Logical reasoning and programming

SAT solving—resolution, DPLL, and CDCL

Karel Chvalovský

CIIRC CTU

A reminder of terminology

We are in propositional logic. A *literal* l is a propositional variable p, also called an atom, or a negation of propositional variable $\neg p$. In this context we write $\overline{p}$ instead of $\neg p$. Moreover, to simplify our notation, we define also $\bar{l}$. If a literal l is $\overline{p}$, then $\bar{l}$ is p. A *clause* is any disjunction of finitely many literals. An important special case is the empty clause, we write $\Box$.

A formula φ is in *conjunctive normal form* (CNF) if φ is a conjunction of clauses.

A formula φ is *satisfiable*, $\varphi \in \mathtt{SAT}$, if there is a valuation v s.t. $v \models \varphi$, that is $v(\varphi) = 1$.

We know that for any formula φ we can obtain a formula φ' in CNF, which is not much longer than φ, and φ and φ' are *equisatisfiable*—either both are satisfiable, or both are unsatisfiable.

Recall two special cases. The empty clause $\Box$ (empty disjunction) is unsatisfiable. The empty conjunction is satisfiable.

SAT problem

Given a formula φ in CNF decide whether $\varphi \in \mathtt{SAT}$.

Why is satisfiability important? Among other things it is possible to express other notions through it.

For any formula φ we have

$$\models \varphi \quad \text{iff} \quad \neg\varphi \text{ is a contradiction} \quad \text{iff} \quad \neg\varphi \notin \mathtt{SAT}.$$

Moreover, for any formula φ and a finite set of formulae Γ we have

$$\Gamma \models \varphi \quad \text{iff} \quad \bigwedge \Gamma \wedge \neg\varphi \text{ is a contradiction} \quad \text{iff} \quad \bigwedge \Gamma \wedge \neg\varphi \notin \mathtt{SAT}.$$

Example

$p, p \to q, q \to r \models r$ iff $p \wedge (p \to q) \wedge (q \to r) \wedge (\neg r) \notin \mathtt{SAT}$.

SAT solving applications

SAT solving is one of success stories in computer science. We are able to solve industrial problems containing millions of variables.

It is used, e.g., in

- formal verification — chip makers check correctness of their designs,
- security,
- bioinformatics — mutations in DNA,
- train safety,
- planning and scheduling,
- automated theorem proving.

CNF as a set of sets

We know that conjunctions and disjunctions are associative, commutative, and idempotent. Therefore a clause can be seen as a set of literals and a formula in CNF as a set of clauses.

Hence from now on we freely use

$$\varphi = \{\{\overline{p}, q\}, \{\overline{q}, r\}, \{\overline{r}, s\}, \{\overline{s}, t\}\}$$

instead of

$$(\overline{p} \vee q) \wedge (\overline{q} \vee r) \wedge (\overline{r} \vee s) \wedge (\overline{s} \vee t).$$

Note that φ is also a representation of

$$(t \vee \overline{s} \vee t) \wedge (\overline{q} \vee r) \wedge (r \vee \overline{q}) \wedge (\overline{r} \vee s) \wedge (\overline{p} \vee q).$$

Resolution rule — example

Assume we want to satisfy two clauses that contain contradicting literals simultaneously

$$\frac{q \vee p \qquad \overline{p} \vee r}{q \vee r}$$

If $v \models (q \vee p) \wedge (\overline{p} \vee r)$, then clearly $v \models q \vee r$.

Resolution rule

Let $l_1, \ldots, l_m, l_{m+1}, \ldots, l_{m+n}$ be literals and p be a propositional variable.

$$\frac{\{l_1, \ldots, l_m, p\} \qquad \{\overline{p}, l_{m+1}, \ldots, l_{m+n}\}}{\{l_1, \ldots, l_m, l_{m+1}, \ldots, l_{m+n}\}}$$

The clause $\{l_1, \ldots, l_m, l_{m+1}, \ldots, l_{m+n}\}$ produced by the resolution rule is called the *resolvent* of the two *input* clauses. We call p and $\overline{p}$ a *complementary pair*. We also say that it is a p*-resolvent* to emphasize the complementary pair.

