Solving problems with CDCL

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

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

Solving problems with MILP

A user's guide to mixed-integer linear programming

Preface

This guide is the user-facing companion to *Mixed-Integer Linear Programming*, *Phases 1-2*. The implementation book describes how the solver was built; this one describes how to use it.

MILP is the right tool when your problem has a linear objective, linear constraints, and decisions that are partly continuous (how much fuel, how many hours, what fraction of a resource) and partly discrete (which targets to engage, which tasks to schedule, which routes to fly). The classical operations-research formulations — assignment, knapsack, set cover, transportation, network flow, the budgeted version of weapon-target assignment — all live here. Decades of solver tuning have made MILP the default tool for static optimisation problems with rich numeric structure.

The book assumes you've seen a linear program before, can write a few lines of C, and want a quick path from "I have an optimisation problem" to "I have a provably-best decision". It closes with the comparison you'd expect for any solver in this series: when to reach for MILP versus SAT versus CSP versus LCG, and what hybrid patterns look like in operational systems.

— Stockholm, May 2026

Introduction

A mixed-integer linear program is the constraint-solving paradigm where you say what you want explicitly, in numbers. The objective is a linear function of the decision variables. The constraints are linear inequalities or equalities. Some of the variables are continuous; some are required to be integer (or, the common case, binary). The solver returns the assignment that optimises the objective among all that satisfy the constraints.

Two layers do the work. The inner layer is the *simplex method* — Dantzig's algorithm from 1947, which finds the optimum of a continuous LP by walking from vertex to vertex of the feasible polytope. The simplex is well-behaved on the LP relaxation of a MILP: relax every integrality requirement, solve the resulting LP, and you have a bound on the integer optimum. The outer layer is *branch-and-bound* — at every node of a search tree, solve the LP relaxation; if the LP is integer-feasible, you have a candidate solution; if not, pick a fractional variable, branch on it (force it down on one child, up on the other), and recurse. The LP bound at each node prunes subtrees that cannot beat the incumbent.

The interplay between the two layers is what makes MILP work. The LP relaxation is fast to solve and gives a strong bound for many natural problem structures. Branch-and-bound is dumb in isolation but smart when guided by tight LP bounds. Production solvers (Gurobi, CPLEX, HiGHS, SCIP) layer hundreds of additional techniques on top — cutting planes, primal heuristics, presolve, parallel exploration — but the simplex-plus-branch-and-bound architecture is the substrate.

The library here is about 700 lines of C99 implementing a two-phase revised simplex and recursive branch-and-bound. It is meant to read in an afternoon, not to compete with HiGHS. What it gives you is the same architectural shape the production solvers use, in code small enough that you can follow what happens to every variable.

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

Quick start

A complete program that solves a tiny knapsack: 3 items, capacity 10, maximise value:

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

int main(void) {
    /* 3 binary variables, 1 capacity constraint. */
    MilpModel *m = milp_new(3, 1);

    /* Objective: max 7 x0 + 9 x1 + 5 x2 */
    double c[3] = { 7.0, 9.0, 5.0 };
    milp_set_objective(m, MILP_MAX, c);

    /* Constraint: 4 x0 + 6 x1 + 3 x2 <= 10 */
    double a[3] = { 4.0, 6.0, 3.0 };
    milp_set_row(m, 0, a, MILP_LE, 10.0);

    /* All three variables are binary. */
    for (int j = 0; j < 3; j++) milp_set_var_type(m, j, MILP_BINARY);

    double x[3], obj;
    int status = milp_solve(m, x, &obj);
    if (status == MILP_OPTIMAL) {
        printf("objective = %.1f\n", obj);
        for (int j = 0; j < 3; j++) printf("  x%d = %.0f\n", j, x[j]);
    }

    milp_free(m);
    return 0;
}
```

Build it with:

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

Five calls (`milp_new`, `milp_set_objective`, `milp_set_row`, `milp_set_var_type`, `milp_solve`) are typically enough. The library handles the LP relaxation, the branch-and-bound, and the bookkeeping.

Building a model

The library uses an index-based API: variables are integers `0..n_vars-1`, constraints are integers `0..n_constraints-1`, both declared at construction time.

```
MilpModel *m = milp_new(n_vars, n_constraints);
```

You must populate every row and the objective before solving. The library does not error on missing data — if you skip a row, that row will be all zeros and the constraint will likely be satisfied trivially.

The objective is a vector of `n_vars` coefficients. The sense (minimisation or maximisation) is part of the same call:

```
milp_set_objective(m, MILP_MIN, c);   /* or MILP_MAX */
```

The coefficients `c[j]` are the cost or value of variable `x[j]` in the objective. For variables that don't appear in the objective, use 0.

Constraint rows are vectors of `n_vars` coefficients plus a sense and a right-hand-side:

```
milp_set_row(m, row_index, a, op, rhs);   /* op is MILP_LE, MILP_EQ, MILP_GE
*/
```

This sets row `row_index` to $\sum_j a_j x_j \;\text{op}\; \text{rhs}$. The `op` constants are `MILP_LE` ($\le$), `MILP_EQ` ($=$), and `MILP_GE` ($\ge$).

Variable types are continuous by default. Override with:

```
milp_set_var_type(m, var_index, MILP_BINARY);     /* 0 or 1 */
milp_set_var_type(m, var_index, MILP_INTEGER);    /* any integer */
milp_set_var_type(m, var_index, MILP_CONTINUOUS); /* default */
```

`MILP_BINARY` is shorthand for "integer with bounds [0, 1]". Setting a variable as binary overrides the bounds; explicit bounds set afterward will be respected, but typically you want the default for binaries.

Variable bounds default to $[0, +\infty)$. Override with:

```
milp_set_var_bounds(m, var_index, lo, hi);
```

Use `-INFINITY` (from `math.h`) for unbounded below; the library translates this to the simplex method's free-variable encoding. Use any large positive constant for unbounded above; production solvers represent infinity directly, but the tutorial library uses big-M.

That's the model-building surface. The combination of objective, rows, variable types, and bounds is enough to express any MILP.

Reading the answer

After `milp_solve` returns `MILP_OPTIMAL`, the variable values are in the output array and the objective value is in the output scalar:

```
double x[n_vars], obj;
int status = milp_solve(m, x, &obj);
if (status == MILP_OPTIMAL) {
    /* x[j] holds the optimal value of variable j */
    /* obj holds the optimal objective value */
}
```

The status codes are:

- `MILP_OPTIMAL` (0) — the optimum was found; `x` and `obj` are valid.
- `MILP_INFEASIBLE` (1) — no feasible assignment exists.
- `MILP_UNBOUNDED` (2) — the objective can be made arbitrarily good; usually a modelling bug.
- `MILP_NODE_LIMIT` (3) — the branch-and-bound exhausted its node budget before proving optimality; `x` may hold a feasible but suboptimal incumbent.
- `MILP_ERROR` (4) — internal error; check stderr.

The `MILP_UNBOUNDED` case is almost always a modelling mistake. It means there is some direction in which the objective improves indefinitely without violating any constraint. Common causes: forgetting to bound a continuous variable above, a sign error in the objective, a missing constraint. The fix is to find the unbounded direction (the LP solver reports it) and add the missing constraint.

The `MILP_INFEASIBLE` case is more interesting. Sometimes it indicates a genuinely impossible request; sometimes it indicates an over-specified model. The standard debugging pattern is to relax constraints one at a time until feasibility returns, which identifies the conflicting subset. Production solvers automate this via Irreducible Infeasible Subsystem (IIS) detection; the tutorial library doesn't, but the manual pattern works fine for the sizes that come up.

LP relaxation, separately

For diagnostic and bound-computing purposes, the library exposes the LP relaxation as a separate entry point:

