

Solving problems with MILP

A user's guide to mixed-integer linear programming

Copyright © 2026 by Streck.ai

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 $0 \dots n_vars - 1$, constraints are integers $0 \dots n_constraints - 1$, 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 $\sum_j a_j x_j$; op ; rhs . The op constants are $MILP_LE$ (\leq), $MILP_EQ$ ($=$), and $MILP_GE$ (\geq).

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 $-\infty$ (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 $\sum a_j x_j \leq 0$ ". Encoded as $\sum a_j x_j \leq M y$ 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 $\sum a_j x_j \leq M$ 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. $\sum c_j x_j \leq B$ 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. $\sum_{s: e \in S_s} x_s \geq 1$ 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 $A \leq M(1 - z)$ and B as $B \leq M z$. 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 $\sum_j v_j p_{ij} x_{ij}$, 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 ($\sum_j x_{ij} \leq 1$), each target is engaged by at most one weapon ($\sum_i x_{ij} \leq 1$), and the total cost is within budget ($\sum_{ij} c_{ij} x_{ij} \leq B$).

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