Finding Minimal Unsatis¯able Subformulae in Satis¯ability instances

Renato Bruni and Antonio Sassano

Dipartimento di Informatica e Sistemistica, Universitµa di Roma \La Sapienza", via Buonarroti 12 - I-00185 Roma, Italy. fbruni, sassanog@dis.uniroma1.it

Abstract. A minimal unsatis¯able subformula (MUS) of a given CNF is a set of clauses which is unsatis¯able, but becomes satis¯able as soon as we remove any of its clauses. In practical scenarios it is often useful to know, in addition to the unsolvability of an instance, which parts of the instance cause the unsolvability. An approach is here proposed to the problem of automatic detection of such a subformula, with the double aim of ¯nding quickly a small-sized one. We make use of an adaptive technique in order to rapidly select an unsatis¯able subformula which is a good approximation of a MUS. Hard unsatis¯able instances can be reduced to remarkably smaller problems, and hence e±ciently solved, through this approach.

1 Introduction

We call minimal unsatis¯able subformula (MUS) of a logic CNF formula a set of clauses which is unsatis¯able, but becomes satis¯able removing any of its clauses (see sec. 2). An approach is here proposed to the problem of automatic detection of a MUS, with the double aim of ¯nding it small, and of proving unsatis¯ability faster. As for the ¯rst point, in fact, in practical scenarios it is often useful to know, in addition to the unsolvability of an instance, which parts of the instance cause the unsolvability. As for the second point, the task of proving unsatis¯ability usually turns out to be computationally harder than proving satis¯ability.

We can, roughly speaking, classify the solution methods for the Satis¯ability problem in exact and heuristic ones. The proposed procedure is a complete one. Most of complete methods are branching procedures based on case splitting. On the contrary, we use a new branching scheme, whose branching rule consists in trying to satisfy at ¯rst the hardest clauses while visiting a clause-based branching tree [1] (see sec. 3). Moreover, we use an Adaptive Core Search in order to rapidly select an unsatis¯able subformula, which is a good approximation of a MUS (see sec. 4). Small but hard unsatis¯able instances can be e±ciently solved through this approach. More important, unsatis¯able subformulae are detected in all problems solved. Their size is often remarkably smaller than the original formulae.

2 Minimal Unsatis¯able Subformulae

Let A be the ground set of the literals a_i. De¯ne $\neg a_l = a_{n+l}$ and $\neg a_{n+l} = a_l$.

$A \equiv \{a_i : a_i = \alpha_i \text{ for } i = 1, \dots, n;\ a_i = \neg \alpha_{i-n} \text{ for } i = n+1, \dots, 2n\}$

Every clause is a set $C_j = \{a_i : i \in I_j \subseteq I \equiv \{1, \dots, 2n\}\}$. A CNF formula is collection F of sets C_j over A: $F = \{C_j : j = 1, \dots, m\}$.

A truth assignment for the logical variable is a set $S = \{a_i : a_i \in S \Rightarrow \neg a_i \notin S\}$. It is partial if $|S| < n$, i.e. some variables are not assigned, and complete if $|S| = n$. Given a partial truth assignment S, the set of possible completion is $C(S) = \{a_i : a_i \notin S \wedge \neg a_i \notin S\}$. S satis¯es C if and only if $S \cap C \neq \emptyset$. S falsi¯es C if and only if $C(S) \cap C = \emptyset$.

A minimal unsatis¯able subformula (MUS) is a collection $G \subseteq F$ of clauses of the original instance having the following properties:

1. $\forall S,\ \exists C_j \in G$ such that $S \cap C_j = \emptyset$ (unsatis¯able)
2. $\forall H \subset G,\ \exists S$ such that $\forall C_j \in H,\ S \cap C_j \neq \emptyset$ (every subset is sat.)

Of course, if any subformula is unsatis¯able, the whole problem is. On the other hand, an unsatis¯able formula always contains a MUS.

There are procedures that, given a set of clauses, recognize whether it is a MUS or not in polynomial time [3]. The key point is how to select a MUS. We propose a procedure to rapidly select a good approximation of a MUS, that means an unsatis¯able set of clauses having almost as few clauses as the smallest MUS.