```
double x_lp[n_vars], obj_lp;
int status = milp_solve_lp(m, x_lp, &obj_lp);
```

This treats every variable as continuous (ignoring `MILP_INTEGER` and `MILP_BINARY`) and returns the LP optimum. For a minimisation problem, the LP value is a *lower bound* on the integer optimum; for maximisation, an *upper bound*. The gap between the LP and the integer solution measures how hard the MIP is for branch-and-bound — a small gap means branch-and-bound prunes aggressively; a large gap means it has to explore most of the tree.

Running the LP separately is useful in three contexts. First, as a sanity check on the formulation: if the LP is infeasible, the MIP is too; if the LP is unbounded, you have a modelling bug to fix. Second, as a fast feasibility filter inside a larger loop: an LP solve is much cheaper than a full MIP solve, and confirms whether the integer problem is worth attempting. Third, as a source of dual values and sensitivity information; the tutorial library doesn't expose these directly, but the LP-solve infrastructure underneath produces them.

What the LP relaxation tells you

Three structural categories of MILP problem, distinguished by how their LP relaxations behave. Knowing which one you have is the most useful single diagnostic for MILP modelling.

Category 1: TUM (totally unimodular). The constraint matrix is structured such that the LP relaxation is always integer at every basic feasible solution. Branch-and-bound never branches; the LP at the root is the MIP answer. The classical assignment problem is the canonical TUM case: each constraint row picks one agent or one task, the matrix is bipartite-incidence, every LP optimum is already a valid assignment. Set partitioning with totally-unimodular structure, transportation problems, and basic network flow all live here.

If you have a TUM problem, the LP relaxation tells you everything. The MIP solve is the LP solve, with a sanity-check pass for integer-feasibility. Branch-and-bound is one node deep.

Category 2: tight LP, real integer gap. The constraint matrix is not TUM but the LP relaxation is still strong — the LP optimum sits close to the integer optimum, with at most a small fractional gap. 0/1 knapsack is the canonical example: Dantzig's classical fractional-knapsack solution has at most one fractional variable, every other variable is already 0 or 1, the gap is bounded by the value of the most valuable single item. Branch-and-bound explores a shallow tree and prunes most subtrees against the incumbent.

If you have a tight-LP problem, the LP relaxation is the dominant cost and branch-and-bound is cheap. You should still solve the full MIP (the LP solution is fractional and not deployable), but the runtime is dominated by the LP layer.

Category 3: loose LP, big integer gap. The constraint matrix is structurally loose — the LP relaxation gives a weak bound and branch-and-bound has to do real work. Set cover is a canonical example: the LP returns a fractional cover with values often very far from 0 or 1, the bound is a $\log n$-approximation rather than a tight number, and the tree can grow exponentially. Mining problems, knapsack with conflicts, multi-commodity flow — these often live here.

If you have a loose-LP problem, the MIP solve will dominate. Production solvers throw cutting planes (Gomory cuts, knapsack cuts, clique cuts) at the relaxation to tighten it; the tutorial library doesn't. For problems in this category that are too large for pure branch-and-bound, the answer is either a stronger formulation (different variables, different constraints, exposing more structure to the LP), a hybrid with another paradigm (constraint propagation to reduce the integer space before MILP sees it), or a production solver.

Knowing which category you're in is half of MILP modelling skill. The other half is shaping the formulation to move it toward category 1 or 2 — adding redundant valid inequalities, lifting weak constraints, exploiting problem-specific structure.

Modelling patterns

A few patterns cover most operational MILP modelling.

Indicator variables. A binary y that means "this thing happens". Use the variable directly in the objective (its coefficient is the cost or value of the thing). Couple it to other variables via big-M or implication-style constraints.

Big-M. "If $y = 0$, then $\sum a_j x_j \le 0$". Encoded as $\sum a_j x_j \le M y$ for a sufficiently large constant M. When $y = 0$, the constraint forces the sum to be at most 0; when $y = 1$, the constraint becomes $\sum a_j x_j \le M$ which is essentially vacuous. Big-M is numerically painful — too small a M excludes valid solutions; too large weakens the LP relaxation; finding the right M requires domain knowledge. Use sparingly.

Aggregated assignment. A 2D assignment problem with n agents and m tasks has nm binary variables and $n + m$ constraints (each agent assigned at most once, each task covered at most once). This is the natural shape; it propagates through TUM if the structure is bipartite.

Knapsack-style budget. $\sum c_j x_j \le B$ where each x_j is binary. The single linear inequality is the canonical example of a constraint that destroys TUM and introduces a real integer gap, but the gap is bounded and branch-and-bound handles it well.

Set covering. $\sum_{s : e \in S_s} x_s \ge 1$ for each element e. The variables are sets; each constraint says "every element must be covered". The natural relaxation is the fractional cover; the integer gap is $O(\log n)$ in the worst case. Set partition (constraints are equalities, not inequalities) has a smaller gap because the LP can't double-cover.

Time-indexed scheduling. A task running over time interval $[s, s+d]$ becomes T binary variables $y_{j,t}$ meaning "task j is running at time t", with constraints $\sum_t y_{j,t} = d$ (the task runs for exactly d time units) and the contiguity constraints. The variable count blows up linearly with the horizon length; the LP relaxation is typically loose. This is the formulation where MILP is at a disadvantage to CP/LCG. Use it for small-horizon problems; reach for LCG when the horizon is operationally meaningful.

Disjunctions via big-M. "Either constraint A or constraint B holds". Introduce a binary z, write A as $A \le M(1 - z)$ and B as $B \le M z$. When $z = 0$, A is active; when $z = 1$, B is. The same numerical pain as basic big-M, and worse if the two constraint bodies have very different scales.

The implementation book covers more elaborate patterns (Benders decomposition, column generation, polyhedral cuts) for problems where the basic patterns scale poorly.

Worked example: budgeted weapon-target assignment

The MILP-Solver's `examples/ex4_wta.c` shows the budgeted version of weapon-target assignment, which is the classical OR formulation of static WTA. Variables x_{ij} indicate "weapon i engages target j". The objective maximises expected damage $\sum v_j p_{ij} x_{ij}$, where v_j is target j's value and p_{ij} is the kill probability of weapon i against target j. Constraints: each weapon engages at most one target ($\sum_j x_{ij} \le 1$), each target is engaged by at most one weapon ($\sum_i x_{ij} \le 1$), and the total cost is within budget ($\sum_{ij} c_{ij} x_{ij} \le B$).

Without the budget row, the matrix is the bipartite-incidence matrix and TUM applies: every LP optimum is integer, the MIP is one LP solve, branch-and-bound never branches. Add the budget row and TUM is destroyed; the LP can return a fractional engagement that splits the budget across targets in a way no real shooter can. But the integer gap is small (the budget is one constraint among $n+m+1$), branch-and-bound prunes most of the 2^{nm} space using the LP bound, and the solve is fast.

This is the kernel that sits inside a static-WTA optimiser. In a real TEWA system, you wrap it in a time loop, recompute p_{ij} from current geometry and seeker models, re-solve at each engagement update, and feed the result downstream to the engagement scheduler.

The interesting modelling decisions in this kernel:

Why expected damage rather than expected leakage? They are equivalent in the static case ($\sum v_j - \text{leakage} = \text{damage prevented}$), but the maximisation objective has a slightly cleaner formulation — no constant offset, no sign manipulation. In the dynamic case where the time of leakage matters (a threat that gets through later is partly engaged later, partly engaged earlier, accumulating expected damage over time), expected leakage is the natural objective and the sign flip in the formulation reflects this.

Why log-space for combined kill probabilities? When multiple weapons can engage one target, the combined kill probability is $1 - \prod_i (1 - p_{ij})^{x_{ij}}$ which is not linear in x_{ij}. The standard linearisation introduces $u_{ij} = -\log(1 - p_{ij})$ and writes the objective as a function of $\sum_i u_{ij} x_{ij}$. The library here keeps the simple linear-in-x form because the example assigns at most one weapon per target; the log-space transformation is the standard extension when multiple shots per target are allowed.

Why is the budget constraint loose? In practice, the budget B may correspond to total rounds fired, total fuel burned, total time the radar is illuminating targets, or aggregated cost in some operational metric. The LP relaxation can split a single weapon's commitment across targets fractionally; branch-and-bound rounds those back to integer commitments. The gap is small but non-zero, which is the typical category-2 behaviour.

Tips and limitations

Scale of coefficients matters. Production simplex implementations scale the coefficient matrix automatically; the tutorial library doesn't. If your problem mixes coefficients spanning many orders of magnitude (kilometre-scale ranges next to fractional-probability scales), the simplex may produce numerically poor solutions. Manually scaling the rows and the objective to similar magnitudes before passing to the solver is the cheap fix.

Tolerance on integrality. The library uses a small epsilon (around 10^{-6}) to decide whether a fractional LP value should be considered integer. If your problem has natural fractional structure right at the epsilon boundary (a coefficient that happens to make $x = 0.0000005$ a valid LP solution), the solver may report integer-feasible when the value is suspiciously close to zero. The fix is to tighten the formulation; the symptom is a feasible answer that doesn't satisfy your downstream integrality expectation.

Branching strategy. The library uses the simplest possible strategy: most-fractional variable first. Production solvers use elaborate strategies (strong branching, pseudo-costs, reliability) that can change the tree size by orders of magnitude. For tutorial-scale problems, most-fractional is fine; for hard problems where the branch-and-bound is the bottleneck, this is the first place to look for speedup.

The node limit. Default is 2^{20} nodes, which is generous for the kinds of problems the library can handle in reasonable time. If you hit the node limit, the problem is too large for the tutorial solver — either reformulate to expose more structure, or move to a production solver.

Diagnostic stats. `milp_nodes_explored` and `milp_lps_solved` tell you whether the runtime was dominated by tree exploration (many nodes, similar LP count) or by hard LP relaxations (few nodes, many simplex iterations per LP, visible only via `verbose = 2`). `milp_lp_relaxation` returns the root LP value, which combined with the integer optimum gives you the integrality gap. A gap that exceeds your expectation is a sign that the formulation is loose; tightening it (adding valid inequalities, lifting weak rows) is the standard response.

Choosing your solver: MILP vs SAT vs CSP vs LCG

MILP has a specific niche. There are problems where it is unambiguously the right tool; there are others where SAT, CSP, or LCG win.

MILP

Strengths. Native handling of linear objectives and inequalities. Continuous variables, weighted sums, probabilistic objectives — all natural. LP-relaxation bounds give you sensitivity analysis, dual prices for tight constraints, warm-start information for re-solving under perturbation. Decades of solver maturity in the commercial and open-source tools; production solvers handle millions of variables routinely.

Optimisation is native. The objective is part of the formulation; the search descends toward it without an outer loop.

The LP relaxation is itself a useful artefact. Even when the MIP is hard, the LP gives a lower bound that supports approximation guarantees, a feasibility verdict at low cost, and a basis for column-generation extensions.

Weaknesses. Logic-rich constraints encode awkwardly. "If A then B" requires big-M (numerically painful) or auxiliary binaries (formulation bloat). The LP relaxation of these encodings is often very loose, leading to lots of branching. Combinatorial structure exploitation depends on the solver's branching strategy and cutting planes; the LP itself doesn't "know" about Hall sets, alldifferent, or matching unless you encode them explicitly.

Time-window scheduling is painful. Time-indexed binary variables blow up the variable count and produce loose LP relaxations. Cumulative resources have the same problem at a larger scale. CP and LCG handle these natively with one or two global constraints; MILP needs hundreds of binaries to express the same thing.

No clause learning. Branch-and-bound prunes by bounding; it doesn't analyse conflicts and learn from them the way CDCL does. For problems where pruning depends on combinatorial structure rather than numeric bounds, branch-and-bound can revisit similar dead ends many times.

Air-force verdict. Use MILP for static optimisation with rich numeric structure: budgeted WTA, ATO-level resource allocation, fleet routing, sortie scheduling, fuel-flow optimisation. Don't use it for time-window engagement scheduling or for logic-heavy doctrinal decision problems where the constraints are clauses rather than inequalities.

SAT

Strengths. Pure combinatorial decision problems. The CDCL machinery (conflict-driven learning, watched literals, VSIDS) handles propositional structure aggressively. Production SAT solvers are tiny, fast, and produce DRAT proofs for high-assurance contexts.

Weaknesses. No arithmetic. Everything that involves "how much" rather than "whether" needs to be discretised or encoded with cardinality constraints. No native optimisation — feasibility only.

Air-force verdict. Use SAT for purely propositional decision problems — doctrinal rule compliance, finite-domain CSP feasibility encoded propositionally, configuration consistency, IFF code checking. Combine with MaxSAT for the natural optimisation extension.

CSP

Strengths. Native finite-domain variables. Global constraints (alldifferent, table, cumulative) that propagate problem-specific structure. The classical CSP family (sparse-set domains, MAC, dom/wdeg) handles permutation-heavy problems beautifully. DLX is hard to beat for exact-cover problems specifically.

Weaknesses. No clause learning in the classical algorithms (LCG fixes this). No optimisation in the basic formulation. Less general than SAT for arbitrary combinatorial encodings.

Air-force verdict. Use CSP for permutation- or assignment-heavy problems with rich combinatorial structure and small enough scale that smart pruning beats clause learning. Use DLX specifically for exact-cover problems with the magazine/slot structure of DWTA assignment.

LCG

Strengths. The CP-SAT hybrid: native scheduling primitives (cumulative, no-overlap), bounded-consistent alldifferent, custom propagators, all with CDCL learning underneath. Time-windowed scheduling that MILP encodes awkwardly becomes natural here. Clause learning across the search prunes similar dead ends globally.

Weaknesses. No native optimisation. Less mature than MILP — fewer presolve transformations, no commercial-grade tuning. LP-relaxation bounds are not available; the search has to use combinatorial bounds only.

Air-force verdict. Use LCG for engagement scheduling once allocation is decided. Time-windowed engagements through fire-control channels with cumulative capacity is exactly what cumulative is for. The Streck.ai unified planning system uses LCG for the schedulability layer underneath MaxSAT-based assignment.

Comparison summary

Concern	MILP	SAT	CSP	LCG
Linear objective	strong	weak	weak	weak
Continuous variables	strong	none	none	none
LP-relaxation bounds	strong	none	none	none
Pure combinatorial decision	medium	strong	strong	strong

Logic-rich constraints	weak	strong	medium	medium
Finite-domain CSP	weak	medium	strong	strong
Exact-cover / enumeration	weak	medium	strong	medium
Time-window scheduling	weak	weak	strong	strong
Cumulative resources	medium	weak	strong	strong
Re-solve under perturbation	strong	medium	medium	medium
Maturity and tuning	strong	strong	medium	medium

Combining multiple technologies

MILP composes well with other paradigms. A few patterns from operational systems:

MILP master + CP subproblem (logic-based Benders). The MILP master decides the high-level allocation (which weapons engage which targets, optimising expected damage with kill-probability matrices). The CP subproblem checks that the allocation can be scheduled through the fire-control channels with time windows. If the schedule is infeasible, the subproblem returns a Benders cut — a constraint that excludes the offending allocation — and the master re-solves. The pattern handles WTA at a scale where neither pure paradigm would work alone, because the master is fast at the numeric optimisation and the subproblem is fast at the combinatorial scheduling.

MILP relaxation as bound for any combinatorial solver. Solve the LP relaxation of your problem; use the LP value as a bound for a separate combinatorial search. The LP gives you a "no integer solution can be better than this" guarantee that bounds the search regardless of which solver runs the integer search. This is the standard pattern for problems where the LP relaxation is strong but the combinatorial structure is poorly suited to MILP branch-and-bound (alldifferent-heavy permutation problems, for instance).

Column generation. When the variable count is huge but only a small subset will be active in the optimum, generate columns (variables) on demand. Solve a restricted master problem with the current columns; price out new columns by solving a subproblem that finds the most profitable variable not yet in the master; add it; re-solve; repeat until pricing fails to find an improving column. Production crew scheduling, vehicle routing, and cutting-stock problems all use this pattern. Implementing it on top of the tutorial MILP library is a moderate engineering project; production solvers expose column-generation interfaces directly.

Re-solving under perturbation. A real operational system rarely solves one MILP and ships the answer. It solves continuously: every minute or every event, the threat picture changes, the optimisation re-runs, the plan updates. MILP supports this well through warm-starting — solve the LP from the previous basis rather than from scratch, and you save most of the simplex work. The tutorial library doesn't expose warm-start; production solvers do, and the speedup on adjacent solves can be 10x or more.

Cuts and valid inequalities. When the LP relaxation is loose, adding valid inequalities tightens it. The simplest are Gomory cuts — automatically derivable from the simplex tableau when a basic variable is fractional. More elaborate cuts (knapsack covers, clique cuts, flow cuts) require problem-specific reasoning but can dramatically reduce the branch-and-bound tree. Production solvers integrate cut generation into the search; tutorial libraries leave it to the user.

API summary

Function	Purpose
`milp_new(n_vars, n_constraints)` / `milp_free`	Model lifecycle
`milp_set_objective(m, sense, c)`	Set the linear objective; sense is `MILP_MIN` or `MILP_MAX`
`milp_set_row(m, row, a, op, rhs)`	Set constraint row; op is `MILP_LE`, `MILP_EQ`, or `MILP_GE`
`milp_set_var_type(m, var, type)`	Variable type: `MILP_CONTINUOUS`, `MILP_INTEGER`, `MILP_BINARY`
`milp_set_var_bounds(m, var, lo, hi)`	Variable bounds; default is [0, +infinity)
`milp_solve_lp(m, x_out, obj_out)`	Solve the LP relaxation only
`milp_solve(m, x_out, obj_out)`	Solve the full MILP via branch-and-bound
`milp_nodes_explored(m)`	Branch-and-bound nodes explored in most recent solve
`milp_lps_solved(m)`	LP relaxations solved during the search
`milp_lp_relaxation(m)`	Value of the root LP relaxation
`milp_set_verbose(m, level)`	Trace verbosity: 0 silent, 1 tree summary, 2 every node
`milp_set_node_limit(m, limit)`	Cap branch-and-bound nodes (default 2^{20})

That is the entire surface. Four model-building functions, two solve entry points, and the stats and tuning accessors. The library is small on purpose — most of the modelling work happens in your code, deciding which variables to use, which constraints to post, and which structural simplifications to exploit. The library does the simplex, the branch-and-bound, the LP bookkeeping, and the optimisation.

Solving problems with SMT

A practical guide to modelling and solving with SAT + EUF + LRA + Nelson-Oppen

Preface

If you have read the companion book that builds the solver --- *Satisfiability Modulo Theories, Phases 1-4* --- you have a solver in your hands and a fairly good idea of what is happening inside it when you call `smt_check`. This book is about how to *use* it: how to translate a problem you have into a formula the solver can answer, how to read what it tells you back, and where the edges of what it can do actually are.

The arc of the book mirrors the arc of *Predictive Algorithms*: we begin with the smallest possible working examples, build a mental model of what the solver is doing as it searches, and finish with a real worked problem --- this time, the placement of widgets in a user interface. UI placement turns out to be a good showpiece for SMT because it sits exactly in the seam between two kinds of constraint: alignment and spacing are linear arithmetic (LRA's home turf), while non-overlap is *disjunctive* (the SAT layer's home turf). Tools that handle only one of the two --- linear solvers like Cassowary, or pure constraint-satisfaction backtrackers --- end up either over-relaxing the problem or over-discretising it. SMT handles both at once.

A short reading guide. If you already have a problem in mind, you can skip the first two chapters and jump to the modelling-patterns chapter (4). If you want to understand what the solver is doing inside, the companion book is the right place; chapter 3 here gives only enough of the mental model to make the patterns and worked examples land.

We assume you have built the solver via `./build.zsh` and seen its sanity tests pass; that puts a `build/` directory next to your sources with a `smt.o` object file you can link against. All code in this book is written against the public interface in `smt.h`; nothing here pokes at solver internals.

— Stockholm, May 2026

1. Quick start

The smallest interesting interaction with the solver is three lines:

```
SmtCtx *ctx = smt_new();
smt_assert_eq(ctx, smt_mk_const(ctx, "a"), smt_mk_const(ctx, "b"));
int r = smt_check(ctx);
// r == SMT_SAT: there is a model where a and b are the same value.
smt_free(ctx);
```

Three different theories, three "hello world" problems.

1.1. Hello, EUF

Two constants `a` and `b`; the equality `a = b` is satisfiable.

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

int main(void) {
    SmtCtx *ctx = smt_new();
    int a = smt_mk_const(ctx, "a");
    int b = smt_mk_const(ctx, "b");
    smt_assert_eq(ctx, a, b);
    printf("%d\n", smt_check(ctx));     // 10 (SMT_SAT)
    smt_free(ctx);
}
```

A slightly less trivial EUF problem: transitivity. `a = b` and `b = c` imply `a = c`, so asserting `a != c` alongside is unsatisfiable.

```
int a = smt_mk_const(ctx, "a");
int b = smt_mk_const(ctx, "b");
int c = smt_mk_const(ctx, "c");
smt_assert_eq (ctx, a, b);
smt_assert_eq (ctx, b, c);
smt_assert_neq(ctx, a, c);
// smt_check returns SMT_UNSAT.
```

The solver finds this contradiction without exploring any choices: the EUF theory propagates `a = c` from the chain, the negated equality already on the trail conflicts with it, and the search ends on the first call to `euf_check`. We will come back to what "propagates" and "conflicts" mean in chapter 3.

