PRACTICAL SOLVERS

A working tour through SAT, MaxSAT, SMT,
lazy clause generation, and mixed-integer programming

STRECK.AI STOCKHOLM

This book was typeset with TEX in Donald Knuth's Computer Modern types, using the `taomac` macros. The illustrations were drawn in METAPOST.

CONTENTS

PREFACE

This is a book about five solvers with one temperament. Each takes a problem you have stated declaratively—variables, constraints, perhaps an objective—and searches for an answer you could not conveniently compute yourself. Each is implemented in a small C99 library that you can read in an afternoon, compile with one command, and link into a program of your own. The code is the primary text; the prose is here to explain how to state your problem so that the search succeeds, and how to read what the solver tells you back.

The five, in the order the book treats them:

CDCL—conflict-driven clause learning—is the modern SAT algorithm. One kind of variable (a Boolean), one kind of constraint (a clause), one question (is there a satisfying assignment?). Everything else is encoding. The library is about 650 lines and demonstrates the techniques—watched literals, first-UIP learning, VSIDS, Luby restarts—that production SAT solvers use under the hood.

MaxSAT extends SAT with soft clauses: preferences with weights, to be honoured when possible and sacrificed at minimum cost when not. The library layers two optimisation algorithms—model-improving linear search and core-guided Fu–Malik—directly on the CDCL engine.

SMT—satisfiability modulo theories—keeps the CDCL engine as a propositional substrate and adds theory solvers above it: congruence closure for equality with uninterpreted functions, an exact- rational simplex for linear real arithmetic, and the Nelson–Oppen channel that lets the two theories exchange what they learn.

LCG—lazy clause generation—is the hybrid of constraint programming and CDCL. Finite-domain integer variables ride on an order encoding; global constraints (`alldifferent`, `cumulative`) propagate as in CP, but every deduction is explained by a clause that enters the CDCL learnt database, so a dead end met once prunes its relatives everywhere.

MILP—mixed-integer linear programming—is the outlier: no clauses at all. A two-phase revised simplex solves linear relaxations exactly; branch-and-bound resolves the integrality. It is the tool for problems that are about *how much* rather than *whether*, and its bounds-driven pruning is the counterpoint to everything clause-driven in the first four.

The intended reader is a competent C programmer who has not used these tools before. You should be comfortable with pointers, arrays, structs, and basic algorithm analysis. You do not need a background in formal logic or operations

research; where mathematical notation appears it is kept loose and tied to the code.

The libraries are deliberately small. None is a competitor to CaDiCaL, Z3, CP-SAT, or Gurobi; each is a concrete artefact you can hold in one head, exhibiting the same architecture as its production cousins. Every function in each public API appears in this book. Every tutorial program compiles against the corresponding library.

A note on lineage. The backtracking toolkit that precedes these five—trails, sparse sets, classical constraint propagation, and exact cover by dancing links—is the subject of a companion volume, *Predictive Algorithms*. This book stands alone, but the two describe complementary halves of one toolbox, and Chapter 17 draws the map that covers both.

A note on style. The code listings are taken from the libraries and their example programs directly. Where I have abbreviated for space, I have said so; where I have not said so, the listing is verbatim. The chapters are short on purpose.

— Stockholm, May 2026

HOW TO READ THIS BOOK

Read Part One first regardless of your destination: the CDCL engine of Chapter 2 is the substrate on which MaxSAT, SMT, and LCG are all built, and the encoding patterns of Chapter 3 recur in every later part.

After that, the parts are independent of one another. If you arrive with a problem in hand, the priority order is roughly:

1. Chapters 1–3, the substrate and the encoding patterns.

2. The part matching your problem's shape: preferences over clauses (Part Two), arithmetic mixed with case analysis (Part Three), time windows and resources (Part Four), or numeric optimisation (Part Five).

3. Chapter 17, which compares all five paradigms on one problem family and describes the hybrid patterns operational systems actually use.

Each part ends with a tutorial chapter that works one problem from statement to running program. The tutorials are independent; you can read the one whose subject most appeals.

A second note on style. Throughout the book I use *we* in places where I am inviting you to follow a derivation, and *you* where I am asking you to take an action—write a program, change a weight, rerun a benchmark. Both meanings are intended literally.

