International Journal of Computational Intelligence and Applications

Systematic versus Local Search and GA Techniques for Incremental SAT

MALEK MOUHOUB
Department of Computer Science, University of Regina
3737 Wascana Parkway,
Regina Saskatchewan, Canada, S4S 0A2
email : mouhoubm@cs.uregina.ca

Propositional satisfiability (SAT) problem is fundamental to the theory of NP-completeness. Indeed, using the concept of "polynomial-time reducibility" all NP-complete problems can be polynomially reduced to SAT. Thus, any new technique for satisfiability problems will lead to general approaches for thousands of hard combinatorial problems. In this paper, we introduce the incremental propositional satisfiability problem that consists of maintaining the satisfiability of a propositional formula anytime a conjunction of new clauses is added. More precisely, the goal here is to check whether a solution to a SAT problem continues to be a solution anytime a new set of clauses is added and if not, whether the solution can be modified efficiently to satisfy the old formula and the new clauses. We will study the applicability of systematic and approximation methods for solving incremental SAT problems. The systematic method is based on the branch and bound technique while the approximation methods rely on stochastic local search (SLS) and genetic algorithms (GA). A comprehensive empirical study, conducted on a wide range of randomly generated consistent SAT instances, demonstrates the efficiency in time of the approximation methods over the branch and bound algorithm. However these approximation methods do not guarantee the completeness of the solution returned. We show that a method we propose that uses non systematic search in a limited form together with branch and bound has the best compromise, in practice, between time and the success ratio (percentage of instances completely solved).

Keywords: Propositional Satisfiability, Local Search, Genetic Algorithms, Branch and Bound.

1. Introduction

A boolean variable is a variable that can have one of two values: true or false. If X is a boolean variable, $\neg X$ is the negation of X. That is, X is true if and only if $\neg X$ is false. A literal is a boolean variable or its negation. A clause is a sequence of literals separated by the logical or operator ($\vee$). A logical expression in conjunctive normal form (CNF) is a sequence of clauses separated by the logical and operator ($\wedge$). For example, the following is a logical expression in CNF:

$$(X_1 \vee X_3) \wedge (\neg X_1 \vee X_2) \wedge \neg X_3$$

The CNF-Satisfiability Decision Problem (also called SAT problem) is to determine, for a given logical expression in CNF, whether there is some truth assignment (set of assignments of true and false to the boolean variables) that makes the expression true. For example, the answer is "yes" for the above CNF expression since the truth assignment

$\{X_1 = true, X_2 = true, X_3 = false\}$ makes the expression true. SAT problem is fundamental to the theory of NP-completeness. Indeed, using the concept of "polynomial-time reducibility" all NP-complete problems can be polynomially reduced to SAT [a]. This means that any new technique for SAT problems will lead to general approaches for thousands of hard combinatorial problems.

One important issue when dealing with SAT problems is to be able to maintain the satisfiability of a propositional formula anytime a conjunction of new clauses is added. That is to check whether a solution to a SAT problem continues to be a solution anytime a set of new clauses is added and if not, whether the solution can be modified efficiently to satisfy the old formula and the new clauses. More formally, we define a dynamic SAT problem as a sequence of static SAT problems $SAT_0, \ldots, SAT_i, SAT_{i+1}, \ldots, SAT_n$ each resulting from a change in the preceding one imposed by the "outside world". This change can either be a restriction (adding a new set of clauses) or a relaxation (removing a set of clauses because these later clauses are no longer interesting or because the current SAT has no solution). In this paper we will focus only on restrictions (we talk then about incremental SAT). More precisely, SAT_{i+1} is obtained by performing an addition of a set of clauses to SAT_i. We consider that SAT_0 (initial SAT) has an empty set of clauses. An incremental SAT over a set X of boolean variables is a sequence of static SAT problems using only the variables in X. Solving an incremental SAT problem consists of maintaining the satisfiability of the related static SAT problems anytime a new set of clauses is added. To illustrate this, let us consider the following example. We assume we have the following SAT formula: $(X_1 \vee X_2) \wedge (\neg X_1 \vee \neg X_2)$ and that when using a given solving method we obtain the following solution: $\{ X_1 = true, X_2 = false \}$. Let us assume now that we add the clause $\neg X_1$ to our SAT formula. As we can see, the solution obtained for the old SAT formula is no longer a solution for the new formula $(X_1 \vee X_2) \wedge (\neg X_1 \vee \neg X_2) \wedge \neg X_1$. We need here to restart the search in order to find a solution for the old and new clauses. In our case, the following satisfies the new formula: $\{ X_1 = false, X_2 = true \}$. Our aim here is, instead of restarting the search from scratch, we use some of the effort made to solve the old formula in order to find a solution for the new one.

Dealing with incremental SAT problems is relevant in so many real world discrete combinatorial problems including reactive scheduling and planning, timetabling, resource allocation, conceptual design, network management and configuration, and interactive graphic. One example, in the case of scheduling problems, is when a solution, corresponding to an ordering of tasks to be processed, has to be reconsidered after a given machine becomes unavailable. We have then to look for another solution (ordering of tasks) satisfying the old constraints and taking into account the new information. Another example, in the area of engineering conceptual design, is when the designers add constraints after specifying an initial statement describing the desired properties of a required artifact during the conceptual phase of design.

In this paper, we will investigate different systematic and approximation methods for solving the SAT problem in an incremental way. The systematic method is a branch and

[a] We will refer the reader to the paper published by Cook[1] proving that if CNF-Satisfiability is in P, then P = NP.

bound technique based on the Davis-Putnam-Loveland algorithm (DPLL) [2,3,4,5,6,7]. More precisely, we use here the well known zChaff solver [7] including the most advanced heuristics improving the DPLL algorithm. The second method relies on stochastic local search [8,9,10,11]. Indeed the underlying local search paradigm is well suited for recovering solutions after local changes (addition of constraints) of the problem occur. We adopt here the Scaling and Probabilistic Smoothing (SAPS) algorithm [12] from the UBCSAT experimentation environment [11]. The third method, based on genetic algorithms [13], is similar to the second one except that the search is multi-directional and maintains a list of potential solutions (population of individuals) instead of a single one [14,15,16,17,18,19,20]. This has the advantage to allow the competition between solutions of the same population which simulates the natural process of evolution. Experimental comparison of the time performance of the different methods have been conducted on randomly generated consistent SAT instances. With no surprise, the results favor the approximation methods (especially SLS) over the systematic one (branch and bound). The approximation methods however do not guarantee the correctness of the solution provided. The most relevant and new result we show is that a method we propose that uses non systematic search in a limited form together with the branch and bound method has the best compromise, in practice, between time cost and success ratio (percentage of instances completely solved). Indeed, while this hybrid method does guarantee, in practice, the completeness of the solution returned, the time returned by this technique is comparable to the running time of the approximation methods.

