

# Solving problems with MaxSAT

*A user's guide to weighted partial MaxSAT*

Copyright © 2026 by Streck.ai

# 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  $O(n_{\text{softs}})$  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  $\leq 1.0$ . 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
<code>maxsat_new/maxsat_free</code>	Context lifecycle
<code>maxsat_new_var(m)</code>	Allocate a SAT variable
<code>maxsat_add_hard_clause(m, lits, n)</code>	Post a hard clause
<code>maxsat_add_soft_clause(m, lits, n)</code>	Post a unit-weight soft
<code>maxsat_add_soft_clause_weighted(m, lits, n, w)</code>	Post a weighted soft
<code>maxsat_add_at_most_one(m, lits, n)</code>	Pairwise at-most-one (hard)
<code>maxsat_set_algorithm(m, alg)</code>	Choose linear or core-guided
<code>maxsat_get_algorithm(m)</code>	Read back the current algorithm
<code>maxsat_solve(m)</code>	Solve to optimality; returns optimum cost or <code>MAXSAT_HARD_UNSAT</code>
<code>maxsat_value(m, var)</code>	Read the value of a variable in the optimal model
<code>maxsat_iterations(m)</code>	Number of outer-loop iterations
<code>maxsat_cores_found(m)</code>	Number of cores discovered (core-guided only)
<code>maxsat_total_decisions(m)</code>	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.