Theorem (correctness)

For any valuation v*, if* $v \models \{l_1, \ldots, l_m, p\}$ *and* $v \models \{\overline{p}, l_{m+1}, \ldots, l_{m+n}\}$*, then* $v \models \{l_1, \ldots, l_m, l_{m+1}, \ldots, l_{m+n}\}$.

Hence the resolution rule preserves satisfiability.

Resolution calculus

The resolution calculus has no axioms and the only deduction rule is the resolution rule.

Resolution proof

A (resolution) proof of clause c from clauses $c_1, \ldots, c_n$ is a finite sequence of clauses $d_1, \ldots, d_m$ such that

- every d_i is among $c_1, \ldots, c_n$ or is derived by the resolution rule from input clauses d_j and d_k, for $1 \leq j < k < i \leq m$,
- $c = d_m$.

We say that a clause c is *provable* (derivable) from a set of clauses $\{c_1, \ldots, c_n\}$, we write $\{c_1, \ldots, c_n\} \vdash c$, if there is a proof of c from $c_1, \ldots, c_n$.

Resolution proof

Example

$$\dfrac{\dfrac{\dfrac{\{p\} \qquad \{\overline{p}, q\}}{\{q\}} \qquad \{\overline{q}, r\}}{\{r\}} \qquad \{\overline{r}\}}{\square}$$

is a proof of $\{\{p\}, \{\overline{p}, q\}, \{\overline{q}, r\}, \{\overline{r}\}\} \vdash \square$. Strictly speaking the presented derivation is not a sequence, but it is easy to produce a sequence from it.

Completeness of resolution calculus

It is not true that we can derive every valid formula in the resolution calculus, e.g., from the empty set we derive nothing. However, it is so called *refutationally complete*.

Theorem (completeness)

Let φ be a set of clauses. If φ is unsatisfiable, then $\varphi \vdash \square$.

Note that from the correctness theorem we already know the converse implication.

Theorem

Let φ be a set of clauses. If $\varphi \vdash \square$, then φ is unsatisfiable.

Deciding SAT using resolution

If we have a formula φ in CNF, a finite set of clauses, then we can clearly derive only finitely many clauses from it, say $\psi = \{c \colon \varphi \vdash c\}$. Note that if $\psi \vdash c$, then $c \in \psi$. We call such a set of clauses *saturated*—it is closed under the resolution rule.

This gives us a decision procedure for SAT. Either we produce the empty clause and hence $\varphi \notin \mathtt{SAT}$, or we produce a saturated set of clauses and hence $\varphi \in \mathtt{SAT}$.

Example

Let $\varphi = \{\{\overline{p}, \overline{q}\}, \{p, r\}, \{q, s\}\}$. A set of clauses $\{\{\overline{p}, \overline{q}\}, \{p, r\}, \{q, s\}, \{\overline{q}, r\}, \{\overline{p}, s\}, \{r, s\}\}$ is saturated. Hence $\varphi \in \mathtt{SAT}$.

Ordered resolution

Assume a formula $\{\{\overline{p}, \overline{q}\}, \{p, r\}, \{q, s\}\}$ and two possible derivations that differ only in the order of performed steps

$$\dfrac{\dfrac{\{\overline{p}, \overline{q}\} \qquad \{p, r\}}{\{\overline{q}, r\}} \qquad \{q, s\}}{\{r, s\}} \qquad\qquad \dfrac{\dfrac{\{\overline{p}, \overline{q}\} \qquad \{q, s\}}{\{\overline{p}, s\}} \qquad \{p, r\}}{\{r, s\}}$$

Is it necessary to try all such possible orderings? No, we can use an ordered resolution. We can always impose an order on variables and resolve using this order. Say $p < q$, meaning all p-resolvents must precede all q-resolvents.

Why is this enough? We try to produce the empty clause, it does not matter in which order we eliminate literals to achieve that goal.

Davis–Putnam algorithm

It was originally developed for first-order logic.

