Restoring Satisfiability or Maintaining Unsatisfiability by finding *small* Unsatisfiable Subformulae

R. Bruni and A. Sassano

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

{bruni,sassano}@dis.uniroma1.it

Abstract

In several applicative fields, the generic system or structure to be designed can be encoded as a CNF formula, which should have a well-defined satisfiability property (either to be satisfiable or to be unsatisfiable). Within a complete solution framework, we develop an heuristic procedure which is able, for unsatisfiable instances, to locate a set of clauses causing unsatisfiability. That corresponds to the part of the system that we respectively need to re-design or to keep when we respectively want a satisfiable or unsatisfiable formula. Such procedure can guarantee to find an unsatisfiable subformula, and is aimed to find an approximation of a *minimum* unsatisfiable subformula. Successful results on both real life data collecting problems and Dimacs problems are presented.

Key words: Consistency restoring, MUS location, (Un)Satisfiability.

1 Introduction

Several problems arising from fields as different as Artificial Intelligence, Cryptography, Database Systems, Machine Vision, VLSI design and testing, are usually encoded into propositional logic formulae. A propositional formula $\mathcal{F}$ in *conjunctive normal form* (CNF) is a conjunction of clauses C_j, each clause being a disjunction of literals, each literal being either a positive (x_i) or a negative ($\neg x_i$) propositional variable. By denoting with l_i the cardinality of C_j, and with $[\neg]$ the possible presence of $\neg$, this is

$$\bigwedge_{j=1..m} \Big(\bigvee_{i=1..l_j} [\neg]\, x_i \Big)$$

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. This problem is well-known NP-complete. Generally, when an instance $\mathcal{F}$ encodes the system or structure one must design, we desire $\mathcal{F}$ 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, 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*. Max-SAT is well-known NP-hard, and, unless P=NP, it cannot be approximated in polynomial time within a performance ratio greater than 7/8 [2]. Note that Max-SAT gives an answer to the SAT problem as well. 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 $\mathcal{F} \setminus \mathcal{S}$. If we have for example an inconsistent knowledge base, consistency can be restored by deleting some of the information contained in it. 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, it often happens that we want to modify the system, e.g. to reduce its cost. 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, as explained in Sect. 2. 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. While solving SAT by means of an enumeration approach, an analysis on the history of the search guide us to the individuation of "hard-to-satisfy" clauses, as illustrated in Sect. 3. We therefore propose, in Sect. 4, an algorithm to find an approximation of the minimum unsatisfiable subformula, by selecting "hard-to-satisfy" clauses until we obtain an unsatisfiable subformula. This procedure is applied to real world problems arising from data collecting, where we want the resulting logic formula to be satisfiable. Moreover, the procedure is applied to widely-known unsatisfiable problems from the Dimacs collection.

2 Unsatisfiable Subformulae

A CNF instance $\mathcal{F}$ can be viewed simply as a set of clauses C_j. Throughout the rest of this section, we assume $\mathcal{F}$ unsatisfiable (otherwise no unsatisfiable subformula could be found in it).

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 $\mathcal{F}$. However, some formulae $\mathcal{F}$ do not admit proper unsatisfiable subformulae, because they become satisfiable as soon as we remove any of its 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, we can have more than one MUS 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. Formally, the collection of all MUS of $\mathcal{F}$ is a *clutter*. The concept of MUS have analogies with that one of IIS (irreducible infeasible systems) in the case of systems of linear inequalities [1].

A *minimum unsatisfiable subformula* (minMUS) of $\mathcal{F}$ is a minimum cardinality MUS. We denote with μ its cardinality.

To find a MUS within a formula is an NP-hard problem, since it implies solving the SAT problem. Moreover, finding a MUS typically requires 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 [6]. We therefore introduce the concept of approximation for an unsatisfiable subformula. Being m the number of clauses in $\mathcal{F}$, and μ the cardinality of any minMUS, an unsatisfiable subformula $\mathcal{U}$ of *approximation* ε (with $0 \leq \varepsilon \leq m - \mu$) of $\mathcal{F}$ is an unsatisfiable subformula $\mathcal{U}$ having cardinality $c = \mu + \varepsilon$. A 0-approximation unsatisfiable subformula is a minMUS.