Related work on solving SAT problems in an incremental way has already been reported in the literature. These methods rely solely on SLS [21,22] or systematic search (backtrack search or branch and bound) [23,24,25] and handle the addition of one clause at a time. Our goal, in this paper, is to explore and compare different systematic and approximation methods to tackle the dynamic satisfiability problem. Also, as we will see in the next section, our method handles the addition of more than one clause at a time and depends on the structure of the CNF formula. Note that incremental SAT problems can be solved using the dynamic CSP paradigm. Indeed CSP[b] is formally equivalent to SAT as it is well indicated in [27,28]. Managing dynamic CSPs has already been reported in the literature [29,30,31,32,33,34,35,36]. [29] introduced the notion of *Dynamic Constraint Satisfaction Problems* for configuration problems (renamed *Conditional Constraint Satisfaction Problems (CCSPs)* later). In contrast with the standard CSP paradigm, in a CCSP the set of variables requiring assignment is not fixed by the problem definition. A variable has either *active* or *nonactive* status. An activity constraint enforces the change of the status of a given variable from *nonactive* to *active*. In [30], Freuder and Sabin have extended the traditional CSP framework by including the combination of three new CSP paradigms : *Meta CSPs, Hierarchical Domain CSPs, and Dynamic CSPs*. This extension is called *composite CSP*. In a composite CSP, the variable values can be entire sub CSPs. A domain can be a set of variables instead of

[b] A CSP consists of a finite set of variables with finite domains, and a finite set of constraints restricting the possible combinations of variable values [26]. Solving a CSP consists of obtaining a set of values of variables satisfying all the constraints.

atomic values (as it is the case in the traditional CSP). The domains of variable values can be hierarchically organized. The participation of variables in a solution is dynamically controlled by activity constraints. Jónsson and Frank [34] proposed a general framework using procedural constraints for solving dynamic CSPs. This framework ha s been extended to a new paradigm called Constraint-Based Attribute and Interval Planning (CAIP) for representing and reasoning about plans [35]. CAIP and its implementation, the EUROPA system, enable the description of planning domains with time, resources, concurrent activities, disjunctive preconditions and conditional constraints. The main difference, comparing to the formalisms we described earlier, is that in this latter framework [34] the set of constraints, variables and their possible values do not need to be enumerated beforehand which gives a more general definition of dynamic CSPs. Note that the definition of dynamic CSPs in [34] is also more general than the one in [33] since in this latter work variable domains are predetermined.

In the next section we present a general procedure we propose for solving Incremental SAT. Sections 3, 4 and 5 are respectively dedicated to the systematic method based on branch and bound, the approximation method based on stochastic local search and the approximation method based on genetic algorithms. Section 6 is dedicated to the empirical experimentation evaluating and comparing the solving methods. Concluding remarks and possible future works are finally presented in Section 7.

2. New Procedure for Solving Incremental SAT

Let us assume that we have the following situation: $SAT_{i+1} = SAT_i \wedge NC$ where:

- SAT_i is the current SAT formula,
- NC is a new set of clauses to be added to SAT_i,
- and SAT_{i+1} is the new formula obtained after adding the new set of clauses.

Both SAT_i and NC (and by consequence SAT_{i+1}) are defined on a set X of boolean variables. Assuming that SAT_i is satisfiable, the goal here is to check the consistency of SAT_{i+1} when adding the new set of clauses denoted by NC. To do so, we have defined the following procedure:

(1) If $x \wedge \neg x$ is contained in NC, return that NC is inconsistent. NC cannot be added to SAT_i.
(2) Simplify NC by removing any clause containing a disjunction of the form $x \vee \neg x$.
(3) Let $NC = NC_1 \wedge NC_2$ where NC_1 is the set of clauses, each containing at least one variable that appears in SAT_i and NC_2 the set of clauses that do not contain any variable that appears in SAT_i and NC_1. Let $SAT_i = S_1 \wedge S_2$ where S_1 is the set of clauses, each containing at least one variable that appears in NC and S_2 the set of clauses that do not contain any variable that appears in NC or S_1. S_2 will be discarded from the rest of the procedure since any assignment to the variables of NC will not affect the truth assignment already obtained for S_2.

(4) Using a search method look for a truth assignment for NC_2. If no such assignment is found return NC cannot be added as it will affect the satisfiability of SAT_i.
(5) Assign the truth assignment of SAT_i to NC_1. If NC_1 is satisfiable goto 7.
(6) Using a search method flip the variables of NC_1 that do not appear in S_1. If NC_1 is satisfied goto 7.
(7) Using a search method, look for a truth assignment for both S_1 and NC_1. If no such assignment is found return NC cannot be added as it will affect the consistency of SAT_i.

Step 5 requires a search procedure that starts from an initial configuration and iterates until a truth assignment satisfying NC_1 is found. To perform this step we can use one of the following methods:

(1) A randomized local search method starting from the initial configuration. The local search algorithm will iterate by flipping the values of the variables of NC_1 which do not appear in S_1 until a truth assignment for NC_1 is found. Details about the stochastic local search method are presented in Section 4.
(2) A genetic algorithm starting from a population containing instances of the initial configuration. The genetic algorithm iterates performing the mutation and crossover operators on only the part of the vectors containing the variables of NC_1 that do not appear in S_1. The method based on genetic algorithms is presented in Section 5.
(3) A branch and bound method which starts with a lower bound equal to the number of non satisfied clauses of the initial configuration, and explores a subset of the search space by assigning values to the variables that appear in NC_1 but not in S_1. The algorithm will stop when the lower bound is equal to zero or when the entire subset of the search space is explored. The detail of the branch and bound method is presented in Section 3.

In step 6 the search procedure starts from the best configuration (assignment) found in step 5, and iterates until a truth assignment satisfying both NC_1 and S_1 is obtained. The search procedure has also to make sure to avoid checking any configuration already explored in step 5. This can be done by checking, at each variable assignment, that the subset of the variables belonging to S_1 and that do not belong to NC_1 has an assignment different from the old one satisfying SAT_i. Step 7 requires a search procedure for determining a truth assignment for NC_2.

3. Branch and Bound for Incremental SAT

The exact algorithms for solving SAT problems include the well known Davis-Putnam-Loveland algorithm (DPLL) [2,3] and the integer programming approaches [37].

DPLL starts with an upper bound (UB) corresponding to the number of unsatisfied clauses of a given complete assignment. The algorithm will then iterate updating the value of UB anytime a new complete assignment with a lower number of unsatisfied clauses is found. The algorithm will stop when UB is equal to zero (which corresponds to a solution

satisfying all the clauses) or when the entire search tree has been explored. More precisely, this backtrack search method compares at each node the upper bound UB with a lower bound (LB) corresponding to the estimation of the minimum number of unsatisfied clauses. LB is equal to the sum of the number of unsatisfied clauses of the current partial assignment and an underestimation of the number of clauses that become unsatisfied if we extend the current partial assignment into a complete one. More formally, LB is defined as follows.

$$LB = current\_unsatisfied + \sum_{x \in Xunsat} min(unsat(x), unsat(\neg x))$$

where :

- $current\_unsatisfied$ is the number of unsatisfied clauses of the current assignment,
- $Xunsat$ is the set of variables that have not been assigned yet,
- $unsat(x)$ (resp $unsat(\neg x)$) is the function returning the the number if unsatisfied clauses if x is assigned true (resp false).

