

Solving problems with CDCL

A user's guide to conflict-driven clause-learning SAT

Copyright © 2026 by Streck.ai

Preface

This guide is the user-facing companion to *CDCL, Phases 1-3*. The implementation book describes how the solver was built; this one describes how to use it.

CDCL is the right tool when your problem reduces to a purely combinatorial yes-or-no question over binary decisions, and when the structure of "this rules out that" matters more than the structure of "this much of this plus this much of that". Pigeonhole-style infeasibility, graph colouring, Sudoku, finite-domain CSPs encoded as SAT, propositional planning, hardware verification — these are CDCL's natural home. The library here is small enough to read in an afternoon and demonstrates the techniques (two-watched literals, first-UIP analysis, VSIDS, Luby restarts) that production SAT solvers use under the hood.

The book assumes you've used a propositional logic encoding before, can write a few lines of C, and want a quick path from "I have a yes/no decision problem" to "I have a satisfying assignment, or proof there isn't one." It closes with the comparison you'd expect for any solver in this series: when to reach for SAT versus SMT versus CSP versus MILP, and what hybrid patterns look like in operational systems.

— *Stockholm, May 2026*

Introduction

SAT is the simplest constraint-solving paradigm in the toolkit. There is one kind of variable (a Boolean), one kind of constraint (a clause — a disjunction of literals), and one question (does there exist an assignment to the variables that satisfies every clause?). Everything else is a modelling exercise: take the problem you have, encode it as Boolean variables and clauses, hand the formula to the solver, read back the answer.

The CDCL algorithm — conflict-driven clause learning — is what makes SAT practical at scale. The high-level loop is straightforward: pick a variable, assign it, propagate the consequences, recurse. When a conflict arises (some clause is violated under the current assignment), the algorithm doesn't just back up one decision — it *analyses* the conflict, derives a new clause that summarises why the conflict happened, adds that clause to the formula, and jumps back to the level where the new clause forces a different choice. The learned clauses accumulate; they cut off whole regions of the search space that share the same conflict structure; over time the solver becomes very good at not making the same kind of mistake twice.

The library implements this in about 650 lines of C99. The two-watched-literal scheme makes propagation almost free per assignment. The first-UIP analysis builds learned clauses in linear time. The VSIDS heuristic biases decisions toward variables that have been recently active in conflicts. Luby restarts periodically clear the partial assignment to escape from unproductive regions of the search.

This guide focuses on how to model your problem; the implementation book covers the algorithms.

Quick start

A complete program that finds an assignment to three variables satisfying two clauses:

```
#include "sat.h"
#include <stdio.h>

int main(void) {
    Sat *s = sat_new(3);    /* variables 1, 2, 3 */

    /* (x1 or not x2 or x3) and (not x1 or x2 or not x3) */
    sat_add_clause(s, (int[]){ 1, -2, 3 }, 3);
    sat_add_clause(s, (int[]){ -1, 2, -3 }, 3);

    int status = sat_solve(s);
    if (status == SAT_SATISFIABLE) {
        for (int v = 1; v <= 3; v++)
            printf("x%d = %d\n", v, sat_value(s, v));
    } else if (status == SAT_UNSATISFIABLE) {
        printf("UNSAT\n");
    }

    sat_free(s);
    return 0;
}
```

Build it with:

```
gcc -std=c99 -O2 -Isrc your_program.c src/sat.c -o your_program
```

That is the entire surface for typical use: create a context with `n_vars` variables, add clauses, solve, read off the assignment. The library is one header file and one implementation file.

Variables and literals

Variables are positive integers from 1 to N, following the DIMACS convention used by every SAT competition entry and most published instances. A literal is a signed integer: +k means "variable k is true", -k means "variable k is false". A clause is an array of literals representing their disjunction.

There is no separate `sat_new_var` call. You declare the variable count when creating the context (`sat_new(n_vars)`) and the variables `1..n_vars` exist from then on. If you don't know the variable count in advance, count them up first by walking your problem structure, then create the context — this keeps the solver's internal allocation single-pass.