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, and we have $\mu = 2$. An 1-approximation unsatisfiable subformula is $\mathcal{U} = \{C_3, C_4, C_5\}$. The following general result holds:

Relation between Max-SAT and MUS
Let $\mathcal{F}$ be an unsatisfiable CNF formula. The complement $\mathcal{F} \setminus \mathcal{S}$ of any set of clauses $\mathcal{S}$ corresponding to a Max-SAT solution of $\mathcal{F}$ is a cover of the clutter of all MUS of $\mathcal{F}$.

Proof: $\mathcal{S}$ cannot entirely contain any MUS. However, $\mathcal{S}$ can partially contain all MUS. Therefore, every MUS has at least one clause in $\mathcal{F} \setminus \mathcal{S}$.

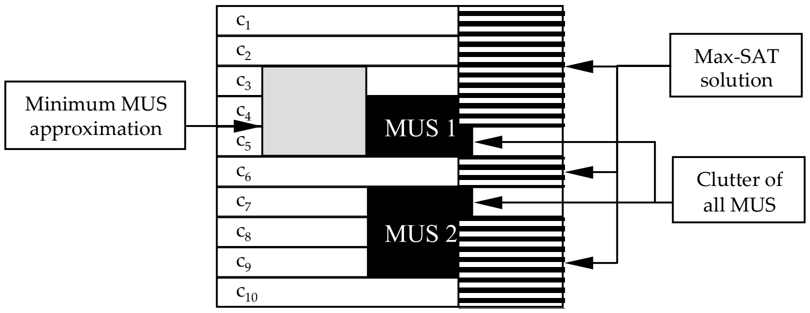

Fig. 1. Relations among the solution of Max-SAT, the clutter of all MUS, and an 1-approximation unsatisfiable subformula

The problems depicted in Sect. 1 correspond to the problem of selecting a minMUS, as follows. In the case we want to restore satisfiability by locating only the small part of the system causing the problem, this actually means locating a minMUS, or at least a MUS or a low-approximation unsatisfiable subformula. Re-design of that part is another issue, and, typically, requires the work of the original human designer. *Postinfeasibility analysis*, in fact, always requires the cooperation of *algorithmic engine* and *human intelligence* [6]. The process could need to be repeated until all MUS are removed from the formula. On the contrary, the set of clauses not satisfied by a Max-SAT solution is not, in general, an unsatisfiable subset (although it may be), hence 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 minMUS, or at least a MUS or a low-approximation unsatisfiable subformula. That is the part of the system that should not be changed.

3 Clause Hardness Evaluation

Many algorithms for the SAT problem have been proposed (see for instance [5,8,9,14] for extensive references). Most of complete methods are based on enumeration techniques and perform a tree search. They are also called Davis-Putnam-Loveland variants [7,12], and have the following general structure:

DPL scheme

(1) Choose a variable x according to a branching rule (see e.g. [10]). Generally, priority is given to variables appearing in unit clauses (unit resolution).
(2) Fix x to a truth value and remove all satisfied clauses and all falsified literals.
(3) If an empty clause is obtained, backtrack and change a truth assignment. Usually, change the last replaceable one (depth-first exploration of the search tree).
Repeat (1,2,3) until *i*) a satisfying solution is found: the formula is satisfiable, or *ii*) the search tree is completely explored and no satisfying solution is found: the formula is unsatisfiable.

During such enumeration, we could consider the history of the search as an information on the structure of the formula. Such analysis guide us to the individuation of "hard-to-satisfy" clauses within the formula. Note that hardness of a clause is typically not due to the single clause in itself, but to its combination with the rest of the clauses in $\mathcal{F}$. Therefore, we will speak of *hardness* of a clause C_j in the case when C_j belongs to the particular instance $\mathcal{F}$ we are considering. This, however, will be implicit in the following.

Our SAT solution algorithm uses a clause-based search tree [13,10]. 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 x_a. If we need to backtrack, the next assignment would not be just the opposite truth value for the same variable x_a, because this would not satisfy C_s. Instead, we select another variable x_b in C_s, and fix x_b in order to satisfy C_s. Moreover, since the previous truth assignment for x_a was not successful, we can also fix the opposite truth value for x_a. Clause selection is as follows. Priority is given to unit clauses (unit resolution). After them, since starting assignment by satisfying the more difficult clauses is known to be very helpful in reducing backtracks [10], we select hardest clauses. Hardness of a clause is evaluated by means of the below $\varphi(C_j)$.

