Approximating Minimal Unsatisfiable Subformulae by means of Adaptive Core Search

Renato Bruni

Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Buonarroti 12, I-00185 Rome, Italy

Abstract

The paper is concerned with the relevant practical problem of selecting a small unsatisfiable subset of clauses inside an unsatisfiable CNF formula. Moreover, it deals with the algorithmic problem of improving an enumerative (DPLL-style) approach to SAT, in order to overcome some structural defects of such approach. Within a complete solution framework, we are able to evaluate the difficulty of each clause, by analyzing the history of the search. Such clause hardness evaluation is used in order to rapidly select an unsatisfiable subformula (of the given CNF) which is a good approximation of a minimal unsatisfiable subformula (MUS). Unsatisfiability is proved by solving only such subformula. Very small unsatisfiable subformulae are detected inside famous Dimacs unsatisfiable problems and in real world problems. Comparison with the very efficient solver SATO 3.2 used as a state-of-the-art DPLL procedure (disabling learning of new clauses) shows the effectiveness of such enumeration guide.

Key words: Consistency Restoring, Enumeration, (Un)Satisfiability.

1 Introduction

A propositional formula $\mathcal{F}$ in *conjunctive normal form* (CNF) is a conjunction of m clauses C_j, each clause being a disjunction of literals, each literal being either a positive (α_i) or a negative ($\neg\alpha_i$) propositional variable. The number of different variables appearing in the formula is n. By denoting with l_j the

Email address: bruni@dis.uniroma1.it (Renato Bruni).

cardinality of C_j, and with $[\neg]$ the possible presence of $\neg$, this is

$$\mathcal{F} = \bigwedge_{j=1..m} (\bigvee_{i=1..l_j} [\neg]\alpha_i)$$

A truth assignment for the variables provides a truth value {*True, False*} for the formula. The *satisfiability* problem (SAT) consists in finding a truth assignment for the variables such that $\mathcal{F}$ evaluates to *True*, or proving that such truth assignment does not exist. Such problem plays a protagonist role both in mathematical logic, since the problem of logical implication can be formalized as a SAT problem, and in computing theory, being SAT the prototype of NP-complete problems. Moreover, many problems arisen from different applicative fields, e.g. VLSI logic circuit design and testing, programming language project, computer aided design, are usually encoded as SAT problems.

We deal here with the practical problem of locating a small unsatisfiable subset of clauses inside an unsatisfiable CNF formula, and with the algorithmic problem of improving an enumerative approach to SAT. As for the first question, when an instance $\mathcal{F}$ encodes a system or structure one must design, we generally require $\mathcal{F}$ to have a well-defined solution property (either to be satisfiable or to be unsatisfiable). When $\mathcal{F}$ is unsatisfiable, and we want it to be satisfiable, we would like to modify the system in order to make $\mathcal{F}$ satisfiable. Conversely, when $\mathcal{F}$ is unsatisfiable and we want it to be so, if we need to re-design the system (for instance to reduce its cost), we would like to keep $\mathcal{F}$ unsatisfiable. An approach to the first problem leads to the solution of maximum satisfiability problems. The *maximum satisfiability* problem (Max-SAT) consists in finding a truth assignment for the variables maximizing the number of clauses C_j which evaluates to *True* [2]. By denoting with $\mathcal{S}$ such maximum set of clauses which can be simultaneously satisfied, $\mathcal{S} \subseteq \mathcal{F}$, satisfiability can be restored by removing from the system all elements corresponding to clauses of $\mathcal{F} \setminus \mathcal{S}$. However, such approach is not desirable in many practical cases. Very often, in fact, we cannot just delete a part of our system, because we need the functionalities contained in that part. Instead, we would like to locate and understand the problem, and, basing on this information, re-design only the small part of the system causing the problem. As for the second problem, when we want $\mathcal{F}$ to be unsatisfiable (and we cannot just add a contradiction), we typically would like to know which part of the system should not be changed, and which one can be modified (or possibly removed). Both of the above problems can be approached by looking for a subset of clauses $\mathcal{U}$ within an unsatisfiable formula $\mathcal{F}$ such that $\mathcal{U}$ is still unsatisfiable. More than one unsatisfiable subformula can be contained within the same $\mathcal{F}$. Unsatisfiable subformulae are characterized with respect to the number of their clauses, and relations between them and the solution of Max-SAT are investigated in Sect. 2. A procedure to select a small unsatisfiable subformula is in Sect. 4.

As for the algorithmic question, many procedures for solving the SAT problem have been proposed, based on different techniques (among others, [3,6,9–12,14,16,17,20,22,29,26,31,34], see also [7,15,18,21,32] for extensive references). A solution method is *complete* if it guarantees (given enough time) to find a solution if it exists, or report lack of solution otherwise. Most of complete methods are based on enumeration techniques, in particular splitting and backtracking, such as the Davis-Putnam-Logemann-Loveland (DPLL) [10,24]. Their flow of control is often represented by a search tree, where the root corresponds to the original formula $\mathcal{F}$, and the arcs correspond to variable assignments. A splitting and backtracking procedure visits the search tree as follows. If the formula has an empty clause, exit and report *unsatisfiable*. If the formula has no variables, exit and report *satisfiable* (the current variable assignment is the solution). Otherwise, select, according to a *branching rule*, a variable α that does not yet have a value. Generate two subformulae, by fixing α respectively to *True* and *False* and removing from the formula all satisfied clauses and all falsified literals. Solve subproblems recursively. A main drawback of this approach is that the search can be very slow if we do not have procedures to avoid visiting most of the branches of the search tree. In the field of satisfiability, effective techniques to tackle such problem are for instance *learning of new clauses* [3], *non-chronological backtracking* [30], and *necessary assignments* [33]. However, we did not use them in our implementation. Instead, we investigate additional techniques. In order to understand if our techniques by themselves are able to speed up an enumerative approach, we compare them to the very efficient solver SATO 3.2 [34] used just as a state-of-the-art DPLL procedure (by disabling learning of new clauses with the option `-g0`). A recognized technique to speed up the search consists in starting assignment satisfying the more difficult clauses at the beginning of the search. We propose a technique to evaluate clause hardness, which is based on the history of the search, as shown in Sect. 3. When dealing with large scale problems, both computation of the generic branching rule and propagation of the variable fixings are demanding operations. Moreover, the task of proving unsatisfiability is usually computationally harder than proving satisfiability, since it implies exploring all the nodes of the search tree that we could not prune. In order to overcome the above problems, it is customary in mathematical programming to use techniques of *delayed row generation* (see [4] for details). We therefore particularize this to SAT. A set of hard clauses, called *core*, is selected and dynamically updated, in order to be kept small and yet hard to solve. Clauses are chosen by using the above hardness evaluation criterion. The procedure stops as soon as this set becomes unsatisfiable. Details are in Sect. 4.