1.2. Hello, LRA

Real-valued variables and linear inequalities. `x + y <= 5` with `x > 3` and `y > 3` is unsatisfiable, because the two lower bounds together give `x + y > 6`, which contradicts the upper bound on the sum.

```
int x = smt_mk_real_var(ctx, "x");
int y = smt_mk_real_var(ctx, "y");

SmtLinTerm xy[2] = { {x, 1, 1}, {y, 1, 1} };       // 1*x + 1*y
smt_assert_lra_le(ctx, xy, 2, 5, 1);               // x + y <= 5
```

```
// x > 3 is the negation of x <= 3.
SmtLinTerm xt = { x, 1, 1 };
int a_x_le_3 = smt_mk_lra_le_atom(ctx, &xt, 1, 3, 1);
int unit = -a_x_le_3;
smt_assert_clause(ctx, &unit, 1);                  // x > 3

// Same for y.
SmtLinTerm yt = { y, 1, 1 };
int a_y_le_3 = smt_mk_lra_le_atom(ctx, &yt, 1, 3, 1);
unit = -a_y_le_3;
smt_assert_clause(ctx, &unit, 1);                  // y > 3

// smt_check returns SMT_UNSAT.
```

Two things to notice in the API. First, the LRA assertion functions take explicit coefficients and a right-hand-side as separate numerator/denominator pairs --- the solver is exact-rational throughout; there is no floating-point representation of any constant. Second, there is no "assert greater than"; you make a strict-less-than atom and assert its negation. The solver's atoms only ever face one way (<= or <); the opposite direction is the negated literal in a unit clause. The reasons for this design are explained in chapter 3, but the practical takeaway is that you assemble atoms and then assemble clauses out of them.

1.3. Hello, Nelson-Oppen

`SmtShared` is the API for variables that participate in *both* theories. The classic Nelson-Oppen example: `f(x) != f(y)` and `x = y`, where `x` and `y` are shared variables, is unsatisfiable. EUF receives the equality, merges the classes, congruence on `f` gives `f(x) = f(y)`, and that contradicts the inequality.

```
SmtShared x = smt_mk_shared(ctx, "x");
SmtShared y = smt_mk_shared(ctx, "y");
int fx = smt_mk_app(ctx, "f", &x.euf, 1);
int fy = smt_mk_app(ctx, "f", &y.euf, 1);
smt_assert_neq(ctx, fx, fy);
smt_assert_shared_eq(ctx, x, y);
// smt_check returns SMT_UNSAT.
```

The interesting case is the *other* direction: `x = 5`, `y = 5`, `f(x) != f(y)`. LRA pins both `x` and `y` to the same value, propagates the bridge atoms back to EUF (more on that in chapters 3 and 4), and the contradiction closes again. From the API side it just looks like:

```
SmtLinTerm xt = { x.lra, 1, 1 };
SmtLinTerm yt = { y.lra, 1, 1 };
smt_assert_lra_eq(ctx, &xt, 1, 5, 1);
smt_assert_lra_eq(ctx, &yt, 1, 5, 1);
smt_assert_neq   (ctx, fx, fy);
// UNSAT.
```

2. The shape of the API

There are five kinds of object the user constructs.

`SmtCtx` is the solver state. One per problem instance; you create it, build a formula on it, call `smt_check`, optionally inspect the model, and free it. The solver is one-shot in Phases 1-4: there is no push/pop API and calling `smt_check` twice on the same context is not supported. You make a new context per problem.

Terms (type `SmtTerm`, an opaque integer id) belong to EUF. They are constants made with `smt_mk_const`, or function applications made with `smt_mk_app`. Application terms are hash-consed: `smt_mk_app(f, [a, b])` always returns the same `SmtTerm` no matter how many times you call it. This matters for congruence reasoning --- the solver discovers that `f(a)` and `f(b)` are the same term as soon as it knows `a = b`, and the hash-consing is what makes that discovery a single union-find merge instead of a search.

Variables (type `SmtVar`, also an opaque integer id) belong to LRA. They are real-valued and made with `smt_mk_real_var`. The two namespaces do not overlap; an `SmtTerm` is not interchangeable with an `SmtVar` even if their underlying integer ids happen to match.

Shareds (type `SmtShared`, a struct of `{ euf, lra }`) live in both worlds at once. They are how mixed-sort terms get into the solver: a shared variable can be the argument of an EUF function and at the same time take part in an LRA constraint. The two views are kept consistent by the Nelson-Oppen bridges installed at `smt_mk_shared` time.

Atoms (type `SmtAtom`, again an integer id) are Boolean propositions the SAT layer can hold a truth value for. There are three constructors:

- `smt_mk_eq_atom(a, b)` builds the EUF atom `a == b`.
- `smt_mk_lra_le_atom(terms, n, rhs_num, rhs_den)` builds `sum(c*x) <= r`.
- `smt_mk_lra_lt_atom(...)` builds `sum(c*x) < r`.

Atoms are 1-indexed positive integers. A *literal* is a signed atom: `+atom` for the atom asserted true, `-atom` for it asserted false. Clauses are arrays of literals plus a length. The shape of a unit assertion is therefore just "make an atom, put its id (with sign) in a 1-element array, push the clause."

There are convenience shortcuts that bundle atom creation with a unit-clause assertion: `smt_assert_eq`, `smt_assert_neq`, `smt_assert_lra_le`, `smt_assert_lra_lt`, `smt_assert_lra_eq`, `smt_assert_shared_eq`, `smt_assert_shared_neq`. These are what you write when you do not need the atom for anything else (e.g. you are not also putting it in a larger clause). For any *disjunctive* assertion you build the atoms first and then call `smt_assert_clause` with the resulting literal array.

3. A mental model of what the solver is doing

You do not need to know how the solver works in order to use it on small problems, but the moment something is unexpectedly slow, returns SAT when you expected UNSAT (or vice versa), or just behaves in a way you cannot explain, the mental model pays off. This chapter sketches the model. The companion book has the full version with code; this is the user-facing version: enough to read the statistics and reason about modelling choices, no more.

3.1. The two-level structure

There are two solvers inside, stacked.

At the bottom, **a SAT solver** sees only literals. It has no idea what the atoms mean. It does propositional reasoning --- unit propagation, clause learning, backjumping --- on a flat universe of Boolean propositions.

Above that, **theory solvers** (EUF and LRA in our case) understand what the atoms actually mean and check consistency on whatever subset of atoms is currently true on the SAT trail. When the SAT solver picks a partial assignment, the theory looks at it and reports back: *consistent* (search deeper or declare SAT if the trail is complete), *inconsistent* (here is a clause naming a subset of asserted literals that cannot all be true --- the SAT solver learns this clause as a new constraint), or *I have implications* (here is a literal the theory has derived; please assign it on the trail before deciding further).

That third channel --- theory propagation --- is what powers Nelson-Oppen. Each theory advertises implied literals, the SAT solver pushes them onto the trail, BCP carries them through ordinary clauses, and the next theory sees the consequences. The implementation book describes the channel and its reasons in detail; for modelling purposes, the consequence is simple: **you do not have to manually thread information between theories.** Setting up the right atoms and clauses is enough; the propagation channel does the threading.

3.2. What the statistics mean

After `smt_check` you can ask the context for several counters:

- `smt_decisions` --- how many literals the SAT solver had to guess. - `smt_conflicts` --- how many learned-clause cycles ran. - `smt_theory_conflicts` --- how many of those conflicts came from a theory rather than from BCP itself. - `smt_theory_props` --- how many literals a theory pushed onto the trail in advance of a SAT decision. - `smt_lra_pivots` --- how many simplex pivots the LRA solver did.

These numbers tell you which layer is doing the work. A pure-LRA problem with `decisions = 0` is one the simplex tableau solves directly without any propositional case analysis. An EUF problem with `theory_props > 0` is one where the congruence-closure structure forced equalities the user did not assert. A Nelson-Oppen problem where you see `theory_props` from LRA *and* a `theory_conflict` from EUF is the full back-and-forth working as intended.

If you see a huge number of decisions for a problem you expected to be tractable, that is a signal the disjunctive structure of the formula is larger than the theory side. Reducing disjunctions --- often by making implicit ordering constraints explicit, as in section 4.5 below --- is usually the right response.

3.3. The two LRA atom kinds and what they imply

The LRA front-end has exactly two atom shapes: <= and <. Equality `a = b` is asserted as the *conjunction* of `a <= b` and `b <= a`, internally via two atoms; the convenience function `smt_assert_lra_eq` does this for you. The four polarities of the two atom kinds cover every inequality you might want:

`-+ atom` of <= says `expr <= r` (weak upper bound) `-- atom` of <= says `expr > r` (strict lower bound) `-+ atom` of < says `expr < r` (strict upper bound) `-- atom` of < says `expr >= r` (weak lower bound)

If you want to say `expr >= r`, build the < atom for `expr < r` and assert its negation. This feels backwards at first; with a few problems it becomes second nature, and the symmetry of the encoding pays off in the solver's internals (delta-rationals, see the companion book).

3.4. The four sources of "unexpected"

Roughly four kinds of surprise come up in modelling. Each has a typical cause.

Unexpected SAT. The most common reason is missing a constraint you thought you had. Re-read the formula: every atom you create is *optional* unless you put it in a unit clause or assert it via one of the `smt_assert_*` shortcuts. An atom that exists but is not asserted is just a proposition the SAT solver will freely set true or false.

Unexpected UNSAT. Usually means two constraints are tighter than you realised. A common culprit in LRA modelling is a strict inequality where you meant a weak one (or vice versa). The four-polarity table above is the place to start.

Unexpected slowness. Almost always one of two things. Either there are many disjunctive constraints (non-overlap of N boxes is N*(N-1)/2 four-literal clauses) and the search space is genuinely big; or you have inadvertently asked LRA to handle a constraint that needs SAT case analysis (e.g. a non-convex region encoded as a single inequality). Statistics tell you which: many decisions = SAT-side, many pivots without many decisions = LRA-side.

Unexpected `unknown`. The solver returns this only when you have already called `smt_check` once and called it again on the same context. Make a fresh `SmtCtx`.

4. Modelling patterns

Each section here shows a small pattern with the C code, the formula it expresses, and the typical statistics profile when the solver runs it. The patterns build on each other; the UI worked examples in chapter 5 use them in combination.

4.1. The unit assertion

The simplest atom-and-clause combination is asserting an atom alone:

```
int a = smt_mk_const(ctx, "a");
int b = smt_mk_const(ctx, "b");
smt_assert_eq(ctx, a, b);              // shortcut
```

or equivalently:

```
int atom = smt_mk_eq_atom(ctx, a, b);
int unit = atom;                       // +atom means "atom is true"
smt_assert_clause(ctx, &unit, 1);
```

The shortcuts (`smt_assert_eq`, `smt_assert_lra_le`, …) exist precisely to skip the second form for the common case. Use them by default; reach for the explicit form only when you also want the atom id to put in another clause.

4.2. Negation by sign flip

The two ways to make `a != b` look like this:

```
smt_assert_neq(ctx, a, b);             // shortcut

int atom = smt_mk_eq_atom(ctx, a, b);
int unit = -atom;
smt_assert_clause(ctx, &unit, 1);      // -atom means "atom is false"
```

This is what is happening when you assert `x > 3`: there is no `smt_assert_lra_gt`, because `x > 3` is just `-(x <= 3)`:

```
SmtLinTerm xt = { x, 1, 1 };
int a_x_le_3 = smt_mk_lra_le_atom(ctx, &xt, 1, 3, 1);
int unit = -a_x_le_3;
smt_assert_clause(ctx, &unit, 1);
```

4.3. Equality as two inequalities

`expr == r` is `expr <= r` *and* `expr >= r`. The first is one atom; the second is the negation of `expr < r`. The library provides `smt_assert_lra_eq` that does both:

```
SmtLinTerm xt = { x, 1, 1 };
smt_assert_lra_eq(ctx, &xt, 1, 5, 1);   // x == 5
```

Internally that produces two atoms (`x <= 5` and `x < 5`) and two unit clauses (`+a_le` and `-a_lt`). If you want the equality to be *one* atom of a larger clause (a disjunction like "x == 5 or y == 7"), you have to build the two-atom-and-clause structure by hand. The example in section 5.4 does this.

