SAT Solving: Theory and Practice of efficient combinatorial search

Oliver Kullmann

Computer Science Department
Swansea University
http://cs.swan.ac.uk/~csoliver/

Southwest Jiaotong University, April 18, 2013

Mathematics Department

Solving boolean equations I

Consider the boolean equation

$$(a \vee \neg b) \wedge (\neg a \vee c) \wedge (b \vee \neg c) = 1. \tag{1}$$

Recall:

- $a, b, c \in \{0, 1\}$
- $x \wedge y = \min(x, y)$
- $x \vee y = \max(x, y)$
- $\neg x = 1 - x$.

Do you see the solutions?

Solving boolean equations II

$$(a \vee \neg b) \wedge (\neg a \vee c) \wedge (b \vee \neg c) = 1$$

has exactly two solutions:

$$a = b = c = 0, \quad a = b = c = 1.$$

We also have an explanation for them, using $x \to y := \neg x \vee y$:

$$(a \vee \neg b) \wedge (\neg a \vee c) \wedge (b \vee \neg c) = 1 \iff$$
$$b \to a = 1, \quad a \to c = 1, \quad c \to b = 1 \iff$$
$$a = b = c.$$

Restriction to $F = 1$ I

We considered an equation of the form $F = 1$, where F is a boolean term.

What about general boolean equations $F = G$?

Using $x \leftrightarrow y := (x \to y) \wedge (y \to x)$ we get

$$F = G \iff F \leftrightarrow G = 1.$$

So at least if we have all seven operations $\vee, \wedge, \neg, 0, 1, \to, \leftrightarrow$, then we can restrict ourselves to equations

$$F = 1$$

for boolean terms F.

Restriction to $F = 1$ II

Note however that in

$$x \leftrightarrow y := (x \rightarrow y) \wedge (y \rightarrow x)$$

the variable x occurs two times, and so without $\leftrightarrow$ in general we need an exponentially longer term.
This can be avoided by using auxiliary variables.

Satisfiability

We said we are "solving" equations $F = 1$. What does this mean?

- Normally we would like to have a good description of *all* solutions.
- Since such questions have been considered, from 1850 until today, not so much progress has been achieved on this question (in general).
- It is asking too much.

It is much more fruitful to ask about
the *existence* of a solution!

We call a boolean formula F

- **satisfiable**, if $F = 1$ has a solution.
- Otherwise F is called **unsatisfiable** ("contradictory").

SAT solvers

We have now our first formulation of what a “SAT solver” is:

A “SAT solver” is an algorithm
which takes as input a propositional formula F,
and returns “SAT” or “UNSAT”.

We will see important refinements on this formulations:

1. The input is restricted to “conjunctive normal form” (as for our example).
2. We might require certificates on the validity of the output.
3. For practice, there is a huge difference between an (abstract) “algorithm” and a (concrete) “implementation”.

Remarks on certificates

If a SAT solver returns “SAT” for input F, then usually we want to see the **satisfying assignment**, a setting for the variables in F making $F = 1$ true.

- These satisfying assignments are the (standard) certificates for the output “SAT”.
- They are **short**!
- Certificates for output “UNSAT” are also needed.
- They seem to be very long, and for harder examples are currently not feasible in practice.

Remarks on tautology I

Since antiquity, **tautologies** have attracted a lot of attention.

- These are boolean formulas F which are always true (i.e., 1).
- For example

  $$(a \wedge (a \rightarrow b)) \rightarrow b$$

  is a tautology (the “modus ponens”).
- Our example $(a \vee \neg b) \wedge (\neg a \vee c) \wedge (b \vee \neg c)$ is obviously not a tautology: it can be falsified by setting, e.g., $a = 0$ and $b = 1$.

Having a SAT solver, we can check for being a tautology (or not):

F is a tautology if and only if $\neg F$ is unsatisfiable.

