Logical reasoning and programming

SAT solving (cont'd)—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:

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

We have seen various approaches how to solve this problem

- resolution,
- Davis–Putnam algorithm,
- DPLL algorithm.

DPLL algorithm

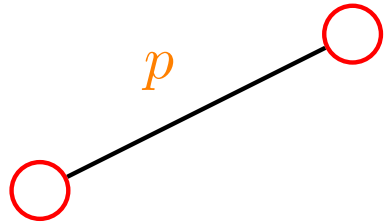
Require: A set of clauses φ
  function $\mathrm{DPLL}(\varphi)$
    while φ contains a unit clause $\{l\}$ **do** ▷ unit propagation
      delete clauses containing l from φ ▷ unit subsumption
      delete $\bar{l}$ from all clauses in φ ▷ unit resolution
    if $\square \in \varphi$ **then return** `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`

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).

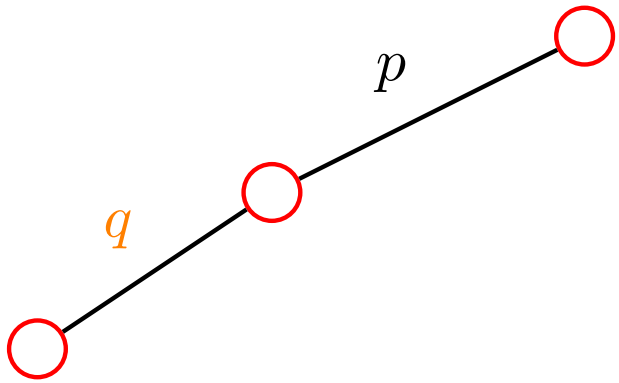
Example: DPLL (without pure literal elimination)



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

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).

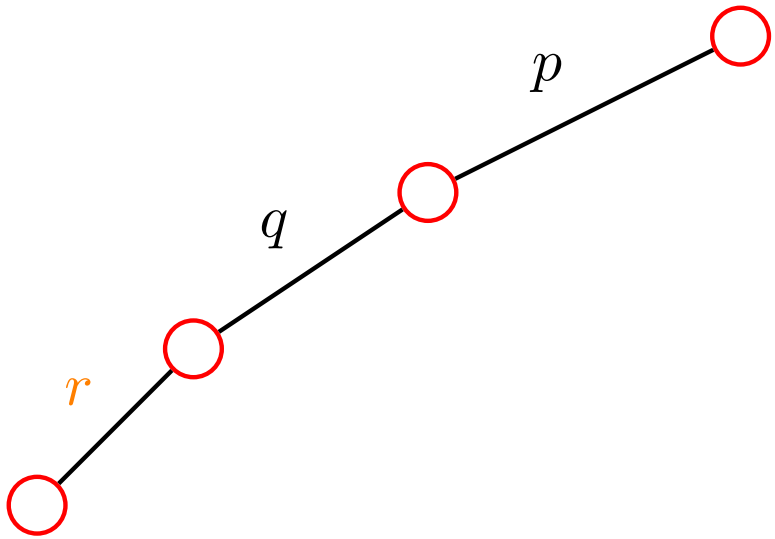
Example: DPLL (without pure literal elimination)



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

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).

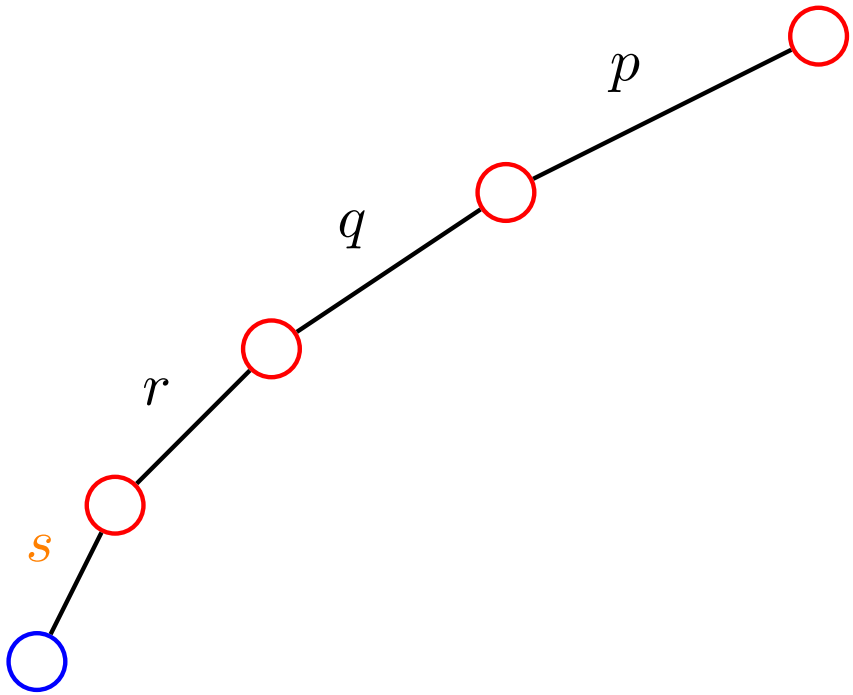
Example: DPLL (without pure literal elimination)



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

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).

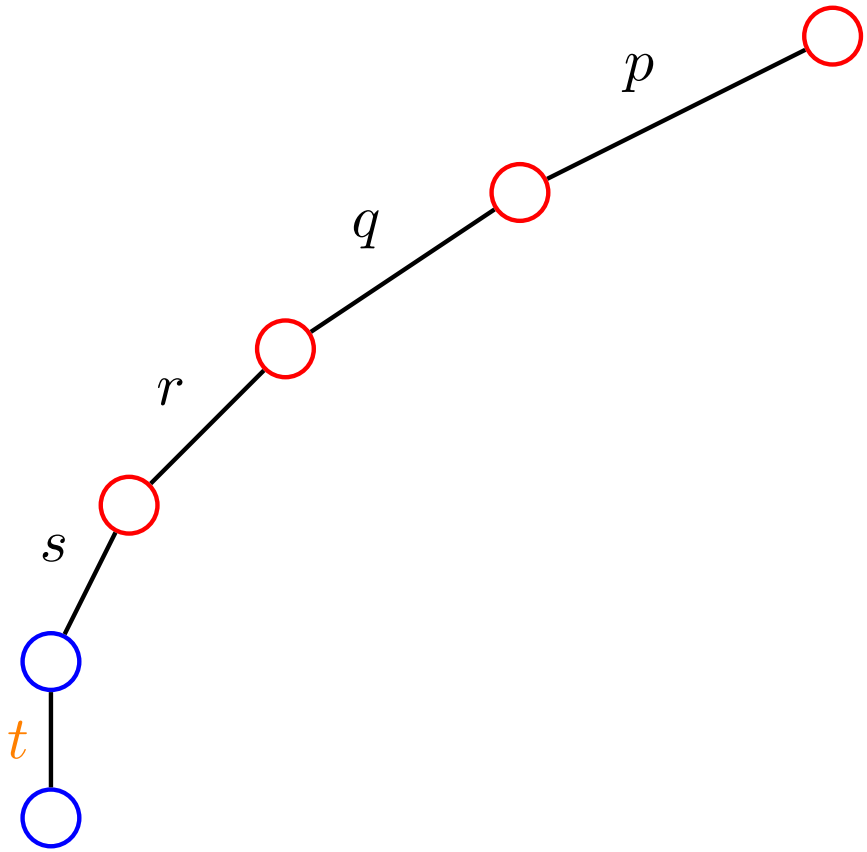
Example: DPLL (without pure literal elimination)