4.4. Case analysis as a clause

A disjunction is just a clause with more than one literal. The non-overlap constraint between two horizontal rectangles is the canonical example:

```
// A is to the left of B    OR    B is to the left of A.
SmtLinTerm a_minus_b[2] = { { ax, 1, 1 }, { bx, -1, 1 } };
SmtLinTerm b_minus_a[2] = { { bx, 1, 1 }, { ax, -1, 1 } };
int a_left = smt_mk_lra_le_atom(ctx, a_minus_b, 2, -BOX_W, 1);
int b_left = smt_mk_lra_le_atom(ctx, b_minus_a, 2, -BOX_W, 1);
int cl[2] = { a_left, b_left };
smt_assert_clause(ctx, cl, 2);
```

Each branch of the disjunction is one literal in the clause. The SAT solver decides which is true; LRA either confirms or refutes the choice.

4.5. Chains beat pairwise disjunctions

If you know a *total order* on the items being placed --- "B0 is the leftmost, B1 next, then B2..." --- you can replace the pairwise non-overlap clauses with a chain of inequalities:

```
for (int i = 0; i + 1 < N; i++) {
    SmtLinTerm gap[2] = { { x[i], 1, 1 }, { x[i+1], -1, 1 } };
    smt_assert_lra_le(ctx, gap, 2, -(BOX_W + GAP), 1);    // x[i] + BOX_W +
GAP <= x[i+1]
}
```

This is dramatically tighter than the disjunctive encoding: there are no choices for the SAT layer to make, just a sequence of bounds for LRA to enforce. Example 11 (`ex11_ui_row_layout.c`) solves four buttons in a row in 0 decisions and 4 pivots; the equivalent disjunctive formulation has six four-literal clauses and forces the SAT solver to enumerate orderings.

The reverse rule is the same: when you do *not* have a natural total order (a free 2D layout, see ex12) you have to bite the disjunctive bullet. Try to discover orderings in your problem and exploit them.

4.6. Linear equality and alignment

Centering, equal spacing, axis sharing --- all of these are linear equalities and live entirely inside LRA. There is no theory dispatch at all on a problem that consists only of equalities and inequalities.

Vertical centering of a button in a toolbar, for instance:

```
// y == (TOOLBAR_H - BUTTON_H) / 2
SmtLinTerm yt = { y, 1, 1 };
smt_assert_lra_eq(ctx, &yt, 1, (TOOLBAR_H - BUTTON_H) / 2, 1);
```

Equal horizontal spacing between three buttons:

```
// 2 * B1.x == B0.x + B2.x
SmtLinTerm spacing[3] = { { x1, 2, 1 }, { x0, -1, 1 }, { x2, -1, 1 } };
```

```
smt_assert_lra_eq(ctx, spacing, 3, 0, 1);
```

The coefficient field is a (num, den) pair, so fractional coefficients ("two-thirds of the parent width") are not a problem.

4.7. Shared variables across theories

Whenever a variable participates in *both* EUF (as a function argument) and LRA (as a constraint variable), make it a `SmtShared` rather than two separate objects:

```
SmtShared x = smt_mk_shared(ctx, "x");

int fx = smt_mk_app(ctx, "f", &x.euf, 1);
SmtLinTerm xt = { x.lra, 1, 1 };
smt_assert_lra_le(ctx, &xt, 1, 10, 1);
```

The `smt_mk_shared` call installs cross-theory "bridge" atoms (eq, le, ge) and linking clauses for every pair of existing shareds. Those bridges are what carry derived equalities between the two theories. They do nothing unless they fire --- they are purely additional propagation channels --- so the cost is real but bounded. (See chapter 6 for the asymptotic.)

4.8. Reusing atoms across clauses

If the same atom appears in multiple clauses, build it once and reuse the id. The atom-construction functions deduplicate equality atoms (the same `(a, b)` pair returns the same `SmtAtom`), but they do *not* deduplicate LRA atoms. Constructing the same `x + y <= 5` twice gives two atoms; the solver treats them as independent propositions and will not propagate one from the other.

In practice this means: if your problem has natural recurring combinations (think "B[i].x - B[j].x <= -BOX_W" in a non-overlap encoding), store the atom ids and reuse them.

4.9. Soft constraints and preferences

The solver does not have a built-in notion of preference or cost. If two models satisfy the formula, the solver returns whichever it finds first. For "we would prefer this layout but any valid one is acceptable", you have two options:

- **Tighten the bounds.** Replace "x >= 0" with "x >= 5" if 5 is your preferred minimum margin; if that turns out to be infeasible the solver will tell you (UNSAT) and you can relax.

- **Solve incrementally.** With no push/pop API in Phases 1-4, this means making a fresh `SmtCtx` per attempt. It is more verbose but works fine for the kinds of problems this solver is sized for.

A general optimisation layer (MaxSMT, optimisation modulo theories) is not part of the Phase 1-4 implementation.

4.10. Common pitfalls

A short list of mistakes you will probably make at least once.

Forgetting to assert an atom you constructed. Atoms created by `smt_mk_*` are not asserted; they are just registered. They only become constraints when you put them in a clause. The SAT solver may freely choose them either way.

Strict vs weak in the wrong direction. When you negate a `<` atom you get `>=`, not `>`. When you negate a `<=` atom you get `>`, not `>=`. The table in section 3.3 is the canonical reference; print it out.

Mixing namespaces. An `SmtVar` (LRA) cannot be the argument of an EUF function. If you need that, make the variable a `SmtShared` from the start. Trying to retrofit a shared variable in the middle of a problem build is not supported in Phases 1-4.

Integer overflow on coefficients. The internal rational type is int64 numerator over int64 denominator. For tutorial-sized problems this is fine; for adversarial inputs --- a chain of pivots that doubles the numerator each step --- it can overflow. If you are pushing the solver's size, normalise your input coefficients aggressively (divide by gcd, pick the smallest representative) before feeding them in.

5. Worked example: UI component placement

Here is the test problem: given a canvas of width *W* and height *H*, plus a set of UI elements with fixed sizes and a list of layout constraints, find a position for every element such that all constraints are satisfied.

This problem comes up in three places in practice:

1. **Auto-layout engines** in Cocoa and Android. The Cassowary algorithm --- an incremental dual-simplex specialised for soft and required linear constraints --- is the widely-cited reference, and while Apple's Auto Layout is closed-source the two share concepts. The constraint vocabulary is rich (equalities with priorities, multipliers, "greater than or equal" with a target) but fundamentally linear --- no disjunctive non-overlap, no case analysis.

2. **Graph drawing.** Force-directed layout is the popular technique, but for problems where exact placement matters (chip floorplanning, PCB layout, document layout with hard constraints) constraint solvers are used. The classic graphviz `dot` driver uses simplex internally.

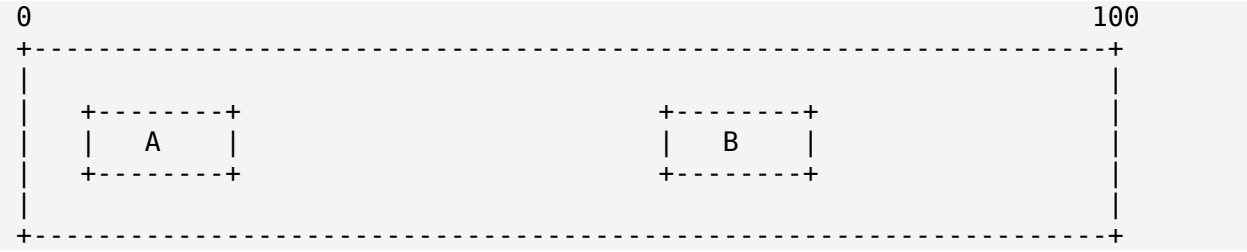
3. **CSS layout.** Flexbox and grid are themselves constraint solvers, hand-coded for speed. They handle the common cases very well and drop to inline expressions for the edge cases.

SMT sits one level above the linear approaches. It can do everything they can (linear inequalities and equalities, alignment, spacing) and also handle the disjunctive constraints they cannot (non-overlap of rectangles, "at least one of these edges aligns", "exactly one of these rows is used"). The cost is a more verbose API and a runtime cost that grows with the disjunctive depth of the problem.

This chapter walks four worked examples from the `examples/` directory. Each is a complete, runnable C program; this text annotates the design choices and reads the statistics.

5.1. Two boxes in a row (`ex10_ui_two_boxes.c`)

The smallest non-trivial layout problem: two rectangles A and B, each 40 wide, in a canvas 100 wide, must not overlap.



The formula has two free variables (the x-coordinates of A and B) and five constraints: A inside the canvas (two bounds), B inside the canvas (two bounds), and the non-overlap disjunction.

The non-overlap is the only piece worth dwelling on. Two horizontal rectangles fail to overlap iff one is entirely left of the other:

A.right <= B.left OR B.right <= A.left

Substituting `right = left + width` and rearranging:

A.x - B.x <= -40 OR B.x - A.x <= -40

That gives a clause with two LRA atoms:

```
SmtLinTerm a_minus_b[2] = { { ax, 1, 1 }, { bx, -1, 1 } };
SmtLinTerm b_minus_a[2] = { { bx, 1, 1 }, { ax, -1, 1 } };
int a_left = smt_mk_lra_le_atom(ctx, a_minus_b, 2, -BOX_W, 1);
int b_left = smt_mk_lra_le_atom(ctx, b_minus_a, 2, -BOX_W, 1);
int cl[2] = { a_left, b_left };
smt_assert_clause(ctx, cl, 2);
```

The solver's output:

```
  canvas width   = 100
  box width      = 40
  result         = SAT (valid layout exists)
  decisions      = 1
  theory conf.   = 0
  LRA pivots     = 2
```

One SAT decision (the disjunction had to be resolved one way or the other), no theory conflicts (the chosen branch was feasible), two LRA pivots to find the actual values. The two coefficients `+1` and `-1` on each atom are what put `B.x` and `A.x` in the tableau; the pivots position them.

5.2. Four buttons in a row (`ex11_ui_row_layout.c`)

If the layout is naturally ordered --- "B0 then B1 then B2 then B3" --- the disjunctive non-overlap can be replaced with a chain of inequalities. This is the single most important optimisation in SMT-based UI layout. For a row of *N* buttons, the disjunctive form has *N(N-1)/2* clauses of four literals each (for 2D) or two literals each (for 1D); the chain form has *N-1* clauses with one literal each.

```
for (int i = 0; i + 1 < N; i++) {
    SmtLinTerm gap[2] = { { x[i], 1, 1 }, { x[i+1], -1, 1 } };
    smt_assert_lra_le(ctx, gap, 2, -(BOX_W + GAP), 1);
}
```

The constraint says `x[i] + BOX_W + GAP <= x[i+1]` for each adjacent pair: B[i] ends, a gap of GAP follows, then B[i+1] starts.

Plus the row-fits-canvas constraint:

```
SmtLinTerm last = { x[N-1], 1, 1 };
smt_assert_lra_le(ctx, &last, 1, CANVAS_W - BOX_W, 1);
```

And the row-starts-at-or-after-zero:

```
SmtLinTerm neg0 = { x[0], -1, 1 };
smt_assert_lra_le(ctx, &neg0, 1, 0, 1);
```

Output:

```
  buttons        = 4
  result         = SAT (valid layout)
  decisions      = 0 (expect 0: pure chain, no disjunction)
  theory conf.   = 0
  LRA pivots     = 4
```

Zero decisions. The SAT layer has nothing to do: every clause is a unit clause and BCP places every literal on the trail before any decision is needed. LRA does four pivots to actually position the buttons.

The price of the chain form is that you have committed to an order. If the row has to lay out elements whose left-to-right order is unknown --- say, dragging a column reorders them --- the chain form does not fit and you need the disjunctive encoding.

5.3. Four free 2D boxes (`ex12_ui_grid_placement.c`)

The general case. Four 40x30 boxes in a 100x100 canvas, no fixed ordering, no overlap. Each pair contributes a four-literal clause: A is left of B, or B is left of A, or A is above B, or B is above A.

```
for (int i = 0; i < N; i++) {
    for (int j = i + 1; j < N; j++) {
        /* ... build left, right, above, below atoms ... */
        int cl[4] = { left, right, above, below };
        smt_assert_clause(ctx, cl, 4);
    }
}
```

Six pairs, six four-literal clauses, twenty-four atoms plus the box-in- canvas bounds. The solver:

```
  canvas         = 100 x 100
  box            = 40 x 30
  pairs          = 6 (each: 4-literal non-overlap clause)
  result         = SAT (valid layout)
  atoms          = 40
  decisions      = 31
  theory conf.   = 6
  conflicts      = 6
```

31 decisions, 6 theory conflicts. The SAT solver enumerates orderings; some of those are infeasible (the boxes do not fit in that order) and LRA learns clauses ruling them out. The solver finds a valid layout after the sixth backjump.

If you want every box positioned in a specific *region* of the canvas ("box 0 is in the top-left quadrant, box 1 in the top-right..."), add bounds to constrain each box's coordinates and the search collapses -- the inequalities are essentially picking the order for the SAT layer.