Remarks on tautology II

Computing $\neg F$ for $F := (a \wedge (a \rightarrow b)) \rightarrow b$:

1. First we eliminate "$\rightarrow$" (recall $x \rightarrow y = \neg x \vee y$):

$$F = (a \wedge (a \rightarrow b)) \rightarrow b = \neg(a \wedge (\neg a \vee b)) \vee b.$$

2. Adding the negation:

$$\neg F = \neg(\neg(a \wedge (\neg a \vee b)) \vee b).$$

3. Recall $\neg(x \vee y) = \neg x \wedge \neg y$ and $\neg(x \wedge y) = \neg x \vee \neg y$ (the de Morgan rules).
4. We get

$$\neg F = \neg(\neg a \vee (a \wedge \neg b)) \vee b) = (a \wedge (\neg a \vee b)) \wedge \neg b.$$

5. Since $\wedge$ is associate, we get

$$\neg F = a \wedge (\neg a \vee b) \wedge \neg b,$$

which is clearly unsatisfiable.

It's just a finite problem ...

What the fuss about a trivial problem like satisfiability – we can just try out all of the finitely many possible solutions!

- If we have n variables in F, then there are 2^n assignments of $0, 1$ to the variables.
- So for input F we have a trivial algorithm solving SAT, which takes

$$O(\ell(F) \cdot 2^{n(F)})$$

  many steps ($\ell(F)$ is the length of the input).
- This algorithm also enumerates all possible solutions.

All problems solved?

We can (and must) do better

A practical SAT problem has typically at least $n = 1000$:

$$2^{1000} = 1.0715 \cdot \ldots \cdot 10^{301}.$$

And many problems of industrial relevance have $n \approx 10^6$ or even $n \approx 10^9$.

And we **can** solve (many) of these problems.

SAT solving

SAT solving has evolved in an astounding fashion:

- With the development of practical SAT solving in the last decade, it got a huge influence on verification.
- Especially the design of modern microchips wouldn't be possible without these algorithms.
- Some speak of the "SAT revolution". (Though it's hidden — it's technical, mathematical, not "glamorous".)

See the SAT handbook Biere, Heule, van Maaren, and Walsh [1] for basic information.

- Theoretically we are much behind.
- However I believe a beautiful (mathematical) theory on SAT is waiting to be discovered.
- That theory should also enable (still) much better SAT solving.

Outline

5 phases of SAT

1. Development of **propositional logic**, starting with the Greeks, especially the Stoics (around 300 BCE).
2. **Boolean equations**: starting with George Boole's (1815-1864) book [2] from 1854.
3. **Circuits**: starting with Claude Shannon's master thesis from 1937.
4. **NP-completeness**: starting with Stephen Cook's proof from 1971 that SAT is NP-complete [3].
5. **Turning NP-completeness from its head to its feet**: SAT can be used efficiently, and many other problems can be efficiently reduced to it. This latest phase started around 2000.

NP-completeness

Boolean equations have the following properties:

- It might not be straightforward to find a solution, but to check, whether an alleged solution is really one, can be done efficiently.
- And if there is a solution at all, then there exists one which is (at most) of similar size as the size of the equation itself.

This means that the (algorithmic) decision problem of deciding whether a boolean equation is solvable (satisfiable) or not is **in NP**.

The **NP-completeness** of this decision problem means, that every other problem in NP is efficiently reducible to it.

P versus NP

The NP-completeness of Boolean Equations (BE) means:

If we have a good solver for it, then,
with some overhead, we have a good solver
for every problem in NP.

- How well we can solve BE is representative for NP!
- The famous “P versus NP” problem is thus precisely the question, whether BE (or SAT) can be solved in polynomial time.
- It is considered as one of the most important open problems in mathematics, and definitely it is the main open problems in computer science.

See 7 Millennium Problems.