PART ONE

CLAUSES

CHAPTER ONE

THE SHAPE OF A SOLVER

Every library in this book presents the same face. You create a context. You declare variables. You post constraints. You call solve. You read the answer back, and you free the context. The five differ in what a variable is, what a constraint is, and what happens inside the call to solve—but the shape of a program that uses any of them is the same twenty lines.

This chapter fixes that shape, names the family relationships among the five, and states the two questions—feasibility and optimality—that everything later refines.

1.1. Declare, post, solve, read back

Here is the shape, instantiated for the SAT library of the next chapter:

```
Sat *s = sat_new(n_vars);       /* a fresh, empty problem;   */
                                /* the vocabulary            */
sat_add_clause(s, lits, n);     /* what must hold, repeated  */
int status = sat_solve(s);      /* the search                */
if (status == SAT_SATISFIABLE)
    v = sat_value(s, x);        /* the model, read back      */
sat_free(s);
```

Substitute `maxsat`, `smt`, `lcg`, or `milp` for the prefix and the outline survives: what changes is only what a variable is, what posting looks like, and what solve returns.

Two conventions run through all five libraries and are worth naming now.

First, *declaration precedes posting.* A constraint can only mention variables that already exist, and in several of the libraries the variable count is fixed at context creation. If you do not know how many variables your encoding needs, count them by walking your problem structure first, then create the context. This keeps every library's internal allocation single-pass and is a habit worth forming even where a library is more forgiving.

Second, *one context answers one question.* None of the five libraries supports incremental re-solving: you build a context, call solve once, and free it. A program that must re-solve as its inputs change—and Chapter 17 argues that most operational programs must—builds a fresh context per cycle. Contexts are cheap to build; the discipline costs little and removes a whole class of stale-state bugs.

1.2. Feasibility and optimality

The five solvers answer two kinds of question.

A *feasibility* question asks whether any assignment satisfies every constraint, and for one such assignment if so. SAT, SMT, and LCG are feasibility solvers: their answer is a model or a verdict of unsatisfiability. When they say UNSAT they mean it universally—no assignment exists, not merely that none was found.

An *optimality* question asks for the best assignment under an objective. MILP is natively an optimality solver: the objective is part of the model, and the search descends toward it. MaxSAT is an optimality solver over clauses: it minimises the total weight of unsatisfied soft clauses.

The boundary is crossable in both directions, at a price. A feasibility solver optimises by repeated solving—post *objective better than the incumbent*, solve, tighten, repeat until UNSAT; the last model is optimal. An optimality solver answers feasibility by optimising a constant objective. Both tricks appear in the chapters ahead; neither is as good as a solver whose native mode matches the question. Matching the question to the solver is half the content of Chapter 17.

1.3. One family, five members

Four of the five libraries are blood relatives.

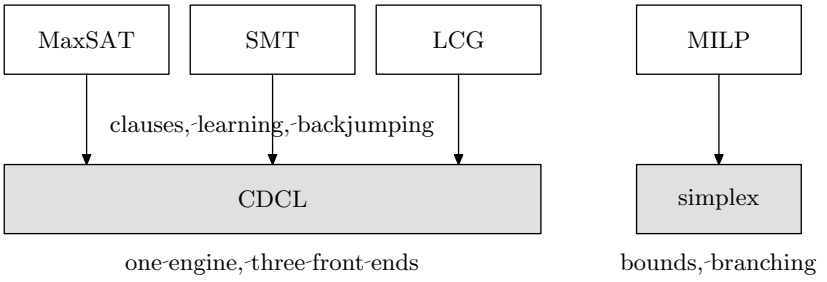

Fig. 1. The family. MaxSAT, SMT, and LCG are front ends to one CDCL engine; MILP stands apart on the simplex and branch-and-bound.

The CDCL SAT engine of Chapter 2 sits at the bottom. MaxSAT drives it in a loop, feeding it formulas whose satisfiability encodes "can we do better than cost k?". SMT runs it as the propositional layer of a two-level architecture, with theory solvers checking each partial assignment for arithmetic and equality consistency. LCG runs it as the search-and-learning layer beneath constraint propagators, which inject their deductions as clauses.