The empty clause `(int[]){}` with `n_lits = 0` is the contradiction; adding it makes the formula trivially UNSAT. `sat_add_clause` returns 1 if this happens (or if a unit clause directly conflicts with an existing assignment); your code can either bail out or continue calling `sat_solve` which will return UNSAT.

Tautologies (clauses containing both a literal and its negation) are silently dropped. Duplicate literals within a clause are de-duplicated. These cleanups happen at clause-posting time, before the solver sees the formula.

Encoding patterns

A handful of patterns cover the great majority of real SAT modelling.

At-most-one. "At most one of these variables can be true". The pairwise encoding posts $\binom{n}{2}$ binary clauses:

```
for (int i = 0; i < n; i++)
    for (int j = i + 1; j < n; j++) {
        int lits[2] = { -vars[i], -vars[j] };
        sat_add_clause(s, lits, 2);
    }
```

This is quadratic in n but absolutely fine up to n around 30. For larger groups, use a sequential-counter or ladder encoding; both add auxiliary variables in exchange for a linear clause count. The trade-off is worth it past $n \approx 50$ or so.

At-least-one. A single clause:

```
sat_add_clause(s, vars, n);
```

The simplest encoding; nothing more is needed.

Exactly-one. At-most-one plus at-least-one. The pairwise at-most-one combined with the single at-least-one clause is the standard idiom. You can also combine them via a one-hot encoding using auxiliary variables, but the direct combination is usually fine for the sizes that come up.

Implication. $a \rightarrow b$ is the clause $\neg a \vee b$:

```
sat_add_clause(s, (int[]){ -a, b }, 2);
```

Conjunction in the antecedent and disjunction in the consequent decompose into separate clauses by standard CNF transformations. $a \wedge b \rightarrow c \vee d$ becomes the single clause $\neg a \vee \neg b \vee c \vee d$. $a \rightarrow b \wedge c$ becomes two clauses, $\neg a \vee b$ and $\neg a \vee c$.

Equivalence. $a \leftrightarrow b$ is two implications: $\neg a \vee b$ and $a \vee \neg b$.

Finite-domain variable. A variable taking one of k values is encoded by k Booleans $x_{v_for_value_1}, x_{v_for_value_2}, \dots, x_{v_for_value_k}$, with the constraint that exactly one is true. Domain pruning becomes "force this value-Boolean to be false". The dual encoding (a comparator chain) uses $k-1$ Booleans instead of k and is more compact for large domains, but the direct encoding is easier to read and good enough for most problems.

Permutation. n items to n positions, every item in exactly one position, every position holding exactly one item. Use n^2 Booleans $x_{i_in_position_j}$; one exactly-one constraint per row and one per column. This is the SAT encoding of an `alldifferent` constraint.

These five patterns plus the basic clause posting cover most of what real SAT modelling needs. For more elaborate structure (cardinality constraints of the form "exactly k of these n are true"), see the implementation book's encoding chapter.

Reading the answer

After `sat_solve` returns `SAT_SATISFIABLE`, every variable has a value: `sat_value(s, v)` returns `SAT_TRUE` (1) or `SAT_FALSE` (0). Variables that the solver didn't need to decide (because the formula didn't constrain them either way) get an arbitrary value; the standard behaviour is to return whichever value the most recent decision heuristic picked, which is typically false but is not contractually so. If you care about the values of unconstrained variables, post unit clauses to pin them.

The `SAT_UNDEF` return value (-1) appears only if you query a variable's value before solving or after an `UNSAT` result. Don't rely on this; check the solver status first.

After `SAT_UNSATISFIABLE`, the formula has no satisfying assignment. The library here does not produce an `UNSAT` proof; production solvers do (DRAT format, typically), and re-implementing that is a moderate engineering project rather than a research one. For tutorial purposes, the `UNSAT` verdict alone is enough.

`SAT_UNKNOWN` indicates the conflict budget was exhausted before the solver could decide. The default budget is unlimited; you only see this if you set `sat_set_conflict_limit` to a finite value. This is occasionally useful in incremental SAT contexts where you want bounded-time queries.