5.4. A real toolbar (`ex13_ui_toolbar.c`)

Three icon buttons (24x24) in a horizontal toolbar (240x40) with the constraints any reasonable toolbar imposes:

1. Buttons fit inside the toolbar (with 8px padding). 2. Buttons don't overlap. 3. Buttons are vertically centered (y = 8 for each). 4. Equal horizontal spacing: gap(B0, B1) = gap(B1, B2). 5. Left-to-right ordering: B0 left of B1 left of B2.

Vertical centering (constraint 3) is the cleanest: a linear equality.

```
for (int i = 0; i < 3; i++) {
    SmtLinTerm yt = { ys[i], 1, 1 };
    smt_assert_lra_eq(ctx, &yt, 1, (TOOLBAR_H - ICON) / 2, 1);
}
```

The ordering (5) replaces the disjunctive non-overlap with a chain:

```
SmtLinTerm chain01[2] = { { x0, 1, 1 }, { x1, -1, 1 } };
SmtLinTerm chain12[2] = { { x1, 1, 1 }, { x2, -1, 1 } };
smt_assert_lra_le(ctx, chain01, 2, -ICON, 1);
smt_assert_lra_le(ctx, chain12, 2, -ICON, 1);
```

And the equal-spacing (4) is one linear equality:

```
// gap(B0,B1) == gap(B1,B2)  <=>  B1.x - (B0.x + ICON) == B2.x - (B1.x +
ICON)
// <=> 2 B1.x - B0.x - B2.x == 0
SmtLinTerm spacing[3] = { { x1, 2, 1 }, { x0, -1, 1 }, { x2, -1, 1 } };
smt_assert_lra_eq(ctx, spacing, 3, 0, 1);
```

Output:

```
  toolbar        = 240 x 40
  icon           = 24 x 24
  padding        = 8
  result         = SAT (valid layout)
  decisions      = 0
  theory conf.   = 0
  LRA pivots     = 5
```

Zero decisions. The toolbar has structure: alignment is a linear equality, ordering is a chain of inequalities, equal-spacing is a linear equality. None of those need SAT case analysis. The whole problem fits inside the simplex tableau and the solver dispatches it in five pivots.

That is the take-home message of this chapter. When you can express structural design intent --- alignment, ordering, equal-spacing, centering --- as linear equalities, the solver does very little propositional work. Reach for disjunctive non-overlap only when the problem genuinely is not ordered (free placement, drag-and-drop, etc.).

6. What the solver can and cannot do

A solver this small is a tool with sharp edges. Use it for what it is good for; reach for something else when the problem is outside its range.

6.1. Inside the comfortable range

- **Tens to low hundreds of variables.** All the worked examples in this book fall well below the upper bound. EUF problems with a few dozen terms and a few hundred atoms run in milliseconds. LRA problems of similar size run in a small number of pivots.

- **Convex linear arithmetic.** LRA in the Dutertre--de~Moura form is a clean, fast solver for the convex linear-rational problems UI layout, network flow, and basic resource allocation produce. Mixed EUF / LRA via Nelson-Oppen handles formulas where the two interact naturally.

- **Boolean structure over theory atoms.** This is the *defining* advantage of SMT over plain LP: disjunctions of inequalities, conditional constraints ("if this Boolean variable is true then this LRA constraint applies"), case-by-case design rules. The SAT layer handles the case analysis; the theory handles each case.

- **Exact rational arithmetic.** No floating-point rounding. Two rationals you compute will compare exactly equal if they mathematically are, regardless of the path you took to compute them.

6.2. Outside the comfortable range

- **Integer arithmetic.** The solver handles linear *rational* arithmetic; integer variables would require linear integer arithmetic (LIA), which is non-convex (the equality `x = y/2` has integer solutions only for some values of `y`). Nelson--Oppen for LIA needs case-splitting inside the theory and is a substantially larger build. If you need integers, model them as rationals plus a "good enough" rounding pass on the output, or step up to a production solver like cvc5 or Z3.

- **Optimisation.** The solver returns a satisfying assignment if one exists, but it does not pick the *best* among many. MaxSMT (find an assignment maximising the number of satisfied soft clauses) and OMT (optimisation modulo theories) are extensions you can read about in the literature but they are not in this implementation.

- **Very large formulas.** The int64 rational backing store, the eager Nelson--Oppen bridge atoms (quadratic in the shared-variable count), and the dense tableau representation all rule out problems in the thousands-of-variables range. The solver is a tutorial artefact and is sized accordingly.

- **Non-linear arithmetic.** Multiplying two variables is outside LRA. There are SMT solvers for non-linear real arithmetic (the NRA theory), but they are vastly more complex than LRA and not in this implementation.

- **Quantifiers.** Everything in this solver is quantifier-free. The common quantified extensions (E-matching for instantiating quantified EUF axioms, MBQI for model-based quantifier instantiation) are not here either.

6.3. Performance signposts

Some rough numbers from the examples in this book, running on a 2024-era laptop:

- Pure chains of LRA inequalities (ex11, ex13): well under a millisecond. - Disjunctive non-overlap of N rectangles (ex12): scales roughly with N^2 atoms and an exponential worst-case for the SAT search, though the actual cases that arise in UI layout are mild. - Mixed EUF + LRA via Nelson--Oppen with a handful of shared variables (ex8, ex9): also sub-millisecond.

If a problem you have is taking unreasonably long, the statistics will tell you why. Many decisions = the SAT layer is enumerating; many pivots without many decisions = the simplex is doing work; many theory propagations = the cross-theory bridges are firing. Each suggests a different remediation: reduce disjunctions, simplify the linear part, or reduce the number of shareds, respectively.

6.4. When to reach for a different tool

- **For pure linear constraints with priorities** (typical UI auto- layout): use Cassowary or any LP solver. It will be faster and the API is closer to what you want.

- **For pure non-overlap of many rectangles** (chip placement, dense UI grids): a specialised geometric algorithm (corner stitching, binary space partitioning) will beat SMT decisively at the size where SMT starts to struggle.

- **For mixed linear + Boolean + integer** at industrial scale: cvc5 or Z3. Both are production-grade and handle every theory mentioned in this book and many more, with SMT-LIB v2 as the input language.

The reason to use the solver built across Phases 1-4 is the same as the reason to read the implementation book: to understand exactly what is happening, to be able to modify the algorithm when your problem has exotic structure, and to have something small and dependency-free for embedding in another C codebase. For workloads at that scale and shape, it is the right tool.

7. Further reading

- **The companion book**, *Satisfiability Modulo Theories*, *Phases 1-4*. The implementation side of everything in this guide.

- Daniel Kroening and Ofer Strichman, *Decision Procedures: An Algorithmic Point of View*, 2nd ed., Springer 2016. The standard textbook treatment of EUF, DPLL(T), and combined theories.

- Bruno Dutertre and Leonardo de Moura, "A Fast Linear-Arithmetic Solver for DPLL(T)," CAV 2006. The primary reference for the LRA algorithm in this solver.

- Greg Badros, Alan Borning, and Peter Stuckey, "The Cassowary Linear Arithmetic Constraint Solving Algorithm," ACM TOCHI 8(4), 2001. The linear-arithmetic solver behind Auto Layout-style UI engines. Reading this alongside Dutertre and de Moura is the fastest way to understand the design space of solvers for layout-like problems.

- The source of cvc5 (https://cvc5.github.io) and Z3 (https://github.com/Z3Prover/z3). The reference implementations.

- Clark Barrett, Pascal Fontaine, and Cesare Tinelli, *The SMT-LIB Standard*, *Version 2.6*, 2017. The input format you would target if you wanted to add a parser to this solver.

Solving problems with LCG

A user's guide to Lazy Clause Generation

Preface

This guide is the user-facing companion to *Lazy Clause Generation, Phases 1-4*. The implementation book describes how the solver was built; this one describes how to use it.

It covers the constraint vocabulary the library offers, the modelling decisions you face when reaching for LCG, and two worked examples: the canonical N-Queens puzzle, and the slightly less canonical case of scheduling air-defence engagements through a multi-channel fire-control radar. It closes with a comparison of CSP/DLX, MILP, and LCG as choices for dynamic weapon-target assignment, since that is the kind of problem the library was written for in the first place.

The book assumes you have used a SAT or CP solver before, can write a few lines of C, and want a quick path from "I have a scheduling problem" to "I have a schedule".

— Stockholm, May 2026

Introduction

Lazy Clause Generation is a hybrid of constraint programming and CDCL SAT. It looks like CP from the outside — you declare integer variables, post constraints, ask for a model — and it looks like SAT on the inside, with clause learning and conflict-driven backjumping doing the heavy lifting on hard instances.

The "lazy" part is what distinguishes it from a pure SAT encoding. A textbook SAT encoding of, say, an `alldifferent` constraint over five variables in a domain of size ten produces hundreds of pairwise inequality clauses up front. LCG never writes those clauses. Instead, when a propagator decides that some bound has to tighten, it constructs an explanation clause *on demand* describing exactly why. That clause enters the SAT engine's learnt database, where it cuts off similar dead ends elsewhere in the search tree. The clauses you never need are never generated, and the ones you do need are exactly the ones the solver was about to derive anyway.

This positioning is the reason to reach for LCG: it gives you global constraints (alldifferent, cumulative, no_overlap) with the propagation strength of CP, but with CDCL learning underneath. Pure SAT lacks the propagators; pure CP lacks the learning.

You should not reach for LCG when the problem is naturally linear with a clear objective — that is MILP territory. You should not reach for it for pure combinatorial exact cover — that is DLX territory. You should reach for it when the problem mixes combinatorial structure with bound reasoning and time windows, especially when you expect to revisit the same subproblem many times in a search.

Quick start

A complete program that finds two integers in `[0, 9]` summing to at most 5, with the first at least 2:

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

int main(void) {
    LcgCtx *ctx = lcg_new();
    IntVar x = lcg_new_int_var(ctx, 0, 9, "x");
    IntVar y = lcg_new_int_var(ctx, 0, 9, "y");

    int    c[2] = { 1, 1 };
    IntVar v[2] = { x, y };
    lcg_post_linear_le(ctx, c, v, 2, 5);    /* x + y <= 5 */

    int unit = lcg_lit_ge(ctx, x, 2);
    lcg_post_clause(ctx, &unit, 1);         /* x >= 2 */

    if (lcg_solve(ctx) == LCG_SAT) {
        printf("x = %d, y = %d\n",
            lcg_value(ctx, x), lcg_value(ctx, y));
    }
    lcg_free(ctx);
    return 0;
}
```

Build it with:

```
gcc -std=c99 -O2 -Isrc your_program.c src/sat.c src/lcg.c \
    src/prop_alldiff.c src/prop_cumulative.c -o your_program
```

That is essentially what `build.zsh` in the distribution does, and what every example file in `examples/` is built with.

Integer variables and the order encoding

Every variable in LCG is a finite-domain integer with a fixed `[lo, hi]` range:

```
IntVar x = lcg_new_int_var(ctx, 0, 9, "x");
```

Internally, the library represents the domain through an *order encoding*: one Boolean atom per threshold, where `b[k]` means "x ≤ k". A monotonicity chain `b[k] → b[k+1]` ensures the encoding stays consistent. You never have to think about this — the encoding is the library's, not yours — but it has two consequences worth knowing about.

The first is that domains have to be finite and small-ish. A variable with domain `[0, 1000]` allocates a thousand SAT atoms. The library will handle it, but at some point the SAT engine's clause database grows enough that you would have done better with a coarser model.

The second is that the encoding is *bound-oriented*. Asserting "x ≤ 5" is one literal; asserting "x ≠ 7" is two literals (the disjunction "x ≤ 6 OR x ≥ 8"). Constraints that turn on individual values can be expressed but are not native — and global constraints that reason about individual values (domain-consistent `alldifferent`, for example) are not available in the current library. Reasoning about *intervals* — bounds, ranges, time windows — is what LCG does best.

The literal helpers expose the encoding directly when you need them:

```
int unit = lcg_lit_le(ctx, x, 5);     /* literal for "x <= 5" */
int unit = lcg_lit_ge(ctx, x, 3);     /* literal for "x >= 3" */
```

These return integer literals you can pass to `lcg_post_clause`. The two special return values `LCG_LIT_TRUE` and `LCG_LIT_FALSE` indicate the requested relation is trivially true or trivially false given the variable's domain, so the caller can short-circuit.

During and after a solve you can query current bounds and the final value:

```
int lo = lcg_lb(ctx, x);     /* current lower bound */
int hi = lcg_ub(ctx, x);     /* current upper bound */
int v  = lcg_value(ctx, x);  /* the model's value (after LCG_SAT) */
```

The bounds accessors are live during search — propagators use them to look at where each variable currently sits.

Linear constraints

The most common building block is the linear inequality:

```
void lcg_post_linear_le(LcgCtx *ctx,
                        const int *coeff, const IntVar *vars, int n_terms,
                        int k);
```

This posts the constraint Σ `coeff[i] · vars[i]` $\leq$ `k`. Coefficients are integers; negative coefficients are fine. To encode an equality you post two inequalities:

```
/* x - y == 3, encoded as two inequalities */
int     c1[2] = {  1, -1 };
int     c2[2] = { -1,  1 };
IntVar  v [2] = { x, y };
lcg_post_linear_le(ctx, c1, v, 2,  3);
lcg_post_linear_le(ctx, c2, v, 2, -3);
```

To encode Σ `a_i x_i` $\geq$ `k`, negate everything: Σ `(-a_i) x_i` $\leq$ `-k`.

The propagator behind `linear_le` does bounds reasoning. It computes the minimum and maximum possible value of the left-hand side from the current bounds of every term, detects infeasibility when the minimum exceeds `k`, and tightens individual variables when the slack room shrinks.

The explanations are surgical: when the propagator pushes `x` $\leq$ `4` because the other variables' current bounds leave only that much room, the clause it generates cites *exactly* the other variables' relevant bound literals, nothing more. The clause goes into the SAT engine's learnt database and contributes to nogood learning across the rest of the search.

There is no built-in `lcg_post_linear_ge`, `lcg_post_linear_eq`, or `lcg_post_linear_lt`. They are all expressible through `linear_le` with sign flips and offsets. If you find yourself writing the same transformation often, wrap it.

The alldifferent constraint