The consequence for you as a user is pleasant: one mental model of search—decide, propagate, conflict, learn, backjump—carries across four of the five libraries. Learn to read the CDCL statistics once, in Section 2.9, and you can diagnose a slow MaxSAT, SMT, or LCG run with the same eyes.

MILP is the deliberate outsider. Its pruning comes from bounds, not conflicts: at every node of its search tree it solves a linear relaxation whose value limits what the subtree could achieve. The contrast between bound-driven and conflict-driven pruning is a running theme, and Chapter 17 makes it explicit.

1.4. The libraries

Each library is one or two C files plus a header, built with a single `gcc` invocation and no dependencies beyond libc (and `-lm` for MILP). Appendix A gives the exact command lines; Appendix B collects the five public APIs in one place. The sizes are worth recording, because they are the point:

```
sat.c           655 lines   CDCL: watched literals, 1UIP,
                            VSIDS, Luby restarts
maxsat.c        620 + sat   linear search and Fu-Malik, weights
smt/euf/lra/   2381 + sat   EUF + LRA + Nelson-Oppen bridges,
  rat.c                     exact rationals
lcg.c + two    1240 + sat   order encoding, propagators,
  propagators               explanations
milp.c          495 lines   two-phase revised simplex,
                            branch-and-bound
```

Nothing here is pseudocode. When Chapter 2 says the propagation loop watches two literals per clause, the loop is forty lines of `sat.c` that you can open next to the book.

1.5. What the rest of the book does

Part One—this part—covers the CDCL engine and the craft of encoding problems into clauses, closing with a graph-colouring tutorial.

Part Two adds soft clauses: weighted partial MaxSAT, the two optimisation algorithms, and a tutorial on assignment under saturation, where there is provably not enough of everything.

Part Three adds theories: equality with uninterpreted functions, linear rational arithmetic, and the Nelson–Oppen combination, with a tutorial on placing rectangles under alignment and non-overlap constraints.

Part Four adds integers back on top of clauses: lazy clause generation, the global constraints `alldifferent` and `cumulative`, and a tutorial on time-window scheduling.

Part Five changes substrate entirely: linear programs, the simplex, branch-and-bound, and the art of formulations whose relaxations are tight, with a tutorial on budgeted assignment.

Part Six is one chapter: which solver for which problem, and how real systems combine them.

CHAPTER TWO

CONFLICT-DRIVEN CLAUSE LEARNING

SAT is the simplest constraint 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 an assignment to the variables exist that satisfies every clause? Everything else is a modelling exercise—take the problem you have, encode it as Booleans and clauses, hand the formula over, read back the answer.

The CDCL algorithm is what makes SAT practical at scale. The outer loop is plain backtracking: pick a variable, assign it, propagate the consequences, recurse. What is not plain is the response to failure. When a clause is violated under the current assignment, the solver does not merely back up one decision. It analyses the conflict, derives a new clause summarising *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.

This chapter covers the API and enough of the machinery that the statistics in Section 2.9 read as diagnosis rather than trivia.

2.1. Variables, literals, clauses

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 call to create a variable. You declare the count when creating the context, and variables $1..n$ exist from then on:

```
Sat *s = sat_new(n_vars);
sat_add_clause(s, lits, n_lits);
```

The empty clause is the contradiction; adding it makes the formula trivially UNSAT, and `sat_add_clause` returns 1 to say so (it also returns 1 when a unit clause directly conflicts with an existing assignment). Your code can bail out early or simply continue calling `sat_solve`, which will return UNSAT.

Two cleanups happen at posting time, before the solver sees the formula: tautologies—clauses containing both a literal and its negation—are silently dropped, and duplicate literals within a clause are merged.