The procedure is applied to widely-known unsatisfiable problems from the Dimacs test set and to real world problems arising from *data collecting*, where we want the resulting logic formula to be satisfiable.

2 Unsatisfiable Subformulae

Throughout the rest of this section, we assume $\mathcal{F}$ unsatisfiable (otherwise no unsatisfiable subformula could be found in $\mathcal{F}$). An *unsatisfiable subformula* of $\mathcal{F}$ is a set $\mathcal{U}$ of clauses such that:

(1) $\mathcal{U} \subseteq \mathcal{F}$ (in the sense of clause-subset, i.e. $C_j \in \mathcal{U} \Rightarrow C_j \in \mathcal{F}$).
(2) $\mathcal{U}$ is unsatisfiable.

An unsatisfiable subformula can be a proper subformula of $\mathcal{F}$ or coincide with $\mathcal{F}$. Note that some unsatisfiable formulae do not admit proper unsatisfiable subformulae, because they become satisfiable as soon as we remove any of their clauses (e.g. the famous *pigeon hole* Dimacs problems). A *minimal unsatisfiable subformula* (MUS) of $\mathcal{F}$ is a set $\mathcal{M}$ of clauses such that:

(1) $\mathcal{M} \subseteq \mathcal{F}$ (in the sense of clause-subset).
(2) $\mathcal{M}$ is unsatisfiable.
(3) Every proper clause-subset of $\mathcal{M}$ is satisfiable.

In the general case, more than one MUS can be contained in the same $\mathcal{F}$. Some of them can overlap, in the sense that they can share some clauses, but they cannot be fully contained one in another. Therefore, the structure of all MUS of a formula is described by the following straightforward lemma.

Lemma 1 *The collection of all MUS of an unsatisfiable CNF formula $\mathcal{F}$ is a clutter $\mathcal{T}$.*

The concept of MUS has analogies with that one of IIS (irreducible infeasible systems) in the case of systems of linear inequalities [1]. Relations between the concepts of Max-SAT solution and MUS can be investigated. Considering example in Fig. 1, we have the set of clauses corresponding to the solution of the Max-SAT problem $\mathcal{S} = \{C_1, C_2, C_3, C_4, C_6, C_8, C_9, C_{10}\}$, and its complement $\mathcal{F} \setminus \mathcal{S} = \{C_5, C_7\}$. The clutter of all MUS is given by $\mathcal{M}_1 = \{C_4, C_5\}$ and $\mathcal{M}_2 = \{C_7, C_8, C_9\}$. The first is the minimum unsatisfiable subformula. An unsatisfiable subformula approximating $\mathcal{M}_1$ is $\mathcal{U} = \{C_3, C_4, C_5\}$. The following general result holds:

Theorem 2 (**Relation between Max-SAT and MUS**) *Let $\mathcal{F}$ be an unsatisfiable CNF formula. Given any set of clauses $\mathcal{S} \subseteq \mathcal{F}$ corresponding to a Max-SAT solution of $\mathcal{F}$, the complement $\mathcal{F} \setminus \mathcal{S}$ is a minimum transversal*[1] *of the clutter $\mathcal{T}$ of all MUS of $\mathcal{F}$.*

[1] A transversal of a collection of sets $\mathcal{A} = \{A_1, \ldots, A_n\}$ over a ground set A is a set $B \subseteq A : |B \cap A_i| \geq 1$ for all $A_i \in \mathcal{A}$. A minimum transversal is a transversal having minimum cardinality.

Proof: $\mathcal{S}$ cannot entirely contain any MUS, though $\mathcal{S}$ can partially contain any MUS. Therefore, every MUS has at least one clause in $\mathcal{F} \setminus \mathcal{S}$. This proves $\mathcal{F} \setminus \mathcal{S}$ to be a transversal of $\mathcal{T}$. $\mathcal{F} \setminus \mathcal{S}$ has the minimum number $(m - s)$ of clauses, since $\mathcal{S}$ has the maximum number (s) of clauses. Any clause-subset of $\mathcal{F}$ with a number of clauses $u < m - s$ would be the complement of an unsatisfiable set of clauses $\mathcal{S}'$ (by the definition of Max-SAT solution). $\mathcal{S}'$ would therefore contain at least one MUS which is not covered by $\mathcal{F} \setminus \mathcal{S}'$. Consequently, any clause-subset of $\mathcal{F}$ with a number of clauses $u < m - s$ is not a transversal of $\mathcal{T}$. This proves $\mathcal{F} \setminus \mathcal{S}$ to be minimum. $\square$

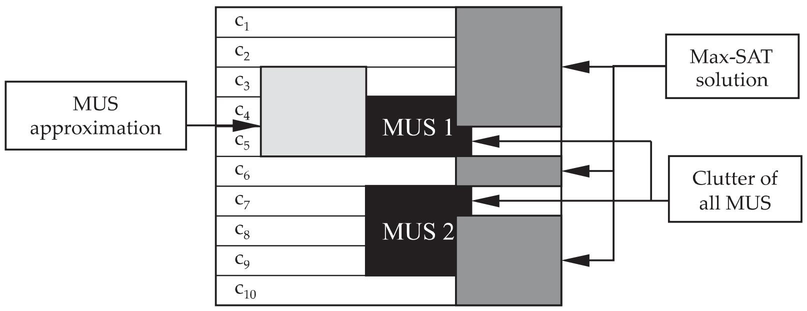

Fig. 1. Relations among the solution of Max-SAT, the clutter of all MUS, and an approximation of the minimum MUS.