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

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).

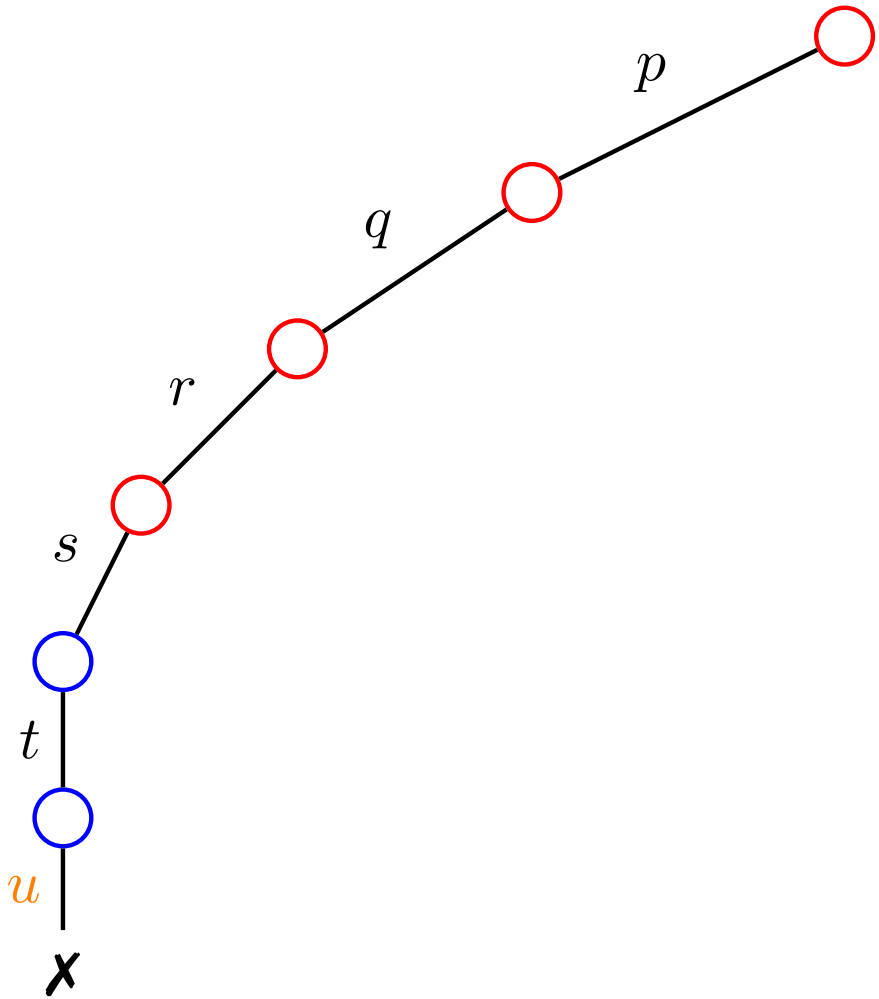
Example: DPLL (without pure literal elimination)



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

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).

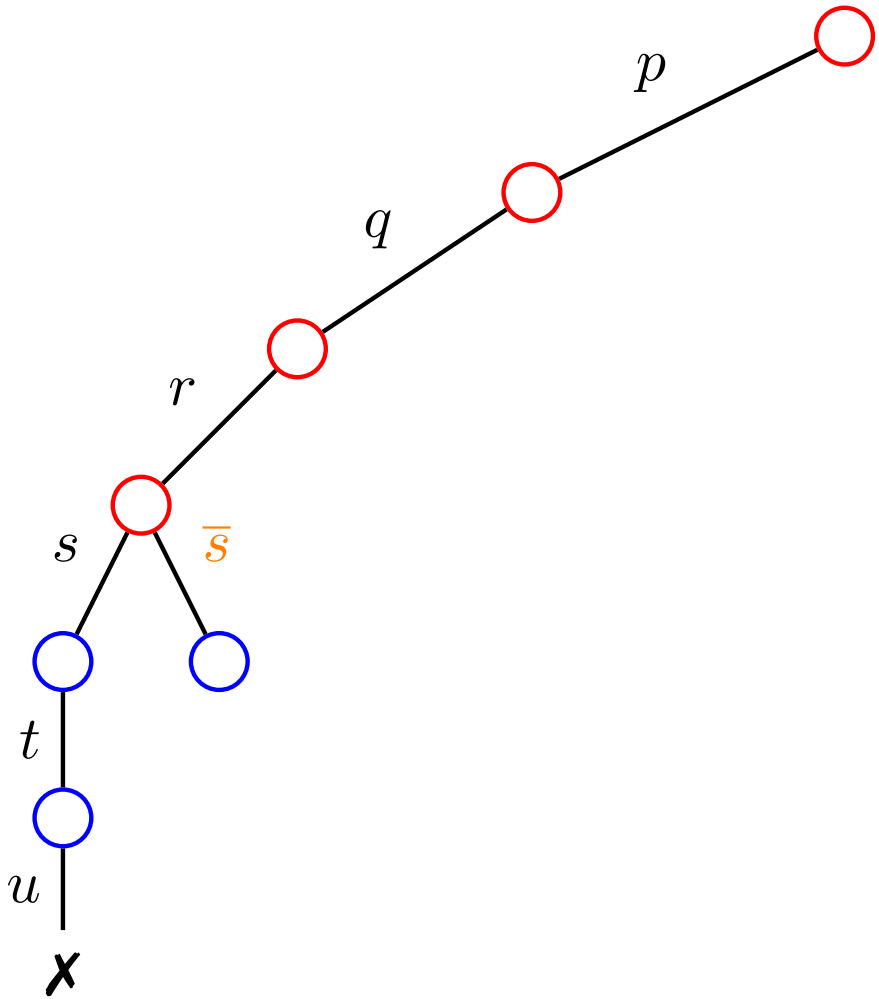
Example: DPLL (without pure literal elimination)



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

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).