We have a set of clauses φ. We choose a variable p such that both p and $\overline{p}$ occur in φ and eliminate it—we produce all possible p-resolvents and add them to φ and then we remove all clauses in φ that contain p or $\overline{p}$. This operation preserves satisfiability.

Example

From $\{\{\overline{p},\overline{q}\},\{p,r\},\{q,s\}\}$ we obtain $\{\{\overline{q},r\},\{q,s\}\}$ by eliminating p and then we obtain $\{r,s\}$ by eliminating q. We cannot proceed and hence the original formula is satisfiable.

We can use many tricks, many proposed already by Davis and Putnam, to simplify searching, but in general the size of space needed to store clauses can grow exponentially.

Some properties of resolution

Subsumption

A clause c_1 is said to (syntactically) *subsume* a clause c_2 if $c_1 \subseteq c_2$.

If $c_1, c_2 \in \varphi$ and $c_1 \subseteq c_2$, then $\varphi \in \mathtt{SAT}$ iff $\varphi \setminus c_2 \in \mathtt{SAT}$. Moreover, this can shorten a derivation of the empty clause.

Example

From $\{\{p\}, \{p, r\}, \{\overline{p}, q\}, \{\overline{r}, q\}\}$ we obtain $\{\{p\}, \{\overline{p}, q\}, \{\overline{r}, q\}\}$ that is equisatisfiable.

Multiple resolvents

If it is possible to obtain more different resolvents from two clauses, then all these resolvents are tautologies, hence always satisfiable, and hence we can ignore them.

Example

$$\frac{\{p, \overline{q}\} \qquad \{\overline{p}, q\}}{\{q, \overline{q}\}} \qquad\qquad \frac{\{p, \overline{q}\} \qquad \{\overline{p}, q\}}{\{p, \overline{p}\}}$$

Conditioning — simplifications

To avoid space problems of the Davis–Putnam algorithm we use a different approach. We try to produce a satisfying valuation by assigning values to variables and we backtrack if necessary.

We select a literal l and replace it by `true` ($\top$). Hence $\bar{l}$ is replaced by `false` ($\bot$). This can lead to many simplifications of our set of clauses.

```
Require: A set of clauses $\varphi$, a literal $l$
  function $\textsc{Simplify}(\varphi, l)$
      $\varphi' \leftarrow \varphi$
      for $c \in \varphi'$ do
          if $l \in c$ then remove $c$ from $\varphi'$              ▷ satisfied clause
          else if $\bar{l} \in c$ then remove $\bar{l}$ from $c$     ▷ unsatisfied literal
      return $\varphi'$
```

Chronological backtracking algorithm

Using the previous simplification function, we can chronologically try to create a satisfying valuation.

```
Require: A set of clauses $\varphi$
  function $\textsc{IsSAT}(\varphi)$
      if $\varphi = \emptyset$ then return true                      ▷ no clause
      else if $\square \in \varphi$ then return false                ▷ empty clause
      else
          $l \leftarrow$ select a literal occurring in $\varphi$
          if $\textsc{IsSAT}(\textsc{Simplify}(\varphi, l))$ then return true
          else if $\textsc{IsSAT}(\textsc{Simplify}(\varphi, \bar{l}))$ then return true
          else return false
```

DPLL algorithm

The name stands for Davis, (Putnam), Logemann, and Loveland.

We improve our backtracking algorithm by the following two ideas:

Unit propagation

If a clause contains only a single literal l, then it is forced that l has to be true.

Example

For $\{\{p\}, \{\overline{p}, q\}, \{\overline{q}, r\}, \{\overline{r}\}\}$ we obtain unsatisfiability immediately after unit propagations and simplifications.

Note that unit propagation is a very powerful technique.

Pure literal elimination

A literal l is *pure*, if $\bar{l}$ does not occur in the formula. Hence we can satisfy all clauses containing l by assigning true to l.

Note that pure literal elimination is often ignored for efficiency reasons.

