Logical reasoning and programming

SAT solving—DPLL and CDCL

Karel Chvalovský

CIIRC CTU

Recap

We deal with formulae in conjunctive normal form (CNF)

$$(\cdots \vee \cdots \vee \cdots) \wedge \cdots \wedge (\cdots \vee \cdots \vee \cdots)$$

and we represent them using

$$\{\{\ldots\}, \ldots, \{\ldots\}\}.$$

Our problem, given a set of clauses φ:

$$\text{Is } \varphi \in \texttt{SAT}?$$

Davis–Putnam algorithm

We produce all possible p-resolvents and add them to φ and then we remove all clauses in φ that contain p or $\overline{p}$.

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.

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 φ, a literal l
  function $\textsc{Simplify}(\varphi, l)$
    $\varphi' \leftarrow \varphi$
    for $c \in \varphi'$ **do**
      if $l \in c$ **then** remove c from φ' ▷ satisfied clause
      else if $\bar{l} \in c$ **then** remove $\bar{l}$ from c ▷ unsatisfied literal
    return φ'

Chronological backtracking algorithm

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

Require: A set of clauses φ
  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 φ
      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 [Davis, Logemann, and Loveland 1962]

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

We improve our backtracking algorithm by the following two ideas:

Unit propagation (also called Boolean constraint 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 φ
  function $\mathrm{DPLL}(\varphi)$
    while φ contains a unit clause $\{l\}$ **do** ▷ unit propagation
      delete clauses containing l from φ ▷ unit subsumption
      delete $\bar{l}$ from all clauses in φ ▷ unit resolution
    if $\square \in \varphi$ **then return** `false` ▷ empty clause
    while φ contains a pure literal l **do**
      delete clauses containing l from φ
    if $\varphi = \emptyset$ **then return** `true` ▷ no clause
    else
      $l \leftarrow$ select a literal occurring in φ ▷ 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.

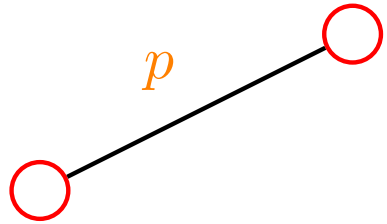
For details see, e.g., Knuth 2015; Biere, M. J. H. Heule, et al. 2009; Biere, M. Heule, et al. 2021.

Example: DPLL (without pure literal elimination)

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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

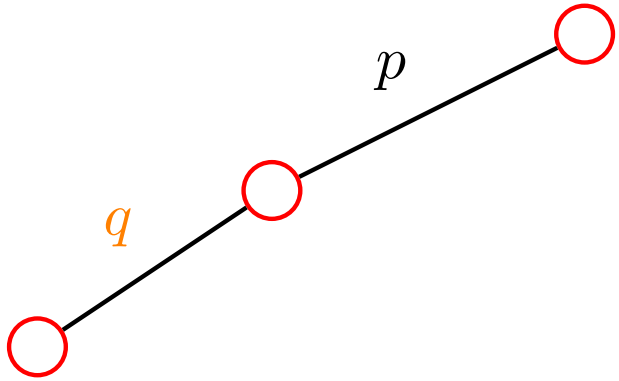
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

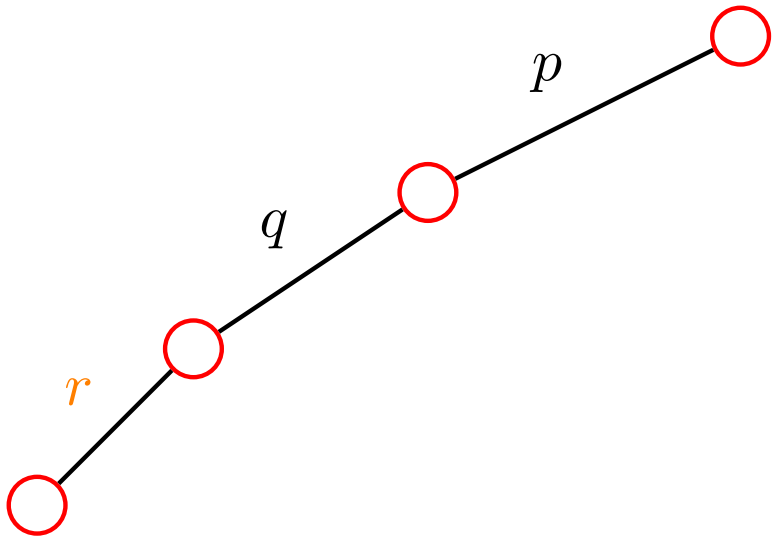
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

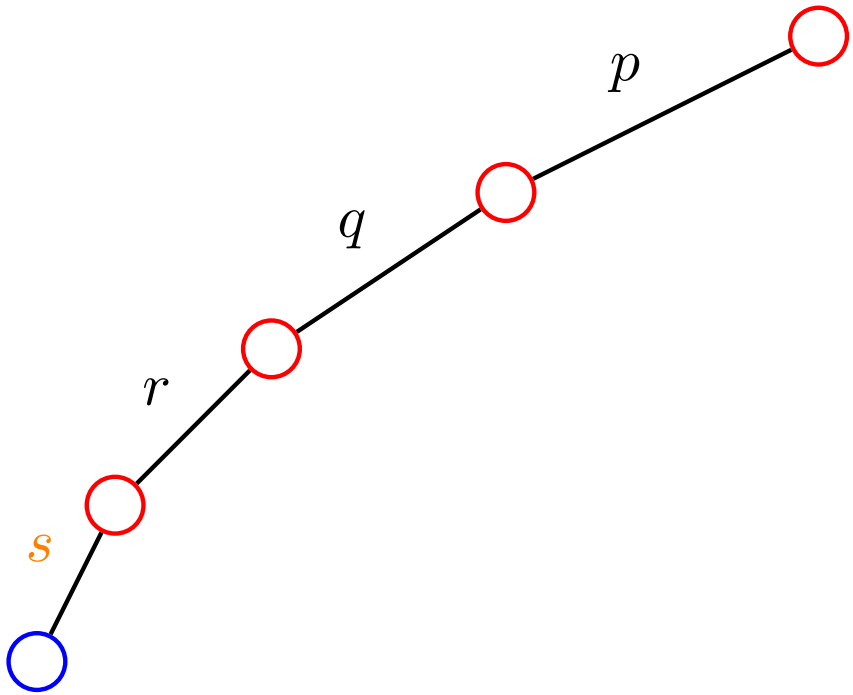
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

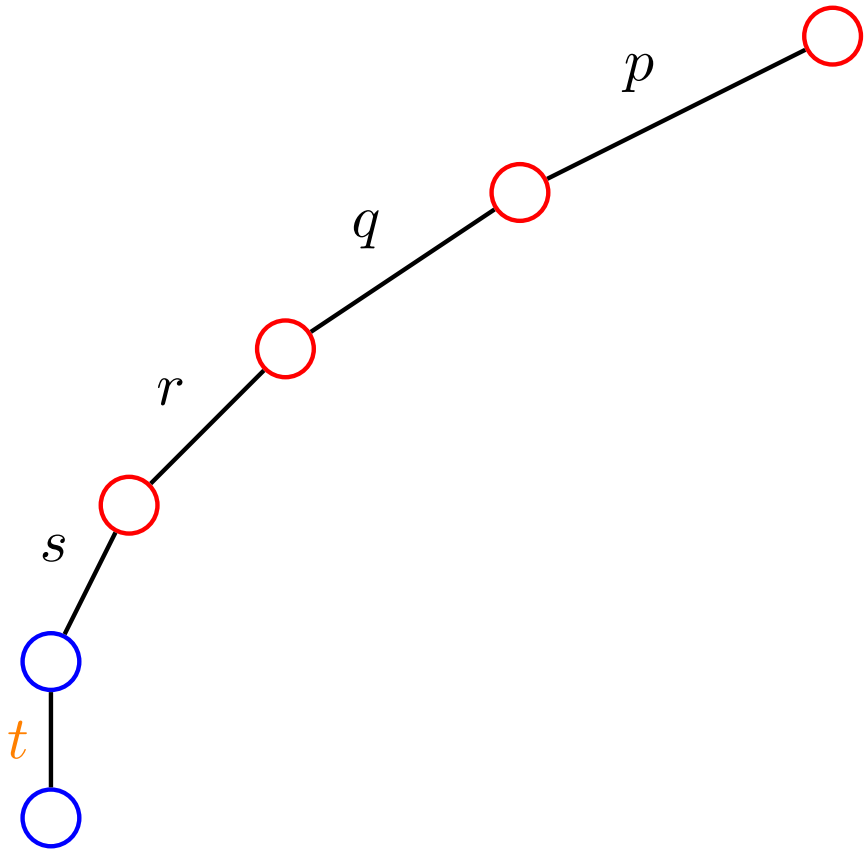
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