The first question depicted in Sect. 1 corresponds to the problem of selecting a MUS, as follows. In the case we want to restore satisfiability by locating, one after another, the problems of the system, this actually means locating each time a MUS, or at least a small unsatisfiable subformula. Within a small number of clauses the original human designer can understand the problem, and re-design only the small parts of the system involved in it. This would hardly be done without such localization process. On the contrary, the complement $\mathcal{F} \setminus \mathcal{S}$ of the set of clauses $\mathcal{S}$ corresponding to a Max-SAT solution is not, in general, an unsatisfiable subset (although it may be), and its location would not help in understanding the problem. In the case we want to keep unsatisfiability, while modifying the system, this again means locating a MUS, or at least a small unsatisfiable subformula. That is the part of the system that should not be changed, while the rest need to be modified according to new specifications. Under special conditions (defined depending on the difference between number of clauses and number of variables), it can be recognized in polynomial time whether a set of clauses is a MUS or not [13,23]. However, to find a MUS within a generic formula is an NP-hard problem, since it implies solving the SAT problem. Moreover, finding a MUS could require much more time than just solving the SAT problem, just like finding an IIS requires much more time than just solving the feasibility of a system of linear inequalities [8]. We propose a procedure to rapidly select a good approximation of a small

MUS, that means an unsatisfiable set of clauses having almost as few clauses as a MUS of small size.

3 Adaptive Branching

Although all clauses of a formula $\mathcal{F}$ should be satisfied, there are clause-subsets of $\mathcal{F}$ which are more "hard-to-satisfy", i.e. which have a small number of satisfying truth assignments, and clause-subsets which are rather "easy-to-satisfy", i.e. which have a large number of satisfying truth assignments. Hardness of a single clause C_j is typically not due to C_j in itself, but to its combination with the rest of the clauses in $\mathcal{F}$. Therefore, *hardness of a clause* C_j will (sometimes implicitly in the following discussion) mean hardness of C_j in the case when C_j belongs to the particular instance $\mathcal{F}$ we are solving.

Our enumeration procedure uses a *clause-based* search tree [28], as follows. At every iteration, a clause C_s to be satisfied is selected. Variables from C_s are therefore selected, and fixed in order to satisfy C_s. Let the first be α_a. If we need to backtrack, the next assignment would not be just the opposite truth value for the same variable α_a, because this would not satisfy C_s. Instead, we select another variable α_b in C_s, and fix α_b in order to satisfy C_s. Moreover, since the previous truth assignment for α_a was not successful, we can also fix the opposite truth value for α_a. If we have no more free variables in C_s, we backtrack to the truth assignments made to satisfy previous clauses, visiting the search tree in a depth-first manner.

Clauses to be satisfied are selected as follows. To begin with, as generally performed in DPLL solution schemes, unit clauses are selected as soon as we have them in the formula, and satisfied by performing unit propagations. After this, starting assignment by satisfying the more difficult clauses is known to be very helpful in reducing backtracks (see e.g. [19]). The difficult point is how to find the hardest clauses. Hardness of a clause is here evaluated by analyzing the history of the search. We say that a clause C_j is *visited* during the exploration of the search tree if we make a truth assignment aimed at satisfying C_j. We *fail* on a clause C_j either when a truth assignment aimed at satisfying C_j produces an empty clause, or when C_j itself becomes empty due to some other truth assignment. Visiting C_j many times shows that C_j is difficult, and failing on it shows even more clearly that C_j is difficult.

Clause hardness adaptive evaluation. *Let v_j be the number of visits of clause C_j, f_j the number of failures due to C_j, p a constant penalty considered for failures, and l_j the length of C_j. A hardness evaluation of C_j in $\mathcal{F}$ is given by*

$$\varphi(C_j) = (v_j + p f_j) \;/ l_j$$

Computing such evaluation requires very little overhead, and its quality improves as the search proceeds. Altogether, we visit our clause-based search tree using the following clause selection criterion:

Adaptive clause selection

(1) *Select all unit clauses* C_{unit}.
(2) *When no unit clauses are present, select clause* $C_{\max}$ *as follows:*

$$C_{\max} = \underset{\substack{C_j \in \mathcal{F} \\ C_j \text{ still unsatisfied}}}{\arg\max} \; \varphi(C_j)$$

Within the set of variables appearing in $C_{\max}$, the order of variable fixings is the following. Let $J_2(\alpha_k)$ be the number of binary clauses containing literal α_k. We select at each step the truth assignment corresponding to the literal $\alpha_{\max}$.

$$\alpha_{\max} = \underset{\substack{\alpha_k \in C_{\max} \\ \alpha_k \text{ still unassigned}}}{\arg\max} \; (1 + J_2(\alpha_k))(1 + J_2(\neg\alpha_k))$$

This, introduced in [9], is an approximation of the two-sided Jeroslow-Wang rule [20,19].

The above is a complete scheme: if a satisfying truth assignment exists, it will be reached, or, if the search tree is completely explored, the instance is unsatisfiable. We refer to the above branching and backtracking scheme as *adaptive branching*.

4 Unsatisfiable Subformula Selection

The practical question introduced in Sect. 1 can be solved by locating the subsets of clauses causing unsolvability, as described in Sect. 2. In order to reach such purpose, within a complete solution framework, we develop an heuristic procedure which can guarantee to find an unsatisfiable subformula, and is aimed to find an approximation of a small MUS.