If UB $\leq$ LB the algorithm backtracks and changes the decision at the upper level. If UB $>$ LB the current partial assignment is extended by instantiating the current node to true or false. If the current node is a leaf node, UB will take the value of LB (a new upper bound has been found). The algorithm will stop when UB is equal to zero or when the entire search tree has been explored. The underestimation is equal here to the minimum between the number of clauses that become unsatisfied if true is chosen for the next assignment and the number of clauses that become unsatisfied if false is chosen for the next assignment. For choosing the next variable to assign, we use the in-most-shortest clause heuristic as reported in [38].

DPLL has been improved using the following rules :

- Unit propagation [2,3]. This technique is used during the backtrack search as follows.
  - Look for a unit clause : a clause containing just one literal l.
  - Fix the value of l to true.
  - Remove all clauses containing l.
  - Remove all occurrences of l's negation.
- The Jeroslow-Wang branching rule [39] as variable selection heuristic. This rule indicates which child should be generated first when the algorithm branches. Roughly, the literal added to the latest generated node should occur in a large number of short clauses. Let us consider the formula F corresponding to the current node and a variable x_i to consider for branching. The Jeroslow-Wang rule is used through the following function : $\omega(F, j, v) = \sum N_{ikv} 2^{-k}$ where N_{ikv} is the number of clauses of length k that contain x_j (if $v = 1$) or $\neg x_j$ (if $v = 0$). v represents here a logical value (0 or 1). If (j, v) maximizes $\omega(F, j, v)$, then x_i is chosen first ($F \bigcup \{x_i\}$) if $v = 1$, and otherwise select $\neg x_i$ first ($F \bigcup \{\neg x_i\}$).

We have chosen the zChaff solver [7] which is the best and most recent improvement to the DPLL algorithm. Indeed, zChaff was the best complete solver in the industrial category

in the SAT 2002 and SAT 2004 competitions [40,41]. zChaff includes the following advanced heuristics and strategies.

- **Variable State Independent Decaying Sum (VSIDS).** This heuristic keeps a score for each variable literal which allows the focus on the recent conflicts. The goal here is to increase the locality of the search. A variable ordering scheme is also used here to increase the frequency of score decay.
- **Two Literal Watching**. This strategy is used for boolean constraint propagation as follows. During backtracking there is no need to modify the watched literals in the clause database.
- **Conflict Driven Clause Learning**. This heuristic is important when solving structured problems.
- **BerkMin Type Decision Heuristic.** Like for VSIDS, the goal here is to increase the search locality. This is done using BerkMin's rule [42] which consists of using the most recent unsatisfied conflict clauses.
- **Choose Short Antecedent Clauses First.** This is motivated by the fact that short clauses prune large spaces from the search and enable the early detection of conflicts.
- **Frequent Restarts.** This consists of using a rapid fixed interval restart policy in the hope of making the solver more robust.

4. SLS for Incremental SAT

One of the well known randomized local search algorithms for solving SAT problems is the GSAT procedure [8,9] presented in figure 1. GSAT is a greedy based algorithm that starts with a random assignment of values to boolean variables. It then iterates by selecting at each step a variable, flips its value from false to true or true to false and records the decrease in the number of unsatisfied clauses. The algorithm stops and returns a solution if the number of unsatisfied clauses is equal to zero. After MAX-FLIPS iterations, the algorithm updates the current solution to the new solution that has the largest decrease in unsatisfied clauses and starts flipping again until a solution satisfying all the clauses is found or MAX-TRIES is reached. In order to prevent the search from getting trapped in a local minima, a new approach called Dynamic Local Search (DLS) has been proposed. DLS associates weights with the clauses of the formula to solve. The goal here is to minimize the total weight rather than the number of satisfied clauses as we mentioned above. Many variants of the DLS strategy have been developed. The most powerful one is the Exponentiated Sub-Gradient (ESG) method [43]. Through the UBCSAT experimentation environment [11], we have adopted an improved version of ESG called Scaling and Probabilistic Smoothing (SAPS) algorithm [12] with the following modifications. When used in step 5, SAPS starts from the initial configuration (corresponding to the truth assignment found for the formula before adding the new clauses) instead of a random configuration. Also, only the variables belonging to NC_1 and which do not appear in S_1 can be chosen for the flip. For step 6, SAPS starts from the best configuration found in step 5. Also, all the configurations explored in step 5 are avoided in step 6 as shown in subsection 2.2.

```
Procedure GSAT
begin
  for i ← 1 until MAX-TRIES do
  begin
    T ← a randomly generated truth assignment
    for j ← 1 until MAX-FLIPS do
      if T satisfies the formula then return T
      else make a flip
  end
  return ("no satisfying assignment found")
end
```

Fig. 1. GSAT Procedure

5. Genetic Algorithms for Incremental SAT

Genetic Algorithms (GAs) perform multi-directional non systematic searches by maintaining a population of individuals (called also potential solutions) and encouraging information formation and exchange between these directions. It is an iterative procedure that maintains a constant size population of candidate solutions. Each iteration is called a generation and it undergoes some changes. *Crossover* and *mutation* are the two primary genetic operators that generate or exchange information in GAs. Under each generation, *good solutions* are expected to be produced and *bad solutions* die. It is the role of the objective (evaluation or fitness) function to distinguish the goodness of the solution.

The idea of crossover operators is to combine the information from parents and to produce a child that obtains the characteristics of its ancestors. In contrast, mutation is a unary operator that needs only one input. During the process, mutation operators produce a child by selecting some bad genes from the parent and replacing them with the good genes. The two operators may behave differently but they both follow the characteristic of GAs in that the next generation is expected to perform better than the ancestors.

Let us see now how to solve the SAT problem using genetic algorithms. Since we are dealing with variables that can only take on two states, true or false, the representation we choose is a binary vector of length n where n is the number of boolean variables with the coding of 1 being true and 0 being false. Each entry in the vector corresponds to the truth assignment for the variable which corresponds to that location in the vector. For example, if we consider the formula in introduction, then the vector (010) will correspond to the truth assignment $\{x_1 = false\ ,\ x_2 = true\ ,\ x_3 = false\}$ which does not satisfy the formula. The pseudo code of the GA based search algorithm we use is presented in figure 2.

The GA search method will start from an initial randomized population of binary vectors and evaluate each vector using a fitness function to see if we have discovered the optimum solution. We define the fitness function as the number of true clauses corresponding

```
1.  Begin
2.    t ← 1
3.    // P(t) denotes a population at iteration t
4.    P(t) ← p random binary vectors of size n
5.    eval ← evaluate P(t)
6.    while t ≤ max_iterations do
7.      begin
8.        t ← t + 1
9.        select P(t) from P(t − 1)
10.       alter P(t)
11.       evaluate P(t)
12.     end
13. end
```

Fig. 2. Genetic Algorithm.

to a given vector. For example, if we consider the following formula :

$$(X_1 \vee X_3) \wedge (\neg X_1 \vee X_2) \wedge \neg X_3$$

then the fitness function of the vector (101) (corresponding to $X_1 = 1$, $X_2 = 0$ and $X_3 = 1$) is equal to 1 (since it violates the last two clauses). If the fitness function is equal to C (C is the number of clauses of the formula) then the CNF expression is satisfied. After evaluating the randomized population, if the optimum function is not found then the crossover and mutation operators will be applied to the selected individuals of the current population respectively with probability p_c and p_m. The way we use to select the M individuals at each iteration (where M is the population size) is to assign a probability of being selected to each individual in proportion of their relative fitness. That is, an individual with the fitness function equal to 10 is 10 times more likely to be chosen than an individual with a score of 1. Note that we may obtain multiple copies of individuals that happened to be chosen more than once (case, for example, of individuals with good fitness function) and some individuals very likely would not be selected at all. Note also that even the individual with the best fitness function might not be selected, just by random chance. We have implemented the crossover and mutation operators in the same way as reported in [17]. Figure 3 describes an example of a crossover. After randomly selecting 2 positions a and b ($a = 3$ and $b = 8$ in the figure) in each parent, each of the two children is generated by taking all the bits in the neighborhood of a and b from one parent and the remaining bits from the other parent. The neighborhood is obtained using a given parameter d. The mutation operator is performed by flipping exactly one randomly selected bit of the parent (this is similar to the MutOne operator in [15]).