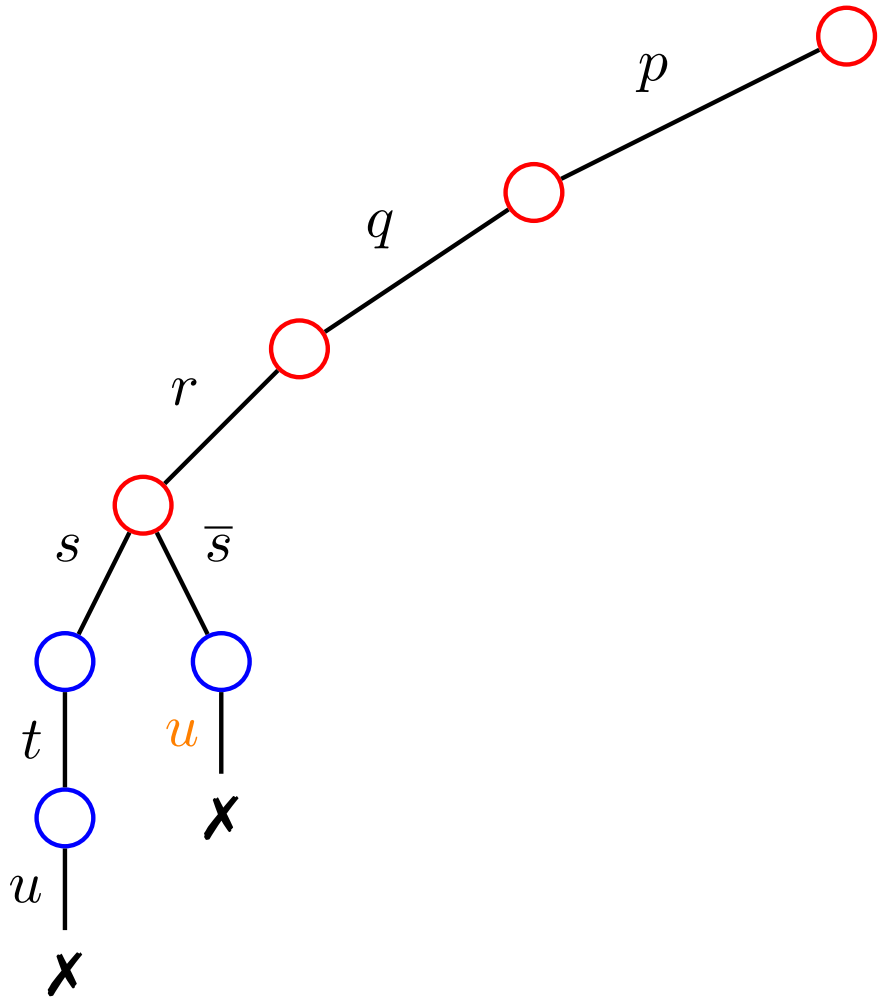
Example: DPLL (without pure literal elimination)



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

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).

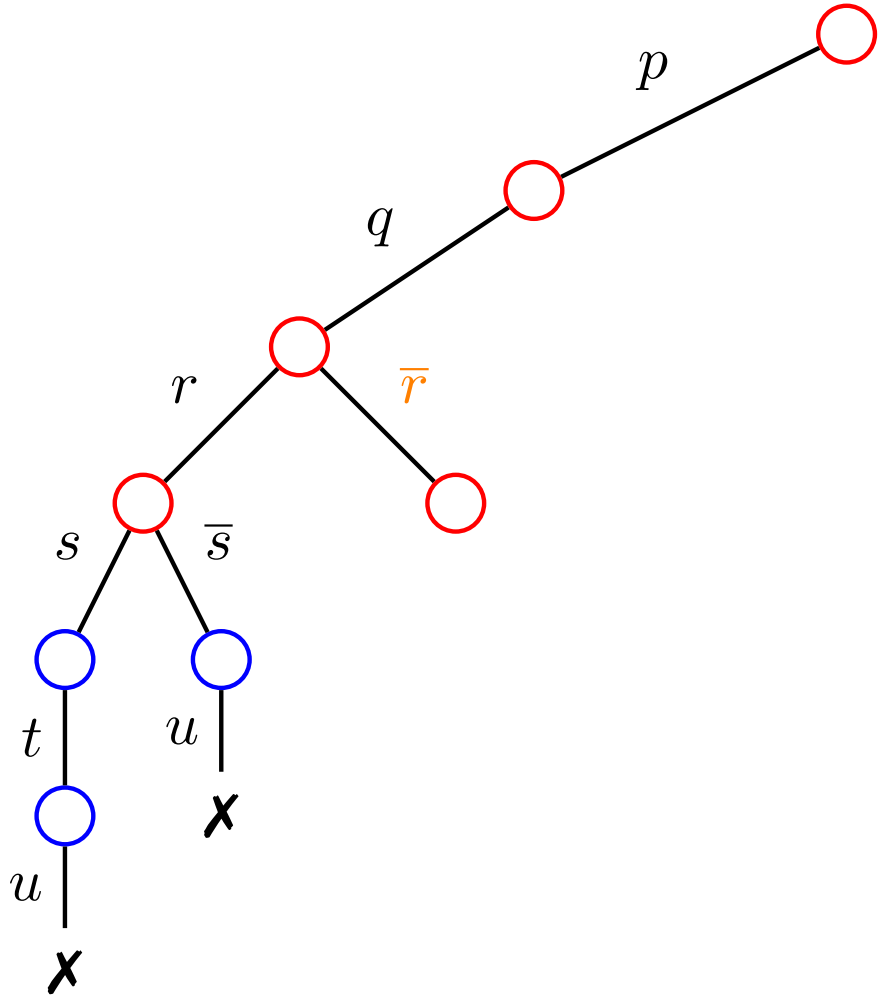
Example: DPLL (without pure literal elimination)



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

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).