The algorithmic question introduced in Sect. 1 is about overcoming some structural defects of a DPLL approach. In this approach, two computationally demanding operations are computation of the generic branching rule, that is to choose the variable fixings to perform, and propagation of such variable fixings, that is to remove from the formula all satisfied clauses (unit subsumption) and all falsified literals (unit resolution). Modern solvers try in different ways to overcome this, for instance by postponing some operations during unit propagation [35]. Moreover, the task of proving unsatisfiability is usually computationally harder than proving satisfiability, since it implies exploring all the nodes of the search tree that we could not prune. When dealing with large-scale problems, it is customary in mathematical programming to use techniques of *delayed row generation* (see [4] for details). Such approaches are motivated by the speed-up we obtain when considering only a portion of the entire problem at every single step. The key issue is that the solution obtained by solving such subproblems is valid for the entire problem. We are therefore particularizing this to SAT. By using the above hardness evaluation, we progressively select a subset of hard clauses, that we call a *core*. We solve the core without propagating assignments to clauses out of the core. If the core is unsatisfiable, this proves that the whole formula is unsatisfiable. If the core is satisfiable, we extend current (partial) solution to a larger subset of clauses (a bigger core), until solving the whole formula, or stopping at an unsatisfiable subformula. Core composition is dynamically updated, in order to keep it small and yet hard to solve.

The procedure developed to tackle both of the above questions is called *adaptive core search* (ACS, see also [25]), and works as follows.

Adaptive core search

- **Preprocessing** *Perform d branching iterations on $\mathcal{F}$ (or less than d if $\mathcal{F}$ is solved before), using shortest clause rule. Initial core $\mathcal{C}_0$ is empty. (If the instance is already solved, Stop.)*
- **Base** *Add to $\mathcal{C}_0$ a fixed percentage c of the clauses of $\mathcal{F}$, giving priority to hardest clauses. Obtain a new core $\mathcal{C}_1$. Remaining clauses form $\mathcal{O}_1$.*
- **Iteration k** *Perform b branching iteration on current core $\mathcal{C}_k$ (or less than b if $\mathcal{C}_k$ is solved before), ignoring $\mathcal{O}_k$, using adaptive branching. We have one of the following cases* **a, b, c**:
  - · **a** *$\mathcal{C}_k$ is unsatisfiable $\Rightarrow$ $\mathcal{F}$ is unsatisfiable, $\mathcal{C}_k$ is the selected unsatisfiable subformula. Stop.*
  - · **b** *No answer after b iterations $\Rightarrow$ Contraction: Form a new core $\mathcal{C}_{k+1}$ by selecting a fixed percentage c of the clauses of $\mathcal{C}_k$, giving priority to hardest clauses. Put $k := k + 1$, goto* **k.**
  - · **c** *$\mathcal{C}_k$ is satisfied by solution S_k $\Rightarrow$ Expansion: Form a new core $\mathcal{C}_{k+1}$ by adding to $\mathcal{C}_k$ a fixed percentage c of the clauses of $\mathcal{O}_k$. First give priority to clauses falsified by S_k, and then give priority to hardest clauses. Put $k := k + 1$, goto* **k.**

Preprocessing serves to give initial values of visits and failures, in order to compute φ. After this, we try to solve the subset of the hardest clauses as if they were our entire instance. If they are an unsatisfiable instance, we stop. If current core $\mathcal{C}_k$ is not solved by b iterations of adaptive branching, this means that $\mathcal{C}_k$ is too large, and must be reduced. In such case, the current truth assignment also should be changed, and it is faster to completely rebuild it.

Finally, if we find a satisfying solution for $\mathcal{C}_k$, we try to extend it to the rest of the clauses. If some clauses are falsified, this means that they are difficult (together with the clauses of the core), and therefore they should be added to $\mathcal{C}_k$. In this case, the current truth assignment falsifies some clauses now in the core, and should be changed. Changing it by backtracking would imply performing a large number of backtracks. Therefore, in such situation also, it is faster to completely rebuild the truth assignment.

The iteration step is repeatedly applied to instances until their solution. In order to ensure termination, solution rebuilding is allowed only a finite number of times r. After that, the *contraction* phase is no longer allowed, and the solution is not entirely rebuilt after the *expansion* phase, but modified by performing backtrack. In other words, the algorithm may evolve until it becomes a branching procedure which can only perform expansion and backtrack. This is called *intensification* phase. Therefore:

Theorem 3 (**Correctness and completeness**) *ACS is a correct and complete solution scheme for the satisfiability problem.*

Proof: After a finite number of contractions without reaching the solution, ACS switches to intensification. The algorithm evolves into a branching and backtracking procedure working on current core $\mathcal{C}_c$. Branching and backtracking is performed until $\mathcal{C}_c$ is solved. It is well known that branching and backtracking is a correct and complete solution scheme for satisfiability. In case $\mathcal{C}_c$ is satisfied by a solution S_c, the procedure adds clauses from $\mathcal{F} \setminus \mathcal{C}_c$. Since clauses of $\mathcal{F} \setminus \mathcal{C}_c$ are a finite number, and clauses added during intensification phase are never removed, termination is guaranteed. Finally, branching and backtracking after clauses addition continues from the partial truth assignment S_c. Since this corresponds to solving each new core with branching and backtracking, correctness is guaranteed. $\square$

As for the practical question of selecting an unsatisfiable subformula, the following holds:

Theorem 4 (**Unsatisfiable subformula selection**) *ACS can guarantee to always find an unsatisfiable subformula $\mathcal{U} \subseteq \mathcal{F}$ if it exists, i.e. when $\mathcal{F}$ is unsatisfiable.*

Proof: In the case of an unsatisfiable instance $\mathcal{F}$, ACS is guaranteed to stop detecting unsatisfiability (by Theor. 3). Such termination can only happen due to the unsatisfiability of a current core $\mathcal{C}_u$ (case **a** of the u-th *iteration* phase). Since the solution is rebuilt at any *contraction*, there is no risk to stop due to an erroneous detection of unsatisfiability of $\mathcal{C}_u$ when $\mathcal{C}_u$ is actually satisfiable. The unsatisfiable subformula $\mathcal{C}_u$ is therefore always selected in the case of an unsatisfiable instance $\mathcal{F}$. $\square$

Moreover, ACS is aimed to find an approximation of a *minimum* unsatisfiable subformula. This because, by progressively selecting hard clauses, and performing several core *expansions* and *contractions* (expecially when b is small), ACS is often able to locate the core on a small MUS, as shown by computational experience in Sect. 5 (although, of course, the size of the minimum MUS is often unknown).

