedures.

3.2 Comparison of Heuristics for Selecting Variables

Similar tests were also carried out on heuristics for selecting variables in a clause. In particular, we have compared the following three heuristics: **v.1 random**, **v.2 best-one** and **v.3 first-improvement** in conjunction with **c.2 shortest** for selecting clauses. The same set of instances and criteria used above for testing clause heuristics were used here.

From the experimental data (not shown here), we observed that although the performance of these heuristics is mixed for the tested instances, it seems that **v.2 best-one** and **v.3 first-improvement** give slightly better results on average.

To have a closer look at the behavoir of these heuristics, we traced once again the evolution of the cost function for each heuristic for some instances. First, we observed that the descending phase of **v.2 best-one** and **v.3 first-improvement** is faster (sharper) than that of **v.1 random**. In other words, to reach a given number of satisfied clauses, **v.1 random** usually needs more iterations. Second, the oscillation phase of **v.2 best-one** is smoother and less chaotic than that of **v.3 first-improvement**, and the oscillation phase of **v.3 first-improvement** is smoother and less chaotic than that of **v.1 random**. This is not surprising. In fact, while **v.2 best-one** and **v.3 first-improvement** guide the search toward improving moves, **v.1 random** may lead to deteriorating moves at any step during the search. Consequently, **v.1 random** may abandon a promising path.

After all, the performance of these heuristics varies according to the type of problem. Indeed, a guided heuristic such as **v.2 best-one** or **v.3 first-improvement** may be stuck in deep local optima.

Note finally that the difference of performance of these variable heuristics is less significant than that for heuristics for selecting clauses. This may be explained by the fact that a clause selection is a *global* decision made among all the unsatisfied clauses while selecting a variable is a local decision limited to a clause. Therefore, the first choice has more influence on the performance of the procedure. This is particularly true if the selected clauses are short. This is what happens with the **c.2 shortest** heuristic we used.

3.3 Results and Comparisons

This section lists results of CH-SAT for some Dimacs benchmarks. Most of these results are obtained with the **c.2 shortest/v.2 best-one** combination, and the only parameter asked is therefore the maximal number of iterations. In our experiments, this number is usually fixed at 1,500,000.

3.3.1 Satisfiable Instances

Table 3 shows the results of CH-SAT for satisfiable instances.

Instances			*CH-SAT*		
Name	Var.	Cl.	Iter.	Add.Cl	Time
aim-100-2_0-yes-1	100	200	42593	2017	28"49
aim-200-2_0-yes-2	200	400	294883	10360	8'3"
aim-200-2_0-yes-3	200	400	244117	15137	31'40"
aim-200-2_0-yes-4	200	400	334586	19137	42'4"
medium	273	2311	251	7	0"14
hanoi	417	2561	14627	790	5"20
huge	937	14519	15374	672	6"83
ii16a1	1650	19368	744	1	0"68
ii16b1	1728	24792	25989	221	14"2
ii16c1	1580	16467	809	3	0"83
ii16d1	1230	15901	883	5	0"89
ii16e1	1245	14766	2208	28	2"22
ii32b3	348	5734	532	11	0"75
f200	200	860	67830	2367	14'35"
f400	400	1700	206307	5119	58'4
par8-2-c	68	270	737	36	0"10
par8-4-c	67	266	592	27	0"9
ssa7552-038	1501	3575	131231	1409	37"4
ssa7552-158	1363	3034	51170	823	11"35
ssa7552-159	1363	3028	18738	439	3"38
ssa7552-160	1391	3126	27824	398	4"96

Table 3. Results for satisfiable instances

From the data in this table and some results not shown here, two general remarks can be made about the performance of CH-SAT.

1. CH-SAT solves the instances of the following classes:
   - *aimxxx*: artificially generated 3-SAT instances having exactly one solution,
   - *iixxx*: instances from inductive inference,
   - *jnhxxx*: random instances which are generated by rejecting unit clauses,
   - *medium, hanoi and huge*: instances from planning,
   - *ssaxxx*: instances from circuit fault analysis,
   - *fxxx*: satisfiable hard random instances until 400 variables.
2. CH-SAT has difficulty solving the following classes of instances or some instances of these classes:
   - *fxxx*: large and hard random instances having more than 400 variables,
   - *gxxx*: graph coloring problems,
   - *par16xx* and *par32xx*: problems in learning the parity function.