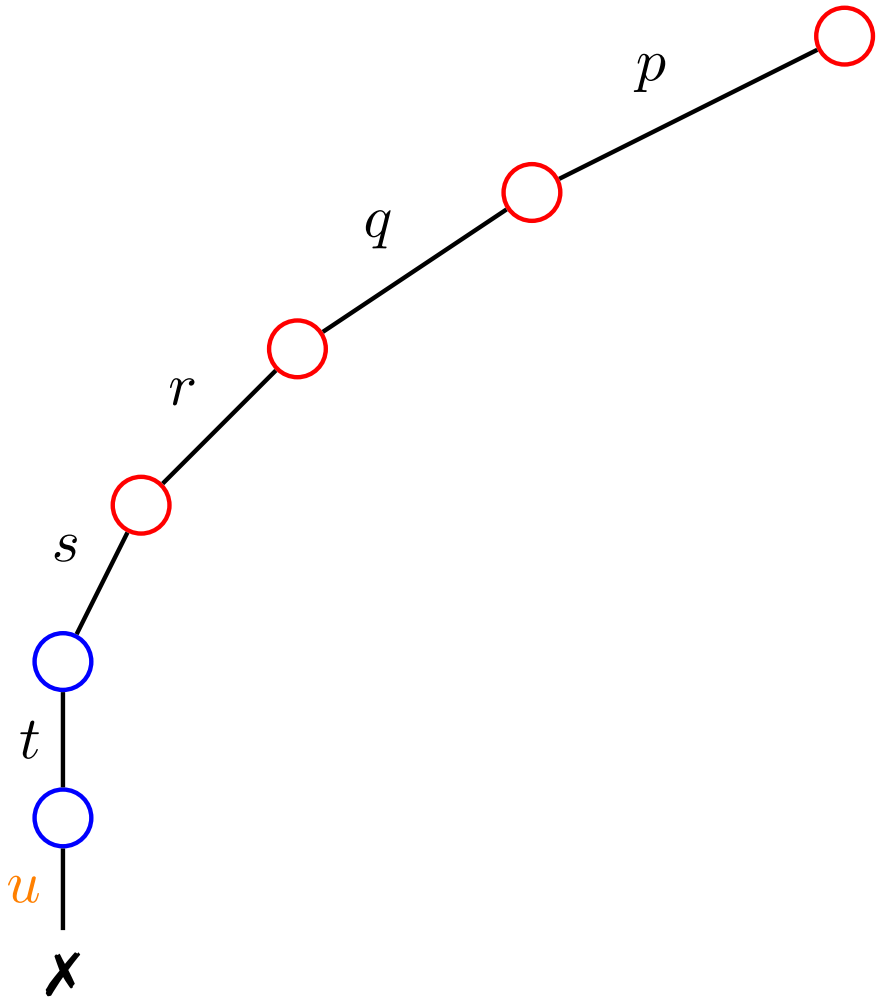
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

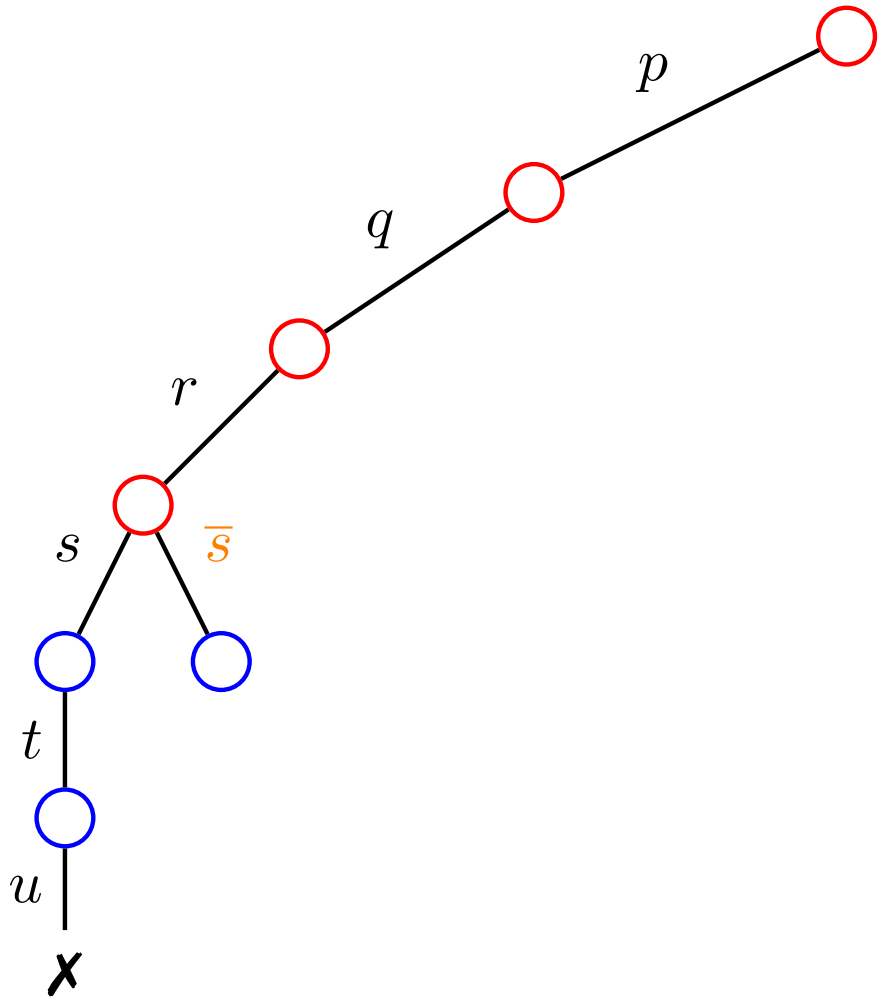
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

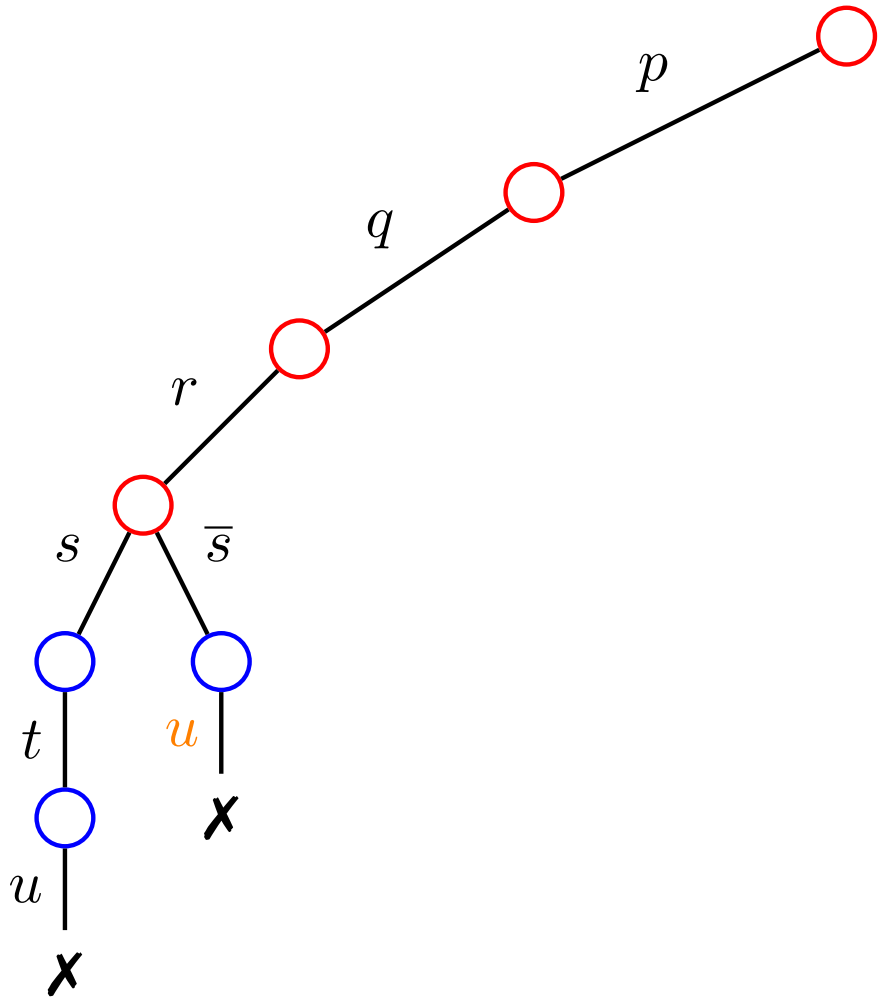
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

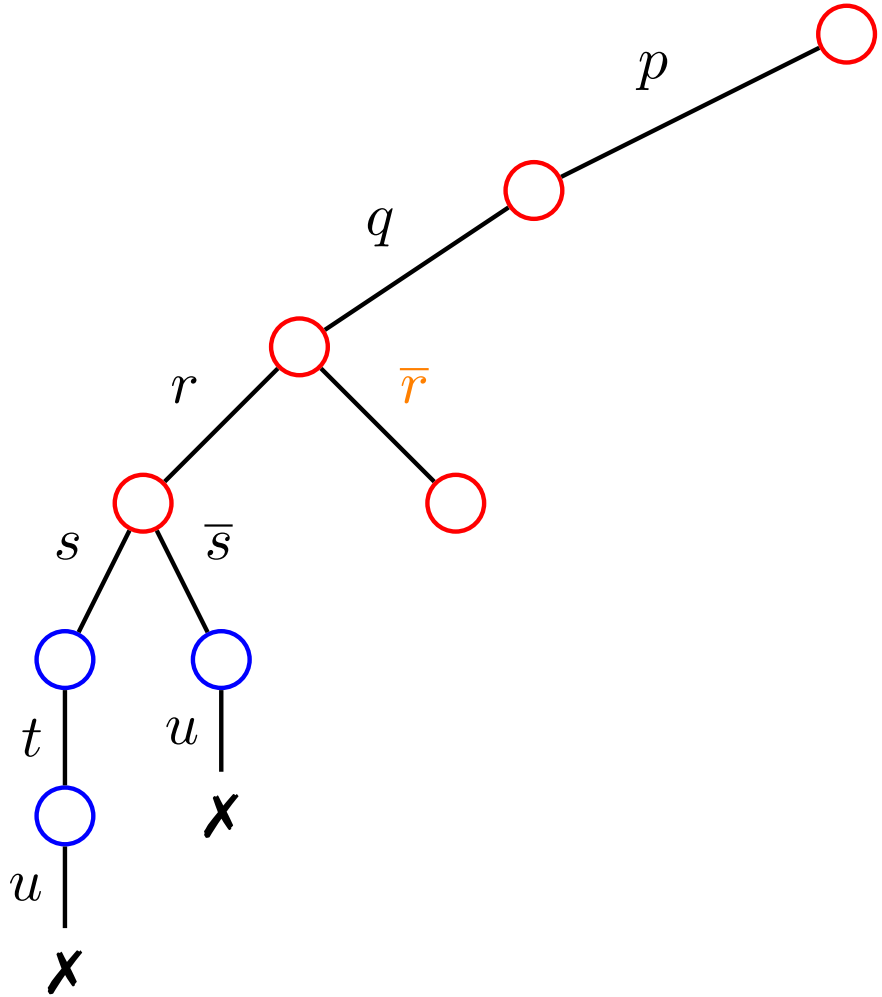
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

