Propositional satisfiability (SAT) and SAT-based decision procedures

Marco Maratea

j.w.w. Enrico Giunchiglia

Dipartimento di Informatica, Sistemistica e Telematica
DIST, University of Genova, STAR-Lab

Motivation

1. Propositional satisfiability (SAT) is one of the most studied fields in AI and CS

2. Very efficient and specialized SAT procedures exist

$\Rightarrow$ use SAT solvers for deciding more expressive logics and formalisms ...

$\Rightarrow$ ...reusing most of the work and knowledge available in SAT

SAT: The problem

A *literal* l is a proposition p or its negation $\neg p$.

Given the literals $l_1, \ldots, l_k$, a *clause* is $l_1 \vee \cdots \vee l_k$.

Given the clauses $c_1, \ldots, c_m$, a CNF (Conjunctive Normal Form) formula is $c_1 \wedge \cdots \wedge c_m$.

An *assignment*, or *valuation* v, is a partial function from the propositions to $\{$TRUE,FALSE$\}$.

We can extend the definition of v in the natural way to assign truth values to literals, clauses and formulas.

Given a CNF formula φ, we define the *propositional satisfiability problem (SAT)*:

Does there exist an assignment v to the propositions in φ such that φ is true?

SAT: Examples

1. $\varphi := \{p, p \vee \neg q, \neg r\}$ has the satisfying assignments
   - $\{p :=$ TRUE, $q :=$ TRUE, $r :=$ FALSE$\}$
   - $\{p :=$ TRUE, $q :=$ FALSE, $r :=$ FALSE$\}$
2. $\varphi := \{\neg p, p \vee \neg q, r \vee \neg p, q\}$ has no satisfying assignments because the clause $\{p \vee \neg q\}$ can not be satisfied.

SAT: Solving methods

- Resolution algorithm
- Local search algorithms
- (Ordered) Binary Decision Diagrams (OBDDs)
- Stalmark's method
- Davis-Logemann-Loveland (DLL) algorithm

Agenda

- DLL
- DLL-based (SAT-based) decision procedures
- Application I: Answer Set Programming (ASP)
- Application II: Separation Logic (SL)
- Application III: SAT-related optimization problems (Max-SAT, Min-ONE)
- Conclusions

The DLL algorithm

function DLL(φ, S)

if $\varphi = \emptyset$ **then return** TRUE ;

if $\emptyset \in \varphi$ **then return** FALSE ;

if $\{l\} \in \varphi$ **then return** DLL$(assign(l, \varphi), S \cup \{l\})$;

$A :=$ an atom occurring in φ;

return DLL$(assign(A, \varphi), S \cup \{A\})$ **or**

DLL$(assign(\neg A, \varphi), S \cup \{\neg A\})$.

DLL-based decision procedures

Given a formula t in the theory T that can be abstracted/compiled into SAT

function BEYSAT(t) **return** DLL$(\textsc{cnf}(\textsc{t2sat}(\mathrm{t})), \emptyset)$;

function DLL(φ, S)

if $\varphi = \emptyset$ **then return** *test*(S, t);

if $\emptyset \in \varphi$ **then return** FALSE ;

if $\{l\} \in \varphi$ **then return** DLL$(assign(l, \varphi), S \cup \{l\})$;

$A :=$ an atom occurring in φ;

return DLL$(assign(A, \varphi), S \cup \{A\})$ **or**

DLL$(assign(\neg A, \varphi), S \cup \{\neg A\})$.

test(S, t) returns TRUE if S is a “solution” for the formula t, and FALSE, otherwise.

From DLL to BEY-SAT: Discussion

1. BEYSAT(t) returns TRUE iff t has a "solution"

2. BEYSAT(t) can be (easily) modified in order to compute all the solutions of t

3.. Most SOTA SAT solvers are a (non-recursive) implementation of DLL

4. If SOTA SAT solvers are based on "learning" in order to backjump irrelevant nodes while backtracking and avoid the exploration of useless parts of the search tree, it is important that *test*(S, t) does not return only FALSE, but also a "witness" of inconsistency (called *reason*)

5. BEYSAT(t) works in polynomial-space if both T2SAT and *test*(S, t) does

Application I: Answer Set Programming (ASP)

Answer Set (stable model) Programming is a programming paradigm proposed by Marek, Truszczynski and Niemela in 1999.

It is a form of declarative programming. It is based on logic rules and on the answer set semantic of Prolog proposed by Gelfond and Lifschitz in 1988.