Remarks on ideology

In the 80s and 90s emerged the ideology of

- "efficient" or "feasible" means "poly-time".
- Worst, not-poly-time means "infeasible".
- Just a term was defined, "infeasible", which could also have been called "Bierseidel" (according to Hilbert). But this term was not used formally, but with "deep meaning".
- The disastrous consequence of that was, that the 80's and 90's were dominated by the rejection of SAT solving as "infeasible", and instead probabilistic or approximative approaches were dominant (which didn't add much to SAT solving).
- And mathematicians learned to abhor "combinatorial explosions". It's there, but you are supposed not to look at it ...

SAT is the theory of combinatorial explosion "at the edge".

Some other applications

Besides EDA ("Electronic Design Automation") here are some other applications:

- Verifying railway safety procedures and hardware.
- Computing numbers from Ramsey theory (especially van der Waerden numbers).
- Breaking cryptographic ciphers (cryptanalysis).
- Solving various (mathematical and non-mathematical) puzzles (latin squares etc.).

The Conjunctive Normal Form (CNF)

We restrict our attention to the standard boolean basis $\vee$, $\wedge$, $\neg$.
Any formula F over this basis can be transformed into an equivalent formula which is a conjunctions of disjunctions of literals, where literals are variables and their negations, using:

1. double negation is the identity: $\neg\neg F = F$;
2. de Morgan rules to move the negations inwards;
3. the distributive law
   $(a \wedge b) \vee (c \wedge d) = (a \vee c) \wedge (a \vee d) \wedge (b \vee c) \wedge (b \vee d)$.

To present the basics, I won't rely on these mechanisms from logic:

- I use an algebraic (not "logical") construction of the basic objects.
- CNFs are treated as **clause-sets**, which are considered as (precise) combinatorial objects, as generalised hypergraphs.

Variables and literals I

- We assume an infinite set $\mathcal{VA}$ of **variables**.
- We define $\mathcal{LIT} := \mathcal{VA} \times \{0,1\}$.
- The elements of $\mathcal{LIT}$ are called **literals**.
- We identify the literals $(v,0)$ with the variables v.
- **var**$((v,\varepsilon))$ $:= v$ and **sgn**$((v,\varepsilon))$ $:= \varepsilon$.
- We have a fixed-point free involution on $\mathcal{LIT}$:

$$(v,\varepsilon) \in \mathcal{LIT} \mapsto \overline{(v,\varepsilon)} := (v, 1-\varepsilon) \in \mathcal{LIT}.$$

- Literal $\overline{x}$ is the **complement** of literal x.
- The defining properties of complementation are

$$\overline{x} \neq x, \quad \overline{\overline{x}} = x.$$

Variables and literals II

For implementation purposes typically the following choice is appropriate, using $\mathbb{N} = \{1, 2, \ldots\}$:

1. $\mathcal{VA} := \mathbb{N}$
2. $\mathcal{LIT} := \mathbb{Z} \setminus \{0\}$.
3. $x \in \mathcal{LIT} \mapsto \overline{x} := -x \in \mathcal{LIT}$.

Nothing is lost if you just use this model.
Remarks:

- In general, the structure given by variables and literals is just a free $\mathbb{Z}_2$-set $\mathcal{LIT}$, freely generated by $\mathcal{VA}$.
- For systematic purposes it can also be useful to allow arbitrary $\mathbb{Z}_2$-sets, i.e., to allow that the complementation has fixed points. Such degenerations can be useful to round off certain constructions.

Clause-sets I

- For $L \subseteq \mathcal{LIT}$ let $\overline{L} := \{\overline{x} : x \in L\}$.
- A **clause** is a finite and clash-free set of literals, the set of clauses is denoted by

$$\mathcal{CL} := \{C \subset \mathcal{LIT} \mid C \text{ finite } \wedge\, C \cap \overline{C} = \emptyset\}.$$

- The simplest clause is $\bot := \emptyset \in \mathcal{CL}$, the **empty clause**.
- A **clause-set** is a set of clauses, the set of all clause-sets is denoted by $\mathcal{CLS} := \mathbb{P}(\mathcal{CL})$.
- The simplest clause-set is $\top := \emptyset \in \mathcal{CLS}$, the **empty clause-set**.