```
void lcg_post_alldifferent(LcgCtx *ctx, const IntVar *vars, int n);
```

This enforces that the `n` variables take pairwise distinct values. The propagator is *bounds-consistent*: it detects when a set of variables is squeezed into too few values (a Hall-set violation) and tightens bounds when an interval of values is fully claimed by some subset of the variables.

What "bounds-consistent" means concretely: the propagator catches cases where some `k` variables all have their domains contained in some `k`-element value interval, and forbids any other variable's domain from overlapping that interval *from the outside*. It does not remove individual values from the middle of a domain. Domain-consistent `alldifferent` (Régin's matching-based algorithm) would do that, but it requires per-value SAT atoms beyond what the order encoding supports, and is not in this library.

The practical implication: `alldifferent` is strongest when the variables' bounds are already tight against each other. It is weak when domains overlap loosely. In a problem like N-Queens — where the `alldifferent`s are over variables with the same wide domain — the propagator mainly contributes when search has narrowed bounds enough that Hall sets emerge.

Example: the smallest non-trivial use is a permutation puzzle. Three variables in `[1, 3]`, all different, must form a permutation of `{1, 2, 3}`:

```
IntVar x = lcg_new_int_var(ctx, 1, 3, "x");
IntVar y = lcg_new_int_var(ctx, 1, 3, "y");
IntVar z = lcg_new_int_var(ctx, 1, 3, "z");
IntVar vs[3] = { x, y, z };
lcg_post_alldifferent(ctx, vs, 3);
```

Combined with a sum constraint or a couple of unit assertions, this nails down a specific permutation.

Cumulative scheduling

```
void lcg_post_cumulative(LcgCtx *ctx,
                         const IntVar *starts,
                         const int    *dur,
                         const int    *demand,
                         int n_tasks,
                         int capacity);
```

This is the most powerful constraint in the library. It says: you have `n_tasks` tasks. Each task `i` has a start time variable `starts[i]`, a fixed integer duration `dur[i]`, and a fixed integer resource demand `demand[i]`. The task occupies the half-open interval `[starts[i], starts[i] + dur[i])`. At no point in time may the total demand from all running tasks exceed `capacity`.

The propagator uses *time-table reasoning*. For each task, it computes the *mandatory part* — the time the task is guaranteed to be running across all currently valid start values. For a task with `lb(s) = 2`, `ub(s) = 4`, and duration 3, the three possible starts cover `[2, 5)`, `[3, 6)`, and `[4, 7)`; the task is guaranteed to run at time 4. The mandatory part is `[ub(s), lb(s) + dur)`, non-empty exactly when `ub(s) − lb(s) < dur`.

Summing demand from all mandatory parts gives a per-time-point demand profile. If that profile exceeds capacity at any time, the constraint is infeasible right now and the propagator emits a conflict. If scheduling some specific task `j` at a particular start would push the demand at some time over capacity, the propagator tightens `j`'s bounds to skip past it.

`no_overlap` is the special case where every task demands 1 and the capacity is 1:

```
void lcg_post_no_overlap(LcgCtx *ctx,
                         const IntVar *starts,
                         const int    *dur,
                         int n_tasks);
```

This is the classic unary-resource constraint: a single machine, a single fire-control channel, a single human-in-the-loop.

The library implements only time-table propagation. There are stronger algorithms (edge-finding, energetic reasoning, the Aggoun-Beldiceanu sweep) that catch more conflicts earlier, but they are significantly more code. For tutorial-scale problems and most real applications under a few dozen tasks, time-table is enough.

Custom propagators

If the built-in vocabulary does not cover what you need, you can write your own propagator and register it. The interface is small:

```
typedef struct {
    void *data;
    int  (*propagate)(LcgCtx *ctx, void *data);
    void (*destroy)(void *data);
} LcgPropagator;

void lcg_register_propagator(LcgCtx *ctx, LcgPropagator p);
```

Your `propagate` function inspects the current bounds via `lcg_lb` and `lcg_ub`, builds an explanation through the helpers `lcg_expl_clear`, `lcg_expl_lo`, `lcg_expl_hi`, and emits derived bounds via `lcg_emit_le`, `lcg_emit_ge`, or `lcg_emit_conflict`. The return value is `LCG_PROP_OK`, `LCG_PROP_TIGHTENED`, or `LCG_PROP_CONFLICT`.

The example `examples/ex2_custom_prop.c` implements a `not_equal(x, k)` propagator in under fifty lines using this API. The two built-in global propagators (`alldifferent` and `cumulative`) use the same API — there is nothing private about it. If a project ends up writing two or three custom propagators, it is using the library the way it is meant to be used.

Worked example: N-Queens

Place `N` queens on an `N × N` board so no two attack. The classical CP encoding uses one integer variable per row: `col[i]` is the column of the queen on row `i`, in `[0, N-1]`.

Three families of attacks need to be ruled out: shared columns, shared `/` diagonals, and shared `\` diagonals. Shared columns become `alldifferent(col_0, ..., col_{N-1})`. Shared `/` diagonals become `alldifferent(col_0 + 0, col_1 + 1, ..., col_{N-1} + (N-1))` — two queens are on the same `/` diagonal iff their row + column sums are equal. Similarly for `\` diagonals with row − column.

LCG does not let you put an arbitrary expression into an `alldifferent`; the propagator wants integer variables. So we introduce auxiliary variables `d1[i] = col[i] + i` and `d2[i] = col[i] - i`, linked through linear equalities:

```
IntVar make_offset_var(LcgCtx *ctx, IntVar src, int offset,
                       int dom_lo, int dom_hi, const char *name)
{
    IntVar aux = lcg_new_int_var(ctx, dom_lo, dom_hi, name);
    int     c1[2] = {  1, -1 }; IntVar v[2] = { aux, src };
    int     c2[2] = { -1,  1 };
    lcg_post_linear_le(ctx, c1, v, 2,  offset);
    lcg_post_linear_le(ctx, c2, v, 2, -offset);
    return aux;
}
```

Two `linear_le`s encode the equality. Then three `alldifferent`s express the queen constraints, and the solver does the rest. The full example in `examples/ex3_nqueens.c` solves 8-Queens in around 30 decisions on a fresh search.

The pattern — globals over auxiliary integer variables linked by linear equalities — is the workhorse of modelling in LCG. Almost every interesting problem ends up looking like it.

Worked example: DWTA scheduling

Six inbound threats are tracked. Each has an arrival time (when it enters the engagement envelope), a deadline (when it impacts the ship if not engaged), and an engagement duration (how long an interceptor needs from launch through intercept). The ship has a single fire-control radar with two illumination channels — every active engagement holds one channel for its full duration.

The doctrine: every threat must be engaged, completely within its `[arrival, deadline)` window, and the radar's two-channel capacity must never be exceeded.

The natural LCG model is a single cumulative constraint:

```
typedef struct {
    const char *id, *kind;
    int arrival, deadline, duration;
} Threat;

Threat threats[] = {
    { "T0", "drone-A",         0,  4,  2 },
    { "T1", "cruise-missile",  0,  5,  3 },
    { "T2", "drone-B",         2,  6,  2 },
    { "T3", "anti-ship",       3,  8,  3 },
    { "T4", "drone-C",         4,  8,  2 },
    { "T5", "cruise-missile",  5, 10,  3 },
};

for (int i = 0; i < N; i++) {
    start[i] = lcg_new_int_var(ctx, threats[i].arrival,
                               threats[i].deadline - threats[i].duration,
                               threats[i].id);
    du[i] = threats[i].duration;
    de[i] = 1;     /* one channel per engagement */
}
lcg_post_cumulative(ctx, start, du, de, N, /*capacity=*/2);
```

That is the whole model. One start-time variable per threat, one duration, one demand, one capacity. The propagator works out the rest. On the six-threat scenario the solver lands on a schedule in seven decisions:

```
    t :   0  1  2  3  4  5  6  7  8  9
    T0    .  .  #  #  .  .  .  .  .  .
    T1    .  #  #  #  .  .  .  .  .  .
    T2    .  .  .  .  #  #  .  .  .  .
    T3    .  .  .  .  #  #  #  .  .  .
    T4    .  .  .  .  .  .  #  #  .  .
    T5    .  .  .  .  .  .  .  #  #  #
    Σ     0  1  2  2  2  2  2  2  1  1     (cap = 2)
```

Every threat is engaged inside its window, and the channel load rides at the cap of 2 for most of the timeline — the defence is fully committed but not over-committed. Drop the capacity to 1 with the same threat picture and the solver returns UNSAT at the root with one conflict and zero decisions: time-table reasoning sees the over-saturation before any branch is needed.

What is *not* in this model: kill probabilities, weapon-target affinity, magazine depth, or any optimization. Those belong to a different layer of the system. The cumulative model answers one specific question — *given that we have decided to engage these threats with these durations, when do the engagements happen?* — and answers it well. The next section is about which layer of the system that question belongs to.

Choosing your solver: CSP/DLX, MILP, or LCG

DWTA is not a single problem. It is a family of related problems — static or dynamic, with or without time windows, with or without uncertain kill probabilities, with or without sensor-pointing constraints, with or without multi-platform coordination. The right solver depends on which sub-problem of the family is in front of you.

This section walks through the three paradigms one at a time, identifies the DWTA sub-problems each is best at, and then describes the hybrid patterns that real operational systems use.

CSP with DLX / Dancing Cells

Pure constraint-satisfaction with backtracking — the Knuth toolkit of sparse sets and Dancing Links — wins where the problem has clean combinatorial structure and small enough scale that smart pruning beats clause learning.

Strengths for DWTA. When the problem reduces to *exact cover* — each target is engaged by exactly one weapon, each weapon has a hard magazine constraint, and there are no time-window subtleties — DLX is hard to beat. The structure of Dancing Cells in particular handles the "and also these resource constraints" extensions naturally. The data structures are cache-friendly and the propagation is essentially free. For modest-sized assignment subproblems within a larger pipeline, DLX is the right choice.

It is also the right choice when you want *all* feasible assignments, not just one. CDCL-based solvers can be coerced into enumeration, but it is awkward and slow; DLX enumerates as a natural mode of operation.

Weaknesses for DWTA. No clause learning. The solver visits the same dead end as many times as the search tree's structure leads it there. For hard combinatorial instances — especially when the constraint set has loose pairwise interactions that only manifest globally — this matters.

No arithmetic. Linear combinations of variables, weighted sums, capacities expressed as inequalities — these are all bolted on rather than native. Probability scoring (`p_kill_total = 1 − ∏(1 − p_ij)`) is awkward.

No native scheduling primitives. Time windows and cumulative resources can be expressed as additional constraints but the propagation is generic, not specialized.

DWTA verdict. Use DLX for the inner combinatorial subproblem of "which weapons can cover which targets given hard binary feasibility constraints", especially when you need enumeration or the subproblem is called many times inside a larger loop. Do not use it for the outer optimization or for time-window scheduling.

MILP

Mixed-integer linear programming is the standard tool for *static* WTA: take a frozen snapshot of the threat picture, decide an optimal assignment, hand the schedule downstream.

Strengths for DWTA. Natural optimization. The MILP formulation has an explicit objective — minimize expected leakage, maximize protected value, minimize engagement cost — and the solver returns a provably optimal solution (or a bounded approximation). Commercial solvers like Gurobi and CPLEX, and good open-source ones like SCIP and HiGHS, throw enormous effort at this exact problem shape.

Linear arithmetic everywhere. Resource budgets, expected-value computations, cost minimization, magazine constraints — all natural. Kill probabilities enter through log-space transformations (linearizing the product `∏(1 − p_ij)` via logs is standard).

LP relaxation as a side benefit. Even when integer feasibility is hard, the LP relaxation gives a lower bound, dual prices for resource constraints (telling you which weapons are bottlenecks), and warm-start information for re-optimization as the picture evolves.

Weaknesses for DWTA. Time-window scheduling is awkward. Encoding "task i runs in some contiguous duration-d interval that starts within `[arr_i, dl_i - d]`" requires either big-M constraints (numerically painful) or time-indexed binary variables (`y_{i,t}` = is task i running at time t?), which blow up the variable count and weaken the LP relaxation.

Cumulative resources require disaggregated time-indexed variables to model accurately. A scheduling problem that LCG expresses in one constraint over `n` integer variables takes hundreds of binary variables in MILP. The LP relaxation of these "discretized" formulations is often very loose.

No combinatorial structure exploitation. The solver does not "know" that an assignment is a permutation, or that a set of variables forms an alldifferent — it sees only the linear inequalities. The branch-and-bound can be slow on problems where CP propagation would prune aggressively.

DWTA verdict. Use MILP for the static WTA optimization — *which weapons engage which targets, with what expected outcomes*. Do not use it for engagement scheduling unless the time discretization is coarse and the number of tasks is small.

LCG

LCG sits between the two, combining CP's propagation with SAT's clause learning.

Strengths for DWTA. Native scheduling. Time-windowed engagements with cumulative or unary fire-control resources are exactly what `cumulative` and `no_overlap` are designed for. The propagator detects over-saturation immediately when the picture cannot be scheduled within the resource budget.

Native `alldifferent`. "Each weapon launches at most one engagement at this time slot" or "every magazine slot is used at most once" express naturally. The bounds-consistent version handles the common case of disjoint time windows; for tighter pruning, write a custom propagator.

Clause learning across the search. When a branch fails, the failure encodes as a CDCL nogood that cuts off similar branches everywhere else. This is the single most important advantage over plain CP backtracking on hard scheduling instances.

Custom propagators are first-class. A doctrinal rule like "shoot-look-shoot requires the second engagement to start at least `delta` time units after the first one's outcome window" can be expressed as a custom propagator without leaving the framework.

Weaknesses for DWTA. No native optimization. LCG answers feasibility questions; "best schedule" requires wrapping it in branch-and-bound on the objective (post `objective ≤ best_known`, solve, lower bound, repeat). This works but is more work than MILP.

Less mature than MILP solvers. Decades of MILP tuning, primal heuristics, and presolve transformations have no counterpart in any LCG implementation. A clever MILP formulation will often outperform an obvious LCG one on the same data.

Order encoding limits some constraint shapes. Constraints expressed naturally over individual values rather than bounds (such as table constraints and domain-consistent alldifferent) are not as efficient as the bound-oriented ones.

DWTA verdict. Use LCG for engagement scheduling once allocation is decided — *when do the engagements happen given the fire-control resources*. Use it for tight combinatorial subproblems with time windows. Do not use it alone for static WTA optimization with rich probabilistic objectives.

Comparison summary

Concern	DLX/CSP	MILP	LCG
Exact-cover allocation	strong	medium	medium
Optimal allocation with probabilities	weak	strong	weak
Time-window scheduling	weak	weak	strong
Cumulative resources	weak	medium	strong
Custom doctrinal rules	medium	weak	strong
Enumeration of solutions	strong	weak	weak
Re-solve under perturbation	weak	strong	medium
Maturity and tuning	medium	strong	weak

Combining multiple technologies

Real operational DWTA systems do not pick one paradigm. They decompose the problem into sub-problems and use the right tool for each, coupling them through one of several standard patterns.

Two-phase allocation-then-scheduling

The simplest decomposition: an MILP master picks the assignment of weapons to targets, optimizing expected leakage or protected-value. A CP/LCG sub-problem checks whether the chosen assignment is schedulable through the fire-control channels within the engagement windows. If yes, done. If no, drop the infeasible assignment as a cut in the MILP and re-solve.