As for the algorithmic question, core search framework has the important feature of considering smaller subproblems at the nodes of the search tree. Therefore, all operations performed, in particular computation of the branching rule and unit propagation consequent to any variable fixing, are performed only on the current $\mathcal{C}_k$. This reduces time needed for them. Moreover, unsatisfiability can be proved by solving only the core subformula, hence exploring a smaller search tree. Evidently, additional techniques to prune the search tree (learning of new clauses to begin with) can be integrated in such framework.

Parameters (d, b, c) greatly affect the result. They can be set in order to minimize the size of unsatisfiable subformulae selected, or to maximize the speed up of a DPLL-style procedure solving the SAT problem. The first result is obtained by using values for d of the order of $2 \times m$, values for b of the order of $5 \sim 100$, and values for c very small, e.g. $0.1 \sim 0.01$, and in any case proportional to the size of the expected unsatisfiable subformula. The latter result is obtained by using values for d of the order of m, values for b in the range of 500 to 5000, and values for c in the range of 10 to 30.

5 Computational Results

We report results on well-known artificially generated instances from the Dimacs [2] test set, and on real-life instances arising from *data collecting* problems. Since parameters can be set in order to minimize the size of unsatisfiable subformulae selected, or to maximize the speed up of a DPLL-style procedure, we report two different kind of tables for the two purposes.

In the tables regarding unsatisfiable subformula selection, we report number

[2] NFS Science and Technology Center in Discrete Mathematics and Theoretical Computer Science - A consortium of Rutgers University, Princeton University, AT&T Bell Labs, Bellcore.

of variables 'n' and of clauses 'm' in the original instance and in the smallest unsatisfiable subformula $\mathcal{U}$ selected. Column 'rest' reports if the formula obtained by removing $\mathcal{U}$ is satisfiable (S) or not (U). Column 'MUS' reports if $\mathcal{U}$ is minimal (Y) or not (N). This could be tested. Parameter p (failure penalty in hardness evaluation) was set at 10. Parameter r (maximum number of solution rebuilding) was set at 500. Parameter d (number of branching iterations in preprocessing), b (number of branching iterations in every branching phase), c (percentage of core variations), have not single preferable values. We give the values corresponding to the smallest unsatisfiable subformula $\mathcal{U}$ selected, and the CPU time elapsed for this. The size of selected subformula is very sensible to parameters' values, expecially c and b. In order to give an idea, the cardinality (ranging form 60 to 850 clauses) of the unsatisfiable subformula selected by varying c (from 1 to 10) and b (from 10 to 20) is also given, in the case of problem *jnh*2 (see Fig. 2). Small parameters variations cause large variations in the cardinality of the selected subformula $\mathcal{U}$. However, several parameters values allow small cardinality for $\mathcal{U}$.

In the tables regarding the speed-up of a DPLL procedure, we compare to SATO 3.2 [34] used as a state-of-the-art DPLL procedure by disabling learning of new clauses with the option `-g0`, and to a simplified version of our procedure which does not use core search, but does use the adaptive branching strategy. All of the three procedures do not use learning of new clauses. Columns labeled 'n' and 'm' are number of variables and clauses. Column 'AdBr' reports time used by the adaptive branching procedure which does not use core search. Column 'ACS Sel.' reports the time elapsed till the selection of the last core (i.e. the unsatisfiable subformula selected) in adaptive core search. Column 'ACS Solv.' reports the time used to solve the last core with the branching procedure of ACS. Column 'ACS Tot.' reports the total time for solving the instance by ACS. Such times are obtained by using the default parameters' values $d = m$ (the number of clauses), $b = 5000$, $c = 30$. Column 'SATO `-g0`' reports solution time for solving the instance using SATO with the option `-g0`. Times are in CPU seconds on a Pentium II 450 MHz. Time limit was 600 seconds.

5.1 Dimacs Problems

We report results on the unsatisfiable series of problems from the Dimacs test set, since they are very widely-known and easily available [3]. Such series are *aim*, *dubois*, *hole*, *jnh*, *pret*. Each problem of the series *dubois*, *hole*, and *pret* is a MUS, hence no smaller unsatisfiable subformula can be found in it. We

[3] Available from
`ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/`

therefore consider the series *aim* and *jnh*. Note that, on the contrary, problems which are MUS are very rare in real world.

The series *aim* is constituted by 3-SAT instances artificially generated by K. Iwama, E. Miyano and Y. Asahiro. They are nowadays easily solved by several SAT solvers, being small in size. Nevertheless, they have a structure more difficult than usual real problems. This results in the presence of unsatisfiable subformulae larger than those selected in the case of real-life problems of Sect. 5.2. Results on them are in Tables 1 and 2.