From now on only finite clause-sets are considered, since we are concentrating on algorithmic issues.

Clause-sets II

For example

$$\{\{a, b\}, \{\overline{a}, c, \overline{d}\}, \bot\}$$

is a clause-set. When writing such examples, it is typically understood that, e.g., a, b, c, d are pairwise different variables.

If considering literals as integers, then:

- Clauses are just finite sets $C \subset \mathbb{Z} \setminus \{0\}$ of non-zero integers, such that for $x \in C$ we have $-x \notin C$.
- Clause-sets are just finite sets of such clauses.

We ask for clauses to be clash-free, since clauses containing clashes would be tautologies. For some constructions however it can be beneficial to allow such degenerations.

Clause-sets III

To spell out the interpretation:

$$F = \underbrace{\{\{a,b\},\{\overline{b},c\},\{\overline{a},\overline{c}\}\}}_{\textbf{clause-set}} \quad \sim \quad \underbrace{(a \vee b) \wedge (\neg b \vee c) \wedge (\neg a \vee \neg c)}_{\textbf{CNF}}.$$

The interpretation of a clause-set as a CNF is the default interpretation, and guides the definitions.

Variables and literals in clause-sets

For a clause $C \in \mathcal{CL}$: $\mathrm{var}(C) := \{\mathrm{var}(x) : x \in C\}$.

For $F \in \mathcal{CLS}$:

$$\begin{aligned} \mathbf{var(F)} &:= \bigcup_{C \in F} \mathrm{var}(C) \\ \mathbf{lit(F)} &:= \mathrm{var}(F) \cup \overline{\mathrm{var}(F)} \end{aligned}$$

i.e.,

- $\mathrm{var}(F)$ is the set of variables of F
- $\mathrm{lit}(F)$ is the set of (possible) literals of F.

Note that the actually occurring literals are given by $\bigcup F$.

Measures

For $F \in \mathcal{CLS}$ the main measures are:

$$\begin{aligned} \boldsymbol{n(F)} &:= |\mathrm{var}(F)| \\ \boldsymbol{c(F)} &:= |F| \\ \boldsymbol{\ell(F)} &:= \sum_{C \in F} |C| \end{aligned}$$

i.e.,

- the number $n(F)$ of variables
- the number $c(F)$ of clauses
- the number $\ell(F)$ of literal occurrences.

Partial assignments

- A “partial assignment” maps some variables to $\{0, 1\}$ (“truth values”).
- The set of all partial assignments is $\mathcal{PASS}$.
- The application of a partial assignment φ to a clause-set F is denoted by $\varphi * F$.

The theory of SAT can be understood
as the theory of the operation
of the monoid $\mathcal{PASS}$ on the set $\mathcal{CLS}$.

$\mathcal{PASS}$

A **partial assignment** is a map

$$\varphi : V \to \{0, 1\}$$

for some finite $V \subset \mathcal{VA}$. We use **var(φ)** $:= V$.

The set $\mathcal{PASS}$ of all partial assignments carries a **monoid structure**:

1. The composition $\circ : \mathcal{PASS} \times \mathcal{PASS} \to \mathcal{PASS}$ is defined for $\varphi, \psi \in \mathcal{PASS}$ as follows:
   1. $\mathrm{var}(\psi \circ \varphi) = \mathrm{var}(\psi) \cup \mathrm{var}(\varphi)$
   2. for $v \in \mathrm{var}(\psi \circ \varphi)$ the result is obtained by using $\varphi(v)$ is possible, and only if this is not possible then $\psi(v)$ is used.
2. $\circ$ is associative (and not commutative).
3. The neutral element is $\langle\rangle := \emptyset \in \mathcal{PASS}$.

