Solving Propositional Satisfiability by Identification of Hard Subformulae

R. Bruni and A. Sassano
University of Roma “La Sapienza”

ISMP 2000
Atlanta, GA, August 2000

Summary

1. Introduction and notation

2. Hard clauses and Minimally unsatisfiable subformulae (MUS)

3. Branching tree structure

4. The algorithm : Adaptive core search (ACS)

5. Computational results and conclusions

Introduction

Every propositional logic formula can be expressed in

Conjunctive Normal Form

$$(\alpha_{i_1} \vee \ldots \vee \alpha_{j_1} \vee \neg\alpha_{j_1+1} \vee \ldots \vee \neg\alpha_{n_1}) \wedge \ldots \wedge (\alpha_{i_m} \vee \ldots \vee \alpha_{j_m} \vee \neg\alpha_{j_m+1} \vee \ldots \vee \neg\alpha_{n_m})$$

Satisfiability Problem (NP-complete)

Is there a truth assignment for the logic variables

(and if yes which is)

such as the whole formula is satisfied (= is True) ?

Notation

Ground set of the literals (posited or negated proposition)

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

Define $\neg a_i = a_{i+n}$, and $\neg a_{i+n} = a_i$

Clause
(set of literals)

$$C_j = \left\{ a_i : i \in I_j \subseteq I \equiv \{1, \ldots, 2n\} \right\}$$

Instance
(collection of sets of literals)

$$\mathcal{F} = \left\{ C_j : j = 1, \ldots, m \right\}$$

Truth Assignment

Truth assignment (set of literals) $S = \{ a_i : a_i \in S \Rightarrow \neg a_i \notin S \}$

Partial if $|S| < n$, complete if $|S| = n$

Completion (set of literals) $C(S) = \{ a_i : a_i \notin S \wedge \neg a_i \notin S \}$

S satisfies $\mathcal{F}$ $\forall C_j \in \mathcal{F}, \quad S \cap C_j \neq \phi$

$\mathcal{F}$ is unsatisfiable $\forall S, \exists C_j \in \mathcal{F} : S \cap C_j = \phi$

Approaches

- Complete methods

Given enough time, are guaranteed to find the solution.

Based on branching (DLL) and/or resolution (DP)

- Heuristics

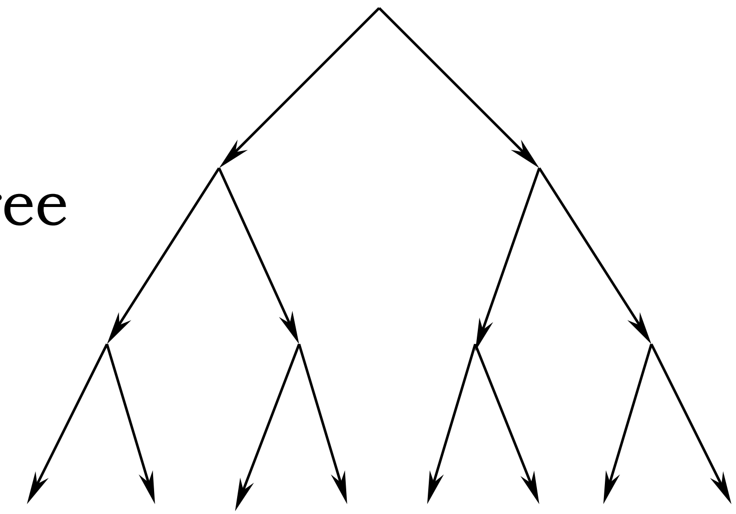
Are faster, but not guaranteed to find the solution.

Not very useful for unsatisfiable instances.

Generic branching procedure

Time depends on :

- number of nodes of the search tree
- time needed at every single node



At every node,

1. Choose variable to branch on (branching rule)

   (needs time)

2. Fix the variable *(fast)*

3. Simplify the formula *(needs time)*

   (unit propagation: unit resolution and unit subsumption)

Possible solutions

- Reduce size of the branching tree,

  e.g. by improving effectiveness of the branching rule

  or by cutting some subtrees.

- Reduce time needed for the selection of the branching

  variable by simplifying its calculation.

- Reduce time needed for unit propagation by delaying

  some operations.