Original formula			Selected $\mathcal{U}$				Parameters			
Problem	n	m	n	m	rest	MUS	d	b	c	time
aim-50-1_6-no-1	50	80	20	22	S	Y	80	10	10	0.1
aim-50-1_6-no-2	50	80	28	32	S	Y	80	10	15	0.1
aim-50-1_6-no-3	50	80	28	31	S	Y	80	10	15	0.1
aim-50-1_6-no-4	50	80	18	20	S	Y	80	10	10	0.0
aim-50-2_0-no-1	50	100	21	22	S	Y	100	10	15	0.1
aim-50-2_0-no-2	50	100	28	31	S	N	100	10	20	0.3
aim-50-2_0-no-3	50	100	22	28	S	Y	100	15	20	0.0
aim-50-2_0-no-4	50	100	18	21	S	Y	100	15	20	0.3
aim-100-1_6-no-1	100	160	43	47	S	Y	160	20	20	1.2
aim-100-1_6-no-2	100	160	46	54	S	N	160	65	15	4.5
aim-100-1_6-no-3	100	160	51	57	S	N	160	60	15	4.6
aim-100-1_6-no-4	100	160	43	48	S	Y	160	48	20	2.5
aim-100-2_0-no-1	100	200	18	19	S	Y	200	12	8	0.5
aim-100-2_0-no-2	100	200	35	39	S	Y	200	16	15	0.9
aim-100-2_0-no-3	100	200	25	27	S	Y	200	30	10	1.8
aim-100-2_0-no-4	100	200	26	32	S	N	200	40	15	1.6
aim-200-1_6-no-1	200	320	52	55	S	Y	320	30	15	2.6
aim-200-1_6-no-2	200	320	76	82	S	N	640	60	24	43.0
aim-200-1_6-no-3	200	320	77	86	S	N	640	65	25	300
aim-200-1_6-no-4	200	320	44	46	S	Y	640	34	10	2.3
aim-200-2_0-no-1	200	400	49	54	S	N	400	40	12	3.7
aim-200-2_0-no-2	200	400	46	50	S	Y	400	35	10	3.0
aim-200-2_0-no-3	200	400	35	37	S	Y	400	35	7	0.4
aim-200-2_0-no-4	200	400	36	42	S	Y	400	12	7	0.8

Table 1: Unsatisfiable subformula selection on the *aim* series: 3-SAT artificially generated problems.

Problem	n	m	AdBr	ACS Sel.	ACS Solv.	ACS Tot.	SATO (`-g0`)
aim-100-1_6-no-1	100	160	1.09	0.17	0.03	0.20	135.96
aim-100-1_6-no-2	100	160	0.67	0.54	0.39	0.93	0.14
aim-100-1_6-no-3	100	160	3.91	0.62	0.73	1.35	0.01
aim-100-1_6-no-4	100	160	0.52	0.61	0.35	0.96	103.30
aim-100-2_0-no-1	100	200	0.03	0.03	0.01	0.04	72.12
aim-100-2_0-no-2	100	200	0.38	0.05	0.04	0.09	105.96
aim-100-2_0-no-3	100	200	0.12	0.04	0.01	0.05	28.65
aim-100-2_0-no-4	100	200	0.11	0.04	0.01	0.05	85.01
aim-200-1_6-no-1	200	320	5.02	0.12	0.09	0.21	>600
aim-200-1_6-no-2	200	320	>600	14.04	32.31	46.35	>600
aim-200-1_6-no-3	200	320	>600	10.80	35.87	46.67	>600
aim-200-1_6-no-4	200	320	5.81	0.09	0.10	0.19	>600
aim-200-2_0-no-1	200	400	15.53	0.20	0.27	0.47	>600
aim-200-2_0-no-2	200	400	3.87	0.17	0.18	0.35	>600
aim-200-2_0-no-3	200	400	1.04	0.05	0.12	0.17	>600
aim-200-2_0-no-4	200	400	0.70	0.16	0.02	0.18	>600

Table 2: Comparison on the *aim* series: 3-SAT artificially generated problems.

Original formula			Selected $\mathcal{U}$				Parameters			
Problem	n	m	n	m	rest	MUS	d	b	c	time
jnh2	100	850	51	60	S	N	850	17	3	3.2
jnh3	100	850	92	173	S	N	8387	110	16	29.7
jnh4	100	850	86	140	S	N	2550	77	15	8.2
jnh5	100	850	85	125	S	N	1700	85	14	7.7
jnh6	100	850	88	159	S	N	2550	80	17	22.9
jnh8	100	850	70	91	S	N	646	37	6	0.6
jnh9	100	850	78	118	S	N	1750	65	9	1.0
jnh10	100	850	95	161	S	N	1700	160	12	0.1
jnh11	100	850	79	129	S	N	1700	160	11	19.0
jnh13	100	850	77	106	S	N	2550	145	10	0.1
jnh14	100	850	87	124	S	N	5100	149	11	0.5
jnh15	100	850	87	140	S	N	850	140	12	1.4
jnh16	100	850	100	321	S	N	1700	160	30	55.8
jnh18	100	850	91	168	S	N	850	146	17	40.6
jnh19	100	850	78	122	S	N	2550	101	10	7.4
jnh20	100	850	81	120	S	N	1700	120	9	0.7

Table 3: Unsatisfiable subformula selection on the *jnh* series: randomly generated hard problems.

The series *jnh* is constituted by random instances generated by J.N. Hooker. Each variable occurs in a given clause with probability p, and it occurs negative or positive with equal probability. Probability p is chosen so that the expected number of literals per clause is 5. Empty clauses and unit clauses are rejected. Results on them are in Tables 3 and 4.

Problem	n	m	AdBr	ACS Sel.	ACS Solv.	ACS Tot.	SATO (`-g0`)
jnh2	100	850	0.13	0.02	0.01	0.03	0.01
jnh3	100	850	1.06	0.45	0.55	1.00	0.02
jnh4	100	850	0.41	0.10	0.07	0.17	0.02
jnh5	100	850	0.22	0.10	0.04	0.14	0.01
jnh6	100	850	0.66	0.43	0.09	0.52	0.02
jnh8	100	850	0.23	0.10	0.03	0.13	0.01
jnh9	100	850	0.16	0.10	0.03	0.13	0.03
jnh10	100	850	0.11	0.08	0.01	0.09	0.03
jnh11	100	850	0.38	0.33	0.03	0.36	0.02
jnh13	100	850	0.28	0.04	0.01	0.05	0.01
jnh14	100	850	0.19	0.35	0.02	0.37	0.03
jnh15	100	850	0.31	0.41	0.03	0.44	0.01
jnh16	100	850	7.02	2.10	3.94	6.04	0.09
jnh18	100	850	0.65	0.45	0.15	0.60	0.02
jnh19	100	850	0.24	0.33	0.04	0.37	0.02
jnh20	100	850	0.26	0.35	0.02	0.37	0.01

Table 4: Comparison on the *jnh* series: randomly generated hard problems.