Extending partial assignments to literals

For a variable $v \in \mathrm{var}(\varphi)$ we define

$$\varphi(\overline{v}) := \overline{\varphi(v)} := 1 - \varphi(v).$$

Thus we have for literals x with $\mathrm{var}(x) \in \mathrm{var}(\varphi)$:

$$\varphi(\overline{x}) = \overline{\varphi(x)}.$$

$* : \mathcal{PASS} \times \mathcal{CLS} \to \mathcal{CLS}$

The application of a partial assignment $\varphi \in \mathcal{PASS}$ to a clause-set $F \in \mathcal{CLS}$ is denoted by

$$\varphi * F \in \mathcal{CLS},$$

and is defined as

$$\varphi * F := \left\{ \{x \in C : \varphi(x) \neq 0\} \mid C \in F \wedge \forall\, x \in C : \varphi(x) \neq 1 \right\}.$$

That is:

- First all satisfied clauses are removed (i.e., clauses containing at least one literal set to 1).
- From the remaining clauses all falsified literals are removed.

SAT and UNSAT

The two central definitions:

$$\begin{aligned} \mathcal{SAT} &:= \{F \in \mathcal{CLS} \mid \exists\, \varphi \in \mathcal{PASS} : \varphi * F = \top\} \\ \mathcal{USAT} &:= \mathcal{CLS} \setminus \mathcal{SAT}. \end{aligned}$$

The two basic examples:

- $\top \in \mathcal{SAT}$, since $\langle\rangle * \top = \top$.
- $\{\bot\} \in \mathcal{USAT}$, since $\bot$ can not be satisfied.
- More generally, every F with $\bot \in F$ is unsatisfiable.

Examples

Consider $F := \{ \{\overline{a}, b\}, \{\overline{b}, c\}, \{\overline{a}, c\} \}$.

$$\begin{aligned} \langle b \to 1 \rangle * F &= \{ \{c\}, \{\overline{a}, c\} \} \\ \langle b, c \to 1 \rangle * F &= \top. \end{aligned}$$

Thus $F \in \mathcal{SAT}$.

$$\begin{aligned} \langle a \to 1 \rangle * F &= \{ \{b\}, \{\overline{b}, c\}, \{c\} \} \\ \langle a \to 1, c \to 0 \rangle * F &= \{ \{b\}, \{\overline{b}\}, \bot \} \in \mathcal{USAT} \\ \langle a \to 1, c \to 1 \rangle * F &= \{ \{b\} \} \\ \langle a \to 1, c \to 1, b \to 1 \rangle * F &= \top \end{aligned}$$

The fundamental laws

$$(\psi \circ \varphi) * F = \psi * (\varphi * F)$$
$$\varphi * \top = \top$$
$$\langle\rangle * F = F$$
$$\varphi * (F \cup G) = (\varphi * F) \cup (\varphi * G).$$

Additionally:

- $\mathrm{var}(\varphi * F) \subseteq \mathrm{var}(F) \setminus \mathrm{var}(\varphi)$.
- Thus, if $\mathrm{var}(F) \subseteq \mathrm{var}(\varphi)$, then $\varphi * F \in \{\top, \{\bot\}\}$.

The special interpretation

If $\mathcal{LIT} = \mathbb{Z} \setminus \{0\}$:

- We can identify partial assignments φ with finite sets of non-zero integers, not containing x and $-x$ at the same time.
- These are the literals set to true by φ.
- φ is a satisfying assignment for F iff φ has non-empty intersection with every element of F.

Backtracking

The basic method is splitting on a variable: For $F \in \mathcal{CLS}$ and $v \in \mathrm{var}(F)$ holds

$$F \in \mathcal{SAT} \text{ if and only if}$$
$$\langle v \to 0 \rangle * F \in \mathcal{SAT} \quad \text{or} \quad \langle v \to 1 \rangle * F \in \mathcal{SAT}.$$

Note that $v \notin \langle v \to \varepsilon \rangle * F$, and thus this procedure terminates.