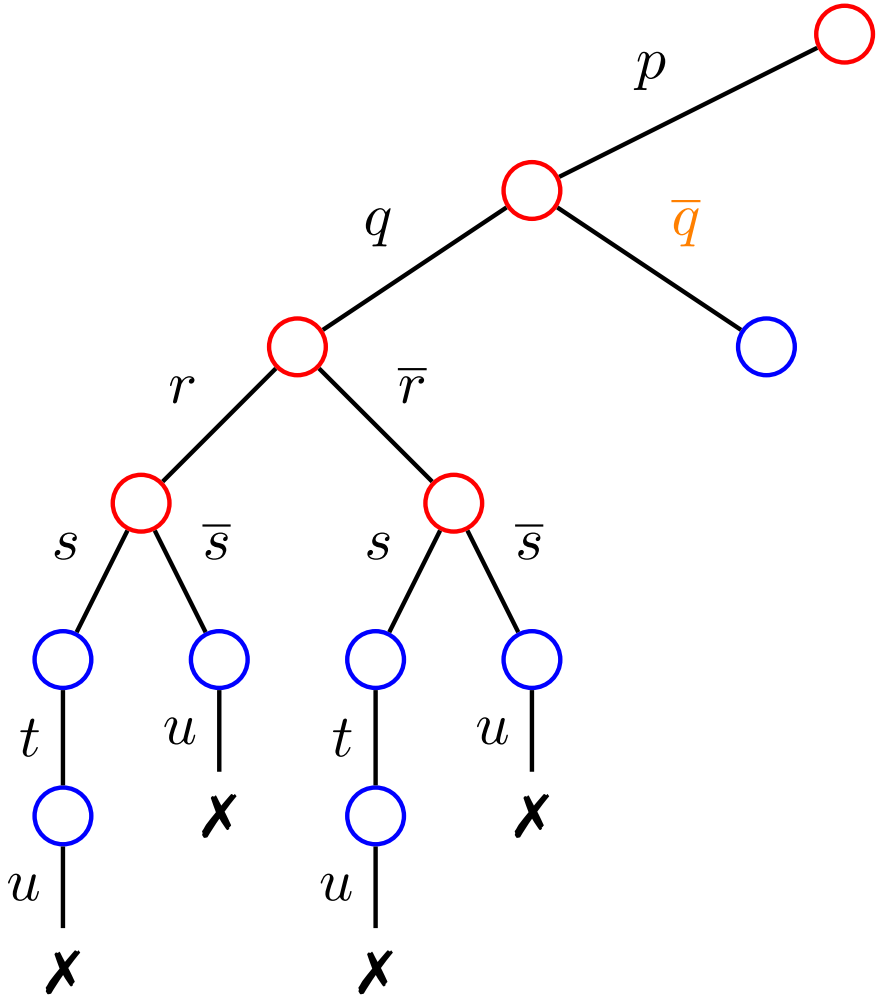
Example: DPLL (without pure literal elimination)



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

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).

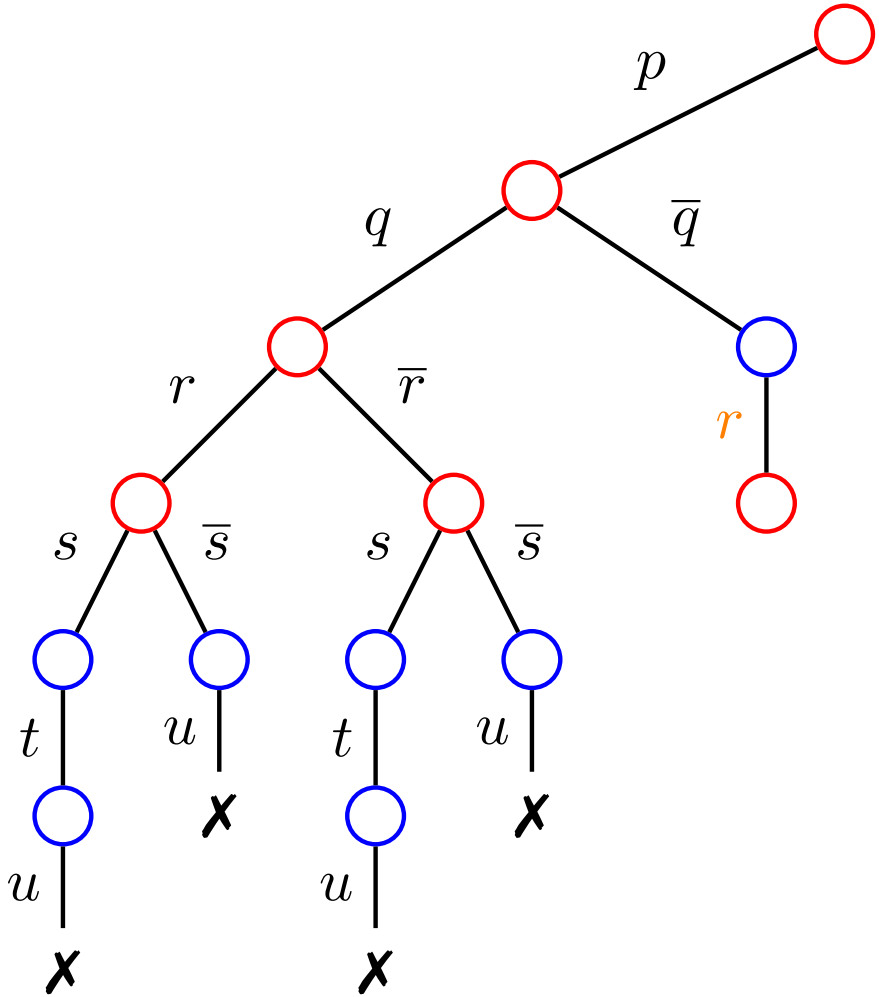
Example: DPLL (without pure literal elimination)



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

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).