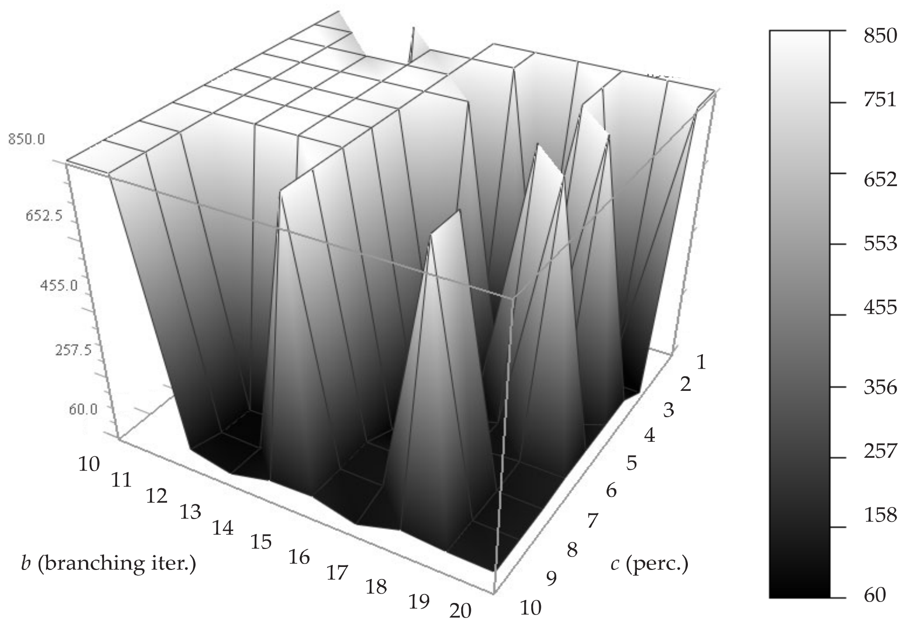

Fig. 2. Cardinality of the unsatisfiable subformula selected in *jnh*2 for different values of b and c.

5.2 Data Collecting Problems

When dealing with a large number of collected information, which could contain errors, the relevant problem of *error detection* arises. Error detection is generally approached by formulating a set of rules that the *data records* must respect in order to be declared *correct*. The more accurate and careful the rules are, the more truthful individuation of correct and erroneous data can be achieved. A first problem arising from this is the validation of such set of rules. In fact, the rules could contain some contradiction among themselves. This could result in erroneous records to be declared correct, and vice versa. The problem of checking the set of rules against inconsistencies can be transformed into a sequence of SAT problems (see [5] for details). Every unsatisfiable instance obtained reveals an inconsistency in the set of rules. In such case, we couldn't just remove some rules to restore consistency. On the contrary, we need to locate the entire set of conflicting rules, in order to let the human expert understand the problem and solve it by modifying some rules. That problem would hardly be understood by the expert without such localization of conflicting rules.

In Table 5 we report results on some instances encoding the set of rules called *data_1* (developed for a real census). They produced a main SAT instance and a sequence of derived instances (data_1.x), some of which resulted unsatisfiable. We show only some of the unsatisfiable ones. Such instances are large but structurally easy. Since inconsistencies are unwanted, they generally contained only one MUS of very small size.

Original formula			Selected $\mathcal{U}$				Parameters			
Problem	n	m	n	m	rest	MUS	d	b	c	time
data_1.0	1960	10420	2	3	S	Y	1000	4	0.01	0.1
data_1.1	1958	10415	2	3	S	Y	1000	4	0.01	0.1
data_1.2	1957	10418	2	4	S	Y	1000	4	0.01	0.1
data_1.3	1953	10410	3	4	S	Y	1000	4	0.01	0.0
data_1.4	1958	10412	2	3	S	Y	1000	4	0.01	0.1
data_1.5	1948	10400	2	3	S	Y	1000	4	0.01	0.1
data_1.6	1956	10416	2	4	S	Y	1000	4	0.01	0.0
data_1.7	1952	10411	2	3	S	Y	1000	4	0.01	0.0
data_1.8	1950	10420	3	5	S	N	1000	4	0.01	0.0
data_1.9	1955	10413	2	4	S	Y	1000	4	0.01	0.0

Table 5: Unsatisfiable subformula selection on instances encoding rules for data collecting problems.

6 Conclusions

In several applicative fields, in addition to solving the SAT problem, one need to locate a MUS, or at least a small unsatisfiable subformula of a given unsatisfiable formula. During the solution of SAT by means of a complete enumeration technique altogether denominated *adaptive core search*, we are able to evaluate clause hardness, by analyzing the history of the search. By progressively selecting hard clauses, in the case of unsatisfiable instances, we are guaranteed to find an unsatisfiable subformula. Moreover, in almost all of the analyzed real problems arising from data collecting, and in several Dimacs problems, our procedure is able to find a minimal unsatisfiable subformula.

Common drawbacks of DPLL procedures for solving the SAT problem are: 1) computation of branching rule can be time-consuming; 2) propagation of variable fixings is even more time-consuming; 3) unsatisfiability requires complete exploration of the search tree. Modern solvers try in several ways to overcome these problems. The procedure of adaptive core search is able to reduce time needed for the above three operations by working only on a subset of hard clauses called core. Comparisons with SATO 3.2 used just as a state-of-the-art DPLL procedure shows the effectiveness of proposed procedure.