Also note for the recursion basis (as already noticed):

$$\mathrm{var}(F) = \emptyset \Leftrightarrow F \in \{\top, \{\bot\}\}.$$

The most basic SAT-algorithm

In C++ pseudo-code:

```
bool A0(CLS F) {
  if ({} in F) return false;
  if (F == {}) return true;
  choose v in var(F); // branching variable
  choose e in {0,1}; // first branch
  if (A0(<v -> e> * F)) return true;
  else return A0(<v -> (1-e)> * F);
}
```

With a reasonable heuristics, already this algorithm can be much faster than 2^n.

Forced assignments

An assignment $\langle x \to 1 \rangle$ for a literal x and $F \in \mathcal{CLS}$ is called **forced**, if $\langle x \to 0 \rangle * F \in \mathcal{USAT}$.

Thus $\langle x \to 1 \rangle * F$ is sat-equivalent to F.

- So we can (and should!) apply the partial assignment $\langle x \to 1 \rangle$.
- The problem in general is, that detection of a forced assignment is coNP-complete.
- So special cases need to be considered.

The easiest case is $\bot \in F$ (why?).

Then comes $\{x\} \in F$.

Unit-clause propagation

A basic mechanism in determining satisfiability is

unit-clause propagation (UCP).

For example:

$$\{\ \underbrace{\{a\}}_{\text{unit-clause}}\ , \{\overline{a}, b\}, \{\overline{b}\}\ \} \xrightarrow{\langle a \to 1\rangle} \{\ \{b\}, \{\overline{b}\}\ \} \xrightarrow{\langle b \to 1\rangle} \{\ \bot\ \}.$$

- Detects and sets some forced assignments, repeatedly.
- Possible in linear time, and is confluent.
- Using the map $\mathbf{r_1} : \mathcal{CLS} \to \mathcal{CLS}$ for UCP we have

$$\mathrm{r}_1(F) := \begin{cases} \{\bot\} & \text{if } \bot \in F \\ \mathrm{r}_1(\langle x \to 1\rangle * F) & \text{if } \exists\, x \in \mathrm{lit}(F) : \bot \in \langle x \to 0\rangle * F \\ F & \text{otherwise} \end{cases}.$$

Generalised unit-clause propagation

Kullmann [7, 8] introduced the notion of

generalised unit-clause propagation

$\mathrm{r}_k : \mathcal{CLS} \to \mathcal{CLS}$, $k \in \mathbb{N}_0$.

$$\mathrm{r}_0(F) := \begin{cases} \{\bot\} & \text{if } \bot \in F \\ F & \text{otherwise} \end{cases}$$

$$\mathrm{r}_1(F) = \begin{cases} \mathrm{r}_1(\langle x \to 1\rangle * F) & \text{if } \exists\, x \in \mathrm{lit}(F) : \mathrm{r}_0(\langle x \to 0\rangle * F) = \{\bot\} \\ F & \text{otherwise} \end{cases}$$

$$\mathrm{r}_k(F) := \begin{cases} \mathrm{r}_k(\langle x \to 1\rangle * F) & \text{if } \exists\, x \in \mathrm{lit}(F) : \mathrm{r}_{k-1}(\langle x \to 0\rangle * F) = \{\bot\} \\ F & \text{otherwise} \end{cases}.$$

$\mathrm{r}_k(F)$ can be computed in time $\ell(F) \cdot n(F)^{2k-2}$.

Example: r_2 is more powerful r_1

Consider

$$F := \{ \{a, b\}, \{a, \overline{b}\}, \{\overline{a}, b\}, \{\overline{a}, \overline{b}\} \}.$$

We have that

1. $r_1(F) = F$ (UCP does nothing).
2. $r_2(F) = r_2(\langle a \to 1 \rangle * F) = \{\bot\}$, since

$$r_1(\langle a \to 0 \rangle * F) = r_1(\{ \{b\}, \{\overline{b}\} \}) = \{\bot\}.$$

Algorithm scheme Ak

For some fixed $k \in \mathbb{N}_0$:

```
bool Ak(CLS F) {
  F = r_k(F);
  if (F == {{}}) return false;
  if (F == {}) return true;
  choose v in var(F); // branching variable
  choose e in {0,1}; // first branch
  if (Ak(<v -> e> * F)) return true;
  else return Ak(<v -> (1-e)> * F);
}
```

- For $k = 0$ we get A0.
- For decent performance at least $k = 1$.
- $k = 2, 3$ (typically not used completely) was restricted to "look-ahead" SAT solvers in the past (see [5, 10]), but $k = 2$ started to become used by "conflict-driven" solvers recently.