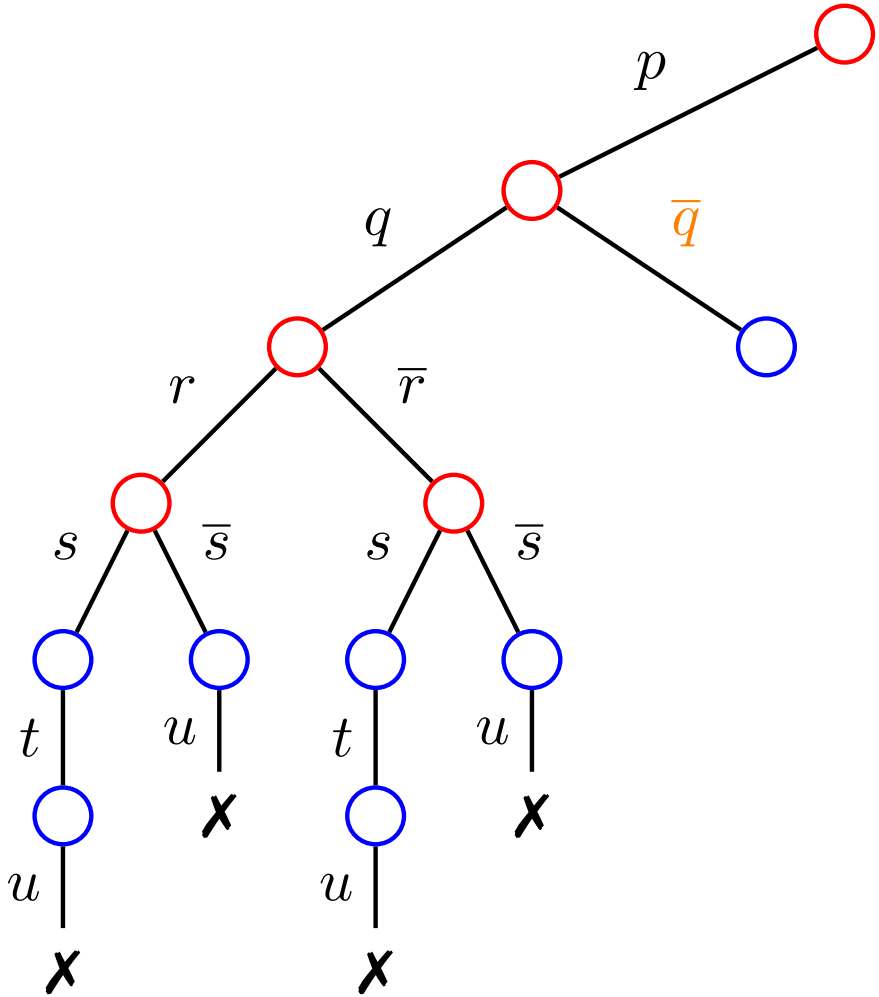
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

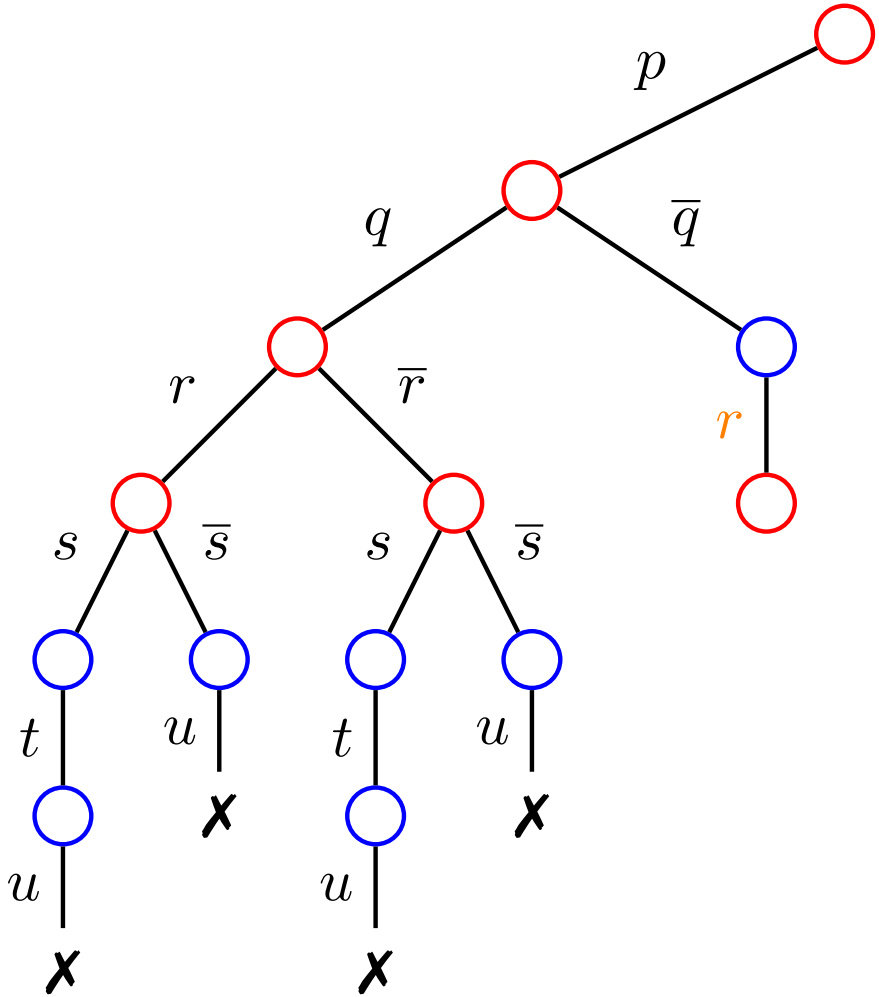
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

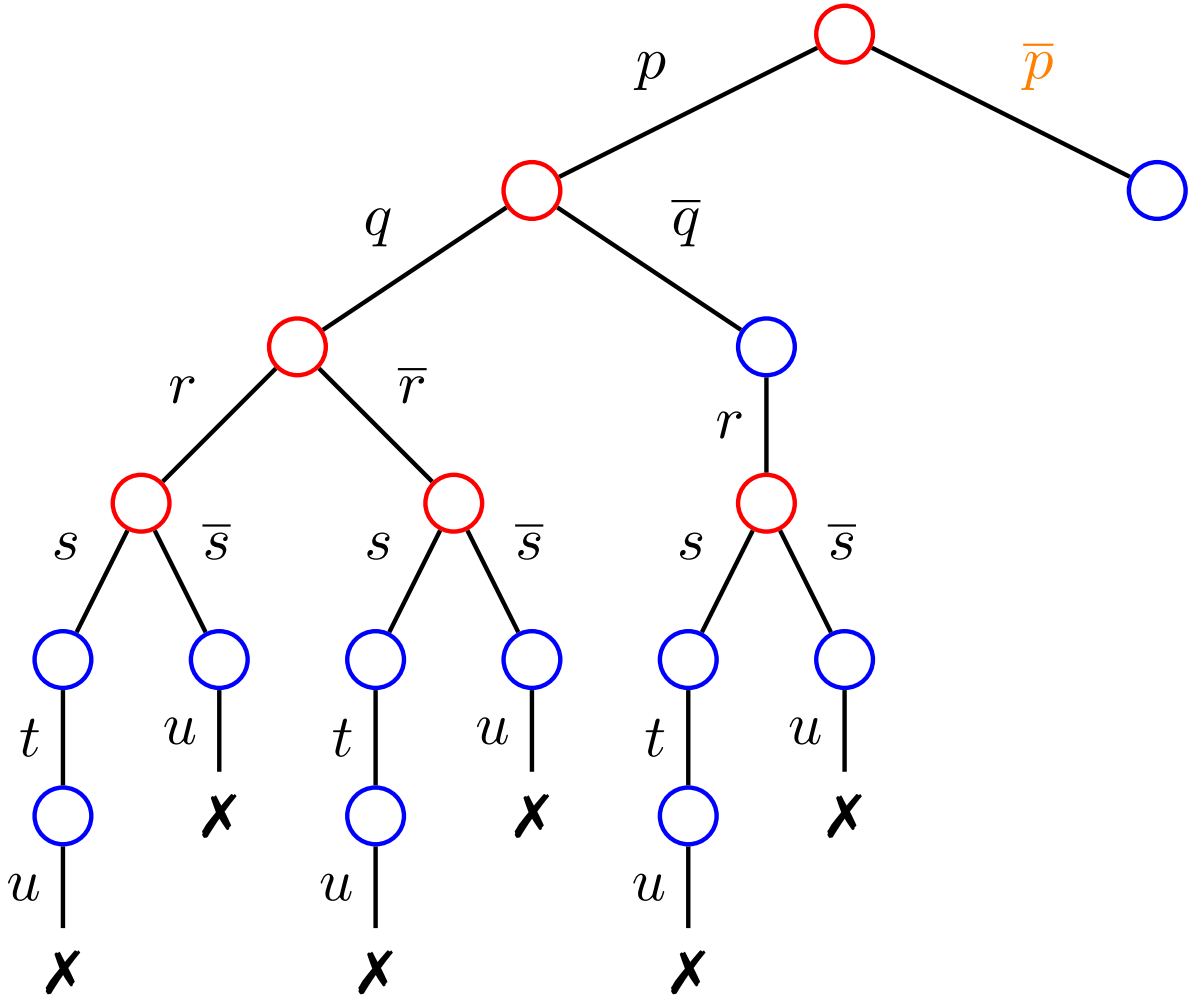
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

