

Solving hard problems in practice

The miracle of SAT-solvers

PART 1

SAT everywhere

Search problems via Boolean constraints.

Scheduling

A hospital assigns Alice, Bob, and Chloé to two shifts: Day and Night.

Subject to conditions:

1. Every shift has at least one nurse.
2. Each nurse works at most one shift.
3. Alice cannot work at night.
4. Bob and Chloé cannot work together.

Let $x_{p,s}$ mean that nurse p works shift s , where $p \in \{A, B, C\}$ and $s \in \{D, N\}$.

Scheduling

The clauses that we get:

$x_{A,D} \vee x_{B,D} \vee x_{C,D}$	(day covered)
$x_{A,N} \vee x_{B,N} \vee x_{C,N}$	(night covered)
$\neg x_{A,D} \vee \neg x_{A,N}$	(Alice: at most one)
$\neg x_{B,D} \vee \neg x_{B,N}$	(Bob: at most one)
$\neg x_{C,D} \vee \neg x_{C,N}$	(Chloé: at most one)
$\neg x_{A,N}$	(Alice cannot work nights)
$\neg x_{B,D} \vee \neg x_{C,D}$	(not together during the day)
$\neg x_{B,N} \vee \neg x_{C,N}$	(not together at night)

For example, $x_{A,D} = T$ and $x_{B,N} = T$, the rest false, is a solution.

Package selection

A user wants to install a code editor.

1. The editor needs a core library and at least one of two codecs.
2. The fast codec also needs platform support.
3. The two codecs cannot be installed together.

Let each package have a Boolean variable meaning it is installed:

e : editor c : core library f : fast codec
 s : safe codec p : platform support.

Package selection

The requirements become these CNF clauses:

- e (the user requests the editor)
- $\neg e \vee c$ (the editor requires the core library)
- $\neg e \vee f \vee s$ (the editor needs at least one codec)
- $\neg f \vee p$ (the fast codec requires platform support)
- $\neg f \vee \neg s$ (the codecs conflict)

For example, $e = T$, $c = T$, $s = T$, and $f = p = F$ satisfies all the clauses.

And also

- ▶ hardware and software verification
- ▶ cryptanalysis
- ▶ bioinformatics
- ▶ program synthesis
- ▶ combinatorial design
- ▶ explainable AI
- ▶ graph-related problems
- ...

Clique of size k

We want a list of k vertices:

$$x_{i,v} = T \iff v \text{ occupies position } i, \quad i \in \{1, \dots, k\}, \quad v \in V.$$

For every position i , require exactly one vertex:

$$\bigvee_{v \in V} x_{i,v}, \quad \neg x_{i,u} \vee \neg x_{i,v} \quad (u \neq v).$$

The table illustrates $k = 3$ (one true entry per row).

	a	b	c	d
position 1	T	F	F	F
position 2	F	T	F	F
position 3	F	F	T	F

Clique of size k

For every pair of positions $i < j$:

- ▶ Do not use a vertex twice: $\neg x_{i,v} \vee \neg x_{j,v}$ for each $v \in V$.
- ▶ Do not choose a non-edge: $\neg x_{i,u} \vee \neg x_{j,v}$ for each $\{u, v\} \notin E$.

With $n = |V|$ and $k \leq n$, the encoding has $O(k^2 n^2)$ clauses.

Proper 3-colouring

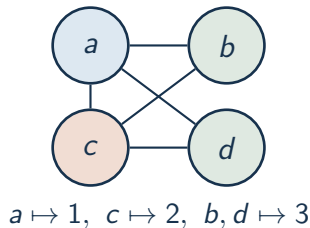
$x_{v,c} = T$ means vertex v has colour $c \in \{1, 2, 3\}$.

Exactly one colour per vertex:

$$\neg x_{v,c_1} \vee \neg x_{v,c_2}, \quad \text{for all } v \text{ and } c_1 \neq c_2$$

For every edge $\{u, v\}$ and colour c :

$$\neg x_{u,c} \vee \neg x_{v,c}.$$



Vertex cover of size at most k

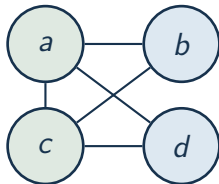
Select vertex v when $y_v = T$. For each edge:

$$y_u \vee y_v.$$

To select at most k vertices, when $k = 2$ forbid each triple:

$$\neg y_u \vee \neg y_v \vee \neg y_w \quad (u, v, w \text{ distinct}).$$

This gives $\binom{n}{k}$ clauses though. For general k , a more tricky encoding is needed.



K_4 without bd
Example cover: $\{a, c\}$

Counting the nodes efficiently

Given choices $y_1, \dots, y_n$, define helper Boolean flags $s_{i,j}$ by:

$$s_{i,j} = T \iff \text{at least } j \text{ of } y_1, \dots, y_i \text{ are true,} \quad 1 \leq i \leq n, \quad 1 \leq j \leq k+1.$$

Set $s_{i,0} = T$ and $s_{0,j} = F$. For each new choice y_i , enforce:

$$s_{i,j} \leftrightarrow s_{i-1,j} \vee (y_i \wedge s_{i-1,j-1}).$$

Either the earlier choices already reached j , or y_i adds the j th.