Tuning knobs

The library exposes two tuning knobs and one verbosity dial.

```
void sat_set_conflict_limit(Sat *s, long limit);  
void sat_set_verbose(Sat *s, int level);
```

`sat_set_conflict_limit` caps the number of conflicts the solver will explore before giving up with `SAT_UNKNOWN`. Default is -1 (unlimited). Useful when you're calling SAT inside a larger loop and want to bound the per-call latency. The right limit is workload-dependent; for sub-millisecond budgets on small problems, a limit in the low thousands is usually adequate.

`sat_set_verbose` controls trace output. Level 0 (default) is silent; level 1 prints summary statistics after solving; level 2 prints a line per conflict. Level 2 is mostly useful for debugging encoding bugs (you can watch the solver thrash on a clause you thought was redundant).

That is the entire tuning surface. Production solvers expose dozens of knobs (restart policies, decay rates, learnt-clause-database management strategies, branching heuristics); the tutorial library doesn't. If you find yourself needing more tuning, the problem is probably big enough to justify reaching for a production solver instead.

Tips and limitations

The encoding dominates the solve time. A well-encoded problem solves in milliseconds; a poorly-encoded version of the same problem can take hours. The standard advice is to use the smallest encoding that captures the structure faithfully, then iterate by profiling. If propagation count is high but conflict count is low, the encoding is propagating well; if both are low and decisions are high, the solver is guessing — you may be missing a structural constraint that would prune.

Symmetry kills SAT solvers. A problem with many symmetric solutions (rotate the colouring, permute the rows, swap two indistinguishable resources) generates many equivalent search paths. The solver doesn't know the symmetries are equivalent; it visits them all. The fix is symmetry breaking: post extra clauses that pick a canonical representative from each equivalence class. The classical N-Queens trick (force the queen in row 0 to be in the left half of the board) is the simplest example.

Watch your variable count, not your clause count. SAT solvers handle large clause counts gracefully — the watched-literal scheme is essentially insensitive to the total number of clauses; only the number of active clauses per literal matters. They handle large variable counts less gracefully because every decision has to choose among them. Encodings that minimise variable count at the cost of more clauses are usually better than the reverse.

Statistics are diagnostic. The stats functions (`sat_decisions`, `sat_conflicts`, `sat_propagations`, `sat_restarts`, `sat_learned_clauses`) are not just curiosity; they tell you what's going wrong when something is slow. A high decisions-per-conflict ratio means the solver is mostly guessing and rarely learning — the encoding is not exposing useful structure. A high propagations-per-decision ratio means the encoding is propagating well — each guess has lots of consequences. The pigeonhole example in `examples/ex1_pigeonhole.c` is the textbook case where propagations and conflicts both stay low while decisions blow up exponentially: pure SAT cannot exploit the counting argument that makes pigeonhole easy for humans.

Unsatisfiability is a result, not a problem. If `sat_solve` returns UNSAT, your encoding is contradictory — either intentionally (you're using SAT to prove infeasibility, as in pigeonhole) or unintentionally (a clause you didn't mean to add is contradicting another). In the unintentional case, the implementation book's debugging-by-relaxation pattern is the right tool: comment out clauses one block at a time until SAT comes back, then narrow down to the specific clause that caused the conflict.

Choosing your solver: SAT vs SMT vs CSP vs MILP

SAT sits at the bottom of the constraint-solving hierarchy. Other paradigms are stronger in specific directions; the trade-off is always between expressiveness and search power.

SAT

Strengths. Pure combinatorial decision problems with binary structure. The cleanest interface (one variable type, one constraint type) and the most mature underlying algorithm. CDCL has been the dominant SAT technique for two decades and is well understood — the clause learning, the watched literals, the VSIDS heuristic all compose into a search that exploits the formula's structure without needing problem-specific tuning.

Encoding flexibility. Almost any finite combinatorial problem can be encoded as SAT given enough variables and clauses. Hardware verification, propositional planning, finite-domain CSPs, graph problems, cryptanalysis of weak ciphers — they all reduce.

Production solvers (MiniSat, Glucose, CaDiCaL, Kissat) are tiny by enterprise-software standards and produce DRAT proofs that can be independently checked. For high-assurance contexts this matters: the trusted computing base is small.