3 The Branching scheme

In order to reduce backtracks, it's better to start assignment satisfying the more di±cult clauses [1], i.e. those which have the fewest satisfying truth assignments, or, in other words, represent the more constraining relations. The point is how to ¯nd hardest clauses. An a priori parameter is the length: unit clauses are universally recognized to be hard, and the procedure of unit propagation, universally performed, satis¯es them at ¯rst.

Our evaluation of clause hardness is based on the history of the search, and keeps improving throughout the computation. We say that a clause C_j is visited when we make a truth assignment aimed at satisfying C_j, and that C_j cause a failure either when an empty clause is generated due to truth assignment made on C_j, or when C_j itself becomes empty. Visiting C_j many times shows that C_j is di±cult, and failing on it shows even more clearly that C_j is di±cult. Counting visits and failures requires a very little overhead. Therefore, we use the following branching rule:

1. Perform all unit resolutions.
2. When no unit clauses are present, make a truth assignment satisfying the clause C_{max} which maximizes our clause hardness measure φ (v_j is the number of visits, f_j the number of failures, p a parameter giving the penalty considered for failures, and l_j the length of the clause).

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

To satisfy C_{max}, we add to the partial solution S a literal $a_l \in C(S)$ such that $a_l \in C_{max}$. If we need to backtrack, the next assignment is not just $\neg a_l$, because it does not satisfy C_{max}. Instead, we add another literal $a_k \in C_{max} \cap C(S)$ [1]. If $C_{max} \cap C(S)$ becomes empty, we obviously backtrack

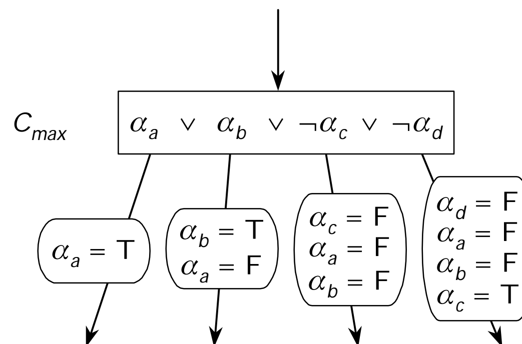

Fig. 1. Example of C_{max} with the consistent branching possibilities.

again to the truth assignments made to satisfy the previous clause, until we have another choice. This is a complete scheme, because, if a satisfying truth assignment exists, it will be reached, and, if the search tree is completely explored, the instance is unsatisfiable.

4 Adaptive Core Search

By applying the above branching scheme, and φ to evaluate clauses hardness, we develop the following procedure of Adaptive Core Search.

0. (Preprocessing) Perform p branching iterations using just shortest clause rule (This gives initial values to φ). If instance is already solved, Stop.
1. (Base) Select an initial collection of hardest clauses whose cardinality is proportional to cardinality of F:
   $C_1 = \{C_j : C_j \in F;\ \varphi(C_j) \geq \varphi(C_k)\ \forall C_k \in F;\ |C| = c|F|;\ c < 1\}$
   This is our first core, i.e. candidate to be a MUS. Remaining clauses form another collection $O_1 = F \setminus C_1$.

k. (Iteration) Perform h branching iteration on C_k, ignoring O_k. Obtain one of the following:

– C_k is unsatisfiable $\Rightarrow$ F is unsatisfiable, then Stop.
– no answer after h iteration $\Rightarrow$ Re-start from (Base) selecting a new set of hardest clauses (allowed only a finite number t of times, in order to ensure termination: after t times, h is increased).
– C_k is satisfied by solution S_k $\Rightarrow$ Test S_k on O_k. One of the following:
  - $\forall C_j \in O_k,\; S_k \cap C_j \neq \emptyset \Rightarrow F$ is satisfied, then Stop.
  - $\exists C_j \in O_k : C(S_k) \cap C_j = \emptyset$. Call this collection of falsified clauses N_k. Add them to the core, obtaining $C_{k+1} = C_k \cup N_k$, delete the partial solution S_k, and apply again (Iteration).
  - $\exists C_j \in O_k : S_k \cap C_j = \emptyset$ and $\nexists C_j \in O_k : C(S_k) \cap C_j = \emptyset$ Try to extend the partial solution S_k to F putting $C_{k+1} = F$ and apply again (Iteration).

The main idea is that we select the clauses that resulted hard during the branching phase, and try to solve them as if they were our entire instance. If they really are an unsatisfiable instance, we have done. If, after h branching iterations we cannot solve them, this means that our instance is still too big, and it must be reduced more. 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, they are difficult (together with the clauses of the core), and therefore they should be added to the core.