Solving SAT problems using GAs has been reported in the literature [14,15,16,17,18,19,20].

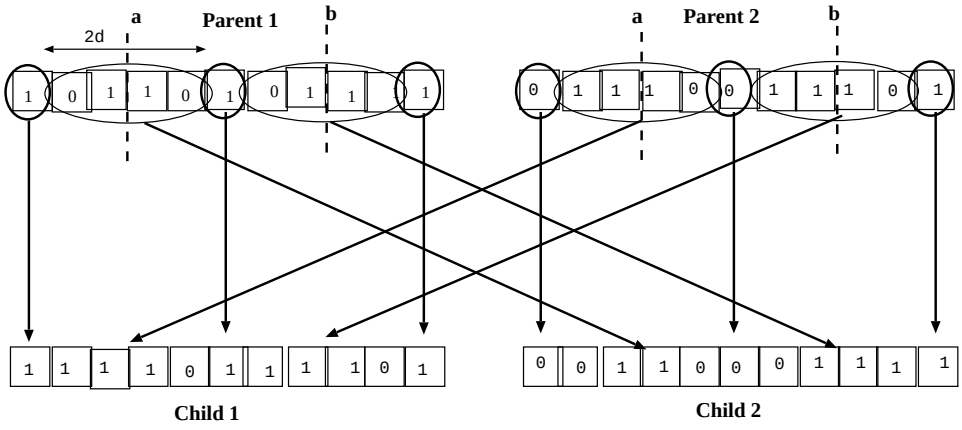

Fig. 3. Crossover operator

The results provided are less promizing than those of SLS methods unless if GAs are coupled with SLS algorithms as it is the case in [17,18,19]. We include GAs in our study in order to see how they behave (comparing to SLS and complete methods) in the case of incremental SAT.

In step 6 of our resolution procedure the above GA method will be used as is to look for the satisfiability of the formula NC_2. In step 5, the initial population contains instances of the initial configuration. Crossover and mutation operators are modified such that only the entries of the vectors corresponding to variables of N_1 which do not appear in S_1 are affected by the operators.

6. Experimentation

In this Section we present an empirical comparative study of the following four methods.

- **SLS**: the stochastic local search method is used here in steps 5, 6 and 7 of the resolution procedure.
- **GA**: the method based on genetic algorithms is used in steps 5,6 and 7.
- **BB**: the branch and bound method is used in steps 5, 6 and 7.
- **SLS+BB**: the branch and bound method is used in steps 5 and 6 while the stochastic local search method is used in step 7.

In order to find the best values for the parameters of the GA method, we have conducted preliminary tests on several randomly generated incremental SAT instances described as shown below. The results show that the following provides the best running time performances.

- Population size $M = 320$.
- Probability for each individual in a population to be selected for the crossover $p_c \in$

$[0.1, 0.5]$ (depending on the problem instance).
- Probability for each individual in a population to be selected for the mutation $p_m \in [0.8, 1]$ (depending on the problem instance).

All tests are performed on a 2.8 GHz Pentium IV computer under Linux and all procedures are coded in C language.

6.1. *Incremental SAT Instances*

Since we did not find libraries providing incremental SAT problems, we build incremental and consistent SAT instances taken from the well known SATLIB library [21]. Each incremental SAT instance is generated from a SAT one in a series of stages. At each stage, a random number of clauses is taken from the SAT instance and added to the incremental SAT one until there are no more clauses to take. In the following we consider *st* the total number of stages and N the total number of clauses of the SAT instance. The number of clauses $(N_1 \ldots N_{st})$ taken at each stage are generated as follows. N_1 and N_2 are randomly chosen form $[1, N/5 - 1]$. N_3 and N_4 will then be generated from $[1, (N - N_1 - N_2)/4 - 1]$, and N_5 and N_6 from $[1, (N - N_1 - N_2 - N_3 - N_4)/3 - 1]$. $N_7, N_8, \ldots, N_{st}$ will be generated in the same manner. This will ensure that the average number of clauses in each stage is almost equal to N/st. The first advantage of this procedure is to guarantee that we handle the addition of more than one clause at each time. The second one, as we will see later, is to be able to generate hard problems (near the phase transition, where the ratio $\frac{\#clauses}{\#variables}$ is close to 4.2). In the following tests, the value of *st* is fixed to 10.

Using the method above, we have generated incremental and consistent SAT instances from the following three types of SAT problems.

- **"Flat" Graph Coloring Problem.** For a given graph, the "flat" graph coloring problem tries to color the nodes of the graph such that any pair of connected nodes have different colors. Here we focus on the decision variant which consists of deciding whether for a particular number of colors, a coloring of the given graph exists. SAT instances corresponding to the "flat" coloring problem are generated as shown in [44].
- **Incremental "Morphed" Graph Coloring Problem.** A particular class of graph coloring problems consists of morphing regular ring lattices with random graphs. $A(typeB)$ p-morph of two graphs $A = (V, E_1)$ and $B = (V, E_2)$ is a graph $C = (V, E)$ where E contains all the edges common to A and B, a fraction p of the edges from $E_1 - E_2$ (the remaining edges of A), and a fraction $1 - p$ of the edges from $E_2 - E_1$. More details regarding the generation of SAT instances for morphed graphs can be found in [45].
- **Random-3-SAT Instances with Controlled Backbone Size Problems.** The backbone of a satisfiable SAT instance is the set of entailed literals. A literal l is entailed by a satisfiable SAT instance S if and only if S AND (NOT l) is unsatisfiable. The backbone size is the number of entailed literals. Random 3-SAT Instances with Controlled Backbone Size are generated as shown in [46].

In order to evaluate and compare the methods on hard incremental SAT problems,

we have generated other incremental 3-SAT instances (from uniform random 3-SAT problems generated as shown in [47]) as follows. Since for 3-SAT problems the phase transition (corresponding to the hardest problems to solve) corresponds to a ratio $\frac{\#clauses}{\#variables} = 4.2$ we first split the random SAT instance into sets of clauses such that the addition of each set to the incremental SAT formula will always lead to a ratio $\frac{\#clauses}{\#variables}$ close to 4.2 . This way the formula to solve at each stage will be close to the phase transition.