Weaknesses. No arithmetic. Everything that involves "how much" rather than "whether" needs to be discretised or encoded with auxiliary cardinality constraints. The encodings work but they bloat the formula and weaken the learning — a single integer comparison may need dozens of auxiliary variables.

Encoding bloat for finite-domain problems. A CSP variable with domain size 100 takes 100 Boolean variables in the direct encoding; an `alldifferent` over n such variables takes $O(n^2)$ inequality clauses. CP solvers handle these natively; SAT solvers brute-force them.

No native optimisation. SAT answers feasibility, not "best". Optimisation requires a search loop on top of the SAT solver (binary search on the objective, or repeated SAT-with-bound) or the MaxSAT extension.

Air-force verdict. Use SAT for the inner combinatorial layer of a planning system — exact-cover decisions, frequency-assignment feasibility, IFF code consistency, mission-plan satisfiability under doctrinal constraints. Don't use it as the outer layer when the question is "best" rather than "feasible".

SMT

Strengths. Built on top of SAT (the library here is in fact the SAT layer beneath the SMT-Solver in this same repository). Adds first-class arithmetic (LRA, LIA), uninterpreted functions, arrays, and the Nelson-Oppen combination of theories. You get the clause-learning power of CDCL together with native handling of " $x + y \leq 5$ " or " $f(a) = f(b)$ implies $a = b$ ".

Modelling expressiveness. Anything you can write in a constraint language with quantifier-free first-order logic over the supported theories. The SMT modelling language (SMT-LIB) is essentially a Lisp for constraint problems.

Weaknesses. Heavier per-call cost than pure SAT. The theory layer does meaningful work on every propagation; for problems that are *really* propositional, this overhead is wasted.

Decidability boundaries. Once you add quantifiers, nonlinear arithmetic, or arrays-with-extensionality, you leave the decidable fragments and the solver may not terminate. Production SMT solvers have heuristics for the undecidable cases but the guarantees become statistical.

Air-force verdict. Use SMT when the problem genuinely mixes Boolean structure with arithmetic — engagement-window arithmetic, separation-minima compliance, continuous-time invariants. The unified-system architecture in the Streck.ai corpus uses SMT exactly here, with pure SAT for the propositional rule-compliance layer.

CSP

Strengths. Native finite-domain variables. Global constraints (`alldifferent`, `cumulative`, `table`) that propagate problem-specific structure aggressively. The classical CSP family (sparse-set domains, MAC, `dom/wdeg`) handles permutation-heavy problems beautifully.

For exact-cover problems specifically, DLX (Dancing Links) is unbeatable: the link-rewiring data structure makes the inner loop almost free, and the natural enumeration mode is rarely matched by other paradigms.

Weaknesses. No clause learning in the classical algorithms. The solver visits dead ends as many times as the search tree's structure leads it there. LCG (the hybrid covered in the LCG-Solver) was designed specifically to address this gap.

Less general than SAT for arbitrary combinatorial encodings. A CSP solver that doesn't have a propagator for your specific constraint shape will either reject your formulation or fall back to a generic propagator that propagates very little.

Air-force verdict. Use CSP when the problem is permutation- or assignment-heavy with rich combinatorial structure, when you want enumeration of solutions, or when the natural propagators of your problem (`alldifferent`, `table`, `regular`) match what the CSP library offers. The CSP/DLX architecture in the Streck.ai DWTA implementation is exactly this pattern.

MILP

Strengths. Decades of tuning and production maturity. Native handling of linear inequalities, continuous variables, weighted objectives, LP-relaxation bounds. Commercial solvers handle problems with millions of variables routinely. The LP relaxation gives dual prices, sensitivity analysis, and warm-start information for re-solving under perturbation.

Weaknesses. Logic-rich constraints encode awkwardly. Each "if-then" rule requires either big-M constants or auxiliary binaries; the resulting LP relaxation is often very loose. Combinatorial structure exploitation depends on the solver's heuristics rather than on the formulation.

No native clause learning. Branch-and-bound prunes by bounding, not by analysing conflicts.

Air-force verdict. Use MILP for static optimisation problems with rich numeric structure — budgeted WTA, fleet routing, sortie scheduling. Don't use it for purely combinatorial decision problems where SAT or CSP would do the job in a fraction of the modelling effort.