At-most- k condition

For at most k , forbid reaching $k+1$:

$$\neg s_{n,k+1}.$$

Size

There are $n(k+1)$ helper flags. Each update takes at most four CNF clauses, so the encoding has $O(n(k+1))$ size.

SAT-solver: an algorithm for CNF-SAT

$$(\alpha_{11} \vee \cdots \vee \alpha_{1n_1}) \wedge \cdots \wedge (\alpha_{m1} \vee \cdots \vee \alpha_{mn_m})$$

SAT An assignment satisfying every clause.

UNSAT No satisfying assignment exists.

A time limit can leave the question **unanswered**.

Build and solve a CNF in Python

```
from pysat.formula import CNF
from pysat.solvers import Glucose3

# 1..6: A_D, B_D, C_D, A_N, B_N, C_N
cnf = CNF(from_clauses=[
    [1, 2, 3], [4, 5, 6],          # both shifts covered
    [-1, -4], [-2, -5], [-3, -6], # one shift per nurse
    [-4], [-2, -3], [-5, -6]      # restrictions
])

with Glucose3(bootstrap_with=cnf.clauses) as solver:
    print(solver.solve(), solver.get_model())
```

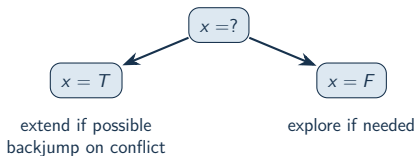
PART 2

How does it work?

Try, propagate, backjump, repeat.

Try

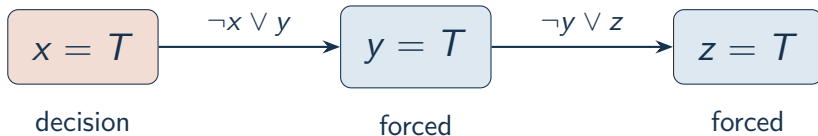
Start with all variables unassigned; extend the assignment step by step.



Propagate

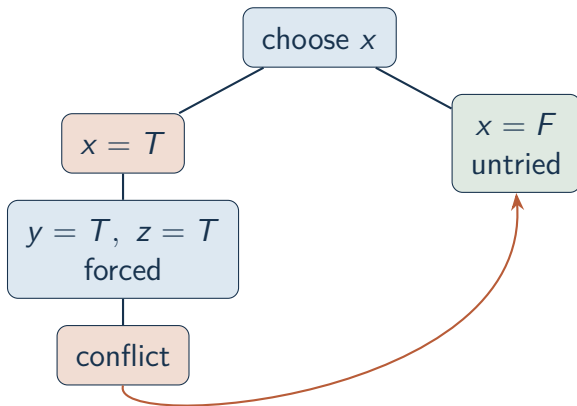
If a clause has one unassigned literal and all others are false, that last literal must be true.

$$(\neg x \vee y) \wedge (\neg y \vee z).$$



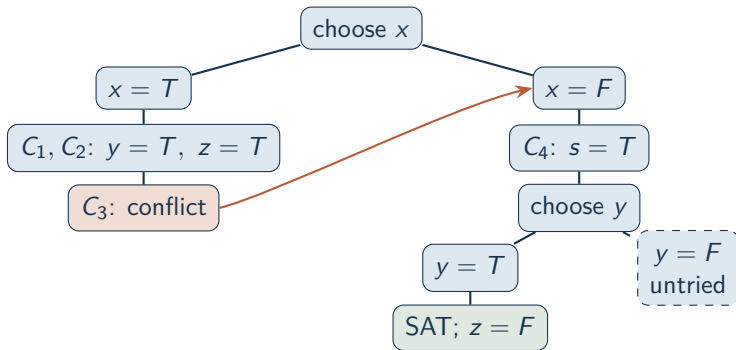
Repeat until nothing more is forced, or a conflict appears.

Backjump to an untried branch



DPLL

$$\varphi = \underbrace{(\neg x \vee y)}_{C_1} \wedge \underbrace{(\neg x \vee z)}_{C_2} \wedge \underbrace{(\neg y \vee \neg z)}_{C_3} \wedge \underbrace{(x \vee s)}_{C_4}.$$



CDCL: Conflict-Driven Clause Learning

Consider the clauses

$$C_1 = (\neg x \vee \neg y \vee p), \quad C_2 = (\neg x \vee \neg y \vee q), \quad C_3 = (\neg p \vee \neg q).$$

DPLL tries $x = T$ at level 1, then $y = T$ at level 2.

Propagation through C_1 and C_2 forces $p = q = T$.

This will cause a conflict on C_3 at the next step.