As put forward in [9], comparing heuristic procedures is very difficult and may be meaningless. However, it will be useful to have some indication about the performance of CH-SAT with respect to other well-known procedures. For this purpose, we have compared the above results with the results of GSAT reported in the literature as well as those we obtained by running GSAT. We can make the following remarks:

- GSAT fails to solve the *aimxxx* family which is easy for CH-SAT to solve,
- For the instances solved by both CH-SAT and GSAT, CH-SAT generally requires far few iterations (flips for GSAT) than GSAT,
- GSAT solves much larger random instances than CH-SAT, thanks to its *Random-Walk* strategy which exploits well the characteristics of these classes of instances,
- GSAT solves the graph coloring instances thanks to the *Clause Weights* and *Averaging in*" strategies.

Although CH-SAT fails to solve some instances which can be solved by GSAT, it is remarkable for CH-SAT to give comparable or better results on many classes of problems since GSAT is a specialized, highly optimized procedure for finding models.

3.3.2 Unsatisfiable instances

Table 4 shows results of CH-SAT for unsatisfiable instances.

Instances			*CH-SAT*		
Name	Var.	Cl.	Iter.	Add.Cl	Time
aim-100-2_0-no-1	100	200	9122	869	5'70"
aim-200-2_0-no-1	200	400	133570	10360	10'10"
aim-200-2_0-no-2	200	400	99318	15137	4'20"
aim-200-2_0-no-3	200	400	28804	19137	35"
aim-200-2_0-no-4	200	400	63233	4910	3'03"
bf2670-001	1393	3434	966774	26891	2h28'48"
jnh2	100	850	448147	25252	1h13'44"
jnh302	100	800	284043	17074	34'04"
jnh302	100	900	12402	821	4'95"
dubois20	60	160	792941	34729	3h33'41"
dubois21	63	168	845671	35723	3h37'01"

Table 4. Results on unsatisfiable instances

From the data in Table 4 and some results not shown here, two general remarks can be made about the performance of CH-SAT.

1. CH-SAT easily solves the following two classes: *aimxxx* and *jnhxxx* and manages to solve one *bfxxxx* and two of the *duboisxx* instances,
2. CH-SAT has difficulty solving the *pretxxx* (2-coloring) class and some large instances of the classes *duboisxx, ssaxxx* and *bfxxx*.

Compared with the results of C-SAT reported in [3] and those we observed by running C-SAT, the current CH-SAT procedure gives poorer results except for the *aimxxx* family. This seems not surprising given that C-SAT is a specialized, highly optimized procedure for detecting the non-existence of solutions. Note also that C-SAT solves some very hard problems such as the *duboisxxx* and *prexxx* instances with the help of advanced techniques such as *symmetry detection* proposed in [1].

4 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented CH-SAT, a complete procedure based on local search heuristics for the satisfiability problem. Some of the combinations of heuristics for selecting clauses/variables have been empirically evaluated. We observed that it is difficult to compare objectively these combinations since very often the performance of a given combination depends on the type of instance to be solved, i.e. the intrinsic structures of these instances. However, we observed that the combination **shortest clause/best (or first-improvement) variable** gives the most consistent results for tested instances.

Preliminary results on the second Dimacs benchmarks show the interest of this procedure for both finding models and proving unsatisfiability. In fact, for satisfiable instances, CH-SAT gives good results for a wide range of problem instances. For unsatisfiable instances, although the current implementation of CH-SAT is not as good as C-SAT, a very efficient procedure specialized in proving unsatisfiability, it gives interesting results for many non-trivial instances.

At the same time, some classes of problems remain unsolvable with CH-SAT. This is the case for large (and hard) *random* and *graph coloring* satisfiable instances, and for some *pretxxx, duboisxx* and *ssaxxx* unsatisfiable instances.

Many possibilities exist for improving or extending CH-SAT. We can mention:

1. **Specialization**: As put forward in [3], searching for solutions and proving the non-existence of solutions are two very different tasks. In fact, the search has to implicitly visit the whole space to prove unsatisfiability while this is not necessary for finding a solution. Specialization has shown its power with C-SAT and GSAT. It would be interesting to specialize CH-SAT for these two different tasks.
2. **One-step selection strategy**: Like other procedures, this alternative to the current two-step selection strategy determines directly the next variable. Once again, many heuristics may be used. We can mention, for instance, 1) random: pick randomly a variable occurring in an unsatisfied clause; 2) the best (min-conflict): pick the variable which gives the greatest decrease in the total number of unsatisfied clauses; 3) next-better: take the first variable which gives any decrease in the total number of unsatisfied clauses; 4) most-constrained: pick the variable which occurs in the biggest number of (unsatisfied) shortest clauses; 5) random-walk: with probability p, pick randomly a variable occurring in an unsatisfied clause; with $1-p$, pick the best.
3. **Learning**: A fundamental element underlying any intelligent problem-solving system is the use of flexible memory. Flexible memory embodies the dual process of creating and exploiting structures in order to take advantage of historical information. Memorizing inextensible partial assignments as clauses may be considered as a primitive learning mechanism. More advanced mechanisms are certainly necessary to tackle very hard problems. The basic idea is that the procedure stores and uses pertinent historical information to better orient the search towards particular regions either to find solutions or to prove the non-existence of solutions.
4. **Clause simplification**: CH-SAT requires many memory to record supplementary clauses. Indeed, it has an worst-case exponential space complexity. Therefore, it should be interesting if some clauses may be simplified or even suppressed during the search.