In answer set programming we obtain the answers by declaring the properties of the answers, by the mean of logic rules.

ASP has been used in several fields like planning, commonsense reasoning, (bounded) model checking, VLSI (wire routing), security protocols.

ASP: Logic rules for 3-colorability problem

```
% Simple graph containing 3 nodes and 3 edges:
% edges between nodes 1 and 2, 2 and 3, 3 and 1.
node(1). node(2). node(3).
edge(1,2). edge(2,3). edge(3,1).
% Declaration of three colors
col(red). col(green). col(blue).
% A node has some color
color(X,red) v color(X,green) v color(X,blue) :- node(X).
% Neighboring nodes should not have the same color
:- edge(X,Y), color(X,C), color(Y,C), col(C).
```

Basic preliminaries

A *(logic) program* Π is a finite set of rules of the form

$$A_0 \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n \tag{1}$$

Let P be the set of atoms in Π, $A_0 \in P \cup \{\bot\}$, $\{A_1, \ldots, A_n\} \subseteq P$. A_0 is the *head*. not is the negation as failure operator.

Comp(Π) consists of formulas of the type

$$A_0 \equiv \bigvee (A_1 \wedge \cdots \wedge A_m \wedge \neg A_{m+1} \wedge \cdots \wedge \neg A_n)$$

for each symbol in $P \cup \{\bot\}$. In the equation, the disjunction extends over all rules (1) in Π with head A_0.

What is an answer set?

Consider first Π in which $m = n$. Let X be a set of atoms.

We say that X is *closed* under Π if for every rule in Π, $A_0 \in X$ whenever $\{A_1, \ldots, A_m\} \subseteq X$.

We say that X is an answer set for Π if X is the smallest set closed under Π.

Consider now the general case $n > m$.

The reduct Π^X of Π related to X is the set of rules

$$A_0 \leftarrow A_1, \ldots, A_m$$

such that $X \cap \{A_{m+1}, \ldots, A_n\} = \emptyset$.

We say that X is an answer set for Π if X is an answer set for Π^X.

Given a (logic) program Π, find if it has a solution is an NP-complete problem.

ASP: Examples

1. Be Π_1:
$$p \leftarrow \ not\ q$$
$$q \leftarrow \ not\ r$$

The only AS is $\{q\}$

2. Be Π_2:
$$p \leftarrow \ not\ q$$
$$q \leftarrow \ not\ p$$

The ASs are $\{p\}$ and $\{q\}$

3. Be Π_3:
$$p \leftarrow \ not\ p$$

Π_3 does not have AS.

Cmodels2 decision procedure

function CMODELS2(Π) **return** DLL(CNF($\underline{Comp(\Pi)}$), $\emptyset$);

function DLL(φ, S)

if $\varphi = \emptyset$ **then return** $\underline{test(S, \Pi)}$;

if $\emptyset \in \varphi$ **then return** FALSE ;

if $\{l\} \in \varphi$ **then return** DLL($assign(l, \varphi), S \cup \{l\}$);

$A :=$ an atom occurring in φ;

return DLL($assign(A, \varphi), S \cup \{A\}$) **or**

DLL($assign(\neg A, \varphi), S \cup \{\neg A\}$).

test(S, Π) returns TRUE if $S \cap P$ is an answer set of Π, and FALSE, otherwise.

From DLL to Cmodels2: Discussion

1. CMODELS2(Π) returns TRUE iff Π has an answer set
2. CMODELS2(Π) can be easily modified in order to compute all the answer sets of a program Π
3. *test*(S, Π) can fail because of "loops" in Π. The reason is extracted from the "loop formulas"

   Ex. Π: $p \leftarrow p$, *Comp*(Π) is $p \equiv p$
4. CMODELS2 works in polynomial-space

Experimental results: Blocks world

		Standard programs			Extended programs	
#b	#s	SMODELS	ASSAT	CMODELS2	SMODELS	CMODELS2
8	i-1	12.32	0.80	1.19	0.81	0.47
11	i-1	71.78	2.97	4.19	2.97	1.01
8	i	40.87	0.89	2.18	1.56	1.40
11	i	71.42	3.17	4.52	3.41	1.16
8	i+1	23.35	0.96	0.97	4.99	0.31
11	i+1	107.48	3.54	3.33	5.21	0.75

Table 1: Blocks world: “#b” is the number of blocks.