The above algorithm is a complete one, and solves, in average case, smaller subproblems at the nodes of the search tree, hence operations performed, such like unit propagation, work only on current core C_k.

5 Computational results

Columns labeled 'branch', 'ACS sel', 'ACS sol', 'ACS tot' respectively are the running times for the branching procedure only (without Adaptive core Search), core selection, core solving, and total times of Adaptive Core Search. n_{core} and m_{core} are the number of variables and clauses appearing in the unsatisfiable subformula selected.

The test problems (all unsatisfiable) are from the DIMACS. Four of them were used in the test set of the Second DIMACS Implementation Challenge [2]. Our running times on them are compared with those of the four faster complete algorithms of that challenge (C-sat , 2cl , TabuS and BRR). Times are normalized according to the DIMACS benchmark dfmax, in order to compare them in a machine-independent way.

6 Conclusions

An approach is here proposed to the problem of automatic detection of a MUS, with the double aim of ¯nding a small-sized one, and of proving unsatis¯ability faster. In particular, we present a clause based tree search paradigm for Satis¯ability testing, a new heuristic to identify hard clauses, that are the clauses most likely to appear in a MUS, and the complete algorithm for approximating a Minimal Unsatis¯able Subformula.

Smaller unsatis¯able subformulae are detected in all the solved unsatis¯able instances. These can be remarkably smaller then the original formula, and give an approximation of a MUS in extremely short times.

Problem	n	m	sol	C sat	2cl	Tabu S	BRR	br.	ACS sel	ACS sol	ACS tot	n_{core}	m_{core}
aim-100-1_6-no-1	100	160	U					1.09	0.17	0.03	0.20	43	48
aim-100-1_6-no-2	100	160	U					0.67	0.54	0.39	0.93	46	54
aim-100-1_6-no-3	100	160	U					3.91	0.62	0.73	1.35	51	57
aim-100-1_6-no-4	100	160	U					0.52	0.61	0.35	0.96	43	48
aim-100-2_0-no-1	100	200	U	52.19	19.77	409.50	5.78	0.03	0.03	0.01	0.04	18	19
aim-100-2_0-no-2	100	200	U	14.63	11.00	258.58	0.57	0.38	0.05	0.04	0.09	37	40
aim-100-2_0-no-3	100	200	U	56.21	6.53	201.15	2.95	0.12	0.04	0.01	0.05	25	27
aim-100-2_0-no-4	100	200	U	0.05	11.66	392.23	4.80	0.11	0.04	0.01	0.05	26	32
Average (on last 4)				30.82	12.50	320.15	3.52	0.16	0.04	0.02	0.06		

Problem	n	m	literals	sol	branch	ACS sel	ACS sol	ACS tot	n_{core}	m_{core}
aim-200-1_6-no-1	200	320	959	U	5.02	0.12	0.09	0.21	52	55
aim-200-1_6-no-2	200	320	959	U	>600	14.04	32.31	46.35	76	87
aim-200-1_6-no-3	200	320	958	U	>600	10.80	35.87	46.67	73	86
aim-200-1_6-no-4	200	320	960	U	5.81	0.09	0.10	0.19	45	48
aim-200-2_0-no-1	200	400	1199	U	15.53	0.20	0.27	0.47	49	55
aim-200-2_0-no-2	200	400	1199	U	3.87	0.17	0.18	0.35	46	50
aim-200-2_0-no-3	200	400	1198	U	1.04	0.05	0.12	0.17	35	37
aim-200-2_0-no-4	200	400	1197	U	0.70	0.16	0.02	0.18	36	42

Table 1. Results of C-SAT, 2cl(limited resolution), DPL with Tabu Search, B-reduction and ACS on the aim series: 3-SAT arti¯cially generated problems. Times are normalized according to dfmax results, as if they were obtained on our same machine.

References

1. J.N. Hooker and V. Vinay. Branching Rules for Satis¯ability. Journal of Automated Reasoning, 15:359{383, 1995.
2. D.S. Johnson and M.A. Trick, editors. Cliques, Coloring, and Satis¯ability, volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996.
3. O. Kullmann. An application of matroid theory to the SAT Problem. ECCC TR00-018, Feb. 2000.