6.2. *Test Results*

The results of the tests performed on Incremental "Flat" Graph Coloring Problems, Incremental "Morphed" Graph Coloring Problems, Incremental Hard Uniform 3-SAT and Incremental Random 3-SAT Instances with Controlled Backbone Size Problems are respectively reported in figures 4, 5, 6 and 7.

Fig. 4. Comparative tests on randomly generated Incremental "Flat" Graph Coloring Problems.

testset	SLS	GA	SLS+BB	BB
flat30-60	0.00450	0.00460	0.00460	0.004
flat50-115	0.014	0.015	0.014	0.08
flat75-180	0.06	0.1	0.06	0.4
flat100-239	0.3	0.5	0.4	0.8
flat125-301	0.9 (98%)	1.8 (98%)	1.2	3.5
flat150-360	1.6 (97%)	2.4 (97%)	2	13.2

Fig. 5. Comparative tests on randomly generated Incremental "Morphed" Graph Coloring Problems.

testset	SLS	GA	SLS+BB	BB
sw100-8-lp0-c5	0.01	0.01	0.01	0.01
sw100-8-lp1-c5	0.02	0.02	0.02	0.1
sw100-8-lp2-c5	0.02	0.022	0.02	0.18
sw100-8-lp3-c5	0.021	0.024	0.228	0.9
sw100-8-lp4-c5	0.07	0.14	0.1	2.7
sw100-8-lp5-c5	0.18 (98%)	0.4 (98%)	0.7	6.3
sw100-8-lp6-c5	0.4 (97%)	1 (95%)	0.9	14.7
sw100-8-lp7-c5	0.7 (92%)	1.2 (92%)	1.2	23.8
sw100-8-lp8-c5	0.9 (93%)	2.1 (93%)	2.5	34.6
sw100-8-lp9-c5	1.2 (95%)	2.1 (95%)	2.8	48.7

In figures 4, 6 and 7, each test set is characterized by the number of variables and clauses of the random instances. For instance, `uf20-91` in figure 6 corresponds to uniform random

Fig. 6. Comparative tests on randomly generated Incremental Hard Uniform 3-SAT Problems.

testset	SLS	GA	SLS+BB	BB
uf50-218	0.001	0.001	0.001	0.002
uf75-325	0.001	0.001	0.0012	0.06
uf100-430	0.0013	0.002	0.002	0.08
uf125-538	0.002	0.003	0.003	0.12
uf150-645	0.0024	0.004	0.003	1.3
uf175-753	0.004	0.008	0.008	2
uf200-860	0.008	0.009	0.01	2.1
uf225-960	0.01 (98%)	0.01 (98%)	0.012	2.8
uf250-1065	0.01 (98%)	0.01 (98%)	0.014	3.9

3-SAT problems having 20 variables and 91 clauses. Similarly, `CBS_k3_n100_m403_b10` in figure 7 corresponds to Random 3-SAT Instances with Controlled Backbone Size having 100 variables and 403 clauses. In figure 5, all instances have 500 variables and 3100 clauses each.

All the figures present the average running time needed by each of the four methods to solve the incremental SAT instances. Indeed, for each test set, each method is executed on 100 instances and the average execution time for solving the instances is taken. Moreover the running time for each instance is obtained by taking the average of 50 independent run on the same instance as this is common procedure when evaluating algorithms on random SAT instances where a phase transition has been observed such as Incremental Hard Uniform 3-SAT Problems [48].

In the case of SLS and GA, we put in brackets the success rate (beside the running time) anytime the approximation method fails to solve the problem for a particular instance. The time is averaged here over the successful instances. For instance, in the case of flat125-301, figure 4, SLS and GA succeeded to solve 98% of the instances and the time is averaged over these 98 instances. Note that a method fails to solve a problem instance if it fails to get a complete solution in at least one of the 50 independent run on this instance. In order to have a more accurate picture regarding the behavior of SLS and GA on these types of instances and especially those from test-sets `uf225-960` and `uf250-1065` [c] we conducted a more in depth study by using a more adequate empirical method called runtime distribution (RTD) [48]. More precisely the RTD of a given randomized algorithm (such as SLS or GA in our case) is a cumulative distribution function F defined as follows.

$$F(t) = Pr\{runtime \leq t\}, F : [0, \infty) \rightarrow [0, 1]$$

For example, figure 8 shows a typical RTD of SLS and GA in the case of a problem

[c]For the other instances where the success ratio is less than 100% both SLS and GA fail to get a complete solution in all the 50 run.

Fig. 7. Comparative tests on randomly generated Incremental 3-SAT Instances with Controlled Backbone.

testset	SLS	GA	SLS+BB	BB
CBS_k3_n100_m403_b10	0.01	0.01	0.01	0.01
CBS_k3_n100_m403_b30	0.012	0.02	0.02	0.02
CBS_k3_n100_m403_b50	0.015	0.034	0.04	0.1
CBS_k3_n100_m403_b70	0.02	0.05	0.06	0.9
CBS_k3_n100_m403_b90	0.03	0.1	0.09	1.2
CBS_k3_n100_m411_b10	0.064	0.12	0.20	2
CBS_k3_n100_m411_b30	0.1	0.2	0.2	2.2
CBS_k3_n100_m411_b50	0.06	0.07	0.07	3
CBS_k3_n100_m411_b70	0.08	0.08	0.1	3.2
CBS_k3_n100_m411_b90	0.09	0.22	0.24	3.8
CBS_k3_n100_m418_b10	0.03	0.04	0.05	0.7
CBS_k3_n100_m418_b30	0.04	0.05	0.1	0.8
CBS_k3_n100_m418_b50	0.04	0.05	0.09	0.9
CBS_k3_n100_m418_b70	0.03	0.05	0.07	1
CBS_k3_n100_m418_b90	0.03	0.05	0.08	1.4
CBS_k3_n100_m423_b10	0.02	0.03	0.09	2
CBS_k3_n100_m423_b30	0.06	0.1	0.09	2.23
CBS_k3_n100_m423_b50	0.06	0.09	0.1	2.87
CBS_k3_n100_m423_b70	0.06	0.05	0.1	1.7
CBS_k3_n100_m423_b90	0.03	0.04	0.09	1.2
CBS_k3_n100_m429_b10	0.01	0.01	0.03	2.8
CBS_k3_n100_m429_b30	0.01	0.01	0.03	0.7
CBS_k3_n100_m429_b50	0.02	0.04	0.13	3.7
CBS_k3_n100_m429_b70	0.03	0.04	0.08	3.6
CBS_k3_n100_m429_b90	0.04	0.04	0.07	3.5
CBS_k3_n100_m435_b10	0.01	0.01	0.03	0.9
CBS_k3_n100_m435_b30	0.01	0.01	0.08	1.1
CBS_k3_n100_m435_b50	0.01	0.01	0.03	0.6
CBS_k3_n100_m435_b70	0.02	0.03	0.090	1.34
CBS_k3_n100_m435_b90	0.02	0.02	0.08	3.34
CBS_k3_n100_m441_b10	0.02	0.03	0.09	1
CBS_k3_n100_m441_b30	0.01	0.02	0.03	0.7
CBS_k3_n100_m441_b50	0.01	0.02	0.04	2.2
CBS_k3_n100_m441_b70	0.03	0.03	0.08	4.3
CBS_k3_n100_m441_b90	0.02	0.02	0.08	5.3
CBS_k3_n100_m449_b10	0.012	0.013	0.03	6.4
CBS_k3_n100_m449_b30	0.01	0.01	0.01	0.6
CBS_k3_n100_m449_b50	0.02	0.022	0.022	0.5
CBS_k3_n100_m449_b70	0.03	0.03	0.03	0.6
CBS_k3_n100_m449_b90	0.06	0.05	0.3	1.6

instance where there is a little variability in the results obtained over the 50 run[d]. `t0` is here the average over the 50 run while `F(t)` is the corresponding RTD.