DPLL algorithm

```
Require: A set of clauses $\varphi$
  function $\mathrm{DPLL}(\varphi)$
      while $\varphi$ contains a unit clause $\{l\}$ do          ▷ unit propagation
          delete clauses containing $l$ from $\varphi$          ▷ unit subsumption
          delete $\bar{l}$ from all clauses in $\varphi$            ▷ unit resolution
      if $\square \in \varphi$ then return false                  ▷ empty clause
      while $\varphi$ contains a pure literal $l$ do
          delete clauses containing $l$ from $\varphi$
      if $\varphi = \emptyset$ then return true                    ▷ no clause
      else
          $l \leftarrow$ select a literal occurring in $\varphi$      ▷ a choice of literal
          if $\mathrm{DPLL}(\varphi \cup \{\{l\}\})$ then return true
          else if $\mathrm{DPLL}(\varphi \cup \{\{\bar{l}\}\})$ then return true
          else return false
```

DPLL — data structures

In real implementations we use trail — we keep whole set and construct a partial assignment during a computation. An efficient implementation of unit propagations is crucial.

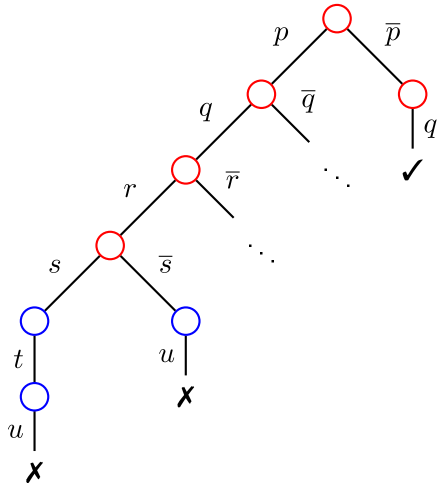
Watched literals

Instead of checking whole clauses all the time we select two distinct literals, called *watched literals*, in each clause. We also remember in which clauses a literal is selected. If we assign a value to a literal l, then we check only clauses where l is a watched literal. In these clauses we try to select another literal as a watched literal. If that is no longer possible, then we have a unit clause.

It has nice properties during backtracking, because there is no need to update current watched literals.

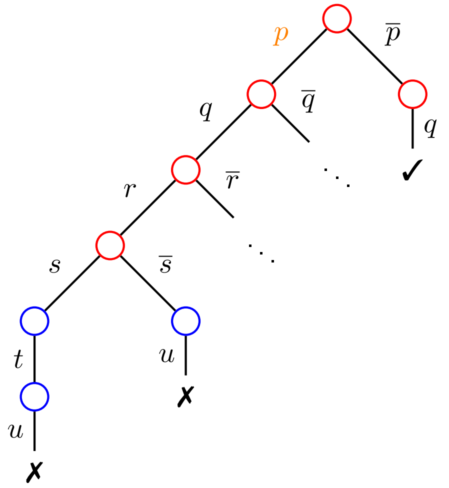
For details see, e.g., Knuth 2015; Biere et al. 2009.

How to improve backtracking in DPLL?



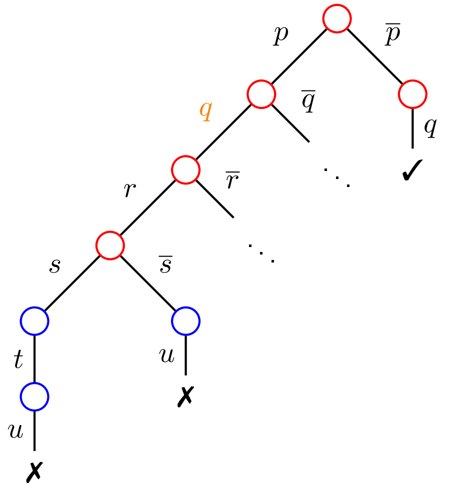
$$c_1 = \{p, q\}$$
$$c_2 = \{q, r\}$$
$$c_3 = \{\overline{p}, \overline{s}, t\}$$
$$c_4 = \{\overline{p}, s, u\}$$
$$c_5 = \{\overline{p}, \overline{t}, u\}$$
$$c_6 = \{\overline{p}, s, \overline{u}\}$$
$$c_7 = \{\overline{p}, \overline{t}, \overline{u}\}$$

How to improve backtracking in DPLL?