```
   MILP allocation  --(assignment)--> LCG scheduler
        ↑                                  |
        |                           (feasible? yes/no)
        +----------(if no, cut)------------+
```

This is *logic-based Benders decomposition* in the lingo. The cuts are problem-specific: the simplest cut is "this exact assignment is infeasible", but tighter cuts ("any assignment that uses these three weapons in this time window is infeasible") cut more search space.

The pattern works well when the allocation has rich objective structure (so MILP is the right tool for the master) and scheduling has rich combinatorial structure (so LCG is the right tool for the subproblem). DWTA fits the pattern naturally.

Column generation and pricing

A more elaborate pattern: the master problem chooses among pre-computed *schedules* (each one a full feasible weapon-target-time triple set), optimizing the objective over a convex combination. A pricing subproblem generates new candidate schedules whose dual prices suggest they would improve the master. The subproblem is naturally a CP/LCG problem; the master is an LP or MILP.

Column generation shines when the number of possible schedules is enormous but only a few are ever active in the optimal solution. The DWTA case fits when the engagement structure has natural decomposition (e.g., per-weapon or per-time-window).

Rolling-horizon re-solve

DWTA is fundamentally dynamic: threats arrive, engagements complete (or fail), and the decision must be re-made every few seconds. Most operational systems use a rolling horizon: solve the next-K-seconds problem with whatever method, execute the first few decisions, then re-plan with updated state.

The solver choice inside the rolling horizon is independent of the rolling-horizon structure itself. MILP with warm-start is one option. CP with persistent learning (passing learnt clauses across solves) is another. The choice depends on which sub-problem dominates the wall-clock budget.

Hybrid in one solver

The most elaborate option: a single solver that mixes integer programming and constraint propagation natively. Commercial tools like CPLEX and Gurobi have rudimentary forms of this (callbacks for cut generation that can run arbitrary code); academic tools like SICStus-Prolog, Choco, and OR-Tools' CP-SAT have richer integration. CP-SAT in particular is essentially a productionized LCG with optimization and a much richer constraint library, and is what most modern operational scheduling pipelines use when they need both CP and optimization in one solver.

For a from-scratch DWTA simulator, the two-phase pattern is usually the right starting point. It is simple to implement, lets you pick the best tool for each sub-problem, and has a clear contract between the layers. Move to column generation or hybrid solvers only when the two-phase decomposition becomes the bottleneck.

Tips and limitations

A few things worth knowing once you start using the library on real problems.

Domain size matters. The order encoding allocates one SAT atom per threshold in each variable's domain. A handful of variables with domains in the hundreds is fine. Hundreds of variables with thousands of values each is not — you'll be better served by a different model (perhaps with auxiliary "bucket" variables) or a different tool.

Bounds are stronger than values. If you can express your constraint in terms of "this variable's bound" rather than "this variable's individual values", the propagator will be happier. The library reasons natively about bounds; individual-value constraints work but are auxiliary.

Linear equalities cost two posts. Every `a · x = b · y + c` is two `linear_le` calls (in opposite directions). The propagator handles them efficiently, but if you have many of them, the constraint count adds up. For a long chain like `aux = src + offset`, consider whether you really need the auxiliary variable or whether you can rewrite later constraints to use the source directly.

One shot per context. The library is one-shot: build an `LcgCtx`, post constraints, call `lcg_solve` once, free. It is not incremental. To re-solve with one more constraint, build a fresh context. (Rolling-horizon DWTA does exactly this in a loop.)

No optimization out of the box. If you need optimization, wrap the solver in a branch-and-bound loop on your objective: post `objective ≤ best_known`, solve, update `best_known` to the result minus one, repeat until UNSAT. The last SAT model is optimal.

Inspect the statistics. After `lcg_solve`, four counters are exposed: `lcg_decisions`, `lcg_conflicts`, `lcg_propagations`, `lcg_explanations`. A high decision count with zero conflicts means the propagators are not pruning enough (the search is essentially DFS). A high conflict count with explanations growing faster than propagations means the solver is finding nogoods — good, that is exactly what LCG should be doing.

API summary

Function	Purpose
`lcg_new` / `lcg_free`	Context lifecycle
`lcg_new_int_var(ctx, lo, hi, name)`	Allocate a finite-domain integer variable
`lcg_lit_le(ctx, x, k)` / `lcg_lit_ge`	Get a literal for "x ≤ k" or "x ≥ k"
`lcg_post_clause(ctx, lits, n)`	Post a CNF clause over literals
`lcg_post_linear_le(ctx, c, v, n, k)`	Post `Σ c[i] v[i] ≤ k`
`lcg_post_alldifferent(ctx, v, n)`	Pairwise-distinct variables (bounds-consistent)
`lcg_post_cumulative(ctx, s, d, h, n, K)`	Time-table cumulative resource
`lcg_post_no_overlap(ctx, s, d, n)`	Unary resource, special case of cumulative
`lcg_register_propagator(ctx, p)`	Install a user-defined propagator
`lcg_expl_clear / _lo / _hi`	Build an explanation buffer inside a propagator
`lcg_emit_le / _ge / _conflict`	Emit a derived bound or a conflict
`lcg_lb(ctx, x)` / `lcg_ub`	Live bound queries during search
`lcg_solve(ctx)`	Run the solver; returns `LCG_SAT`, `LCG_UNSAT`, or `LCG_UNKNOWN`
`lcg_value(ctx, x)`	After `LCG_SAT`: the model's value of `x`
`lcg_decisions / _conflicts / _propagations / _explanations`	Search statistics

That is the entire surface. Six built-in constraint posters, two query functions, a small custom-propagator API, and a solve. The library is small on purpose — most of the modelling work happens in the user code that decides which constraints to post in what form. The library does the propagation and the learning.

Solving problems with MaxSAT

A user's guide to weighted partial MaxSAT

Preface

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

MaxSAT is the right tool when your problem has hard rules that must hold and soft preferences you'd like to honour but can't always afford to. Operational systems run into this constantly: every realistic threat picture is over-constrained, every realistic scheduling problem has more demands than resources, every realistic configuration has compromise built in. SAT and CP solvers tell you whether a configuration exists; MaxSAT tells you which compromise costs least.

The book assumes you've used a SAT solver before, can write a few lines of C, and want a quick path from "I have an over-constrained problem" to "I have a least-bad answer." It closes with the comparison you'd expect for any solver in this series: when to reach for MaxSAT versus MILP versus weighted CSP, and what hybrid patterns look like in operational systems.

— Stockholm, May 2026

Introduction

A *partial* MaxSAT problem has two kinds of clauses. Hard clauses must be satisfied in any solution the solver returns — they encode the physics, the rules of engagement, the kinematic envelopes, the resource capacities, whatever cannot be negotiated. Soft clauses are preferences. Each soft clause has a weight; the solver minimizes the sum of weights of unsatisfied soft clauses.

The library implements two algorithms for finding the optimum. The first, *model-improving* (linear search), is the conceptually obvious approach: find any model, post a cardinality constraint forcing strict improvement, repeat until UNSAT. The second, *core-guided* (Fu-Malik), is the conceptually richer approach used by every modern competition solver: find an unsatisfiable subset of soft clauses, relax exactly that subset by exactly the right amount, iterate. Both produce the same answer; the core-guided approach is what scales to production-sized problems with proper SAT engine support behind it.

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 two variables subject to a hard constraint and two competing soft preferences:

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

int main(void) {
    MaxSatCtx *m = maxsat_new();
    int x = maxsat_new_var(m);
    int y = maxsat_new_var(m);

    /* Hard: x and y cannot both be true. */
    int hard[2] = { -x, -y };
    maxsat_add_hard_clause(m, hard, 2);

    /* Soft: prefer x true (weight 7) and y true (weight 3). */
    int la = x, lb = y;
    maxsat_add_soft_clause_weighted(m, &la, 1, 7);
    maxsat_add_soft_clause_weighted(m, &lb, 1, 3);

    long cost = maxsat_solve(m);
    /* cost = 3 (the lighter soft is sacrificed). */
    /* x = 1, y = 0. */

    maxsat_free(m);
    return 0;
}
```

Build with:

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

The library is small enough that you can read every line in an afternoon. The whole thing is two files plus the SAT engine reused from the SMT/LCG codebase.

Variables and clauses

Variables are positive integers, allocated through `maxsat_new_var`. Literals are signed integers: `+v` for "v is true," `-v` for "v is false." This matches the underlying SAT engine and the standard DIMACS conventions.

Hard clauses are posted with `maxsat_add_hard_clause(m, lits, n)`. They must hold in any solution; the solver returns `MAXSAT_HARD_UNSAT` if the hard clauses are jointly infeasible.

Soft clauses come in two flavours:

```
void maxsat_add_soft_clause(MaxSatCtx *m, const int *lits, int n);
void maxsat_add_soft_clause_weighted(MaxSatCtx *m, const int *lits, int n,
                                     long weight);
```

The unweighted variant is shorthand for weight 1. Weight 0 is a no-op (the soft clause is silently dropped). Negative weights are invalid and dropped with no error — if you find yourself wanting negative weights, you probably want to negate the clause instead.

For at-most-one constraints over a small set of literals, the library provides a convenience helper that posts the pairwise encoding as hard:

```
void maxsat_add_at_most_one(MaxSatCtx *m, const int *lits, int n);
```

For at-most-k where k > 1, you have to encode it manually for now. The implementation book describes the sequential-counter encoding used internally; you can lift that into your own code if you need it.

Choosing an algorithm

Two algorithms are available, selected with `maxsat_set_algorithm`:

```
maxsat_set_algorithm(m, MAXSAT_ALG_LINEAR);          /* Phase 1 */
maxsat_set_algorithm(m, MAXSAT_ALG_CORE_GUIDED);     /* default */
```

For tutorial-scale problems and most over-constrained problems with cleanly-structured cores, core-guided is the right default and the library uses it unless told otherwise. For pathologically small problems where the deletion-based core extraction's O(n_softs) SAT-call overhead matters more than the algorithmic improvement, linear can be faster — the `examples/ex2_core_vs_linear.c` example shows exactly this kind of comparison. In production with proper SAT-engine assumptions the trade-off goes the other way decisively.

Modelling tips

Three things are worth knowing once you start using MaxSAT on real problems.

Weights matter more than counts. With unweighted MaxSAT the solver minimizes the *number* of unsatisfied soft clauses, treating every soft as equally important. This is almost never what you want for a real problem. If T0 is a cruise missile and T5 is a low-flying drone, "either is acceptable to miss" is rarely the actual preference. Weight your soft clauses, even crudely, even when you're not sure about the precise values. A weight of 10 versus a weight of 1 is usually enough structure to drive the search to the right shape of answer, even if neither number is exactly the "real" threat value.

Weights are tutorial-grade in this library. The Phase 3 implementation handles weights by *replication*: a soft clause with weight `w` becomes `w` identical unit-weight clauses internally. This is conceptually transparent (weight = count) and requires no algorithmic changes, but it scales linearly in total weight. For weights up to a few hundred this is fine. For weights in the thousands or millions — common in real models where you want fine-grained relative preferences — you need a production solver with proper weight-splitting Fu-Malik or pseudo-Boolean cardinality encoding. The library will faithfully find the right answer regardless; it will just be slow.