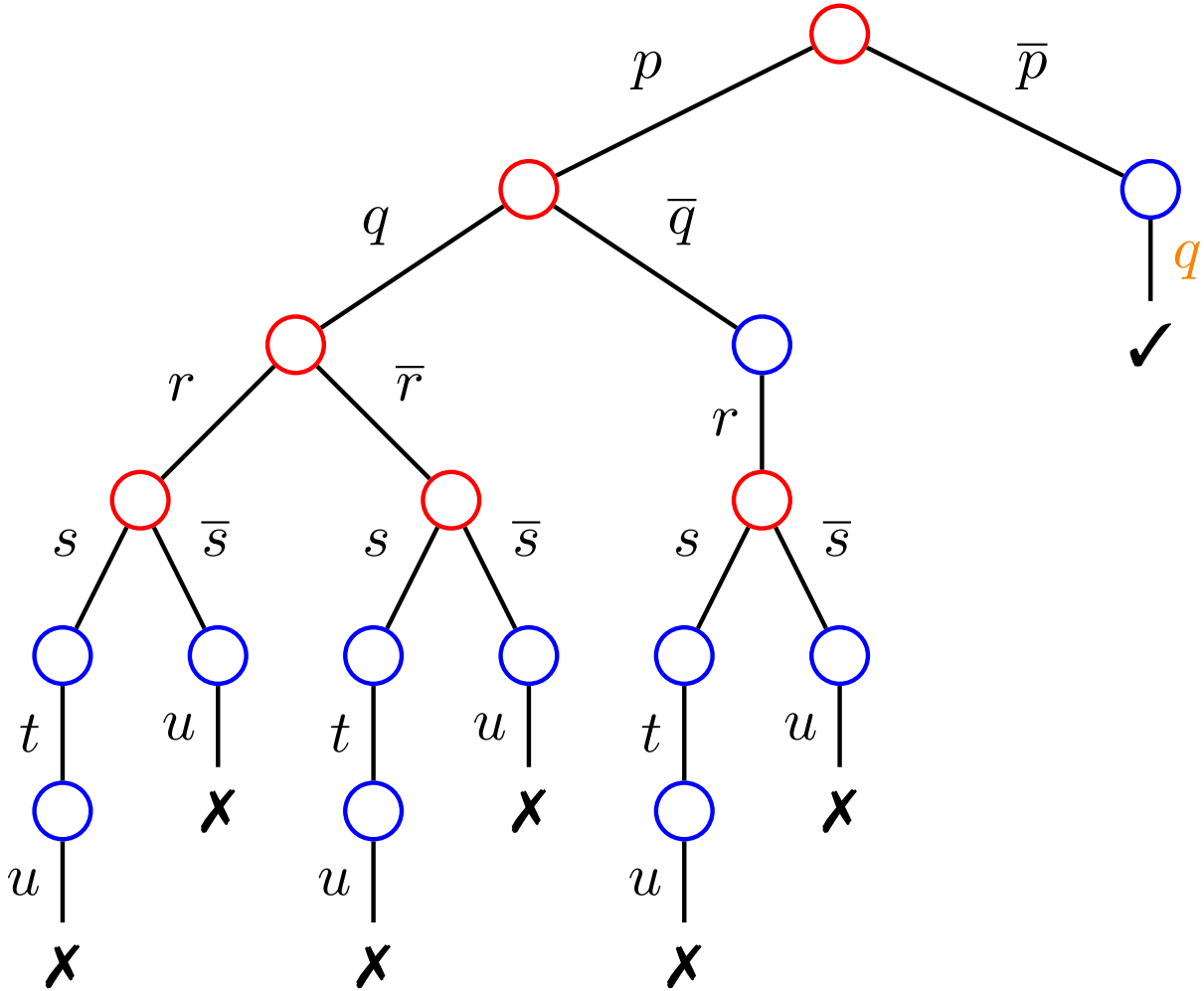
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

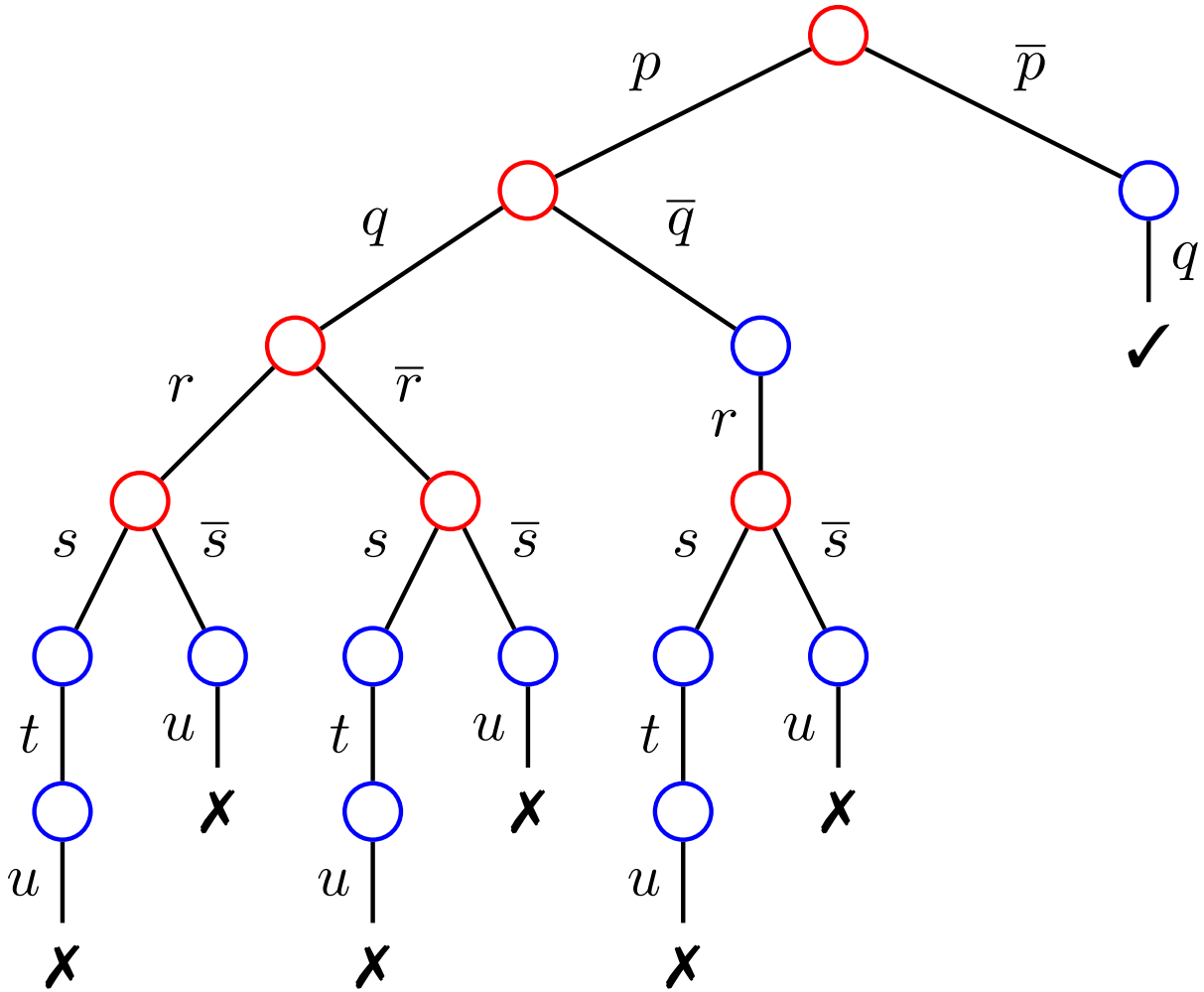
Example: DPLL (without pure literal elimination)



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

For simplicity, we fix the order of choices to $p < q < r < s < t < u$ and always select a positive literal first, but any unselected literal can be chosen and in any order (positive/negative).

How to improve backtracking in DPLL?