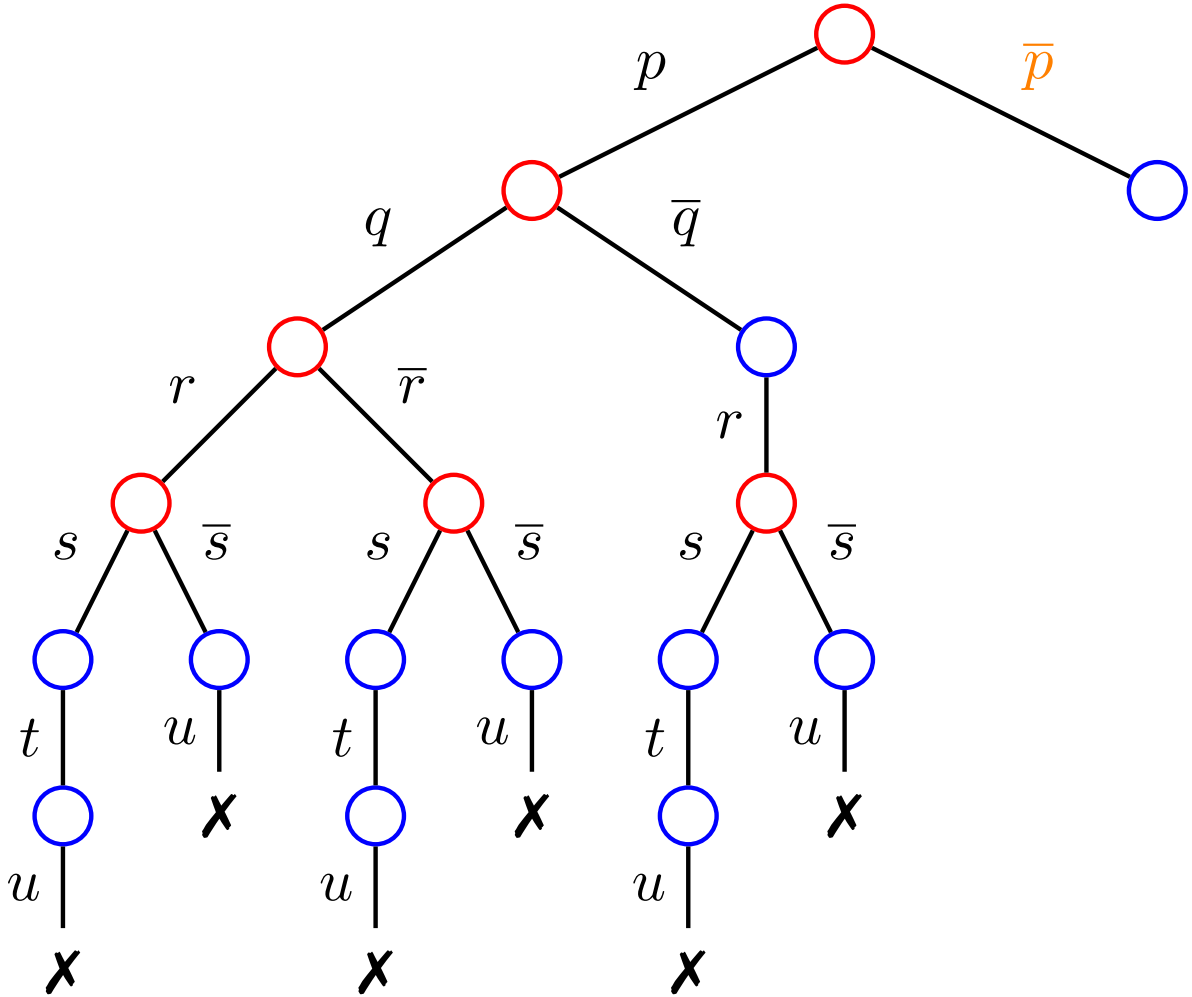
Example: DPLL (without pure literal elimination)



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

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).

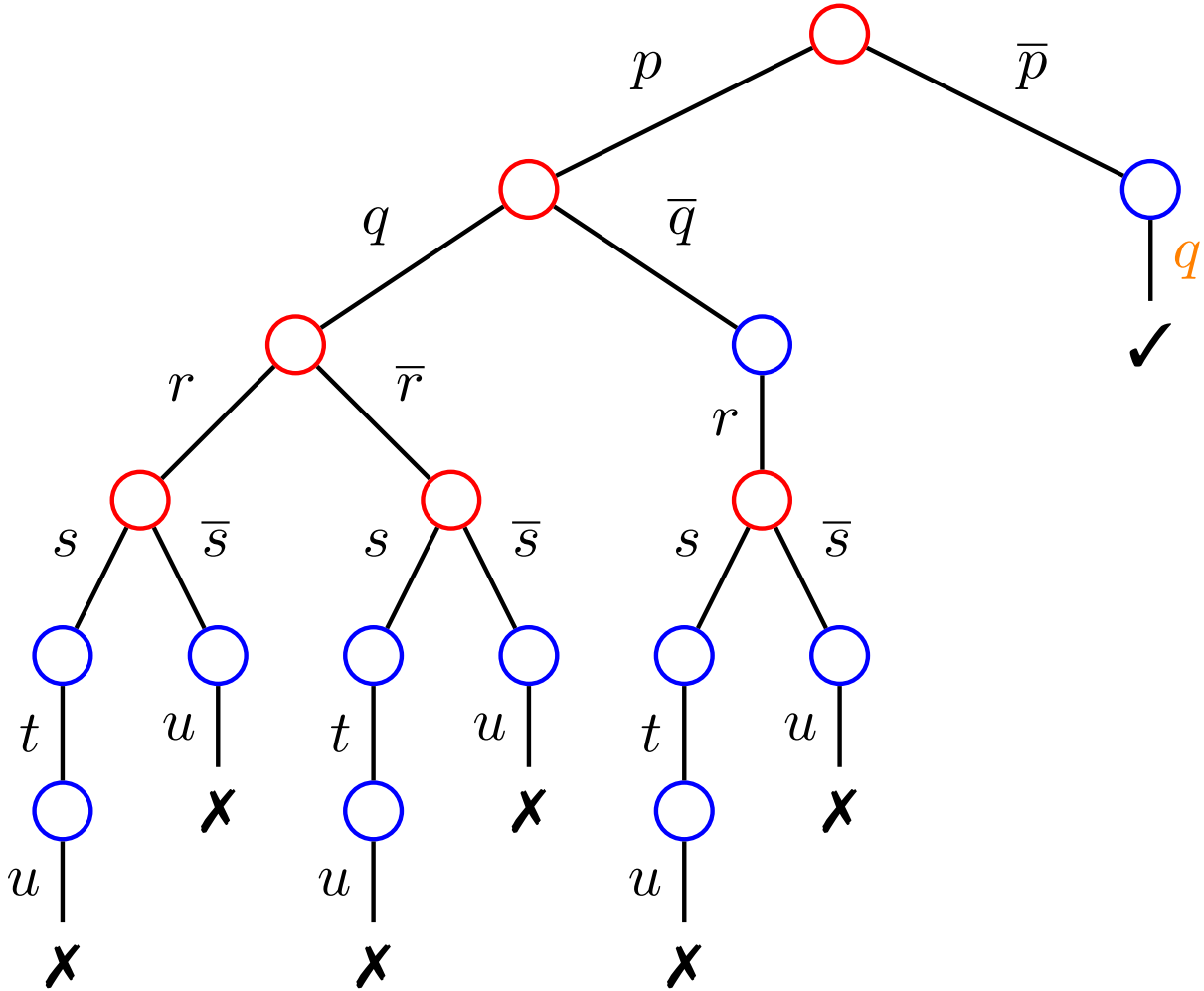
Example: DPLL (without pure literal elimination)



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

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).