Autarkies

Dual to forced assignments (in a sense) are “autarkies”:

- An **autarky** for a clause-set F is a partial assignment φ which satisfies every clause $C \in F$ it touches (i.e., $\mathrm{var}(\varphi) \cap \mathrm{var}(C) \neq \emptyset$).
- Again, for an autarky φ the result $\varphi * F$ of “autarky reduction” is sat-equivalent to F (note that $\varphi * F \subseteq F$).
- Detection of autarkies is NP-complete.
- So, again, schemes for restricted autarkies are needed.

Pure literals and beyond

The simplest case of autarkies are “pure literals”:

If for a literal x we have $\overline{x} \notin \bigcup F$,
then x is called **pure for** F,
and $\langle x \to 1 \rangle$ is an autarky for F.

This is just the beginning, and a nice theory of autarky has emerged; see [6].

Clause learning

A **look-ahead solver** is an extension of scheme Ak, using also autarkies and other reductions.

- They are good for cases which are essentially hard, and so systematic work is required.
- However they are typically less good for examples with a lot of "special structure".
- The "SAT revolution", which took place in the last decade, is based on solvers capable of exploiting such structures.
- Since such structures are typical for practical examples.

This new type of solver is called **conflict-driven** (or "CDCL" – conflict-driven clause-learning); see [11].

The basic idea

A look-ahead solvers has the organisation of the search (the splitting- or backtracking-tree) externally, while a conflict-driven solver *internalises* it:

If at the end of a branch
a partial assignment φ yields $\bot \in \varphi * F$,
then we might "learn" the "cause" of this "conflict".

And if the learning result is a clause, then we can add it to F.

How can this work?

What else can there be in φ —
more than the falsified clause $C \in F$ (i.e., $\varphi * \{C\} = \{\bot\}$)
(which we already have)?

The point is that φ does not only contain "decision variables" (the branch variables), but also forced assignments, typically by r_1.

Via this "conflict analysis",
thus we might replace assignments to variables in C
by some other assignments — we learned something!

Learning a clause

It boils down to the circumstance,

$$\text{that from } \bot \in \varphi * F \text{ we can infer,}$$
$$\text{via conflict-analysis,}$$
$$\varphi' * F \in \mathcal{USAT} \text{ for some } \varphi' \subseteq \varphi.$$

Now we can learn the **negation of** φ'. For example

$$\varphi' = \langle a \to 0, b \to 1 \rangle$$

means "$a = 0 \wedge b = 1$". Negation yields "$a = 1 \vee b = 0$" — a clause!

- So we learn the clause $C_{\varphi'}$.
- $C_{\varphi'}$ contains precisely the literals set to 0 by φ'.
- For the example we have $C_{\varphi'} = \{a, \overline{b}\}$.
- "Learning" means $F \rightsquigarrow F \cup \{C_{\varphi'}\}$.

An example

Consider

$$F := \{ \{a, b, c\}, \{a, \overline{b}, c\}, \{\overline{a}, b, c\}, \{\overline{a}, \overline{b}, c\}, \{a, b, \overline{c}\}, \{a, \overline{b}, \overline{c}\}, \{\overline{a}, b, \overline{c}\}, \{\overline{a}, \overline{b}, \overline{c}\} \}.$$

1. We start with $a \to 0$ (first decision).
2. r_1 yields nothing.
3. Then we assign $b \to 0$ (second decision).
4. Now r_1 yields $c \to 1$.
5. So we have now $\varphi = \langle a \to 0, b \to 0, c \to 1 \rangle$.
6. We have $\bot \in \varphi * F$.
7. Via conflict analysis we get $\varphi' := \langle a \to 0, b \to 0 \rangle$.
8. So we learn $C_{\varphi'} = \{a, b\}$.