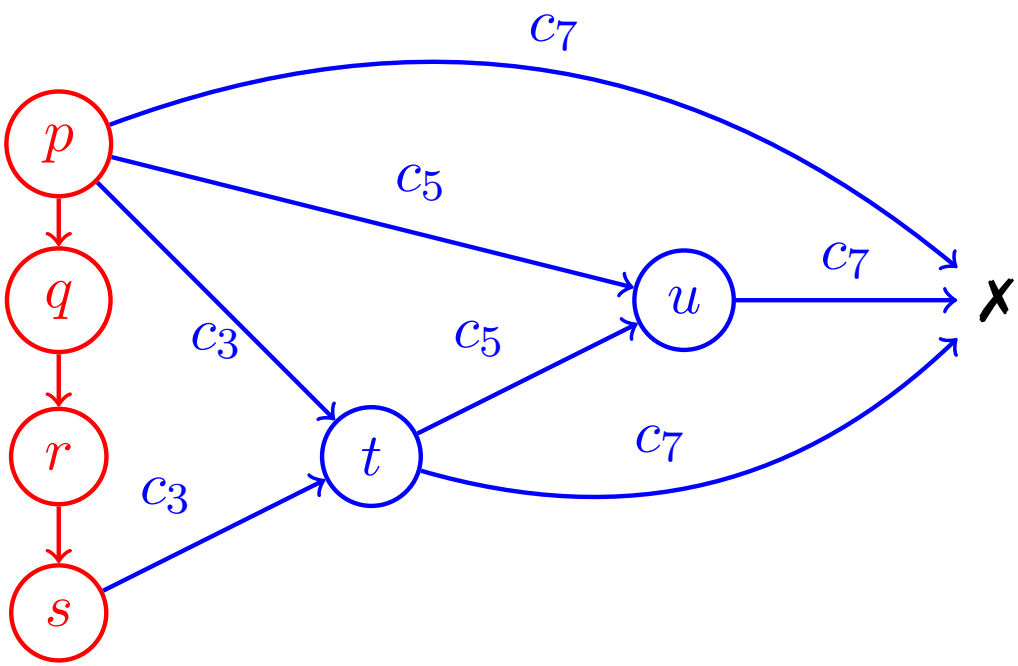


$$c_1 = \{p, q\}$$
$$c_2 = \{q, r\}$$
$$c_3 = \{\overline{p}, \overline{s}, t\}$$
$$c_4 = \{\overline{p}, s, u\}$$
$$c_5 = \{\overline{p}, \overline{t}, u\}$$
$$c_6 = \{\overline{p}, s, \overline{u}\}$$
$$c_7 = \{\overline{p}, \overline{t}, \overline{u}\}$$

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.

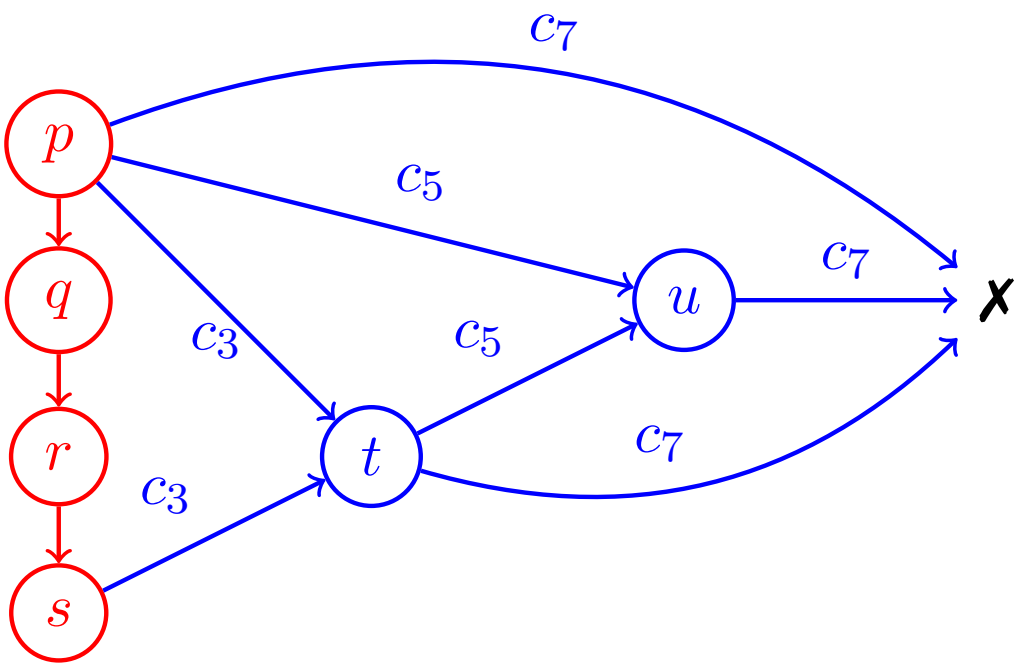


$$
\begin{aligned}
c_1 &= \{p, q\} \\
c_2 &= \{q, r\} \\
c_3 &= \{\overline{p}, \overline{s}, t\} \\
c_4 &= \{\overline{p}, s, u\} \\
c_5 &= \{\overline{p}, \overline{t}, u\} \\
c_6 &= \{\overline{p}, s, \overline{u}\} \\
c_7 &= \{\overline{p}, \overline{t}, \overline{u}\}
\end{aligned}
$$

Hence $(p \wedge s) \rightarrow \bot$ that is equivalent to $\{\overline{p}, \overline{s}\}$. We can learn this clause and add it to our set of clauses. This prevents us from visiting the same conflict in a different branch.

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.

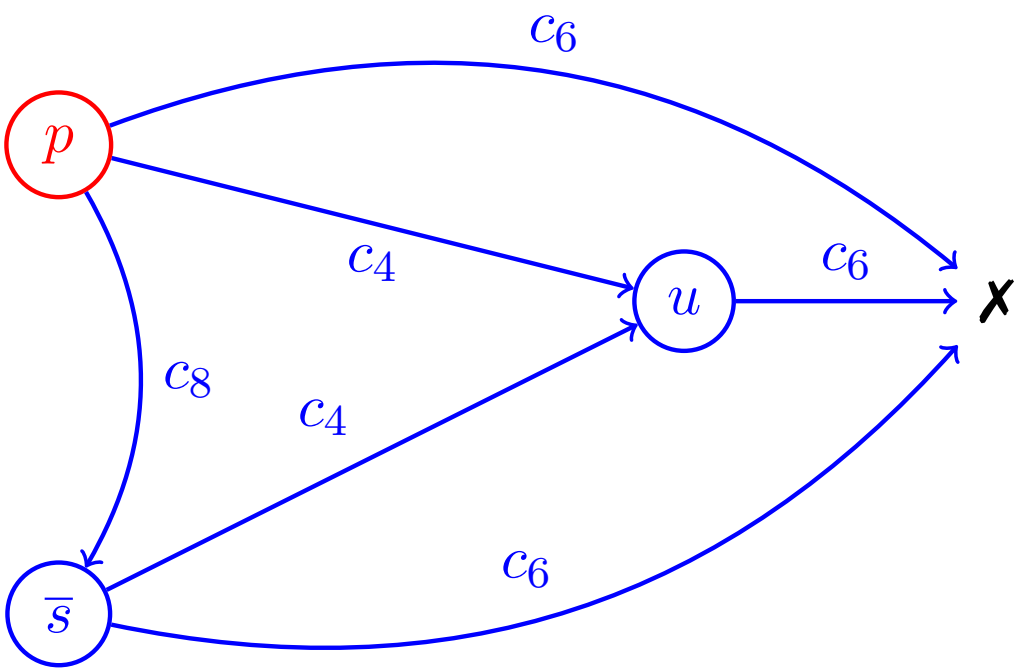


$$
\begin{aligned}
c_1 &= \{p, q\} \\
c_2 &= \{q, r\} \\
c_3 &= \{\overline{p}, \overline{s}, t\} \\
c_4 &= \{\overline{p}, s, u\} \\
c_5 &= \{\overline{p}, \overline{t}, u\} \\
c_6 &= \{\overline{p}, s, \overline{u}\} \\
c_7 &= \{\overline{p}, \overline{t}, \overline{u}\}
\end{aligned}
$$

Moreover, $\{\overline{p}, \overline{s}\}$ is an *asserting clause*; it contains exactly one literal that depends on the last decision. Hence an asserting clause flips a literal on the last decision level. We learn only such clauses.

Implication graph — analyzing conflicts

We can also analyze the second conflict now.