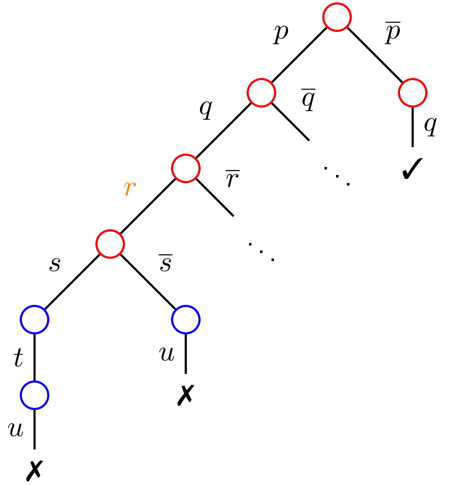
$c_1 = \{p, q\}$
$c_2 = \{q, r\}$
$c_3 = \{\overline{p}, \overline{s}, t\}$
$c_4 = \{\overline{p}, s, u\}$
$c_5 = \{\overline{p}, \overline{t}, u\}$
$c_6 = \{\overline{p}, s, \overline{u}\}$
$c_7 = \{\overline{p}, \overline{t}, \overline{u}\}$

How to improve backtracking in DPLL?



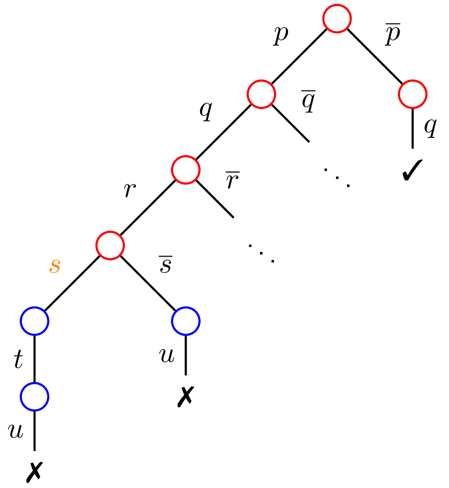
$$c_1 = \{p, q\}$$
$$c_2 = \{q, r\}$$
$$c_3 = \{\overline{p}, \overline{s}, t\}$$
$$c_4 = \{\overline{p}, s, u\}$$
$$c_5 = \{\overline{p}, \overline{t}, u\}$$
$$c_6 = \{\overline{p}, s, \overline{u}\}$$
$$c_7 = \{\overline{p}, \overline{t}, \overline{u}\}$$

How to improve backtracking in DPLL?



$c_1 = \{p, q\}$
$c_2 = \{q, r\}$
$c_3 = \{\overline{p}, \overline{s}, t\}$
$c_4 = \{\overline{p}, s, u\}$
$c_5 = \{\overline{p}, \overline{t}, u\}$
$c_6 = \{\overline{p}, s, \overline{u}\}$
$c_7 = \{\overline{p}, \overline{t}, \overline{u}\}$

How to improve backtracking in DPLL?



$c_1 = \{p, q\}$

$c_2 = \{q, r\}$

$c_3 = \{\overline{p}, \overline{s}, t\}$

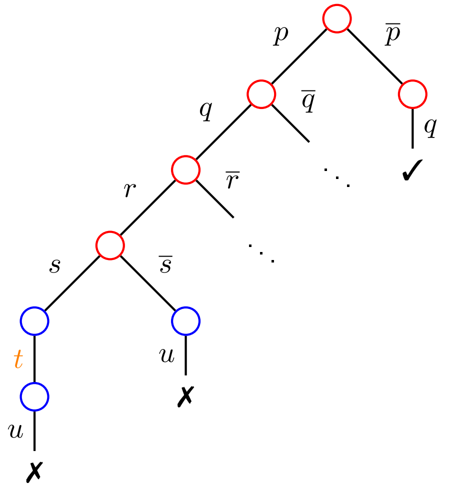
$c_4 = \{\overline{p}, s, u\}$

$c_5 = \{\overline{p}, \overline{t}, u\}$

$c_6 = \{\overline{p}, s, \overline{u}\}$

$c_7 = \{\overline{p}, \overline{t}, \overline{u}\}$

How to improve backtracking in DPLL?