Experimental results: H.C. complete graphs

	Standard programs				Extended programs	
	SMODELS	ASSAT	DLV	CMODELS2	SMODELS	CMODELS2
np30c	11.70	1.14	22.08	0.69	0.36	0.36
np40c	62.89	41.81	97.96	1.63	2.48	0.87
np50c	219.56	14.51	314.46	3.37	8.39	1.79
np60c	594.46	48.80	770.07	5.81	20.47	3.41
np70c	1323.61	291.60	1679.12	8.22	39.41	5.87
np80c	2354.28	32.51	3407.35	14.20	75.36	9.18
np90c	TIME	779.06	TIME	22.23	122.53	14.19
np100c	TIME	—	TIME	28.63	185.52	20.76
np120c	TIME	—	TIME	53.33	418.15	41.84

Table 2: Complete graphs. npXc corresponds to a graph with “X” nodes.

Experimental results: Formal Verification problems

	SMODELS	ASSAT	DLV	CMODELS2
mutex4	33.92	0.62	840.60	0.68
phi4	0.24	2.98	1.44	TIME
mutex2	0.09	1.78		0.12
mutex3	229.57	MEM		24.16
phi3	2.87	236.91		3.91

Table 3: Checking requirements in a deterministic automaton.

Experimental results: BMC problems

BMC	SMODELS	CMODELS2	CMODELS2'
dp-10.i-02-b11	382.72	1476.72	442.14
dp-10.s-02-b8	15.24	8.20	14.22
dp-12.s-O2-b9	336.03	65.41	137.34
dp-8.i-O2-b9	8.08	12.62	10.69
dp-8.s-O2-b7	1.19	1.02	2.28
dp-10.i-O2-b12	445.47	3295.72	163.29
dp-10.s-O2-b9	28.87	16.07	15.03
dp-12.s-O2-b10	971.50	209.29	48.73
dp-8.i-O2-b10	5.05	40.01	6.44
dp-8.s-O2-b8	1.76	1.99	2.03

Table 4: Bounded Model Checking Problems.

Application II: Separation Logic (SL)

Decision procedures able to decide quantifier-free first-order theories are becoming increasingly important in Artificial Intelligence and Formal Verification.

Several properties of hardware, timed automata, and software can be modeled in quantifier-free first-order theories as well as planning and scheduling problems.

Separation Logic (SL) is one of such decidable quantifier-free first-order theories that allows boolean combination of difference constraints ($x - y \leq c$).

Why Separation Logic?

SL is a good compromise between efficiency and expressivity.

It combines propositional atoms with a restricted form of linear arithmetic via the standard boolean connectives.

Many available benchmarks are in SL and a lot of properties of systems and planning/scheduling constraints can be encoded in this logic.

Examples:

"Activity A1 lasts for 10 units of time at most": $e_1 - s_1 \leq 10$

"Activity A1 should start before activity A2 finishes": $s_1 \leq e_2$

"Activity A1 should start before activity A2 finishes, otherwise A3 should start when A2 finishes": $s_1 \leq e_2 \vee s_3 = e_2$

SL: Definitions

Fix a domain of interpretation D for the arithmetic variables (the set of real or the set of integer numbers).

An SL-atom is either a propositional variable or a difference constraint $x - y \leq c$ ($<, >, \geq, =, \neq$ can be (easily) recast in $\leq$), where x and y range on D and c is a numeric constant.

An SL-literal is an SL-atom or its negation.
An SL-clause is a finite disjunction of SL-literals.
An SL-formula is a finite conjunction of SL-clauses.

Deciding an SL-formula (Is there an SL-assignment to propositional atoms and arithmetic variables, such that the SL-formula ψ is true?) is an NP-complete problem.

TSAT++ **decision procedure**

Given a formula ψ in SL,

function TSAT++(ψ) **return** DLL(CNF($\underline{Abstract(\psi)}$), $\emptyset$);

function DLL(φ, S)

if $\varphi = \emptyset$ **then return** $\underline{\textit{test}(S, \psi)}$;

if $\emptyset \in \varphi$ **then return** FALSE ;

if $\{l\} \in \varphi$ **then return** DLL$(assign(l, \varphi), S \cup \{l\})$;

$A :=$ an atom occurring in φ;

return DLL$(assign(A, \varphi), S \cup \{A\})$ **or**

DLL$(assign(\neg A, \varphi), S \cup \{\neg A\})$.

test(S, ψ) returns TRUE if the set of constraints in S is consistent, FALSE otherwise.