Comparison summary

Concern	SAT	SMT	CSP	MILP
Pure combinatorial decision	strong	strong	strong	medium
Boolean + arithmetic	weak	strong	medium	strong
Finite-domain CSP	medium	medium	strong	weak
Continuous quantities	none	strong	none	strong
Weighted optimisation	weak	medium	weak	strong
Exact-cover / enumeration	medium	weak	strong	weak
Time-window scheduling	weak	medium	strong	weak
UNSAT proofs	strong	strong	weak	medium
Maturity and tuning	strong	strong	medium	strong

Combining multiple technologies

SAT is rarely used in isolation in operational systems; the more interesting deployments combine it with other paradigms.

SAT inside MaxSAT. The most natural extension: take SAT's CDCL engine, layer weighted soft clauses on top, optimise over the cost of unsatisfied softs. The MaxSAT-Solver in this repository builds directly on this CDCL implementation. Operationally, MaxSAT is what you reach for when SAT's feasibility question becomes "best feasibility given that some preferences must be sacrificed".

SAT as the propositional substrate of SMT. The SMT-Solver in this repository similarly uses the CDCL engine as its Boolean layer. The theory propagators (LRA, EUF) inject theory-derived clauses into the SAT engine, which then learns them along with the standard conflict-driven clauses. This is the architecture every modern SMT solver uses.

SAT inside LCG. The LCG-Solver uses CDCL underneath, with constraint propagators replacing the pure-SAT inference. When a cumulative or alldifferent propagator derives a bound tightening, it produces an explanation clause that enters the CDCL learnt-clause database. The learning carries across the search; a conflict in one part of the schedule prunes similar conflicts elsewhere.

SAT as feasibility check. A pattern that appears across operational planning: use a heavyweight solver (MILP, CSP, scheduling) to compute a candidate plan, then run a fast SAT check to confirm doctrinal-rule compliance. The SAT formulation captures the rules as clauses; the candidate plan is a unit-clause assignment; UNSAT means a rule was violated. This is the Streck.ai unified-system pattern for rule-compliance certification.

SAT for enumeration. Sometimes you want all satisfying assignments, not just one. The standard pattern is to call SAT, post a clause forbidding the model just found ("at least one variable must differ"), and re-solve. The loop terminates when SAT returns UNSAT. This works but is much slower than DLX for problems that have many solutions; reach for CSP/DLX if enumeration is the primary mode of operation.

The most common operational pattern is SAT as the inner feasibility layer of a larger system. The outer layer (MILP, MaxSAT, LCG, or a custom search) decides "what should we try"; the SAT layer answers "is this consistent with the rules". The roundtrip is cheap because SAT is fast; the combination is more expressive than any single paradigm.

API summary

Function	Purpose
<code>sat_new(n_vars) / sat_free</code>	Context lifecycle; variables 1..n_vars exist from creation
<code>sat_add_clause(s, lits, n)</code>	Post a clause; returns 1 if formula is now trivially UNSAT
<code>sat_solve(s)</code>	Solve; returns SAT_SATISFIABLE, SAT_UNSATISFIABLE, or SAT_UNKNOWN
<code>sat_value(s, v)</code>	Read the value of variable V after a satisfiable solve
<code>sat_set_conflict_limit(s, limit)</code>	Cap conflicts before giving up with SAT_UNKNOWN
<code>sat_set_verbose(s, level)</code>	Trace verbosity: 0 silent, 1 summary, 2 per-conflict
<code>sat_decisions(s)</code>	Total decisions made during the search
<code>sat_conflicts(s)</code>	Total conflicts encountered
<code>sat_propagations(s)</code>	Total unit-propagation steps
<code>sat_restarts(s)</code>	Number of Luby restarts
<code>sat_learned_clauses(s)</code>	Clauses learned by CDCL analysis

That is the entire surface. One constraint-posting function, one solve, the model-query, two tuning knobs, and the stats accessors. The library is small on purpose — most of the modelling work happens in your code, deciding which clauses to post. The library does the propagation, the conflict analysis, the clause learning, and the search.