Main challenges related to SAT solving

I Develop a true hybrid solver

- The 90's were the time of look-ahead and local-search solvers.
- The last decade then was the time of conflict-driven solvers.
- It is clear, that no scheme dominates the others.
- A "true" combination of these three schemes is needed (not just a "hybrid" as an ad-hoc mixture).

For some first remarks, connecting look-ahead and conflict-driven, see [9].

II Theory of good representations

Currently just ad-hoc methods for representing computational problems as SAT problems are used:

- A theory is needed here.
- The practical applications should be developed together with new SAT solvers (which might be needed to understand better representations).

For some approaches, see [4].

III Theory of heuristics

- Heuristics are dominated by engineering approaches — mathematics is needed!
- Especially the heuristics for conflict-driven solvers are obscure.

See [10] for a foundation for look-ahead solvers.

IV Understanding!

- I believe it is possible to understand the fundamental patterns of unsatisfiability.
- Also a much refined proof theory is needed, taking special problem structures into account.
- SAT is more complicated than UNSAT here; I believe considering SAT as a kind of “limit” of UNSAT is the right approach.

Summary and outlook

I I believe most of the interesting things in SAT are still to come!

II Especially theory is needed.

End

(references on the remaining slides).

For my papers see
http://cs.swan.ac.uk/~csoliver/papers.html.

Bibliography I

[1] Armin Biere, Marijn J.H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, February 2009. ISBN 978-1-58603-929-5.

[2] George Boole. *An Investigation of The Laws of Thought, on which are founded The Mathematical Theorie of Logic and Probabilities*. Dover Publication, Inc., first published in 1958. ISBN 0-486-60028-9; printing of the work originally published by Macmillan in 1854, with all corrections made within the text.

[3] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

Bibliography II

[4] Matthew Gwynne and Oliver Kullmann. Towards a theory of good SAT representations. Technical Report arXiv:1302.4421v3 [cs.AI], arXiv, March 2013.

[5] Marijn J. H. Heule and Hans van Maaren. Look-ahead based SAT solvers. In Biere et al. [1], chapter 5, pages 155–184. ISBN 978-1-58603-929-5.

[6] Hans Kleine Büning and Oliver Kullmann. Minimal unsatisfiability and autarkies. In Biere et al. [1], chapter 11, pages 339–401. ISBN 978-1-58603-929-5. doi: 10.3233/978-1-58603-929-5-339.

[7] Oliver Kullmann. Investigating a general hierarchy of polynomially decidable classes of CNF's based on short tree-like resolution proofs. Technical Report TR99-041, Electronic Colloquium on Computational Complexity (ECCC), October 1999.

Bibliography III

[8] Oliver Kullmann. Upper and lower bounds on the complexity of generalised resolution and generalised constraint satisfaction problems. *Annals of Mathematics and Artificial Intelligence*, 40 (3-4):303–352, March 2004.

[9] Oliver Kullmann. Present and future of practical SAT solving. In Nadia Creignou, Phokion Kolaitis, and Heribert Vollmer, editors, *Complexity of Constraints: An Overview of Current Research Themes*, volume 5250 of *Lecture Notes in Computer Science (LNCS)*, pages 283–319. Springer, 2008. doi: 10.1007/978-3-540-92800-3_11. ISBN-10 3-540-92799-9.

[10] Oliver Kullmann. Fundaments of branching heuristics. In Biere et al. [1], chapter 7, pages 205–244. ISBN 978-1-58603-929-5. doi: 10.3233/978-1-58603-929-5-205.

Bibliography IV

[11] Joao P. Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Biere et al. [1], chapter 4, pages 131–153. ISBN 978-1-58603-929-5.