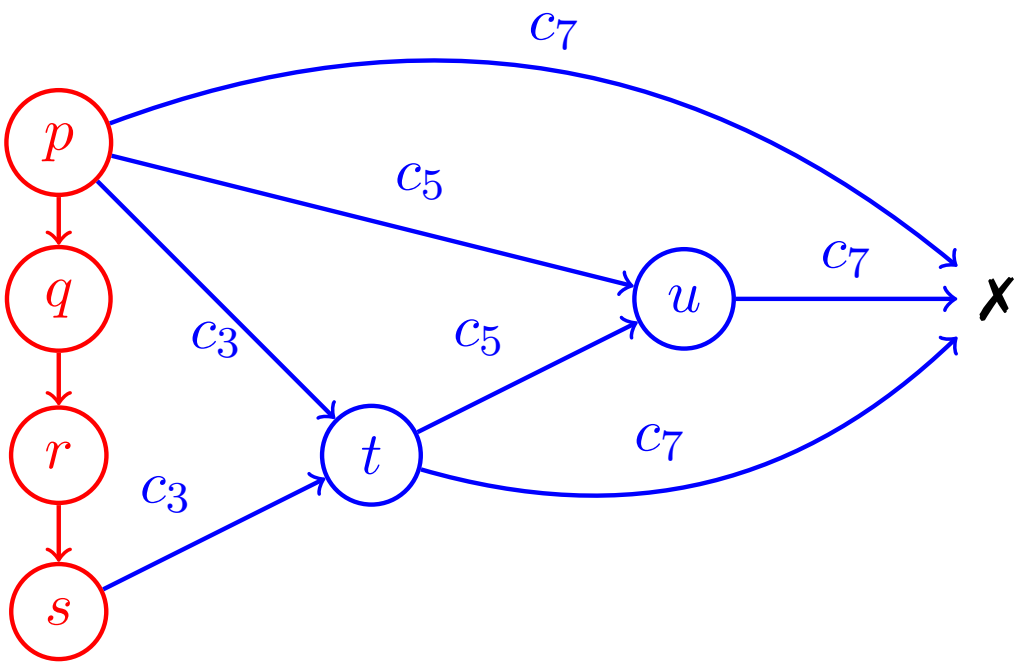


$$
\begin{aligned}
c_1 &= \{p, q\} \\
c_2 &= \{q, r\} \\
c_3 &= \{\overline{p}, \overline{s}, t\} \\
c_4 &= \{\overline{p}, s, u\} \\
c_5 &= \{\overline{p}, \overline{t}, u\} \\
c_6 &= \{\overline{p}, s, \overline{u}\} \\
c_7 &= \{\overline{p}, \overline{t}, \overline{u}\} \\
c_8 &= \{\overline{p}, \overline{s}\}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

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

Implication graph — various cuts

It was possible to learn a different clause.

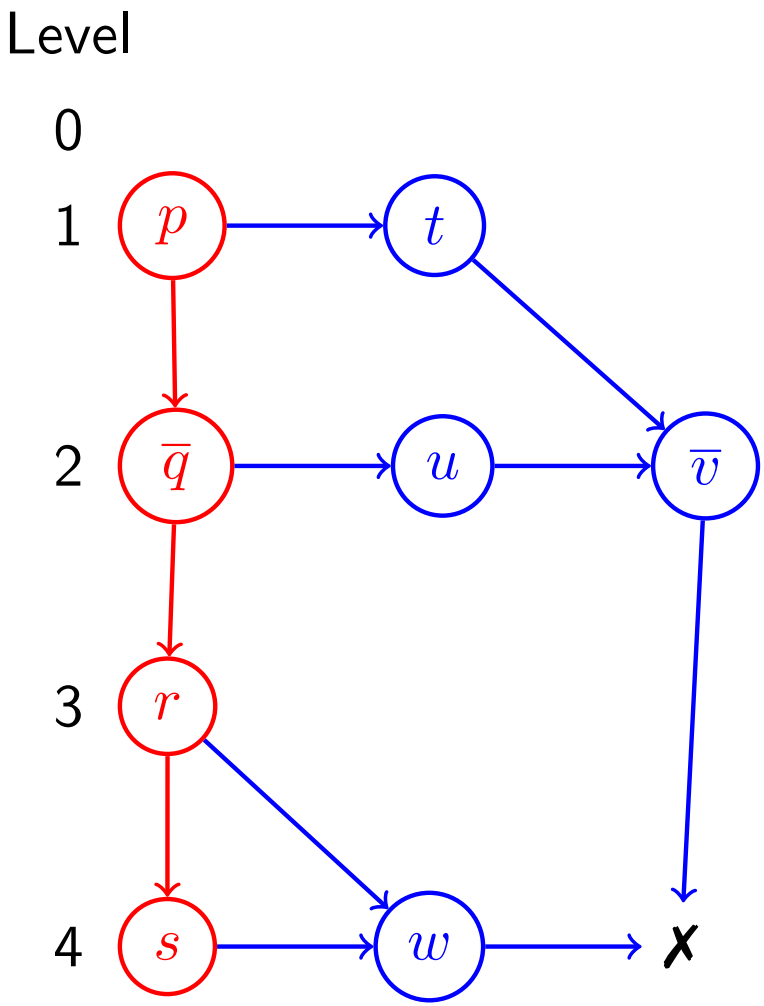


$$
\begin{aligned}
c_1 &= \{p, q\} \\
c_2 &= \{q, r\} \\
c_3 &= \{\overline{p}, \overline{s}, t\} \\
c_4 &= \{\overline{p}, s, u\} \\
c_5 &= \{\overline{p}, \overline{t}, u\} \\
c_6 &= \{\overline{p}, s, \overline{u}\} \\
c_7 &= \{\overline{p}, \overline{t}, \overline{u}\}
\end{aligned}
$$

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) on the path to the last decision point.

Implication graph — various decision levels

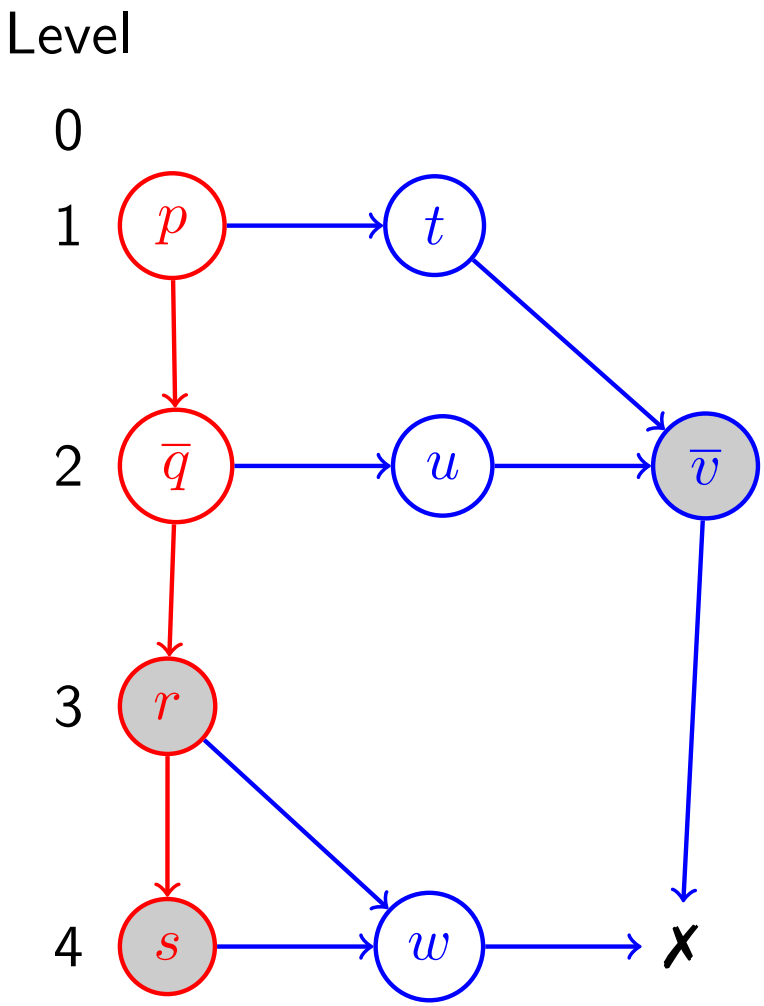


We want

- a literal assigned at the last level,
- literals involved assigned at previous levels.

Not necessarily decision literals!

Implication graph — various decision levels



We want

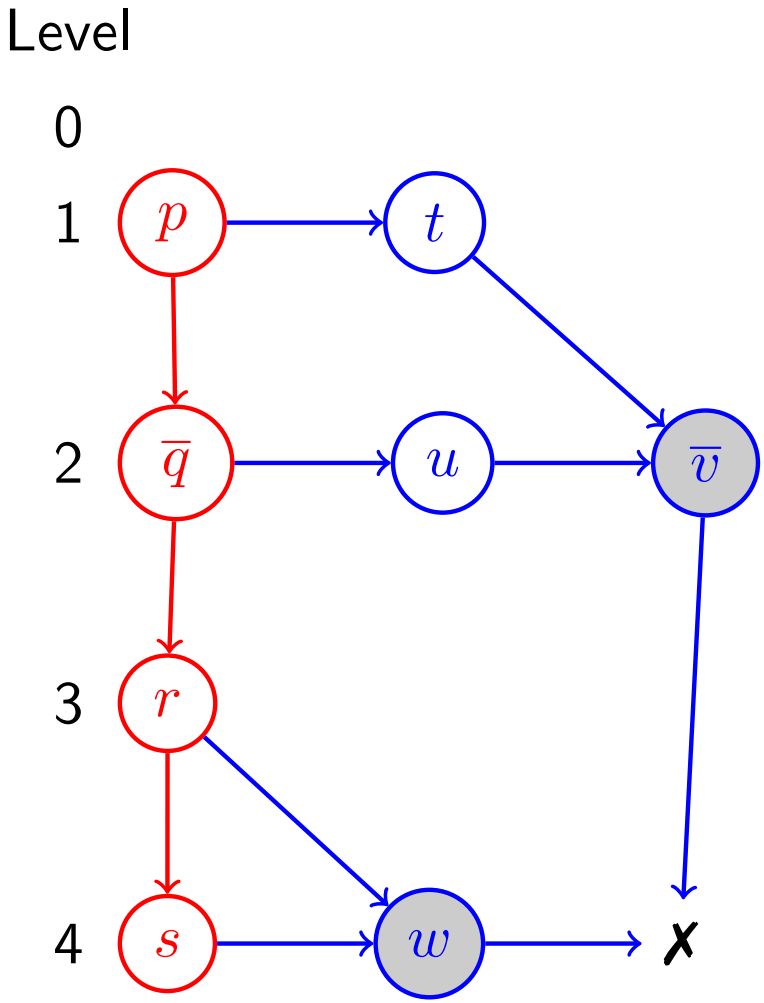
- a literal assigned at the last level,
- literals involved assigned at previous levels.