From DLL to TSAT++: Discussion

1. TSAT++(ψ) returns TRUE iff ψ has a solution
2. *test*(S, ψ) can fail because of sets of inconsistent difference constraints in ψ. The reason is extracted using the Bellman-Ford algorithm considering the difference constraints involved in a negative cycle.
3. TSAT++ works in polynomial-space

Disjunctive Temporal Problem (DTPs)

These are well-known random problems from the AI community.

DTPs are randomly generated by fixing the number k of expressions $x - y \leq c$ per SL-clause, the number n of arithmetic variables, a positive integer L such that all the constants are taken in $[-L, L]$. Then:

1. the number of clauses m is increased in order to range from satisfiable to unsatisfiable instances from 2*n to 14*n step n,
2. for each tuple of values of the parameters, 100 instances are generated and then given to the solvers, and
3. the median of the CPU time is plotted against the m/n ratio.

TSAT++'s performance (1): DTPs

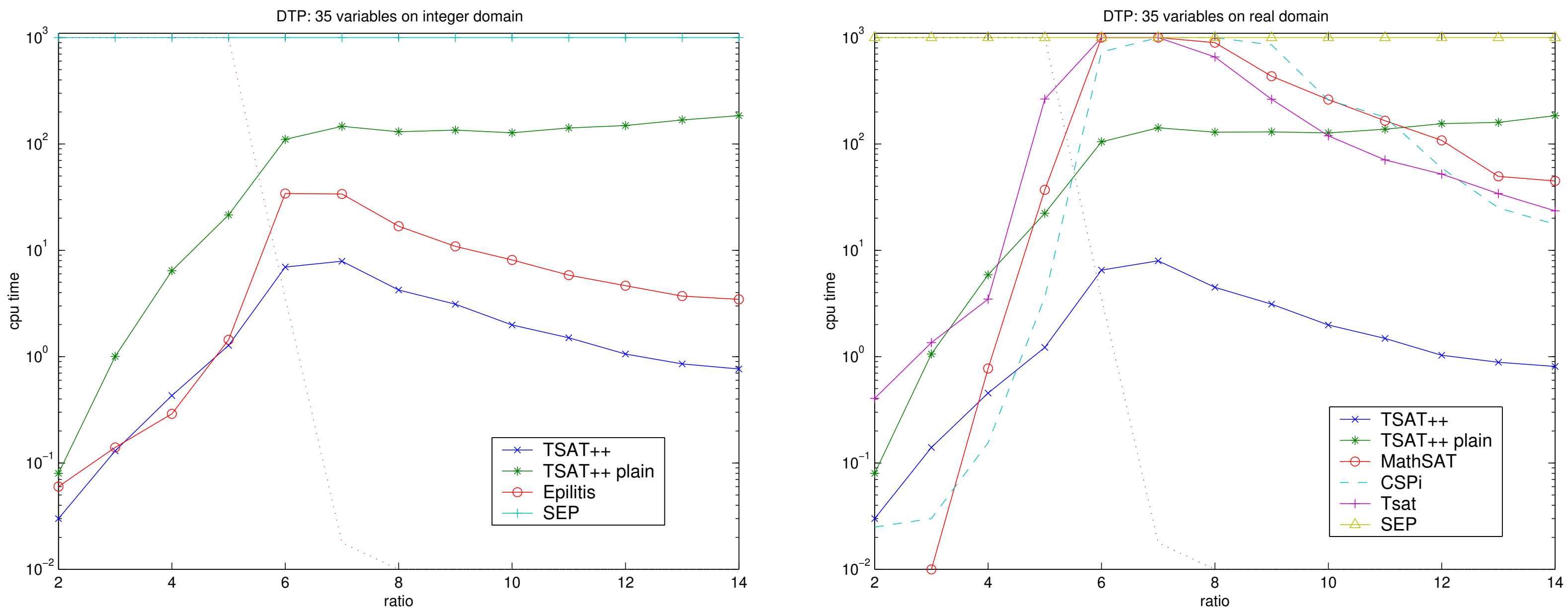

Figure 1: Evaluation on the DTP on 35 variables. Integer domain (left) and real domain (right). Setting: $k = 2, L = 100$.