HARD CLAUSES

Given an instance $\mathcal{F}$, some clauses are more *difficult* to satisfy, that is are more constraining in the context of *that particular instance*

Example of short clauses containing the same variables (hard)

$$C_1 = \{ a_1 \ a_2 \} \quad C_2 = \{ a_1 \ \neg a_2 \} \quad C_3 = \{ \neg a_1 \ a_2 \}$$

Example of long clauses containing different variables (easy)

$$C_1 = \{ a_1 \ a_2 \ \neg a_3 \} \quad C_2 = \{ a_4 \ \neg a_5 \ a_6 \} \quad C_3 = \{ \neg a_7 \ a_8 \ a_9 \}$$

Individuation of hard clauses

A priori :

observations made before, length l_j, ...

In itinere :

(solving the problem with a branching procedure)

# of visits v_j of a clause, # of failure f_j due to a clause

We evaluate clause hardness using

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

Calculation of φ requires extremely small overhead and keeps improving throughout the computation.

Branching rule part 1: clause selection

- If we front hard clauses deep in the branching tree (current partial assignment S almost complete, C(S) small) usually we need to backtrack far.
- If we front hard clauses at the beginning of the branching tree (S small, C(S) wide) we solve them, or we discover unsatisfiability earlier.

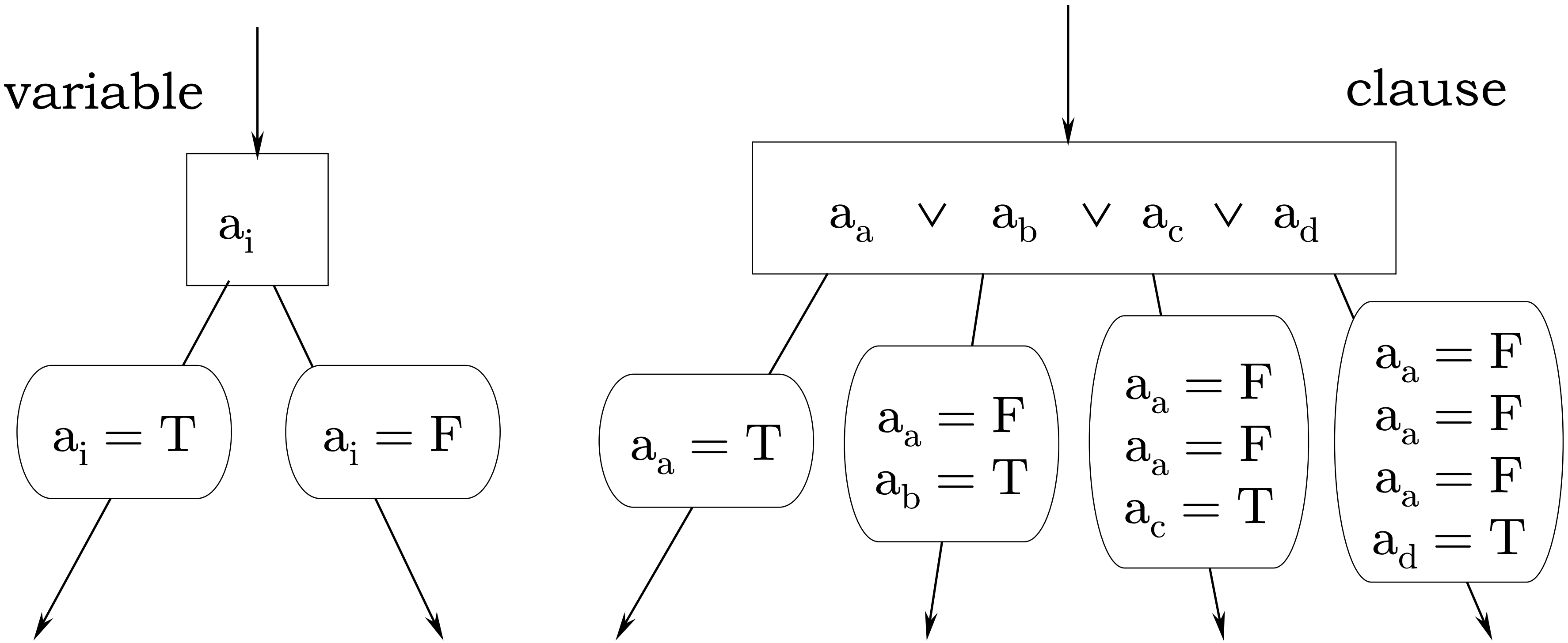
Hard clauses must be fronted first:

We choose $C_j = \text{argmax}\ \varphi(C_j) = (v_j + pf_j)/\ l_j$

Branching scheme

Like most of complete methods, we use a branching procedure.