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 if a truth assignment aimed at satisfying C_j produces an empty clause, or if C_j itself becomes empty due to some other truth assignment. We evaluate the difficulty of a clause C_j by counting how many times C_j is visited, and how many times the enumeration fails on C_j. Visiting C_j many times shows that C_j is difficult, and failing on it shows even more clearly that C_j is difficult. Moreover, as known, the shorter a clause is, the harder that clause is.

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 penalty considered for failures, and l_j the length of C_j. An hardness evaluation of C_j in $\mathcal{F}$ is given by

$$\varphi(C_j) = (v_j + pf_j) \,/\, l_j$$

Counting visits and failures has the important feature of requiring very little overhead. The quality of such evaluation improves as the search proceeds.

4 Unsatisfiable Subformula Selection

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. In case 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. This can be related to well-known techniques of row generation often used in mathematical programming.

Adaptive core search

- **Preprocessing** Perform d branching iterations using just 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 $\mathcal{C}_k$, ignoring $\mathcal{O}_k$, using adaptive clause selection. We have one of the following **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. Then, we try to solve a set of hard clauses as if they were our entire instance. When current core is not solved by b branching iterations, this means that it is too large, and must be reduced. Finally, if we find a satisfying solution for the core, 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 the core. The above is a complete solution scheme.

Parameters (d, b, c) greatly affect the result. They can be set in order to quickly solve the SAT problem, as in [3], or with the aim of selecting small subformulae. This latter 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. 0.01, if we expect MUS much smaller than the original $\mathcal{F}$, or larger otherwise. Such procedure can guarantee to always find an unsatisfiable subformula (for unsatisfiable $\mathcal{F}$). Moreover, it 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), we believe it is able to locate the core on a minMUS.

5 Computational Results

In the tables, we report number of variables (n) and number of clauses (m) of the original instance and of the smaller unsatisfiable subformula selected. Column 'MUS' report if selected unsatisfiable subformula is minimal (Y) or not (N). This could be tested. We are unable to determine if it is a minMUS or not, because we do not have an exact procedure to test this. A simple test of all possible subformulae would be hopeless. Column 'rest' report if the formula obtained by removing the selected unsatisfiable subformula is satisfiable (S) or not (U). Parameter p (failure penalty in hardness evaluation) was set at 10. Such choice gave better and more uniform results. 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 reported unsatisfiable subformulae. The size of selected subformula is very sensible to parameters' values, expecially c and b. In order to give an idea, the cardinality of selected subformula (ranging form 60 to 850) varying c (from 1 to 10) and b (from 10 to 20) is also given, in the case of problem *jnh2* (see Fig. 2). Times are in CPU seconds (on a Pentium II 450 MHz).

5.1 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*. Records not respecting all the rules are declared *erroneous*. 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 [4] 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 1 we report results on some instances encoding the set of rules called *census1* (developed for a real census). They produced a main SAT instance and a sequence of derived instances (census_1.x), some of which resulted unsatisfiable. Such instances are large but structurally easy. Since inconsistencies are unwanted, they generally contained only one MUS of very small size. Inconsistencies on purpose introduced could always be detected.

Original formula			Selected $\mathcal{U}$				Parameters			
Problem	n	m	n	m	rest	MUS	d	b	c	time
census_1.0	1960	10420	2	3	S	Y	1000	4	0.01	0.1
census_1.1	1958	10415	2	3	S	Y	1000	4	0.01	0.1
census_1.2	1957	10418	2	4	S	Y	1000	4	0.01	0.1
census_1.3	1953	10410	3	4	S	Y	1000	4	0.01	0.0
census_1.4	1958	10412	2	3	S	Y	1000	4	0.01	0.1
census_1.5	1948	10400	2	3	S	Y	1000	4	0.01	0.1
census_1.6	1956	10416	2	4	S	Y	1000	4	0.01	0.0
census_1.7	1952	10411	2	3	S	Y	1000	4	0.01	0.0
census_1.8	1950	10420	3	5	S	N	1000	4	0.01	0.0
census_1.9	1955	10413	2	4	S	Y	1000	4	0.01	0.0

Table 1: Unsatisfiable subformula selection on instances encoding rules for data collecting problems.