TSAT++'s performance (2): hand-made problems

K	T	unique	TSAT++	TSAT++p	MathSAT	SEP
100	5	NO	0.01	0.11	0.61	1.18
100	5	YES	0.04	7.57	TIME	0.17
250	5	NO	0.08	0.76	5.40	52.20
250	5	YES	0.21	194.99	TIME	0.77
500	5	NO	0.29	4.46	21.22	742.99
500	5	YES	1.05	TIME	TIME	4.85
1000	5	NO	1.07	22.3	–	TIME
1000	5	YES	6.45	TIME	–	22.53
2000	5	NO	3.76	94.23	–	–
2000	5	YES	29.90	TIME	–	–

TSAT++'s performance (3): real-world problems

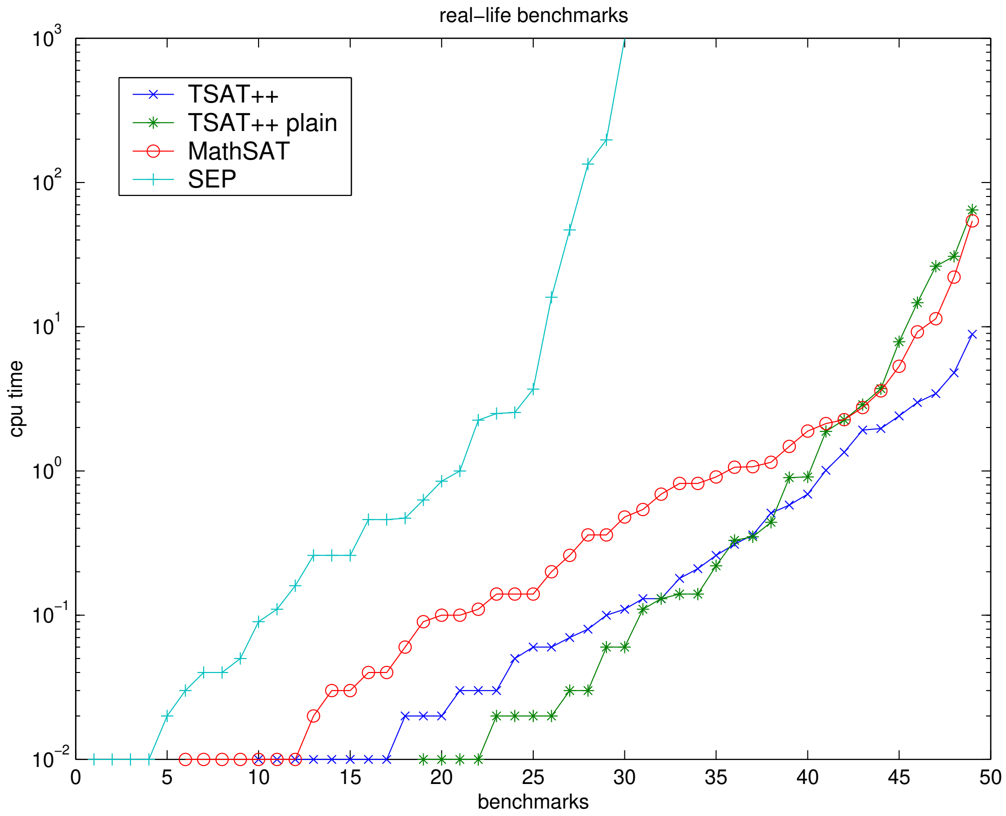

Figure 2: Real-world problems.

Application III: SAT-related optimization problems

There is some interesting research related to SAT, namely

1. Max-SAT: Given an unsatisfiable SAT instance, how many clauses can be satisfied at most (at the same time)?
2. Max(Min)-One: Given a satisfiable SAT instance, find the satisfying assignment with the maximum (minimum) number of variables assigned to TRUE

They have applications in planning, routing problems, (unbounded) model checking, correcting the minimum amount of inconsistent knowledge.

Till now, methods for solving these problems have been focused on branch-and-bound algorithms.

Application III: The new approach

Let focus for the moment on the Max-SAT and Max-One problems.

Both problems can be expressed via an optimization function $max \sum_i x_i$ where the x_i are (a subset of) the variables in the SAT instance.

A cardinality constraint $\sum_i x_i$ can be encoded (via half adders, specialized encoding) into a propositional formula $\text{ENC}(x_i, s_i, b_i)$ where the s_i are some added variables and $b_0, \ldots, b_b$ represent in some way the cardinality constraint (depending on the encoding used).