Not necessarily decision literals!

Here one option is

$$\{\overline{s}, \overline{r}, v\}.$$

Implication graph — various decision levels



We want

- a literal assigned at the last level,
- literals involved assigned at previous levels.

Not necessarily decision literals!

Here one option is

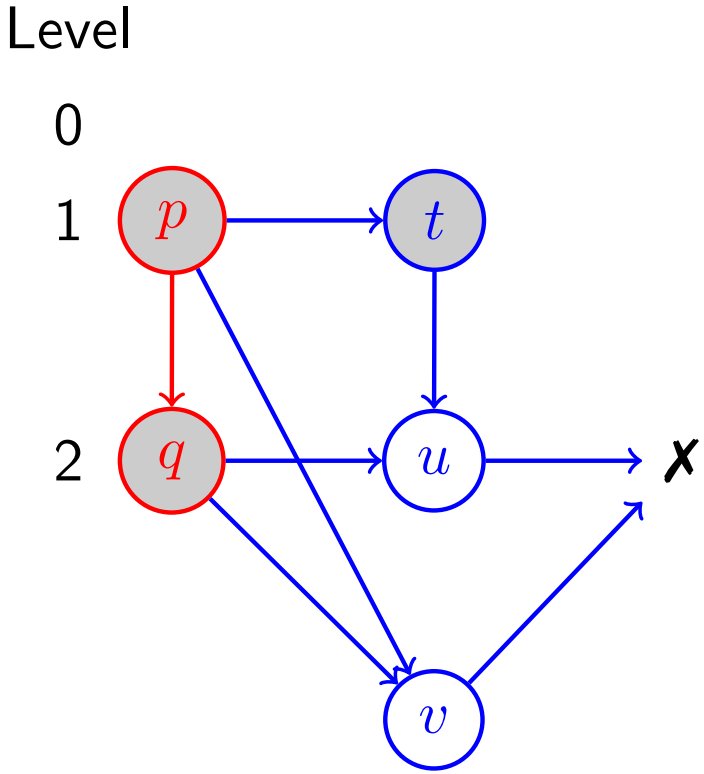
$$\{\overline{s}, \overline{r}, v\}.$$

However, the first UIP gives

$$\{\overline{w}, v\}$$

and we go to the decision level 2.

Learned clause minimization (local)



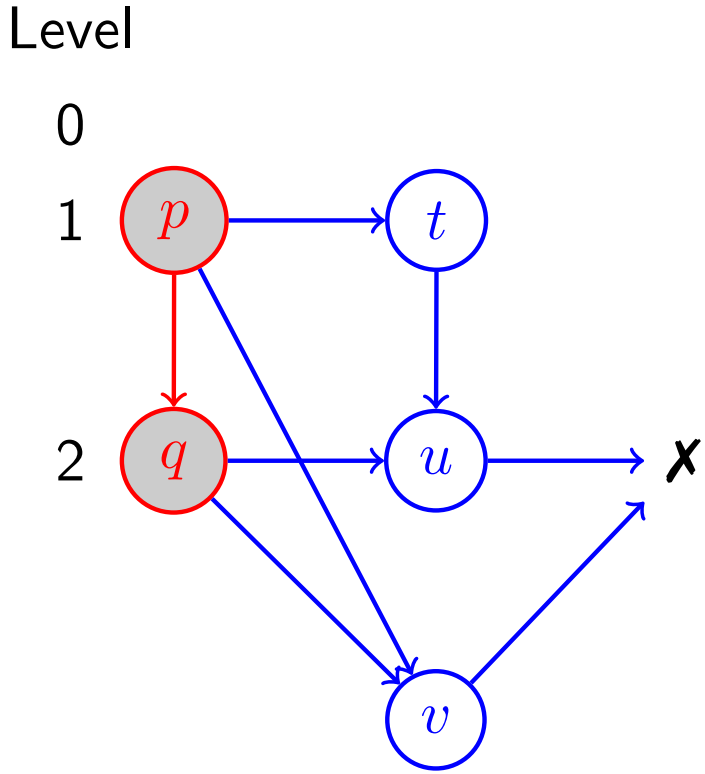
Here we learn

$$\{\overline{q}, \overline{p}, \overline{t}\}.$$

However, we get t using

$$\{\overline{p}, t\}.$$

Learned clause minimization (local)



Here we learn

$$\{\overline{q}, \overline{p}, \overline{t}\}.$$

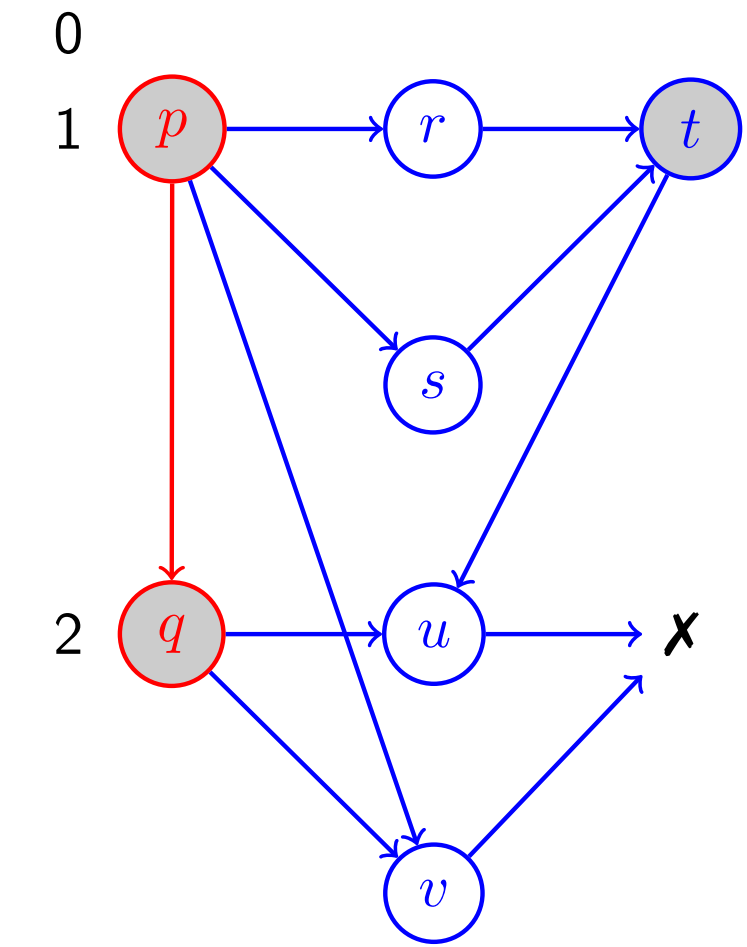
However, we get t using

$$\{\overline{p}, t\}.$$

Hence by self-subsumption resolution we obtain

$$\{\overline{q}, \overline{p}\}.$$

Learned clause minimization (recursive)



Here we learn

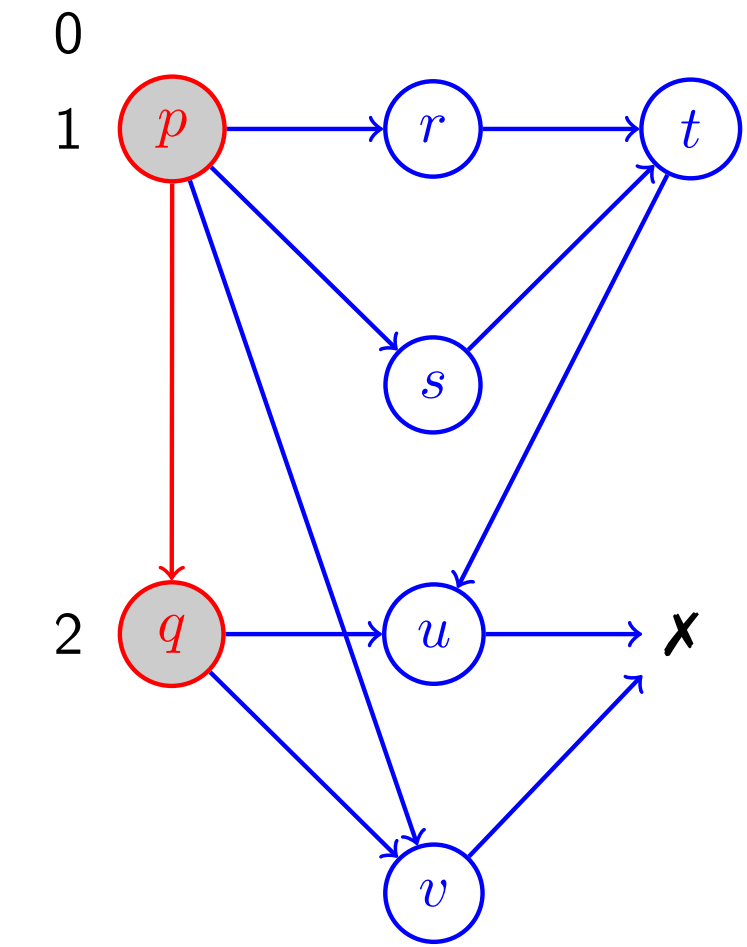
$$\{\overline{q}, \overline{p}, \overline{t}\}.$$

and the previous approach fails here; we get a longer clause using

$$\{\overline{r}, \overline{s}, t\},$$

which gives us t.