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.

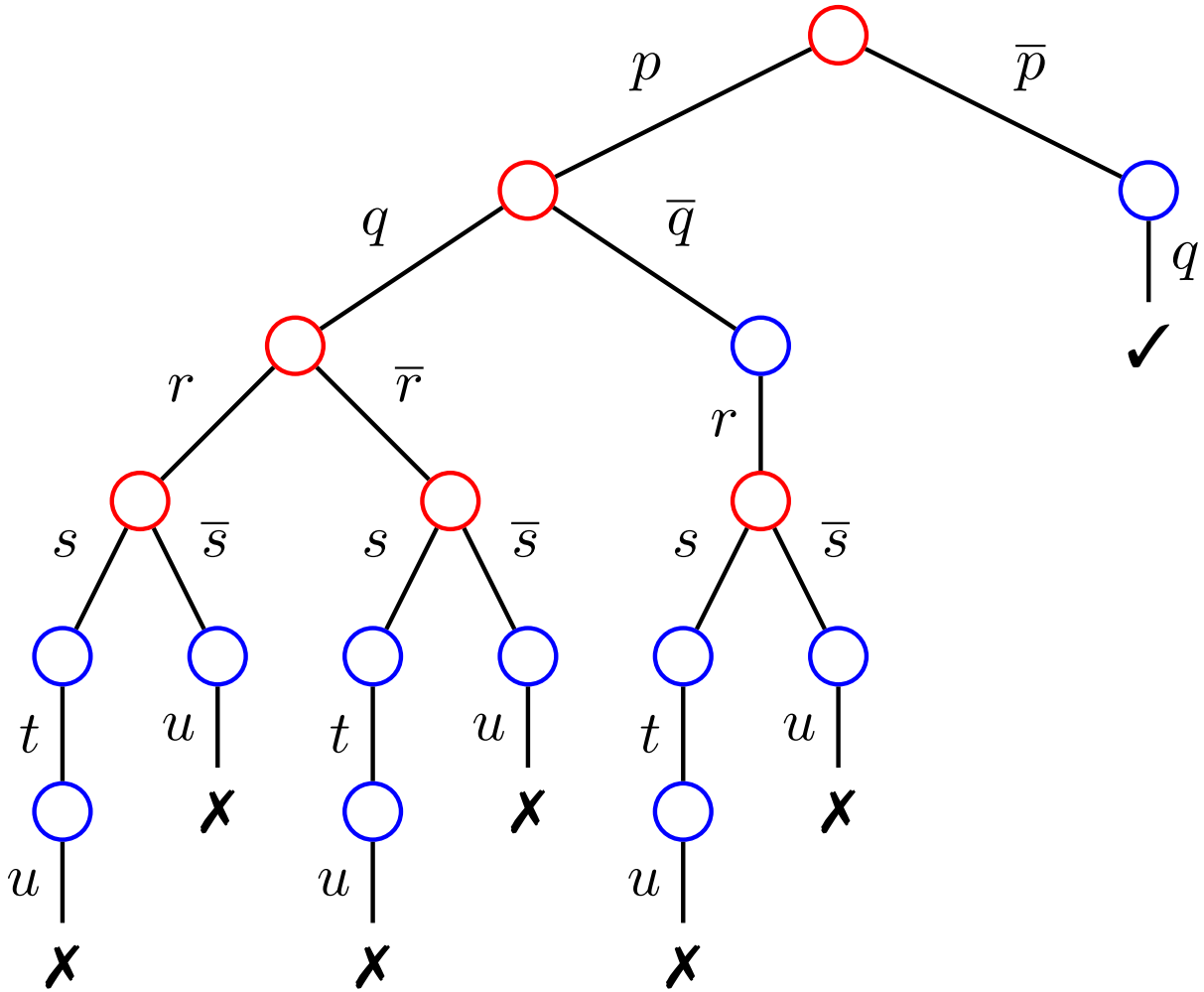
Watched literals

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?

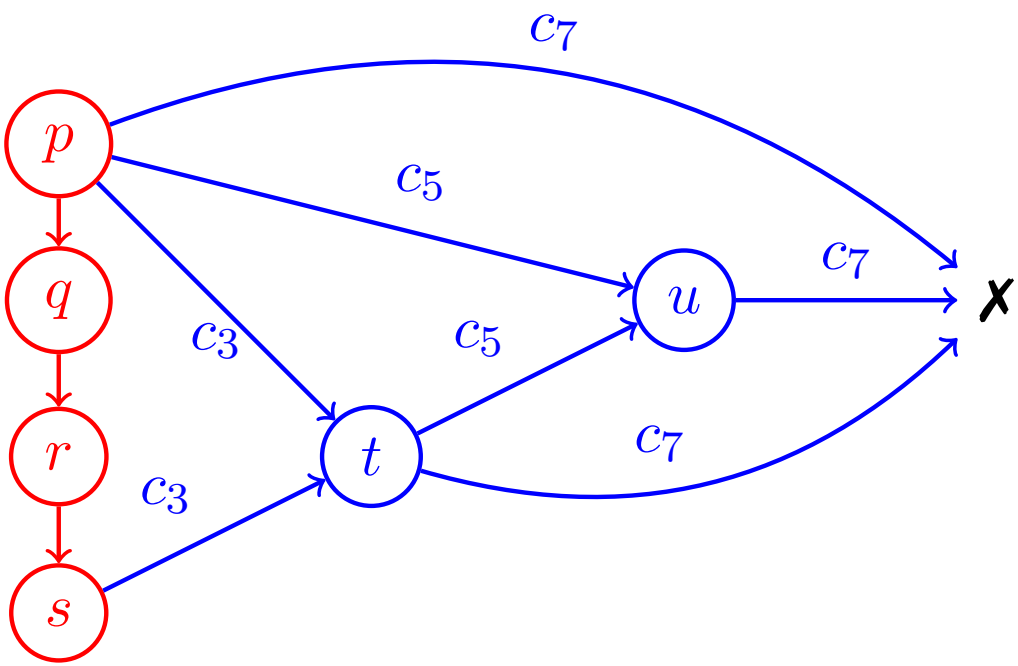


$$c_1 = \{p, q\}$$
$$c_2 = \{q, r\}$$
$$c_3 = \{\overline{p}, \overline{s}, t\}$$
$$c_4 = \{\overline{p}, s, u\}$$
$$c_5 = \{\overline{p}, \overline{t}, u\}$$
$$c_6 = \{\overline{p}, s, \overline{u}\}$$
$$c_7 = \{\overline{p}, \overline{t}, \overline{u}\}$$

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.