Similar work of embedding local search into backtracking procedures has been reported, see for instance [3, 11]. The main difference is that they integrate local search into the Davis-Putnam procedure. In this paper, we have given an alternative way of realizing this integration, i.e. embedding local search into a very simple informed backtracking framework.

To conclude, we make some general remarks on heuristics-based search procedures. The performance of a heuristics depends largely on its capacity to exploit the structure or characteristics of problems. A heuristic has *a priori* some "favorite" problems. Therefore, a long-term goal of our work is to look for a better understanding of the behavior of heuristic procedures such as CH-SAT. This may consist in identifying the characteristics of problems that may be efficiently exploited by the given heuristic procedure.

ACKNOWLEDGEMENTS

We would like to thank O. Dubois of the University of Paris 6, B. Selman and H. Kautz of AT&T, and L. Sais of the University of Lens for making respectively C-SAT, GSAT and DPTWSAT available to us, and the referees for their comments on the paper.

REFERENCES

[1] Benhamou B and Sais L. 'Theoretical Study of Symmetries in Propositional Calculus and Applications', Proc. 11th Conf. on Automated Deduction, Springer-Verlag, 1992, pp281-294.

[2] De Jong K.A. and Spears W.M. 'Using Genetic Algorithms to Solve NP-Complete Problems'. Proc. of Intl. Conf. on Genetic Algorithms, Fairfax, Virginia, June 1989, pp124-132.

[3] Dubois O., André P., Boufkhad Y. and Carlier J. 'SAT versus UNSAT'. in [10].

[4] C. Fleurent and J.A. Ferland, "Object-oriented Implementation of Heuristic Search Methods for Graph Coloring, Maximum Clique, and Satisfiability". in [10]

[5] Garey M.R. & Johnson D.S. 'Computers and Intractability: a Guide to the Theory of NP-Completeness'. Freeman, San Francisco, CA, 1979.

[6] Gent I.P. and Walsh T. 'The SAT Phase Transition'. Proc. of ECAI'94, John Wiley & Sons, Amsterdam, The Netherlands, August 1994, pp105-109.

[7] Hao J.K. and Dorne R. 'A Population-based Method for Satisfiability Problems'. Proc. of ECAI-94, John Wiley & Sons, Amsterdam, The Netherlands, August 1994, pp135-139.

[8] Hensen P. and Jaumard B., Algorithms for the maximum satisfiability problem, Computing Vol.44, pp279-303, 1990.

[9] Hooker J.N. 'Testing Heuristics: We Have it all Wrong'. J. of Heuristics, Vol.1, No.1, 1996.

[10] Johnson D.S. and Trick M.A. (eds.) '2nd DIMACS Implementation Challenge: Cliques, Coloring and Satisfiability'. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26, 1996.

[11] Mazure B., Sais L. and Grégoire E. 'Boosting Complete Techniques Thanks to Local Search Methods'. Workshop on Practical Solving of SAT at ICCP'95, Marseille, France, 1995.

[12] Minton S., Johnston M.D. Philips A.B. and Laird Ph. 'Minimizing Conflicts: A Heuristic Repair Method for Constraint-Satisfaction and Scheduling Problems', Journal of Artificial Intelligence, Vol.58 No.1-3, 1992, pp.161-206.

[13] Mitchell D., Selman B. and Levesque H.J. 'Hard and Easy Distributions of SAT Problems'. Proc. of AAAI-92, San Jose, CA, 1992, pp.459-465.

[14] Selman B., Levesque H.J., and Mitchell M. 'A New Method for Solving Hard Satisfiability Problems'. Proc. of AAAI-92, San Jose, CA, 1992, pp.440-446.

[15] Selman B., Kautz H.A and Bram C. 'Noise Strategies for Improving Local Search'. Proc. of AAAI-94, Seattle, WA, July 1994.

[16] Spears W.M. 'Simulated Annealing for Hard Satisfiability Problems'. in [10].

[17] Yokoo M. 'Weak-Commitment Search for Solving Constraint Satisfaction Problems' Proc. of AAAI-94, Seattle, WA, July 1994, pp313-318.