$c_1 = \{p, q\}$

$c_2 = \{q, r\}$

$c_3 = \{\overline{p}, \overline{s}, t\}$

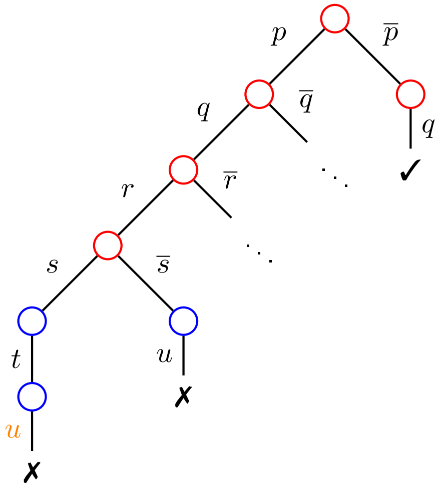
$c_4 = \{\overline{p}, s, u\}$

$c_5 = \{\overline{p}, \overline{t}, u\}$

$c_6 = \{\overline{p}, s, \overline{u}\}$

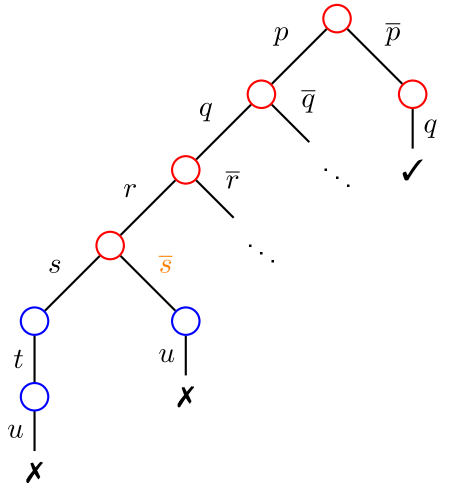
$c_7 = \{\overline{p}, \overline{t}, \overline{u}\}$

How to improve backtracking in DPLL?



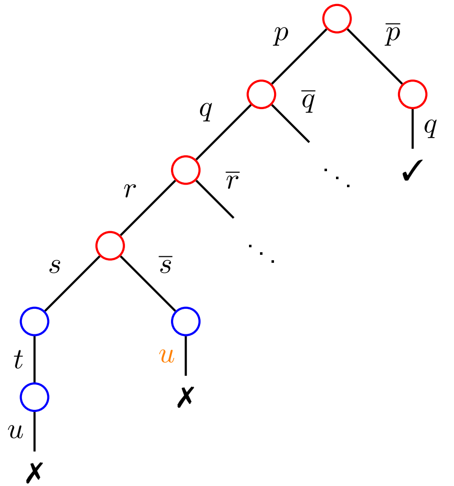
$c_1 = \{p, q\}$
$c_2 = \{q, r\}$
$c_3 = \{\overline{p}, \overline{s}, t\}$
$c_4 = \{\overline{p}, s, u\}$
$c_5 = \{\overline{p}, \overline{t}, u\}$
$c_6 = \{\overline{p}, s, \overline{u}\}$
$c_7 = \{\overline{p}, \overline{t}, \overline{u}\}$

How to improve backtracking in DPLL?



$c_1 = \{p, q\}$
$c_2 = \{q, r\}$
$c_3 = \{\overline{p}, \overline{s}, t\}$
$c_4 = \{\overline{p}, s, u\}$
$c_5 = \{\overline{p}, \bar{t}, u\}$
$c_6 = \{\overline{p}, s, \overline{u}\}$
$c_7 = \{\overline{p}, \bar{t}, \overline{u}\}$

How to improve backtracking in DPLL?