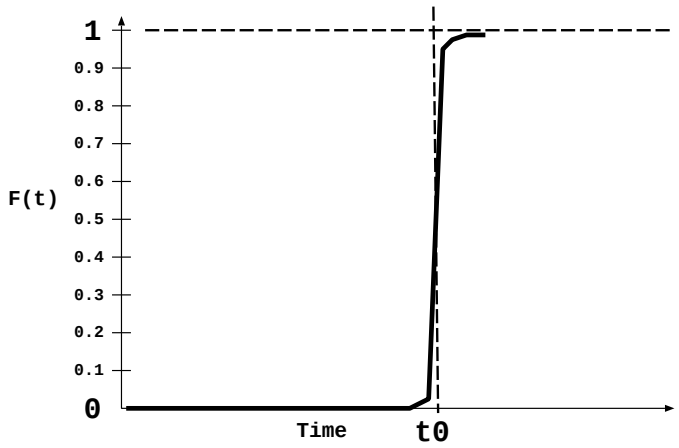

Fig. 8. Typical RTD of SLS and GA when success ratio is 100%.

We run 200 independent tests on both `uf225-960` and `uf250-1065` with SLS and GA. The chart in figure 9 is the RTD corresponding to the SLS with `uf225-960` (the RTDs of the other cases are similar). As we can see from the chart, although there is some variability in the results, we did not notice any behaviour such as heavy-tailed RTD [e]. Actually heavy-tailed RTD was not observed in any of the tests we conducted. This is justified by the fact that heavy-tailed RTD does mainly occur with backtracking or randomized backtracking algorithms and on structured problems [49].

In all the figures (4, 5, 6 and 7), SLS method presents the best results. Due to the exponential running time of the branch and bound method (comparing to the polynomial time cost of the approximation techniques) BB is slower especially for large instances. However BB is a systematic search method that always guarantees the completeness of the solution returned which is not the case of SLS and GA. Indeed, for very large problems SLS and GA fail sometimes to solve the problem completely. For example, in the test-sets `uf225-960` and `uf250-1065`, figure 6, SLS and GA fail to solve 2% of the instances. SLS+BB is the method that has the best compromise between the performance in time and the completeness of the solution returned. Indeed, SLS+BB succeded to solve completely all the problem instances (even for very large problems) in a reasonable execution time. Indeed, as we can see the running time of SLS+BB is, in general, comparable to the running time of SLS and outperfoms the one of GA. The completeness of SLS+BB is guaranteed here because the non systematic part of the method (SLS) is applied only in step 7 on a

[d]This mainly concerns all problem instances where the success ratio is 100%.
[e]Heavy tailed RTD occurs when most runs are short while some others take very long ruining time [49].

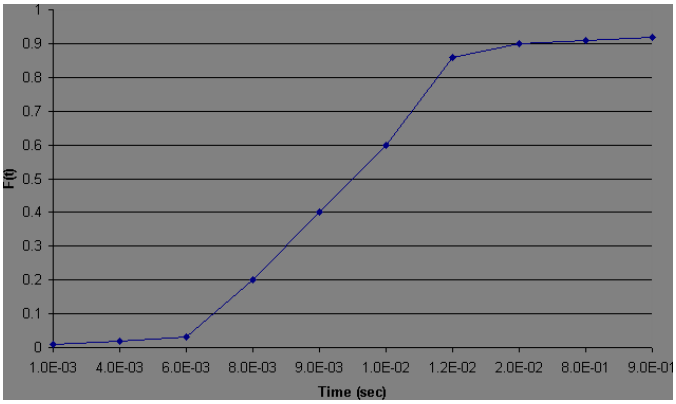

Fig. 9. RTD of SLS for `uf225-960`.

small set of clauses (NC_2 which is a subset of the set of new added clauses at each time). In practice, SLS always finds a complete solution for a SAT formula when its number of clauses is small.

Note that we conducted the same tests reported in figures 4, 5, 6 and 7, on the static versions of SLS (SAPS), BB (zChaff) and GAs. The running time of these static algorithms are very poor (sometimes 100 times slower) than their incremental variants, as indicated in figures 10, 11, 12 and 13. In these latter comparative figures each running time of a static method is reported in brackets beside the one of its incremental variant.

Fig. 10. Comparative tests on randomly generated Incremental "Flat" Graph Coloring Problems.

testset	SLS (SAPS)	GA (Static)	SLS+BB	BB (zChaff)
flat30-60	0.00450 (0.04)	0.00460 (0.04)	0.00460	0.004(0.04)
flat50-115	0.014 (0.15)	0.015(0.15)	0.014	0.08(0.09)
flat75-180	0.06(0.07)	0.1(0.9)	0.06	0.4(4.12)
flat100-239	0.3(3.56)	0.5(6.8)	0.4	0.8(18.12)
flat125-301	0.9 (98%) (22.11)	1.8 (98%)(37.12)	1.2	3.5 (44)
flat150-360	1.6 (97%)(74)	2.4 (97%) (88)	2	13.2 (127)

Fig. 11. Comparative tests on randomly generated Incremental "Morphed" Graph Coloring Problems.

testset	SLS (SAPS)	GA (Static)	SLS+BB	BB (zChaff)
sw100-8-lp0-c5	0.01 (0.2)	0.01(0.2)	0.01	0.01(0.23)
sw100-8-lp1-c5	0.02(0.33)	0.02(0.44)	0.02	0.1(2.3)
sw100-8-lp2-c5	0.02(0.45)	0.022(1.12)	0.02	0.18(3.4)
sw100-8-lp3-c5	0.021(0.56)	0.024 (0.67)	0.228	0.9 (13.45)
sw100-8-lp4-c5	0.07(1.12)	0.14(2.44)	0.1	2.7(34)
sw100-8-lp5-c5	0.18(14.5) (98%)	0.4(19) (98%)	0.7	6.3 (67)
sw100-8-lp6-c5	0.4(27) (97%)	1(34) (95%)	0.9	14.7(98)
sw100-8-lp7-c5	0.7(56) (92%)	1.2(64) (92%)	1.2	23.8 (240)
sw100-8-lp8-c5	0.9(88) (93%)	2.1(112) (93%)	2.5	34.6 (278)
sw100-8-lp9-c5	1.2 (97)(95%)	2.1(144) (95%)	2.8	48.7 (2145)

Fig. 12. Comparative tests on randomly generated Incremental Hard Uniform 3-SAT Problems.