Lex-priority via weight gaps. If you want strict hierarchical priorities — "satisfy every priority-1 soft before paying any priority-2 cost" — give the priority-1 clauses weights big enough that any combination of priority-2 clauses cannot outweigh a single priority-1 clause. With N priority-2 clauses of weight 1, set the priority-1 clauses to weight N+1 or larger. This emulates lex-MaxSAT without needing a separate solver mode, at the cost of some replication overhead. Production lex-MaxSAT solvers handle this more efficiently, but the modelling pattern is the same.

Worked example: over-constrained DWTA

The canonical use case for MaxSAT in air-defence is *over-constrained dynamic weapon-target assignment*. The LCG library handled DWTA's feasibility case — every threat must be engaged, find a schedule that fits. MaxSAT handles the harder case: more threats than the defence can prosecute, every threat has a value, minimize total leakage.

The model is a single weighted MaxSAT instance:

```
typedef struct {
    const char *id;
    long  value;
    int   compat[N_WEAPONS];      /* 1 if weapon w can engage this threat */
} Threat;

int x[N_WEAPONS][N_THREATS];      /* x[w][t] = "weapon w engages threat t" */

for each weapon w:
    if compatible(w, t): x[w][t] = maxsat_new_var(m);

/* Hard: at most one engagement per weapon. */
for each weapon w:
    maxsat_add_at_most_one(m, eligible_x_for_weapon[w]);

/* Hard: at most one weapon per threat. */
for each threat t:
    maxsat_add_at_most_one(m, eligible_x_for_threat[t]);

/* Soft: each threat engaged, weighted by value. */
for each threat t:
    int eligible[] = { x[w][t] for w compatible with t };
    maxsat_add_soft_clause_weighted(m, eligible, n_eligible,
threat[t].value);
```

On the saturated scenario in `examples/ex3_dwta_weighted.c` — three weapons, six threats, values ranging from 2 to 10 — the solver finds the optimum in 10 cores: engage the three highest-value threats (T0 value 10, T2 value 8, T4 value 7), leak the three lowest (T1 value 5, T3 value 3, T5 value 2). Total leakage 10, total engaged value 25 out of a total threat value of 35.

The interesting comparison is with the *unweighted* DWTA example from Phase 1, which solves the same compatibility structure but treats every threat as equally important. There the solver engages T2, T4, and T5 — three threats, three weapons — leaving T0 (value 10) and T1 (value 5) to leak. Total leakage 18 versus the weighted solver's 10. The weighted formulation is not subtly better; it is the right answer where the unweighted one is a tactical disaster.

This is the most important fact about MaxSAT for operational use: **you almost certainly need weights**. A binary "yes/no, was a soft satisfied" cost function does not match operational reality. If you find yourself reaching for unweighted MaxSAT for a real-world problem, sanity-check whether you actually want weights and just defaulted to 1 because it was easier to type.

Worked example: prioritised tactical decisions

The "weights for priorities" pattern shows up everywhere in tactical software. A few examples worth thinking about, with their natural weighted-MaxSAT formulations.

Radar dwell scheduling. Hard: total beam-time per second ≤ 1.0. Soft per track i: "track i gets a revisit this second," weighted by track priority and squared track-error growth rate (a track losing accuracy fast needs revisits more than one in a stable state). The solver picks a subset of tracks for revisit and a leaves the others to coast.

Crew scheduling. Hard: certification matches qualification, rest minimum is respected, simultaneous assignments are physically possible. Soft: "crewmember X gets shift Y," weighted by some combination of currency requirements, fairness, and preferences. The solver respects the hard rules and produces a schedule that maximizes the weighted-soft score.

Maintenance vs. operational sortie scheduling. Hard: required inspection intervals are not violated, aircraft availability is consistent. Soft per sortie: "sortie S is flown," weighted by mission priority. Soft per maintenance window: "maintenance task M is done this period," weighted by how overdue it is. The solver balances the two streams of demand against the airframe pool.

ATO assembly with prioritized targets. Hard: aircraft-target-time triples are physically feasible. Soft per target: weighted by target value. Soft per sortie: weighted by negative cost (fuel, exposure). Lex priorities via weight gaps if you want "cover defensive CAP before any strikes."

Frequency assignment under jamming pressure. Hard: physical interference graph. Soft per emitter: "stays on preferred band," weighted by importance of that emitter remaining unjammed. The solver re-allocates as the jamming picture evolves, minimizing total disruption cost.

The common structure: hard rules from physics or regulation, soft preferences from doctrine or operational value, weights from the relative importance of competing soft preferences. Once you start looking, every C2 problem is shaped like this.

Tips and limitations

A handful of practical considerations.

Replication scales with total weight. Phase 3's weighting is implemented by replicating clauses. Total memory and total SAT calls in each core extraction scale with the sum of weights. For weights in the dozens or hundreds this is fine; for weights in the thousands, the library will be slow. A production solver would handle weight-splitting natively.

One shot per context. The library is one-shot: build a context, add clauses, call `maxsat_solve` once. Rolling-horizon DWTA does this in a loop — build a fresh context for each replanning cycle, solve, execute the first few decisions, discard, repeat. This is mechanical but it means warm-starting is not available; production solvers benefit from incremental SAT, which the library deliberately does not implement.

Inspect the stats. After solving:

```
long iter   = maxsat_iterations(m);
long cores  = maxsat_cores_found(m);
long decs   = maxsat_total_decisions(m);
```

For core-guided, `cores_found` equals the optimum cost (modulo replication: it equals the number of *unit* relaxations, which is the total weight relaxed). For linear, `cores_found` is always 0. If the SAT decision count is implausibly high for the size of the problem, you may be running deletion-based core extraction on a problem with very high total weight; consider whether your weight scale is appropriate.

Optimality is over the *hard* clauses. If the hard clauses are infeasible, `maxsat_solve` returns `MAXSAT_HARD_UNSAT`. Don't conflate "hard infeasible" with "all soft clauses unsatisfied"; they are very different states. A common modelling mistake is making a constraint hard that should be soft — for instance, encoding "every threat must be engaged" as a hard constraint and then getting `MAXSAT_HARD_UNSAT` because the picture is over-saturated. The fix is to make those constraints soft with high weights.

Choosing your solver: MaxSAT vs MILP vs WCSP

MaxSAT sits in a specific niche. There are problem shapes where it is unambiguously the right answer, others where MILP or weighted CSP wins.

MaxSAT

Strengths. Native handling of the "over-constrained problem with weighted preferences" pattern. Clause learning across the search; CDCL machinery underneath, so hard combinatorial structure is exploited well. The unsat-core formulation directly addresses the operational question "what *must* be sacrificed?" — the cores are often human-interpretable as conflicting constraint sets.

Logic-rich constraints are natural. Eligibility rules, doctrinal precedence, capability matchings — these are all clauses, not arithmetic. The LP relaxation that MILP relies on is loose on this kind of structure; MaxSAT propagation cuts through it directly.

Lex-priority is natural via weight gaps. The operational "do A before any of B before any of C" structure maps to MaxSAT cleanly.

Weaknesses. Weights via replication scale poorly for large weights; a production solver needs pseudo-Boolean cardinality encoding. The library here is tutorial-grade, but the technique it demonstrates is what production solvers like Open WBO, RC2, and EvalMaxSAT use under the hood.

No continuous decision variables. Everything is Boolean. If your problem has natural continuous quantities (fuel allocation, beam-time fractions), you have to discretize them and pay the discretization cost.

Less mature than MILP. Decades of solver tuning, primal heuristics, and presolve transformations have no MaxSAT counterpart at the same level of polish. A clever MILP formulation will sometimes beat an obvious MaxSAT one on the same data.

Air-force verdict. Use MaxSAT for over-constrained discrete decision-making with weighted preferences — DWTA, radar dwell scheduling, frequency assignment, crew assignment. Use it when the constraints are clauses rather than inequalities, when preferences are naturally weighted, and when you want explanations in terms of conflicting constraint subsets (the cores).

MILP

Strengths. Decades of solver maturity. Commercial solvers (Gurobi, CPLEX) and good open-source ones (HiGHS, SCIP) handle problems with millions of variables routinely. Natural for continuous quantities, weighted sums, and probabilistic objectives.

LP relaxation as a side benefit. Even when integer feasibility is hard, the LP relaxation gives a lower bound, dual prices, and warm-start information for re-optimization.

Optimization is native. The objective function is part of the formulation; there is no "find feasibility then descend" outer loop.

Weaknesses. Logic-rich constraints encode awkwardly. Each "if-then" rule requires either big-M constants (numerically painful) or an auxiliary binary variable (formulation bloat). The LP relaxation of these encodings is often very loose, leading to lots of branching.

Combinatorial structure exploitation depends on the solver's heuristics, not the formulation. MILP does not "know" about Hall sets, matching, or other CP-level structure unless the formulation makes it explicit.

Air-force verdict. Use MILP for static WTA optimization with rich probabilistic objectives — minimize expected damage given kill-probability matrices — where the objective is a weighted sum and the constraints are mostly linear. Use it for ATO-level resource allocation where the magnitudes matter and continuous slack variables ease the formulation.

Weighted CSP

Strengths. Combines CP-style propagation with weighted preferences. Tighter than MaxSAT on combinatorial structure (better propagators for `alldifferent`, `cumulative`, and so on). Native handling of structured weighted preferences.

Best of both worlds for problems where you have both rich combinatorial structure *and* weighted soft constraints. The natural extension of the LCG library if Phase 5 were written.

Weaknesses. Less mature implementation landscape than MaxSAT or MILP. Solvers like Toulbar2 exist and are excellent, but the ecosystem is smaller.

Pseudo-Boolean / cardinality encodings are sometimes more efficient than weighted CSP cost functions for the same problem; the choice is genuinely problem-dependent.

Air-force verdict. Use WCSP for problems where you need both CP-style global constraints and weighted preferences in the same formulation — DWTA with engagement-scheduling constraints *and* threat-value weights, sensor scheduling with revisit-rate penalties *and* doctrinal precedence. The LCG library's propagator API is the right substrate for this; adding weighted-cost machinery on top is the natural Phase 5 if there ever is one.

Comparison summary

Concern	MaxSAT	MILP	WCSP
Over-constrained discrete problems	strong	weak	strong
Optimal allocation with probabilities	weak	strong	medium
Time-window scheduling	weak	weak	strong
Cumulative resources	weak	medium	strong
Logic-rich constraints	strong	weak	medium
Continuous quantities	none	strong	none
Weighted preferences	strong	strong	strong
Lex / hierarchical priorities	strong	medium	medium

Enumeration	weak	weak	medium
Maturity and tuning	medium	strong	weak

Combining multiple technologies

The same decomposition patterns that worked for the LCG library work here. The most relevant for DWTA-style problems:

MILP master + MaxSAT subproblem. A logic-based Benders decomposition where MILP optimizes the high-value allocation (which weapons engage which targets, expected leakage minimization with kill-probability matrices) and MaxSAT decides the over-constrained subset selection ("given resource constraints, which targets actually get engaged with the assignments we have"). The MILP master can use the MaxSAT subproblem's "best subset" cost as part of its objective.

MaxSAT master + LCG subproblem. The opposite direction: MaxSAT decides the engagement subset under saturation, LCG verifies that the chosen engagements can be scheduled through the fire-control channels within their time windows. The MaxSAT master may need to revise its choice if the schedule is infeasible — feeding the infeasibility back as a "this subset of engagements cannot be scheduled" hard constraint. This is the natural decomposition for the full DWTA pipeline as your simulator describes it.

Rolling-horizon MaxSAT. The realistic dynamic case: every few seconds, build a fresh MaxSAT instance over the current threat picture, solve, execute the first few decisions, discard the rest, replan with updated information. The library's one-shot design is fine for this — you create a fresh context per cycle. The MaxSAT call is in the inner loop of the control system. Production warm-starting and incremental SAT make this dramatically faster than the library's straightforward implementation; the modelling pattern is identical.

A from-scratch DWTA simulator probably wants to start with the MaxSAT master + LCG subproblem pattern. The MaxSAT layer answers "given the picture, what's the best subset?" and the LCG layer answers "given the subset, what's the schedule?" The two answer different operational questions and decompose cleanly. If you find a particular case where the decomposition oscillates (MaxSAT picks a subset that LCG rejects, MaxSAT picks again, etc.), strengthen the MaxSAT layer's awareness of the scheduling constraints — add hard clauses encoding the obvious schedulability checks, leaving only the subtle interaction cases for the LCG round-trip.

API summary

Function	Purpose
`maxsat_new` / `maxsat_free`	Context lifecycle
`maxsat_new_var(m)`	Allocate a SAT variable
`maxsat_add_hard_clause(m, lits, n)`	Post a hard clause
`maxsat_add_soft_clause(m, lits, n)`	Post a unit-weight soft
`maxsat_add_soft_clause_weighted(m, lits, n, w)`	Post a weighted soft
`maxsat_add_at_most_one(m, lits, n)`	Pairwise at-most-one (hard)
`maxsat_set_algorithm(m, alg)`	Choose linear or core-guided
`maxsat_get_algorithm(m)`	Read back the current algorithm
`maxsat_solve(m)`	Solve to optimality; returns optimum cost or `MAXSAT_HARD_UNSAT`
`maxsat_value(m, var)`	Read the value of a variable in the optimal model
`maxsat_iterations(m)`	Number of outer-loop iterations
`maxsat_cores_found(m)`	Number of cores discovered (core-guided only)
`maxsat_total_decisions(m)`	Total SAT decisions across all iterations

That is the entire surface. Four constraint-posting functions, an algorithm switch, a solve, and the model-query and stats accessors. The library is small on purpose — most of the modelling work happens in your code, deciding which clauses to post with which weights. The library does the propagation, the clause learning, the core extraction, and the cost minimization.