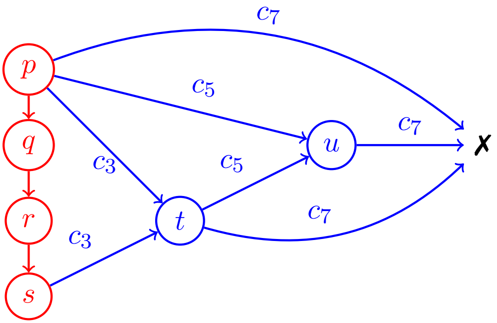


$c_1 = \{p, q\}$
$c_2 = \{q, r\}$
$c_3 = \{\overline{p}, \overline{s}, t\}$
$c_4 = \{\overline{p}, s, u\}$
$c_5 = \{\overline{p}, \overline{t}, u\}$
$c_6 = \{\overline{p}, s, \overline{u}\}$
$c_7 = \{\overline{p}, \overline{t}, \overline{u}\}$

Clearly detected conflicts do not depend on q and r. Hence there is no need to check different assignments for them and we have a non-chronological backtracking.

Implication graph — analyzing conflicts

Red vertices are decision points and blue vertices are caused by unit propagations. Red edges show the direction of decisions and blue edges the reasons for unit propagations.

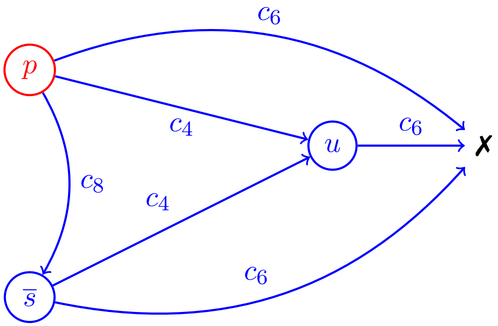


$c_1 = \{p, q\}$
$c_2 = \{q, r\}$
$c_3 = \{\overline{p}, \overline{s}, t\}$
$c_4 = \{\overline{p}, s, u\}$
$c_5 = \{\overline{p}, \bar{t}, u\}$
$c_6 = \{\overline{p}, s, \overline{u}\}$
$c_7 = \{\overline{p}, \bar{t}, \overline{u}\}$

Hence $(p \wedge s) \to \bot$ and hence $\top \to (\overline{p} \vee \overline{s})$ $(=\{\overline{p}, \overline{s}\})$. We can learn this clause and add it to our set of clauses. This avoids visiting the same conflict in a different branch.

Implication graph — analyzing conflicts

We can also analyze the second conflict now.

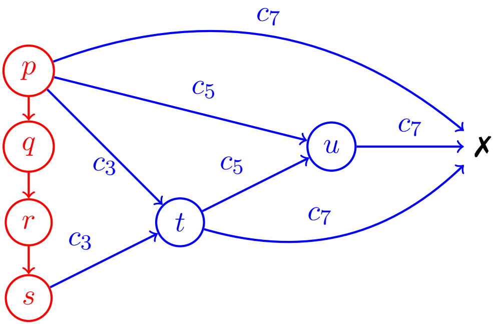


$c_1 = \{p, q\}$
$c_2 = \{q, r\}$
$c_3 = \{\overline{p}, \overline{s}, t\}$
$c_4 = \{\overline{p}, s, u\}$
$c_5 = \{\overline{p}, \overline{t}, u\}$
$c_6 = \{\overline{p}, s, \overline{u}\}$
$c_7 = \{\overline{p}, \overline{t}, \overline{u}\}$
$c_8 = \{\overline{p}, \overline{s}\}$

Hence we learn $c_9 = \{\overline{p}\}$.

Implication graph — various cuts

It was possible to learn a different clause.



$$
\begin{aligned}
c_1 &= \{p, q\} \\
c_2 &= \{q, r\} \\
c_3 &= \{\overline{p}, \overline{s}, t\} \\
c_4 &= \{\overline{p}, s, u\} \\
c_5 &= \{\overline{p}, \overline{t}, u\} \\
c_6 &= \{\overline{p}, s, \overline{u}\} \\
c_7 &= \{\overline{p}, \overline{t}, \overline{u}\}
\end{aligned}
$$

We usually prefer to learn $\{\overline{p}, \overline{t}\}$ instead of $\{\overline{p}, \overline{s}\}$. Because t is so called dominator, all paths from s to the conflict go through t.

We call such dominators *unique implication points* (UIP) and a popular strategy is to learn the first UIP (the one closest to the conflict).

