

# **Solving problems with LCG**

*A user's guide to Lazy Clause Generation*

Copyright © 2026 by Streck.ai

# 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 " $x \leq k$ ". A monotonicity chain  $b[k] \rightarrow b[k+1]$  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 " $x \leq 5$ " is one literal; asserting " $x \neq 7$ " is two literals (the disjunction " $x \leq 6$  OR  $x \geq 8$ "). 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  $\sum \text{coeff}[i] \cdot \text{vars}[i] \leq k$ . 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  $\sum a_i x_i \geq k$ , negate everything:  $\sum (-a_i) x_i \leq -k$ .

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 `alldifferents` 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  $N \times N$  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 `alldifferents` 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 ( $p_{kill\_total} = 1 - \prod(1 - p_{ij})$ ) 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  $\prod (1 - p_{ij})$  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  $a \cdot x = b \cdot y + c$  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
<code>lcg_new/lcg_free</code>	Context lifecycle
<code>lcg_new_int_var(ctx, lo, hi, name)</code>	Allocate a finite-domain integer variable
<code>lcg_lit_le(ctx, x, k) / lcg_lit_ge</code>	Get a literal for " $x \leq k$ " or " $x \geq k$ "
<code>lcg_post_clause(ctx, lits, n)</code>	Post a CNF clause over literals
<code>lcg_post_linear_le(ctx, c, v, n, k)</code>	Post $\sum c[i] v[i] \leq k$
<code>lcg_post_alldifferent(ctx, v, n)</code>	Pairwise-distinct variables (bounds-consistent)
<code>lcg_post_cumulative(ctx, s, d, h, n, K)</code>	Time-table cumulative resource
<code>lcg_post_no_overlap(ctx, s, d, n)</code>	Unary resource, special case of cumulative
<code>lcg_register_propagator(ctx, p)</code>	Install a user-defined propagator
<code>lcg_expl_clear / _lo / _hi</code>	Build an explanation buffer inside a propagator
<code>lcg_emit_le / _ge / _conflict</code>	Emit a derived bound or a conflict
<code>lcg_lb(ctx, x) / lcg_ub</code>	Live bound queries during search
<code>lcg_solve(ctx)</code>	Run the solver; returns <code>LCG_SAT</code> , <code>LCG_UNSAT</code> , or <code>LCG_UNKNOWN</code>
<code>lcg_value(ctx, x)</code>	After <code>LCG_SAT</code> : the model's value of $x$
<code>lcg_decisions / _conflicts / _propagations / _explanations</code>	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.