No binary branching, but clause-based branching.



Branching rule part 2: variable selection

- Within the clause, we need to choose the variable :

Two sided Jeroslow Wang (Hooker)

$$\text{Choose } a_i = \text{argmax} \sum_{C_j \subset a_i} 2^{-|Cj|} + \sum_{C_j \subset \neg a_i} 2^{-|Cj|}$$

approximated as follows:

let $J(a_i) = 1 + \#$ binary clauses containing a_i

choose $a_i = \text{argmax } J(a_i)\, J(\neg a_i)$

Minimally Unsatisfiable Subformulae

Given an unsatisfiable instance $\mathcal{F}$, we can have collections of clauses $\mathcal{G} \subset \mathcal{F}$ which are still unsatisfiable

$\mathcal{G} \subseteq \mathcal{F}$ is **minimally unsatisfiable** (MUS) iff

➡ $\forall$ S, $\exists$ $C_j \in \mathcal{G}$: $S \cap C_j = \phi$ (is unsatisfiable)

➡ $\forall \mathcal{H} \subset \mathcal{G}$, $\exists$ S, $\forall$ $C_j \in \mathcal{H}$: $S \cap C_j \neq \phi$ (every subset is satisfiable)

$\mathcal{F}$ is unsatisfiable $\Leftrightarrow$ $\mathcal{F}$ contains a MUS

MUS approximation

By collecting enough hard clauses (using φ) from an unsatisfiable instance, we can identify an unsatisfiable set of clauses.

If we stop collecting as soon as the set is unsatisfiable, we cannot say that it's minimal.

We have a quick approximation of a minimally unsatisfiable subformula.

The algorithm - Definitions

Given an instance $\mathcal{F}$ and values for φ, define:

$$\varphi(\mathcal{F}) = \mathcal{C} = \{C_j : C_j \in \mathcal{F},\ \varphi(C_j) \geq \varphi(C_k)\ \forall\ C_k \in \mathcal{F},\ |\mathcal{C}| < |\mathcal{F}|\}$$

(selection of the hardest clauses in the clause-set)

and $\mathcal{O} = \mathcal{F} \setminus \mathcal{C}$

Given a partial solution S_k and a set of clauses $\mathcal{O}_k$, define:

$\mathcal{N}_k = \{ C_j \in \mathcal{O}_k : C(S_k) \cap C_j = \phi \}$ (falsified clauses)

$\mathcal{S}_k = \{ C_j \in \mathcal{O}_k : S_k \cap C_j \neq \phi \}$ (satisfied clauses)

Adaptive core search - I

Preprocessing: perform p branching iterations on $\mathcal{F}$ to give initial values to φ

Base step: select an initial collection of the hardest clauses $\mathcal{C}_1 = \varphi(\mathcal{F})$. $\mathcal{C}_1$ is the first *core* , i.e. candidate to be a MUS. Remaining clauses form $\mathcal{O}_1$

Adaptive core search - II

Iteration : apply *h* branching iterations to $\mathcal{C}_k$ ignoring $\mathcal{O}_k$

One of the following:

- $\mathcal{C}_k$ is unsatisfiable $\Rightarrow$ $\mathcal{F}$ is unsatisfiable, then STOP.

- No result after *h* iteration $\Rightarrow$ contraction (allowed only *t* times to ensure termination): put $\mathcal{C}_{k+1} = \varphi(\mathcal{C}_k)$, k = k+1, goto **Iteration**.