CDCL: Conflict-Driven Clause Learning

Consider the clauses

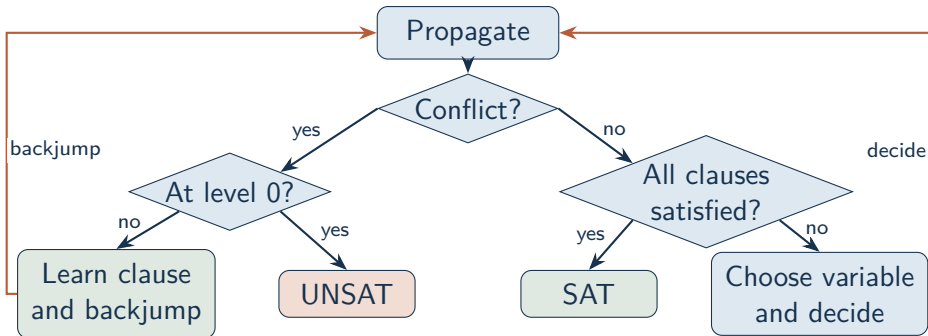
$$C_1 = (\neg x \vee \neg y \vee p), \quad C_2 = (\neg x \vee \neg y \vee q), \quad C_3 = (\neg p \vee \neg q).$$

Analyse before backjumping: setting $x = T$ and $y = T$ led to a conflict.

Learn a new clause: $C_4 = (\neg x \vee \neg y)$.

From now on, work with $C_1 \wedge C_2 \wedge C_3 \wedge C_4$

The CDCL search loop



Backjump: undo decisions conflicting with the learned clause.

Modern SAT-solvers: CDCL + heuristics

Branching: Favour variables active in recent conflicts.

Simplification: Preprocessing and inprocessing use logical rules to shrink the formula, e.g. $(\varphi \vee \psi) \wedge \varphi \mapsto \varphi$.

Fast propagation: Track two literals in each clause. Recheck it only when one becomes false, instead of scanning every clause after each assignment.

Clause quality: Prefer clauses whose literals come from few decision levels: they often link related deductions.

Clause deletion: Learned clauses accumulate. Drop long or rarely useful ones; keep the ones that still help.

Restarts: Clear assignments periodically, but keep learned clauses and scores.

Focus on conflicts, remember useful causes, restart with that memory, and discard what stops helping.

... + machine learning

Promising moves. Predict which variable to try next, especially early, when there is little search history.

Clause quality. Estimate which facts learned during search may help again.

Match the problem. Recognise its shape and choose a suitable solving strategy or simplification.

Change direction. Suggest when to restart.

There is a cost: predictions may require more time than they save. Practical solvers use learning occasionally, often early, then return to carefully designed built-in rules.

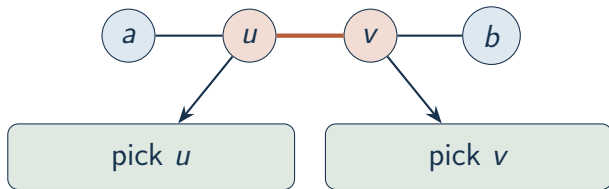
PART 3

Direct graph algorithms

Try, propagate, and exploit graph structure.

Vertex cover: branch-and-bound

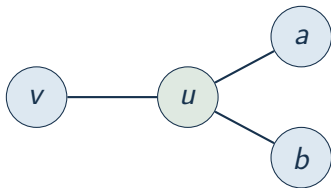
Can we cover every edge using at most k vertices? For an uncovered edge $\{u, v\}$, choose u or v .



Each pick uses one unit of k ; remove the edges it covers.

Vertex cover: a safe shortcut

A leaf v has just one edge: $\{u, v\}$.



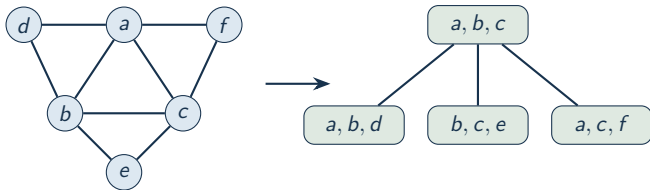
If a cover picks v but not u , swap v for u .

The number picked stays the same; u may cover more edges.

So we can safely pick u : spend 1 from k and remove its covered edges. Ignore isolated vertices; they have no remaining edge to cover.

Graph decompositions

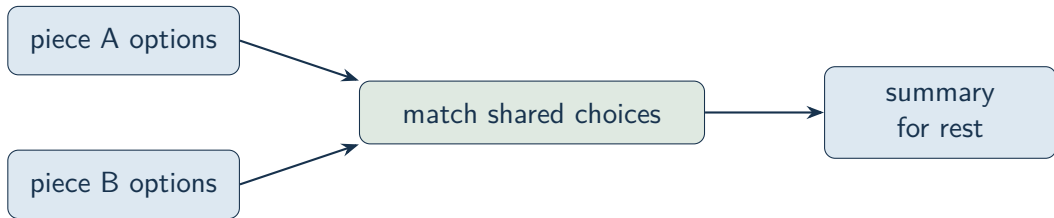
Split vertices into small, overlapping groups called **bags**. Keep every edge inside a bag; bags containing a vertex stay connected.



Every finite graph fits in one bag. The question is: can we choose bags in a way that the graph starts to look like a tree? Like a path?

Small bags mean fewer cases to consider

Each bag reports its possible choices at the shared vertices.



Few boundary vertices mean few combinations.

With a fixed small bag size, many hard problems can be solved in polynomial time.