testset	SLS (SAPS)	GA (Static)	SLS+BB	BB (zChaff)
uf50-218	0.001(0.01)	0.001(0.012)	0.001	0.002(0.03)
uf75-325	0.001(0.01)	0.001(0.022)	0.0012	0.06(0.05)
uf100-430	0.0013(0.017)	0.002(0.034)	0.002	0.08(0.09)
uf125-538	0.002(0.02)	0.003(0.034)	0.003	0.12(2.3)
uf150-645	0.0024(0.03)	0.004(0.05)	0.003	1.3(12)
uf175-753	0.004(0.06)	0.008(0.1)	0.008	2(17)
uf200-860	0.008(0.09))	0.009(0.09)	0.01	2.1(25)
uf225-960	0.01(0.23) (98%)	0.01(0.4) (98%)	0.012	2.8(45)
uf250-1065	0.01(0.46) (98%)	0.01(0.7) (98%)	0.014	3.9(58)

7. Conclusion

In this paper we have presented different ways based respectively on systematic and approximation methods for maintaining the satisfiability of CNF propositional formulas in an incremental way. Although the non systematic methods have the best performance in time, they do not always guarantee the completeness of the solution returned. On the other hand, the systematic method based on branch and bound does not have good performance in time while it guarantees a complete solution. Finally, a method we propose that uses a non systematic search in a limited form has the best compromise between time and the success ratio (percentage of instances completely solved).

Our work is of interest to a large variety of applications that need to be processed in an evolutive environment. This can be the case of real-world problems such as reactive scheduling and planning, dynamic combinatorial optimization, dynamic constraint satisfaction and machine learning in a dynamic environment.

Fig. 13. Comparative tests on randomly generated Incremental 3-SAT Instances with Controlled Backbone.

testset	SLS (SAPS)	GA (Static)	SLS+BB	BB (zChaff)
CBS_k3_n100_m403_b10	0.01(0.08)	0.01(0.09)	0.01	0.01(0.09)
CBS_k3_n100_m403_b30	0.012(0.22)	0.02(0.3)	0.02	0.02(0.26)
CBS_k3_n100_m403_b50	0.015(0.24)	0.034(0.37)	0.04	0.1(0.4)
CBS_k3_n100_m403_b70	0.02(0.32)	0.05(0.44)	0.06	0.9(5.8)
CBS_k3_n100_m403_b90	0.03(0.34)	0.1(1.12)	0.09	1.2(6.8)
CBS_k3_n100_m411_b10	0.064 (0.07)	0.12(1.1)	0.20	2(9.3)
CBS_k3_n100_m411_b30	0.1 (1.2)	0.2(2.3)	0.2	2.2(10.11)
CBS_k3_n100_m411_b50	0.06 (0.77)	0.07(0.8)	0.07	3(12.4)
CBS_k3_n100_m411_b70	0.08(0.8)	0.08(0.87)	0.1	3.2(18.7)
CBS_k3_n100_m411_b90	0.09(9.23)	0.22(3.11)	0.24	3.8(24)
CBS_k3_n100_m418_b10	0.03(0.32)	0.04(0.38)	0.05	0.7(6.18)
CBS_k3_n100_m418_b30	0.04(0.4)	0.05(0.46)	0.1	0.8(8.12)
CBS_k3_n100_m418_b50	0.04(0.43)	0.05(0.49)	0.09	0.9(9.3)
CBS_k3_n100_m418_b70	0.03(0.32)	0.05(0.51)	0.07	1(12)
CBS_k3_n100_m418_b90	0.03(0.3)	0.05(0.55)	0.08	1.4(12.3)
CBS_k3_n100_m423_b10	0.02(0.22)	0.03(0.34)	0.09	2(26)
CBS_k3_n100_m423_b30	0.06(0.65)	0.1(1.2)	0.09	2.23(23)
CBS_k3_n100_m423_b50	0.06(0.7)	0.09(0.9)	0.1	2.87(18)
CBS_k3_n100_m423_b70	0.06(0.7)	0.05(0.6)	0.1	1.7(15.8)
CBS_k3_n100_m423_b90	0.03(0.3)	0.04(0.44)	0.09	1.2(15)
CBS_k3_n100_m429_b10	0.01(0.12)	0.01(0.15)	0.03	2.8(33)
CBS_k3_n100_m429_b30	0.01(0.12)	0.01(0.15)	0.03	0.7(7.8)
CBS_k3_n100_m429_b50	0.02(0.2)	0.04(0.38)	0.13	3.7(36)
CBS_k3_n100_m429_b70	0.03(0.35)	0.04(0.4)	0.08	3.6(33.2)
CBS_k3_n100_m429_b90	0.04(0.45)	0.04(0.49)	0.07	3.5(33)
CBS_k3_n100_m435_b10	0.01(0.18)	0.01(0.22)	0.03	0.9(11)
CBS_k3_n100_m435_b30	0.01(0.2)	0.01(0.23)	0.08	1.1(13)
CBS_k3_n100_m435_b50	0.01(0.12)	0.01(0.13)	0.03	0.6(6.8)
CBS_k3_n100_m435_b70	0.02(0.23)	0.03(0.33)	0.090	1.34(13.6)
CBS_k3_n100_m435_b90	0.02(0.23)	0.02(0.22)	0.08	3.34(32.5)
CBS_k3_n100_m441_b10	0.02(0.23)	0.03(0.29)	0.09	1(11)
CBS_k3_n100_m441_b30	0.01(0.12)	0.02(0.22)	0.03	0.7(8.12)
CBS_k3_n100_m441_b50	0.01(0.12)	0.02(0.23)	0.04	2.2(18.7)
CBS_k3_n100_m441_b70	0.03(0.34)	0.03(0.38)	0.08	4.3(38.6)
CBS_k3_n100_m441_b90	0.02(0.21)	0.02(0.24)	0.08	5.3(45)
CBS_k3_n100_m449_b10	0.012(0.2)	0.013(0.23)	0.03	6.4(48.9)
CBS_k3_n100_m449_b30	0.01(0.12)	0.01(0.14)	0.01	0.6(7.8)
CBS_k3_n100_m449_b50	0.02(0.2)	0.022(0.25)	0.022	0.5(5.3)
CBS_k3_n100_m449_b70	0.03(0.34)	0.03(0.35)	0.03	0.6(6.12)
CBS_k3_n100_m449_b90	0.06(0.5)	0.05(0.6)	0.3	1.6(12)

One perspective of our work is to deal with retraction of clauses in an efficient way. Assume that during the search, a given clause (or a set of clauses) is removed, would it be worthwhile to reconsider any decision made because of these clause(s) or would it be more costly than just continuing on with search. Another idea we will investigate in order to improve the performance of our general procedure consists of processing steps 4 until 6 and step 7 of our procedure in parallel. If any of these two parallel phases fails then the main procedure will stop and returns NC (set of new clauses to be added) inconsistent.