References

[1] E. Amaldi, M.E. Pfetsch, and L. Trotter, Jr. Some structural and algorithmic properties of the maximum feasible subsystem problem. In *Proc. of 10th Integer Programming and Combinatorial Optimization conference.* Lecture Notes in Computer Science 1610 (Springer-Verlag, 1999) 45–59.

[2] R. Battiti and M. Protasi. Approximate Algorithms and Heuristics for MAX-SAT. In D.Z. Du and P.M. Pardalos eds. *Handbook of Combinatorial Optimization* **1** (Kluwer Academic Publishers, 1998) 77-148.

[3] R.J. Bayardo Jr. and R.C. Schrag. Using CSP lookback techniques to solve exceptionally hard SAT instances. In *Proc. of Principles and Practice of Constraint Programming - CP96* (Springer-Verlag, 1996) 46–60.

[4] D. Bertsimas and J.N. Tsitsiklis. *Introduction to Linear Optimization.* (Athena Scientific, Belmont, Massachusetts, 1997).

[5] R. Bruni and A. Sassano. Errors Detection and Correction in Large Scale Data Collecting. In F. Hoffmann et al. eds. *Advances in Intelligent Data Analysis*, Lecture Notes in Computer Science 2189 (Springer-Verlag, 2001) 84–94.

[6] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers* **C35**(8) (1986) 677–691.

[7] V. Chandru and J.N. Hooker. *Optimization Methods for Logical Inference.* (Wiley, New York, 1999).

[8] J.W. Chinneck and E.W. Dravnieks. Locating Minimal Infeasible Constraint Sets in Linear Programs. *ORSA Journal on Computing* **3** (1991) 157-168.

[9] J. Crawford and L. Auton. Experimental results on the crossover point in Satisfiability problems. *In Proceedings AAAI-93* (1993) 22–28.

[10] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. Assoc. for Comput. Mach.* **5** (1962) 394–397.

[11] M. Davis and H. Putnam. A computing procedure for quantification theory. *Jour. Assoc. for Comput. Mach.* **7** (1960) 201–215.

[12] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT versus UNSAT. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, **26** (1996) 415–436.

[13] H. Fleischner and S. Szeider. Polynomial-time Recognition of Minimal Unsatisfiable Formulas with Fixed Clause-Variable Difference. *ECCC* TR-00-049 (2000).

[14] A. Van Gelder and Y.K. Tsuji. Satisfiability testing with more reasoning and less guessing. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **26** (1996) 559–586.

[15] I.P. Gent, H. van Maaren, and T. Walsh editors. *SAT 2000* (IOS Press, Amsterdam, 2000).

[16] J. Groote and J. Warners. The propositional formula checker HeerHugo. In I.P. Gent, H. van Maaren, and T. Walsh eds. *SAT 2000* (IOS Press, Amsterdam, 2000).

[17] J. Gu. Optimization Algorithms for the Satisfiability (SAT) Problem. In Ding-Zhu Du ed. *Advances in Optimization and Approximation* (Kluwer Academic Publishers, 1994) 72–154.

[18] J. Gu, P.W. Purdom, J. Franco, and B.W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. *DIMACS Series in Discrete Mathematics* (American Mathematical Society, 1999).

[19] J.N. Hooker and V. Vinay. Branching Rules for Satisfiability. *Journal of Automated Reasoning* **15** (1995) 359–383.

[20] R.E. Jeroslow and J. Wang. Solving Propositional Satisfiability Problems. *Annals of Mathematics and AI* **1** (1990) 167–187.

[21] D.S. Johnson and M.A. Trick, editors. *Cliques, Coloring, and Satisfiability*, volume **26** of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* (American Mathematical Society, 1996).

[22] E. de Klerk, H. van Maaren, and J.P. Warners. Relaxations of the Satisfiability Problem using Semidefinite Programming. In I.P. Gent, H. van Maaren, and T. Walsh eds. *SAT 2000* (IOS Press, Amsterdam, 2000).

[23] O. Kullmann. An application of matroid theory to the SAT Problem. *ECCC* TR00-018 (2000).

[24] D.W. Loveland. *Automated Theorem Proving: a Logical Basis.* (North Holland, 1978).

[25] C. Mannino and A. Sassano. Augmentation, Local Search and Learning. *AI-IA Notizie* XIII (2000) 34–36.

[26] J.P. Marques-Silva and K.A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers* **48**(5) (1999) 506-521.

[27] D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT Problems. In *Proceedings of AAAI* (1992) 459–465.

[28] B. Monien and E. Speckenmeyer. Solving satisfiability in less than 2^n steps. *Discrete Applied Mathematics* **10** (1985) 287-295.

[29] D. Pretolani. Efficiency and stability of hypergraph SAT algorithms. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **26** (1996) 479–498.

[30] R. Stallman and G.J. Sussman. Forward reasoning and dependency directed backtracking. *Artificial Intelligence* **9**(2) (1977) 135–196.

[31] G. Stålmarck and M. Säflund. Modeling and Verifying Systems and Software in Propositional Logic. *in proc. of Internat. Conf. on Safety of Computer Control Systems* (Pergamon Press, Oxford, 1990) 31-36.

[32] K. Truemper. *Effective Logic Computation.* (Wiley, New York, 1998).

[33] R. Zabih and D. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of AAAI'88* (1988) 155–160.

[34] H. Zhang. SATO: An Efficient Propositional Prover. *in Proc. of International Conference on Automated Deduction (CADE-97)*, Lecture notes in Artificial Intelligence 1104 (Springer-Verlag, 1997) 308–312.

[35] H. Zhang and M.E. Stickel. Implementing the Davis-Putnam Method. In I.P. Gent, H. van Maaren, and T. Walsh eds. *SAT 2000* (IOS Press, Amsterdam, 2000).