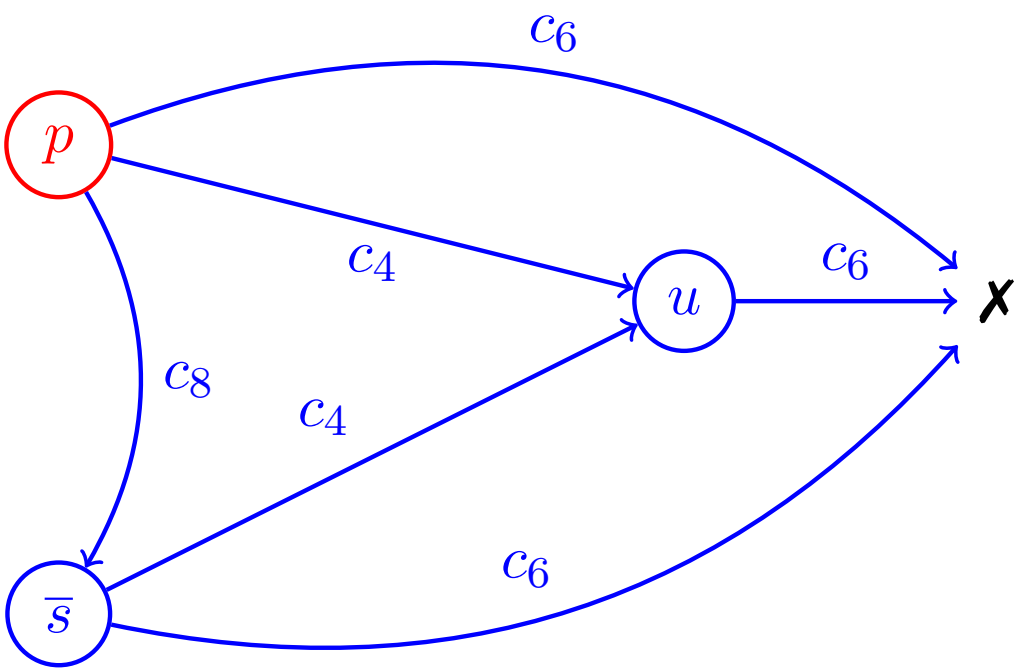


$$
\begin{aligned}
c_1 &= \{p, q\} \\
c_2 &= \{q, r\} \\
c_3 &= \{\overline{p}, \overline{s}, t\} \\
c_4 &= \{\overline{p}, s, u\} \\
c_5 &= \{\overline{p}, \overline{t}, u\} \\
c_6 &= \{\overline{p}, s, \overline{u}\} \\
c_7 &= \{\overline{p}, \overline{t}, \overline{u}\}
\end{aligned}
$$

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

We can also analyze the second conflict now.

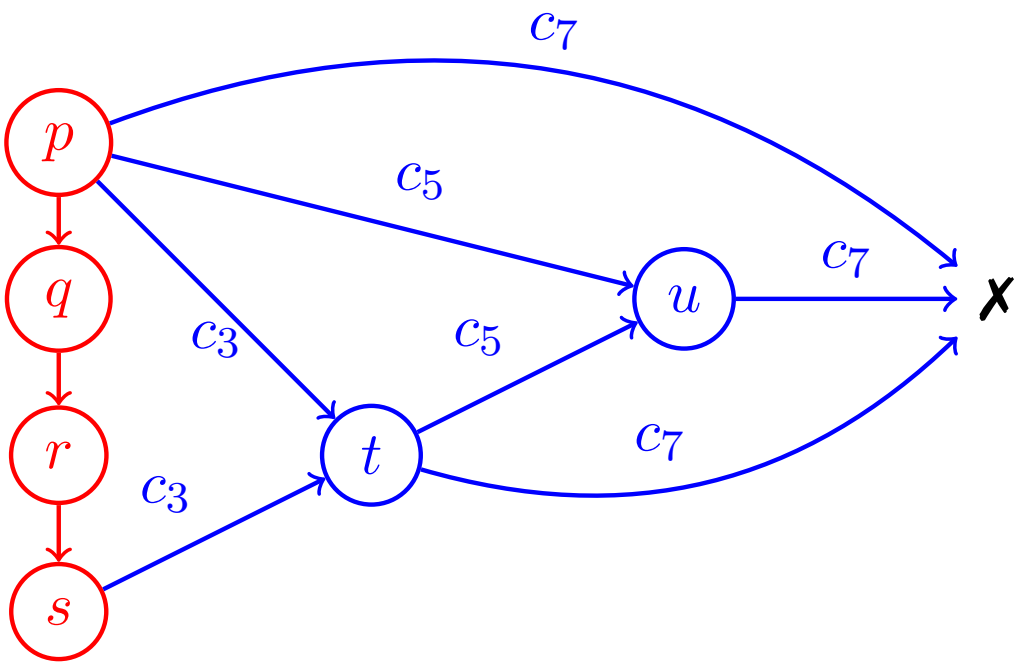


$$
\begin{aligned}
c_1 &= \{p, q\} \\
c_2 &= \{q, r\} \\
c_3 &= \{\overline{p}, \overline{s}, t\} \\
c_4 &= \{\overline{p}, s, u\} \\
c_5 &= \{\overline{p}, \overline{t}, u\} \\
c_6 &= \{\overline{p}, s, \overline{u}\} \\
c_7 &= \{\overline{p}, \overline{t}, \overline{u}\} \\
c_8 &= \{\overline{p}, \overline{s}\}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

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.

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.

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.

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)

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

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 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.

Planning

In classical planning we want to produce a sequence of actions that translate an initial state into a goal state.

It is well-known that the plan existence problem is PSPACE-complete. Hence it is not (assuming $\mathrm{NP} \neq \mathrm{PSPACE}$) easily solvable using SAT. However, if we consider only plans up to some length, then it is solvable by SAT, because the lengths of plans are usually polynomially bounded.

Planning as a SAT problem

We encode as a CNF formula "there exists a plan of length k", denoted φ_k, and search iterativelly.

- If $\varphi_k \in \mathtt{SAT}$, then we extract a plan from a satisfying assignment.
- If $\varphi_k \notin \mathtt{SAT}$, then we continue with φ_{k+1}.

Classical planning (recap)

We have a set of state variables $X = \{x_1, \ldots, x_n\}$ that are assigned values from a finite set. A state s is such an assignment for X, we write $\{x_1 = v_1, \ldots, x_n = v_n\}$. A set of conditions is a subset of a state.