Given a CNF formula ϕ the idea is to build the extended SAT formula $\phi \wedge \text{CNF}(\text{ENC}(x_i, s_i, b_i))$ and than guiding the search with the b_i bits from the MSB to the LSB.

The OPTSAT decision procedure

function OPTSAT(ϕ) **return** DLL$(\underline{\phi \wedge \text{CNF}(\text{ENC}(x_i, s_i, b_i))}, \emptyset)$;

function DLL(φ, S)

if $\varphi = \emptyset$ **then return** $\underline{ComputeSolution(S, x_i, b_i)}$;

if $\emptyset \in \varphi$ **then return** FALSE ;

if $\{l\} \in \varphi$ **then return** DLL$(assign(l, \varphi), S \cup \{l\})$;

$A :=$ an atom occurring in φ, <u>preferentially</u> and <u>in order</u> on b_i;

return DLL$(assign(A, \varphi), S \cup \{A\})$ **or**

DLL$(assign(\neg A, \varphi), S \cup \{\neg A\})$.

$ComputeSolution(S, x_i, b_i)$ computes the solution given the encoding and the values of b_i; it can be used for finding all the solutions with the same “rank”, otherwise return TRUE.

From DLL to OPTSAT: Discussion

1. OPTSAT(ϕ) returns the "optimal" solution of ϕ as first solution
2. OPTSAT(ϕ) can be (easily) modified in order to compute "all the optimal solutions" of ϕ
3. OPTSAT works in polynomial-space

Analysis on randomly generated 2SAT benchmarks: Max-SAT

#var ub	#cl #ist	bf	#CNF b optz b	#CNF bm optz bm	#CNF w optz w
1500	1800		3279104	3280903	66567
0	24	3.94	5.2	5.24	26.76
1	33	7.96	5.55	5.49	27.93
2	22	8.29	5.90	5.80	28.31
3	14	29.27	7.50	6.78	29.90
4	6	131.44	42.82	13.95	46.16
5	1	–	569.10	107.74	145.85
TIME		1	0	0	0
2000	2400		5814208	5816607	93281
0	13	9.89	9.88	9.73	50.71
1	30	14.12	10.55	10.20	51.11
2	29	15.05	11.34	10.80	52.87
3	13	76.88	17.00	12.65	55.45
4	10	180.59	143.18	34.30	142.33
5	1	–	351.25	135.53	273.13
TIME		12	4	2	2

Analysis on randomly generated benchmarks: Min-ONE

	#var	#cl	#CNF b	#CNF bm	#CNF w	PBS	opbdp	optz b	optz bm	optz w
2	200	50	42938	43137	7019	0.19	0.76	**0.17**	**0.17**	0.96
S	200	100	42988	43187	7069	47.12	16.91	4.73	**3.76**	TIME
A	200	150	43038	43237	7119	16.87	14.54	**7.63**	10.04	148.30
T	200	200	43088	43287	7169	26.46	45.95	**5.84**	6.37	TIME
3	120	120	16064	16183	4463	14.61	4.50	0.55	**0.46**	11.39
S	120	240	16184	16303	4583	TIME	30.52	**22.66**	90.62	TIME
A	120	360	16304	16423	4703	TIME	**3.81**	85.7	79.6	TIME
T	120	480	16424	16543	4823	0.64	**0.08**	5.73	5.15	5.84

Table 5: Randomly generated benchmarks. Min-ONE 2SAT e 3SAT problems.

Analysis on real-world benchmarks: Max-SAT

ist	#var	#cl	#CNF b / w	bf	optz b	optz w
b4	578	2035	$> 4\ 10^6$ / 82124	0.65	6.92	40.41
b5	1407	5383	– / 202480	21.42	MEM	260.97
l2	1164	3525	– / 129998	57.11	MEM	117.47
l7	3319	10335	– / 393514	–	MEM	1061.61
l8	3810	11877	– / 435148	–	MEM	1729.87
q8	512	2273	$> 5\ 10^6$ / 89852	0.62	9.80	50.36
q10	886	5622	– / 208933	2.21	MEM	409.09
q14	1370	9313	– / 365920	–	MEM	1145.27
c880	957	2590	$> 6\ 10^6$ / 98411	TIME	12.65	56.86
c1355	1294	3662	– / 136707	TIME	MEM	138.80
c1355-s	1294	3670	– / 137147	TIME	MEM	142.95
c1908	1917	5096	– / 194731	TIME	MEM	237.09
c1908-s	1919	5108	– / 195055	TIME	MEM	229.22
c1908.b	957	2590	– / 194839	1705.69	MEM	232.01