References

1. S. A. Cook. The complexity of theorem proving procedures. In *3rd Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.
2. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of The Association for Computing Machinery (ACM)*, 7:201–215, 1960.
3. D. Loveland. *Automated Theorem-Proving : A Logical Basis*. North Holland, 1978.
4. J.P. Marques-Silva and K.A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.
5. L. Zhang, C.F. Madigan, M.H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solve r. In *Proceedings of International Conference of Computer Aided Design (ICCAD)*, 2001.
6. W. Moskewisc, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Procceedings of the 38th Design Automation Conference*, pages 530–535, June 2001.
7. S. YogeshMahajan, Zhaohui Fu, and Sharad Malik. Zchaff2004: An efficient sat solver. In H.H. Hoos and D.G. Mitchell, editors, *LNCS 3542*. SAT 2004, Springer-Verlag Berlin Heidelberg, 2004.
8. B. Selman and H. A. Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI'93)*, pages 46–51, Washington, DC, 1993. The AAAI Press/The MIT Press.
9. B. Selman, H. A. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *AAAI'94*, pages 337–343. MIT Press, 1994.
10. H. H. Hoos and T. Stutzle. *Stochastic Local Search Foundations and Applications*. Morgan Kaufmann / Elsevier, 2004.
11. D. A. D. Tompkins and H. H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing: Revised Selected Papers of the Seventh International Conference (SAT 2004, Vancouver, BC, Canada, May 10–13, 2004)*, volume 3542 of *Lecture Notes in Computer Science*, pages 306–320, Berlin, Germany, 2005. Springer Verlag.
12. F. Hutter, A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In *Eighth International Conference on Principles and Practice of Constraint Programming (CP-02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248, Berlin, Germany, 2002. Springer Verlag.
13. Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evaluation Program*. Springer-Verlag, 1992.
14. K. A. De Jong and W. M. Spears. Using genetic algorithms to solve np-complete problems. In Schaffer J. D, editor, *Third International Conference on Genetic Algorithms*, pages 124–132, San Mateo, CA, 1989. Morgan Kaufmann.
15. A. E. Eiben and J. K. van der Hauw. Solving 3-sat with adaptive genetic algorithms. In *4th IEEE Conference on Evolutionary Computation*, pages 81–86, Piscataway, NJ, 1997.
16. B. Craenen and A.E. Eiben. Comparing evolutionary algorithms on binary constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 7(5):424–444, 2003.
17. G. Folino, C. Pizzuti, and C. Spezzano. Parallel hybrid method for sat that couples genetic algorithms and local search. *IEEE Transactions on Evolutionary Computation*, 5(4):323–334, 2001.
18. F. Lardeux, F. Saubion, and J.K. Hao. Gasat: A genetic local search algorithm for the satisfiability problem. *Evolutionary Computation*, 14(2):223–253, 2006.
19. J.K. Hao, F. Lardeux, and F. Saubion. Evolutionary computing for the satisfiability problem. In *APPLICATIONS OF EVOLUTIONARY COMPUTING*, volume 2611 of *Lecture Notes in Computer Science*, pages 258–267. Springer Verlag, 2003.
20. J. Gottlieb, E. Marchiori, and C. Rossi. Evolutionary algorithms for the satisfiability problem.

Evolutionary Computation, 10(1):35–50, 2002.
21. H.H. Hoos and K O'Neil. Stochastic Local Search Methods for Dynamic SAT - an Initial Investigation. In *AAAI-2000 Workshop 'Leveraging Probability and Uncertainty in Computation*, pages 22–26, 2000.
22. J. Gutierrez and A.D. Mali. Local search for incremental satisfiability. In *International Conference on Artificial Intelligence*, pages 986–991, 2002.
23. J.N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15:177–186, 1993.
24. H. Bennaceur, I. Gouachi, and Plateau. An incremental branch-and-bound method for satisfiability problem. *INFORMS Journal on Computing*, 10:301–308, 1998.
25. J. Wittemore, J. Kim, and K.A. Sakallah. Satire: A new incremental satisfiability engine. In *DAC 2001*, pages 542–545, 2001.
26. R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.
27. T. Walsh. Sat v csp. In R. Dechter, editor, *sixth International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 441–456, Berlin, Germany, 2000. CP 2000, Springer Verlag.
28. F. Bacchus: . SAT 2006: 10-10. Csps: Adding structure to sat. In A. Biere and C. P. Gomes, editors, *LNCS 4121*, pages 10–10. SAT 2006, Springer-Verlag Berlin Heidelberg, 2006.
29. S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 25–32, Boston, MA, August 1990. AAAI Press.
30. D. Sabin and E. C. Freuder. Configuration as composite constraint satisfaction. In George F. Luger, editor, *Proceedings of the (1st) Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161. AAAI Press, 1996.
31. D. Sabin, E. C. Freuder, and R. J. Wallace. Greater efficiency for conditional constraint satisfaction. *Proc., Ninth International Conference on Principles and Practice of, Constraint Programming - CP 2003*, 2833:649–663, 2003.
32. E. Gelle and B. Faltings. Solving mixed and conditional constraint satisfaction problems. *Constraints*, 8:107–141, 2003.
33. R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. In *7th National Conference on Artificial Intelligence*, pages 37–42, St Paul, 1988.
34. A. K. Jónsson and J. Frank. A framework for dynamic constraint reasoning using procedural constraints. In *ECAI 2000*, pages 93–97, 2000.
35. A. K. Jónsson J. Frank. Constraint-based attribute and interval planning. *Constraints*, 8(4):339–364, 2003.
36. I. Tsamardinos, T. Vidal, and M. E. Pollack. CTP: A New Constraint-Based Formalism for Conditional Temporal Planning. *Constraints*, 8(4):365–388, 2003.
37. R.G. Jeroslow C.E. Blair and J.K. Lowe. Some results and experiments in programming techniques for propositional logic. *Computers and Operations Research*, 13(5):633–645, 1986.
38. R.J. Wallace and E.C. Freuder. Comparing constraint satisfaction and davis-putnam algorithms for the maximal satisfiability problem. In D.S. Johnson and M.A. Trick, editors, *Second DIMACS Implementation Challenge. American Mathematical Society*, 1995.
39. R. E. Jeroslow. Solving propositional satisfiability problems. *Annals of Mathematics and AI*, 1:167–187, 1990.
40. *SAT Competition*. Morgan Kaufmann, 2002.
41. *SAT Competition*. Morgan Kaufmann, 2004.
42. E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver. In *DATE*, 2002.
43. D. Shuurmans, F. Southey, and R.C. Holte. The exponentiated subgradient algorithm for heuristic boolean programming. In *IJCAI-01*, pages 334–341. Morgan Kaufmann, 2001.
44. T. Hogg. Refining the phase transition in combinatorial search. *Artificial Inteligence*, 81:127–

154, 1996.
45. I. Gent, H. H. Hoos, P. Prosser, and T. Walsh. Combining structure and randomness. In *AAAI-99*, pages 654–660, 1999.
46. J. Singer, I. Gent, and A. Smaill. Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research*, pages 235–270, 2000.
47. P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *IJCAI-91*, pages 331–337, 1991.
48. H. H. Hoos and T. Stutzle. Characterizing the run-time behavior of stochastic local search. In *Technical Report AIDA-98-01*, 1998.
49. H. H. Hoos. "heavy-tailed behaviour in randomised systematic search algorithms for sat?". In *Technical Report TR-99-16*, Department of Computer Science, University of British Columbia, 1999.