- $\mathcal{C}_k$ is satisfied by S_k $\Rightarrow$ If $\mathcal{C}_k = \mathcal{F}$, $\mathcal{F}$ is satisfiable, STOP. Otherwise test S_k on $\mathcal{O}_k$. One of the following: [next]

Adaptive core search - III

Test S_k on $\mathcal{O}_k$. One of the following :

- $\mathcal{S}_k = \mathcal{O}_k$, $\mathcal{N}_k = \phi \Rightarrow \mathcal{F}$ is satisfied, then STOP
- $\mathcal{N}_k \neq \phi \Rightarrow$ expansion: add $\mathcal{N}_k$ to the core, obtain $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \mathcal{N}_k$ (note $\mathcal{C}_{k+1} \subseteq \mathcal{F}$), k = k+1, goto **Iteration**.
- $\mathcal{N}_k = \phi$, $\mathcal{S}_k \subset \mathcal{O}_k \Rightarrow$ extension: keep S_k , put $\mathcal{C}_{k+1} = \mathcal{C}_k \cup \varphi(\mathcal{O}_k)$, k = k+1, goto **Iteration**.

Features

- Complete method: In the worst case performs a complete branching.
- Factors of size reduction for the branching tree:

  1) To front hard clauses first.

  2) To prove unsatisfiability exploring only the core subtree.

- Factors of time reduction for branching variable selection:

  1) Easy to compute branching rule.

  2) To choose only within the current core.

- Reduces time needed for unit propagation by delaying it for clauses out of the core.

Usefulness of MUS approximation

Many sat instances encode real world problems.

If a formula is unsatisfiable, it is useful to know which part of the formula cause this unsolvability.

That is the part respectively to remove or to keep when we respectively want such formula to be satisfiable or unsatisfiable.

Series PAR16 (sat)

Problem	*n*	*m*	ACS 1.0	SATO 3.2
par16-1	1015	3310	10.10	24.16
par16-1-c	317	1264	11.36	2.62
par16-2	1015	3374	52.36	49.22
par16-2-c	349	1392	100.73	128.15
par16-3	1015	3344	103.92	40.81
par16-3-c	334	1332	8.19	78.91
par16-4	1015	3324	70.82	1.51
par16-4-c	324	1292	5.10	133.07
par16-5	1015	3358	224.84	4.92
par16-5-c	341	1360	72.29	196.33

Series AIM-100 unsat

Problem	n	m	C-sat	2cl	TabuS	BRR	ACS sel	ACS sol	ACS tot	*n* core	*m* core
aim-100-1_6-no-1	100	160	n.a.	n.a.	n.a.	n.a.	0.17	0.03	0.20	43	48
aim-100-1_6-no-2	100	160	n.a.	n.a.	n.a.	n.a.	0.54	0.39	0.93	46	54
aim-100-1_6-no-3	100	160	n.a.	n.a.	n.a.	n.a.	0.62	0.73	1.35	51	57
aim-100-1_6-no-4	100	160	n.a.	n.a.	n.a.	n.a.	0.61	0.35	0.96	43	48
aim-100-2_0-no-1	100	200	52.19	19.77	409.50	5.78	0.03	0.01	0.04	18	19
aim-100-2_0-no-2	100	200	14.63	11.00	258.58	0.57	0.05	0.04	0.09	37	40
aim-100-2_0-no-3	100	200	56.21	6.53	201.15	2.95	0.04	0.01	0.05	25	27
aim-100-2_0-no-4	100	200	0.05	11.66	392.15	4.80	0.04	0.01	0.05	26	32

Series II32 (sat)

Problem	*n*	*m*	ACS 1.0
ii32a1	459	9212	0.02
ii32b1	228	1374	0.00
ii32b2	261	2558	0.03
ii32b3	348	5734	0.03
ii32b4	381	6918	1.53
ii32c1	225	1280	0.00
ii32c2	249	2182	0.00
ii32c3	279	3272	2.84
ii32c4	759	20862	5.07
ii32d1	332	2730	0.01
ii32d2	404	5153	0.76
ii32d3	824	19478	7.49
ii32e1	222	1186	0.00
ii32e2	267	2746	0.01
ii32e3	330	5020	0.08
ii32e4	387	7106	0.02
ii32e5	522	11636	1.03

Conclusions

- Techniques to perform a fast complete enumeration are widely proposed in literature (e.g. sophisticated data structure, ...).
- Here a technique is presented to reduce the set that enumeration works on.
- In practical scenarios it is useful to know which part of the instance cause the unsolvability.

Results are extremely encouraging. We believe better performances can be obtained by using both of the above techniques.