Analysis on real-world benchmarks: Min-ONE

instance	#var	#CNF b w	#res	PBS	opbdp	optz b	optz bm	optz w
2bitcomp-5	125	17554 / 2700	39	8.02	**0.95**	1.78	1.82	10.69
2bitmax-6	252	68042 / 5739	61	TIME	120.5	60.59	**48.64**	183.08
3blocks	283	94132 / 15230	56	**0.14**	296.92	6.75	5.09	**0.14**
4blocks	758	199624 / 31955	66	**0.25**	TIME	4.14	8.72	7.46
4blocksb	410	419805 / 159562	66	**52.57**	TIME	TIME	TIME	TIME
sat-bw-large.b	1087	1216246 / 35808	131	**0.18**	1.78	15.16	11.31	0.22
sat-bw-large.c	3016	– / 103938	136	34.97	TIME	MEM	MEM	**3.56**
sat-logistics.a	828	707642 / 21399	135	MEM	TIME	14.66	14.06	**11.52**
sat-logistics.b	843	733605 / 22297	138	TIME	TIME	364.34	**200.21**	TIME
cnf-r1-b1-k1.1	307	100789 / 7535	36	7.6	86.13	1.14	1.22	**0.88**
cnf-r2-b3-k1.1	1495	2274580 / 36060	141	**0.05**	67.14	13.41	12.85	0.25
cnf-r3-b1-k1.1	1461	2174994 / 35116	119	**6.88**	TIME	289.75	200.91	12.97
qg1-8	512	419805 / 159562	49	TIME	TIME	**64.03**	105.59	89.24
qg2-8	512	419805 / 159562	64	TIME	TIME	41.75	38.97	**31.02**

Conclusions

- the SAT-based approach is very competitive and often superior w.r.t. rival approaches in a number of application areas
- it can leverage on the work done on SAT in the last years, and our algorithms can (easily) take advantages on future enhancement in the field (SAT competitions every year!)
- here we have focused on NP-complete problem, but the approach can be/has been applied to “harder” problems such as disjunctive logic programming and conformant planning

References (I)

A. Armando, C. Castellini, E. Giunchiglia and M. Maratea - The SAT-based Approach to Separation Logic Accepted to the Journal of Automated Reasoning (JAR). 2005.

E. Giunchiglia and M. Maratea - Evaluating Search Strategies and Heuristics for Efficient Answer Set Programming. Accepted to the 9th Congress of the Italian Association for Artificial Intelligence (AI*IA 2005). To appear LNAI.

Yu. Lierler and M. Maratea - Cmodels2: SAT-based Answer Set Solvers Extended to Non-tight Programs. In Proc. 7th International Conference on Logic Programming and Non Monotonic Reasoning (LPNMR 2004). LCNS series.

E. Giunchiglia, Yu. Lierler and M. Maratea - SAT-based Answer Set Programming. In Proc. 19th American Association for Artificial Intelligence (AAAI 2004). AAAI / MIT Press.

A. Armando, C. Castellini, E. Giunchiglia and M. Maratea - A SAT-based Decision Procedure for the Boolean Combination of Difference Constraints. Accepted to 7th International Conference on Theory and Practice of Satisfiability Testing (SAT 2004). To appear LNCS.

References (II)

A. Armando, C. Castellini, E. Giunchiglia, M. Idini and M. Maratea - TSAT++: An Open Reasoning Platform for Satisfiability Modulo Theory. Accepted to 2nd Workshop on Pragmatic of Decision Procedures in Automated Reasoning (PDPAR 2004). To appear ENTCS.

E. Giunchiglia, M. Maratea and A. Tacchella - (In)Effectiveness of Look-ahead Techniques in a Modern SAT solver. In Proc. 9th International Conference on Principle and Practice of Constraint Programming (CP 2003). LNCS series.

E. Giunchiglia, M. Maratea and A. Tacchella - Dependent and Independent Variables in Propositional Satisfiability. In Proc. 8th European Conference on Logics in Artificial Intelligence (JELIA 2002). LNAI series.

E. Giunchiglia, M. Maratea, A. Tacchella and D.Zambonin - Evaluating Search Heuristics and Optimization Techniques in Propositional Satisfiability. In Proc. 1st International Joint Conference on Automated Reasoning (IJCAR 2001). LNCS.