We have

- an initial state,
- a set of goal conditions—a goal state is such a state that satisfies all the goal conditions.

Moreover, we have a set of actions A where every $a \in A$ has preconditions and effects which are both sets of conditions.

Example

We have a chessboard and $X = \{x_1, \ldots, x_{64}\}$ are the squares of the chessboard. An assignment says how pawns, pieces, and the empty square are distributed. A goal condition can be that the white king and queen are at x_{10} and x_{28}, respectively.

There exists a plan of length k in SAT

We introduce propositional variables for

- actions—meaning the action a is used in the step t,
- assignments—meaning $x = v$ holds before an action in the step t is applied,

for every $a \in A$, $x \in X$, possible value v of x, and step $t \leq k$.

Then we describe all the required properties of a valid plan by a conjunction of clauses:

- the initial state,
- the goal conditions are satisfied after k steps,
- state variables are assigned exactly one value,
- exactly one action is performed in one step,
- the values of state variables change only by actions,
- an applied action must satisfy preconditions and effects.

Planning using SAT

We can do various improvements, e.g.,

- perform more actions in a step if they are non-conflicting,
- introduce variables for transitions instead of assignments,
- symmetry breaking.

Incremental SAT solving

Instead of solving a new problem for every k, we can observe that many parts remain the same—we solve a sequence of similar SAT problems. We want to add and remove clauses, but keep learned clauses and variable scores.

Note that in our problem we only add clauses and change the goal conditions, which are described by unit clauses, when we go from φ_k to φ_{k+1}.

Assumptions

Clearly, adding clauses is possible in CDCL, but removing clauses can lead to various problems. However, we have

- a formula φ and
- assumptions $l_1, \ldots, l_n$, where l_i are literals.

The question is whether $\varphi \wedge l_1 \wedge \cdots \wedge l_n \in \texttt{SAT}$. It is incremental, because we can change assumptions and add new clauses.

We can select all the assumptions as decision variables and continue as always. Hence we can keep all learned clauses from CDCL!

Bounded model checking

It is very similar to planning. We want to verify a property of an automaton with transition states, an initial state, and a given property P that has to be valid at each step.

Bounded model checking as a SAT problem

We bound the number of steps to k and try to reach in k steps a state where P fails. Hence φ_k means "there is a state reachable in k steps where P fails".

- If $\varphi_k \in \texttt{SAT}$, then we extract a bug from a satisfying assignment.
- If $\varphi_k \notin \texttt{SAT}$, then we continue with φ_{k+1}.

How to encode typical constraints

We want to express

$$p_1 + p_2 + \cdots + p_n \bowtie k,$$

where $\bowtie \in \{\leq, \geq, =\}$, k is a positive integer, and $\sum_{1 \leq i \leq n} p_i$ is equal to the number of true p_is.

- $=$ is expressed as both $\leq$ and $\geq$,
- ≥ 1 is $\{p_1, \ldots, p_n\}$,
- ≤ 1 is
  - pairwise—$\mathcal{O}(n^2)$ clauses by $\{\{\overline{p_i}, \overline{p_j}\} : 1 \leq i < j \leq n\}$,
  - sequential counter—$\mathcal{O}(n)$ clauses and $\mathcal{O}(n)$ new variables,
  - bitwise encoding—$\mathcal{O}(n \log n)$ clauses and $\mathcal{O}(\log n)$ new variables,
- $\geq k$ is no more than $n - k$ literals can be false,
- $\leq k$ use generalized pairwise, sequential counters, BDDs, sorting networks, (pairwise) cardinality networks, ...

Or use a pseudo-Boolean (PB) solver for $\sum a_i p_i \bowtie k$.

Consistency and arc-consistency

A very nice property of encodings, e.g., for an encoding of constraints. We say that an encoding is

consistent if any partial assignment that cannot be extended to a satisfying assignment (is inconsistent) leads to a conflict by unit propagation,

arc-consistent if consistent and unit propagations eliminate values that are inconsistent.

Example

For ≤ 1 we have

consistency if two variables are true, then unit propagation produces a conflict,

arc-consistency if a variable is true, then unit propagation assigns false to all other variables. (+consistency)

Finite-domain encoding

We encode that a variable x takes one of the values $\{1, \ldots, n\}$.

One-hot encoding

- we use x_i for x takes value i (n variables),
- we need x has exactly one value,
- easy to use constraints and other rules

Unary encoding (order encoding)

- $\underbrace{1 \ldots 1}_{i-1} \underbrace{0 \ldots 0}_{n-i}$ for x takes value i ($n-1$ variables),
- we need $\{\overline{x_{j+1}}, x_j\}$ for $1 \leq j < n-1$

Binary encoding

- we encode i as a binary number ($\lceil \log n \rceil$ variables),
- if $n \neq 2^k$ some values are not valid,
- using constraints and other rules can be non-trivial

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 works effectively on random k-SAT. Also historically good for planning and circuit design problems.

MaxSAT

There are various variants of SAT. For example, many problems in computer science are expressible as the maximum satisfiability problem—what is the maximum number of clauses that can be satisfied simultaneously.

We usually have (weighted) partial MaxSAT with two types of clauses:

- hard—must be satisfied,
- soft—desirable to be satisfies (possibly with weights)

and we want to maximize the sum of the weights of satisfied soft clauses.

You can check benchmark results at MaxSAT Evaluation 2019. For example RC2 (Python, winner), MaxHS (also MIP solvers), Open-WBO.

Unsatisfiable cores

Let φ and ψ be unsatisfiable formulae in CNF such that $\varphi \subseteq \psi$. We say that

- φ is an *unsatisfiable core* of ψ,
- φ is a *minimal unsatisfiable core* of ψ, if every proper subset of φ is satisfiable.

A very important (and hard) practical problem is to extract minimal unsatisfiable cores. For example, in MaxSAT and formal verification.

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. Also popular CryptoMiniSAT.

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 (includes CaDiCaL and Glucose).

Check results of SAT Competition 2020 and from previous years for the state of the art.

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).

SAT solving summary

SAT solvers are very powerful, among other things, thanks to

- small representations in CNFs,
- preprocessing, (inprocessing),
  - subsumption,
  - variable elimination, (variable addition),
  - symmetry breaking,
- unit propagations,
  - good data structures for backtracking,
- clause learning and back-jumping,
  - restarts,
  - deletion of learned clauses,
- fast decision heuristics,
- and much much more techniques we have not mentioned.

We do clever tricks, but first and foremost they have to be fast!

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.