Learned clause minimization (recursive)



Here we learn

$$\{\overline{q}, \overline{p}, \overline{t}\}.$$

and the previous approach fails here; we get a longer clause using

$$\{\overline{r}, \overline{s}, t\},$$

which gives us t.

Hence a recursive minimization, which is more time-consuming, is necessary to get

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

taking into account also clauses $\{\overline{p}, r\}$ and $\{\overline{p}, s\}$.

Conflict-Driven Clause Learning (CDCL)

It is the DPLL algorithm with non-chronological backtracking, called back jumping, and clause learning. However, CDCL with all the restarts and the deletions of learned clauses has little in common with a systematic search done by DPLL.

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.

SAT/UNSAT modes

Modern solvers use different modes for SAT/UNSAT problems, or alternate these modes during their run.

Preprocessing

We want to obtain an equisatisfiable problem that is “simpler”.

There are many techniques

- unit propagations,
- pure literal eliminations,
- subsumptions,. . .

Bounded Variable Elimination (BVE)

Loosely speaking, we eliminate a variable as in Davis–Putnam only when it does not increase the number of clauses (this can be relaxed over time). Combined with tautology elimination and subsumptions.

Inprocessing

Basically all state-of-the-art solvers interleave search with preprocessing.

Decision heuristics

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

Historically

Based on the number of occurrences of variables in unsatisfied clauses. Many variants, for example,

- considered only the shortest unsatisfied clauses,
- weight their occurrences (Jeroslow–Wang)

$$w(l) = \sum_{c \in \varphi \text{ such that } l \in c} 2^{-|c|}$$

We can compute it at the beginning or dynamically, however, that is expensive to do, cf. watched literals.

Why do we prefer short clauses?

Decision heuristics — modern

Focus heuristics

In CDCL we try to find small unsatisfiable subsets and hence we 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 to prioritize recently learned clauses.

Global heuristics

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. We also learn if possible. Good for random k-SAT.

Decision heuristics — value

We have selected a variable, but what value (positive/negative) should we try first? It is also called phase picking and it is especially important for satisfiable instances.

Historically

- based on the number of occurrences of variables in unsatisfied clauses; many variants
- a version of MiniSAT always sets literals to false

Phase saving

We do not concentrate directly on clauses, but instead we cache the behavior of variables during propagations and backtracking; we want to reach similar regions of the search space. Also very useful in combination with rapid restarts; we keep exploring the same region of the search space.

Parallel solving

SAT solving is difficult to parallelize. Moreover, our data structures, e.g. watched literals, make it even harder.

Cube and conquer (look-ahead and CDLC)

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

Good for hard combinatorial problem, e.g., the Boolean triples problem.

Portfolio approach

We run multiple solvers (usually the same one) with different settings on the same formula. We share clauses, which is especially important for unsatisfiable instances, among solvers. The main problems are how to diversify our portfolio and share clauses (which clauses, how many of them, 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 variables.

These methods are incomplete and it is an open problem how to use these techniques for showing unsatisfiability.

GSAT

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

Many extensions and variants, the most famous one is WalkSAT. You can try some of them in UBCSAT.

WalkSAT

We try to avoid local minima by combining the greedy moves of GSAT with random walk moves.

- Select randomly an unsatisfied clause c.
  - If by flipping a variable x occurring in c no satisfied clause becomes unsatisfied, then flip x. ("freebie" move)
  - Otherwise with a probability
    - p flip a random variable x in c ("random walk" move),
    - $(1-p)$ perform a GSAT step ("greedy" move) on variables from c; flip the best variable $x \in c$.

For details see Walksat Home Page. It is efficient on random k-SAT. Also historically good for planning and circuit design problems.

probSAT

A generalization of WalkSAT that calculates the probability distribution for the potential flip variables. It works also on some hard non-random problems. For details, see here.

CDCL or/and stochastic local search

On some instances stochastic local search methods work very well, you can try UBCSAT.

Moreover, it is even possible to combine CDCL with a local search and many modern solvers take advantage of that,

- for example, we can use a local search to produce a long partial assignment (trail) and then take advantage of this knowledge when we decide the values of variables (phase picking).

How to select a SAT solver?

Try different solvers (based on CDCL), 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. A rapidly developing toolkit is PySAT.

Check results of SAT Competition 2021 and from previous years for the state of the art.

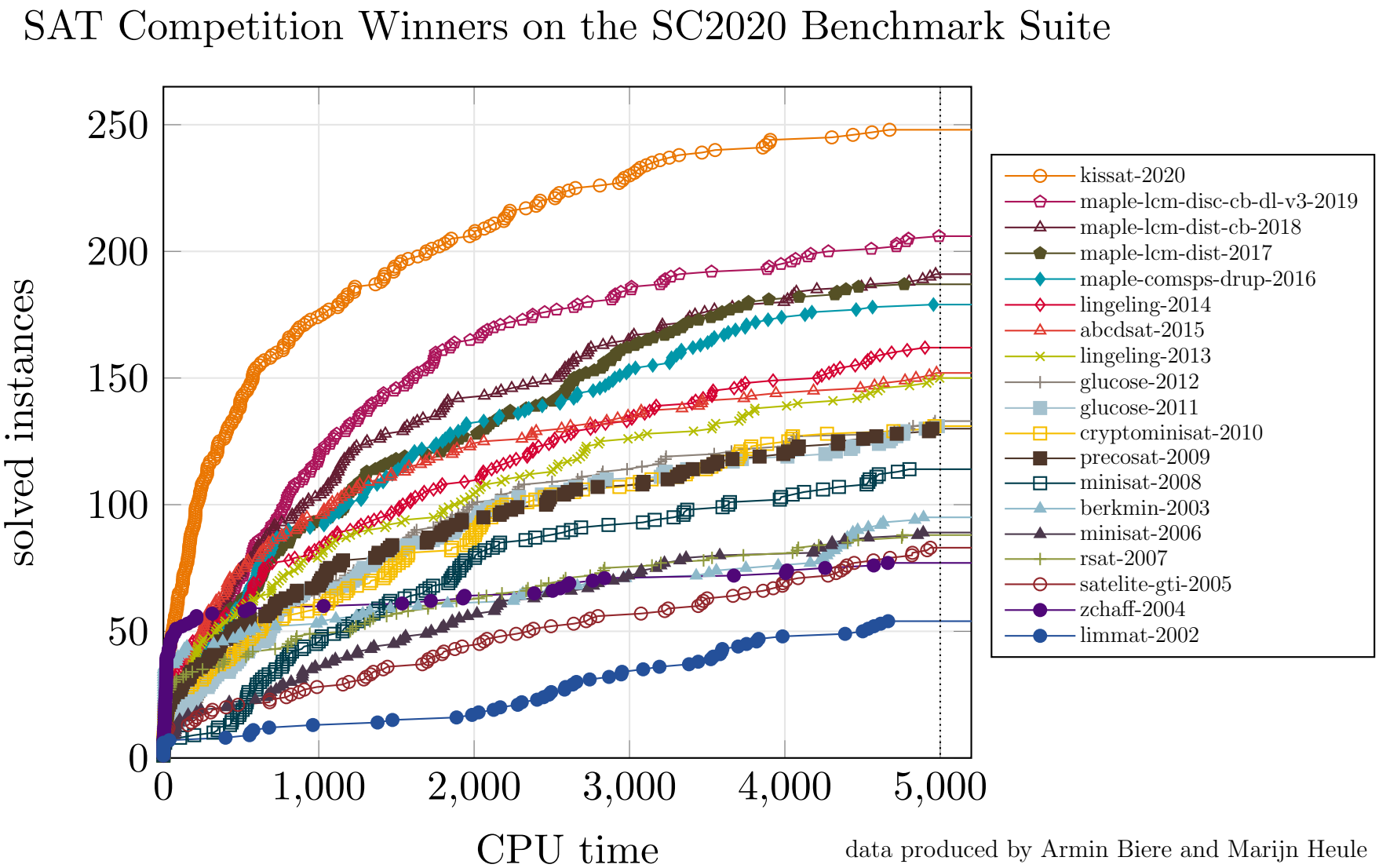
Some modern solvers

There are many modern solvers available, e.g.,

- solvers developed by Armin Biere, for example,
  - PicoSAT,
  - Lingeling,
  - CaDiCaL,
  - Kissat,
  - SATCH—a simple and clean code base for explaining and experimenting with SAT solvers,
- Glucose
  - based on MiniSat,
- MapleSAT
  - based on Glucose,
- CryptoMiniSat,
- Sat4j—a java library; good for Windows.

Progress over years. . .

source



data produced by Armin Biere and Marijn Heule

DIMACS format

The standard 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.

Input

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

encodes

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

DIMACS output format

There are three possible outcomes

- `s SATISFIABLE`
  - a satisfying assignment is returned: `v 1 -2 -3 4 0`
- `s UNSATISFIABLE`
  - a possible certificate in an external file
- `s UNKNOWN`

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, Marijn Heule, et al., eds. (2021). *Handbook of Satisfiability*. 2nd. Vol. 336. Frontiers in Artificial Intelligence and Applications. Washington: IOS Press. ISBN: 978-1-64368-161-0.


Biere, Armin, Marijn J. H. Heule, 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.


Davis, Martin, George Logemann, and Donald Loveland (July 1962). "A Machine Program for Theorem-Proving". In: *Communications of the ACM* 5.7, pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557.


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