Conflict-Driven Clause Learning (CDCL)

It is the DPLL algorithm with non-chronological backtracking, called back jumping, and clause learning.

Restarts

It is useful to restart a CDCL solver from time to time. We forget all assignments but keep the learned clauses.

Delete learned clauses

It is necessary to delete some learned clauses to avoid space problems and hence we try to keep only the most useful clauses.

Preprocessing

We usually try to minimize the input problem using subsumptions and variable eliminations.

Decision heuristics

How to select a literal? Many approaches, but it has to be fast.

Focus heuristics

In CDCL we try to find small unsatisfiable subsets and hence prefer variables involved in recent conflicts.

Modern solvers usually use a variant of VSIDS (Variable State Independent Decaying Sum). We start with the number of occurrences of a variable in all clauses. If a conflict clause c is detected, then the score of all variables in c is increased. Moreover, we periodically divide our scores by a constant.

Global heuristics

Good for hard problems. We look-ahead on a literal l. It means that we assume l, then we apply unit propagations and check clauses that are shortened by this assignment, but not completely satisfied. We prefer literals that produce shorter clauses.

Parallel solving

SAT solving is difficult to parallelize.

Cube and conquer

We generate many partial assignments, e.g., by a breath-first search with a limited maximal depth, and try to solve them.

Portfolio approach

We run multiple solvers (usually the same one) with different settings on the same formula. We share clauses among solvers. The main problems are how to diversify our portfolio and clause sharing — which clauses, how many, when, . . .

It works very well on large problems that are easy to solve.

Probabilistic algorithms — stochastic local search

We start with a random complete valuation and try to minimize the number of unsatisfied clauses by flipping values.
It is an open problem how to use these techniques for showing unsatisfiability.

GSAT

```
function $\mathrm{GSAT}(\varphi)$
    for $i \in (1, MAXITERS)$ do
        $v \leftarrow$ a random valuation on $\varphi$
        for $j \in (1, MAXFLIPS)$ do
            if $v \models \varphi$ then return $v$
            else minimize #unsat clauses by flipping a variable
    return None
```

Walksat

We try to avoid local minima. Select randomly a unsatisfied clause c. If by flipping a variable x occurring in c no new unsatisfied clause emerges, then flip x. Otherwise with a probability p flip a variable x in c and with a probability $(1-p)$ perform a GSAT step.

How to select a SAT solver?

Try different solvers, they use the same input format and hence it is easy to experiment. However, the good encoding of your problem is usually at least as important as a good solver.

MiniSat is free, fast, and very popular implementation in C. It won all three industrial categories in the SAT Competition 2005. A new version is called MiniSat 2. However, it is not the state of the art. A good choice if you want to use a SAT solver in your software.

For playing in Python you can use pycosat, a package that provides bindings to PicoSAT on the C level.

Check results of SAT Competition 2018 and from previous years for the state of the art.

DIMACS format

The standard input format for SAT solvers.

Variables are enumerated $1, 2, \ldots$. A variable x_i is represented by i and $\overline{x_i}$ by $-i$. A clause is a list of non-zero integers separated by spaces, tabs, or newlines. The end of a clause is represented by zero. The order of literals and clauses is irrelevant.

```
c start with comments
c
p cnf 5 3                                   #variables #clauses
1 -5 4 0
-1 5 3 4 0
-3 -4 0
```

encodes

$$(x_1 \vee \overline{x_5} \vee x_4) \wedge (\overline{x_1} \vee x_5 \vee x_3 \vee x_4) \wedge (\overline{x_3} \vee \overline{x_4}).$$

Certifying unsatisfiability

It is easy to convince someone that a formula is satisfiable by showing an assignment. To certificate that it is unsatisfiable is not so easy. It can be exponentially long and usually such a certificate is provided in a form of resolution proof.

A standard format currently used is called DRAT (Delete Resolution Asymmetric Tautologies).

Bibliography I

Biere, Armin et al., eds. (Feb. 2009). *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, p. 980. ISBN: 978-1-58603-929-5.

Knuth, Donald E. (2015). *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. 1st. Addison-Wesley Professional. ISBN: 978-0-13-439760-3.