5.2 Dimacs Problems

We choose some unsatisfiable problems from the Dimacs[1] test set, since they are widely-known and easily available[2]. The series *aim* are 3-SAT instances artificially generated by K. Iwama, E. Miyano and Y. Asahiro, the series *jnh* are generated by J.N. Hooker. They are nowadays easily solved by many SAT solvers, being small in size. Nevertheless, they have a structure much more difficult than usual real problems. This results in the presence of unsatisfiable subformulae much larger than in sec. 5.1.

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

Table 2: Unsatisfiable subformula selection on the *aim*-50 series: 3-SAT artificially generated problems.

Original formula			Core				Parameters			
Problem	n	m	n	m	rest	MUS	d	b	c	time
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

Table 3: Unsatisfiable subformula selection on the *aim*-100 series: 3-SAT artificially generated problems.

[1] NFS Science and Technology Center in Discrete Mathematics and Theoretical Computer Science - A consortium of Rutgers University, Princeton University, AT&T Bell Labs, Bellcore.
[2] Available from
`ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/benchmarks/cnf/`

Original formula			Core				Parameters			
Problem	n	m	n	m	rest	MUS	d	b	c	time
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 4: Unsatisfiable subformula selection on the *aim*-200 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 5: Unsatisfiable subformula selection on the *jnh* series: randomly generated hard problems.

6 Conclusions

In several applicative fields, in addition to solving the SAT problem, one need to locate a minMUS, or at least a MUS or an 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 MUS. We are unable to determine whether it is a minMUS or not, because, contrary to the case of MUS, we do not have a procedure to test this. (Testing every possible subset, even for small instances, would be hopeless.) However, we suspect a minMUS was selected in many of the analyzed real problems and in some Dimacs problems.

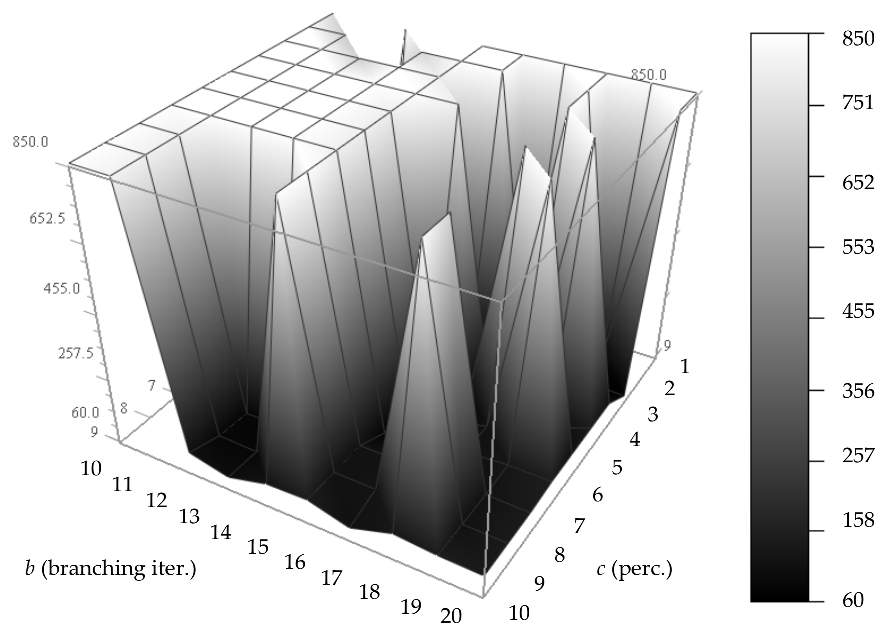

Fig. 2. Cardinality of the unsatisfiable subformula selected in *jnh2* for different values of b and c.

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, 45–59, 1999.

[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*, Kluwer Academic Publishers, 1:77-148, 1998.

[3] R. Bruni and A. Sassano. A Complete Adaptive Algorithm for Propositional Satisfiability. *Technical Report 19-00*, DIS, University of Rome “La Sapienza”, 2000.

[4] R. Bruni and A. Sassano. Optimization Techniques for an Error Free Data Collecting. *Technical Report 01-01*, DIS, University of Rome “La Sapienza”, 2001.

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

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

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

[8] I.P. Gent and T. Walsh, editors. *SAT 2000*, Journal of Automated Reasoning, Volume 24, Issue 1/2, 2000.

[9] 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.

[10] J.N. Hooker and V. Vinay. Branching Rules for Satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[11] 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.

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

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

[14] K. Truemper. *Effective Logic Computation.* Wiley, New York, 1998