2.2. A first program

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;
}
```

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.

2.3. Unit propagation

The workhorse inside the solver is *unit propagation*, also called Boolean constraint propagation or BCP. A clause with all but one literal false forces the remaining literal: if $(x_1 \vee \neg x_2 \vee x_3)$ has x_1 false and x_2 true, then x_3 must be true. The forced assignment may make other clauses unit in turn; propagation runs the cascade to a fixed point after every decision.

Most of a SAT solver's time is spent here, so the data structure that answers "which clauses might have become unit?" sets the cost of the whole search.

2.4. Two watched literals

The naive answer—check every clause containing the variable just assigned—touches far too much memory. The standard answer, and the library's, is the two-watched-literal scheme.

Each clause nominates two of its literals as *watched*. The invariant: as long as both watched literals are non-false, the clause can be ignored—it cannot be unit, and it cannot be violated. Only when a watched literal becomes false does the clause need attention, and then the solver first tries to walk the watch to some other non-false literal in the clause.

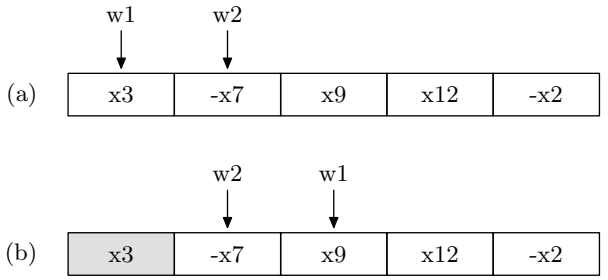

Fig. 2. Two watched literals. In (a) the watches sit on `x3` and `-x7`. When `x3` becomes false (b), the first watch walks along the clause to the non-false `x9`; the clause needs no further attention until a watched literal falls again.

Only if no replacement exists is the clause really interesting: the other watched literal is now forced (the clause is unit), or also false (the clause is a conflict).

Two properties make the scheme fast. Assignments that do not touch a watched literal cost nothing at all—the clause is never visited. And nothing needs to be undone on backtracking: a watch that walked rightward can simply stay where it is, because the invariant only ever speaks of the current assignment. The cost of propagation is essentially insensitive to the total number of clauses; only the number of watches per literal matters. That insensitivity is why the encoding advice of Chapter 3 worries about variable counts and not clause counts.

2.5. Conflict analysis

When propagation finds a violated clause, the solver has more information than "this branch failed". It has the *implication graph*: every propagated literal points

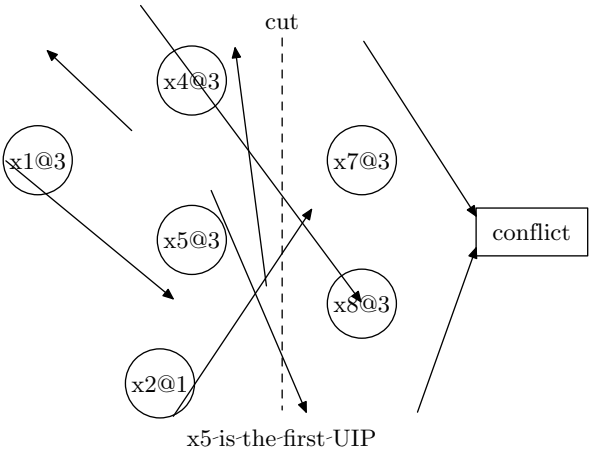

Fig. 3. An implication graph at a conflict; a node `x@`ℓ is the assignment of `x` made at decision level ℓ. The decision `x1` at level 3 propagated `x4` and `x5`; those propagated `x7` and (with the older `x2` from level 1) `x8`; together `x7` and `x8` violated a clause. Every cut separating the decision side from the conflict side yields a learnable clause; the first-UIP cut passes through `x5`, the node closest to the conflict that dominates every path from the current decision.

back to the clause that forced it, and through it to the literals that made that clause unit.

Conflict analysis walks this graph backward from the violated clause, resolving away literals of the current decision level until exactly one remains. That literal is the *first unique implication point*—the first UIP—and the negations of the literals gathered along the way form the learned clause. The learned clause is, by construction, false under the current assignment and a logical consequence of the formula: the search was always going to fail here, and now the formula says so explicitly.

The analysis runs in time linear in the size of the conflict's neighbourhood of the graph, and it happens at every conflict. The library's implementation is the `analyze` function in `sat.c`; it is some fifty lines and repays reading.

2.6. Backjumping

The learned clause does more than record the failure; it directs the recovery. Every literal in it except the UIP belongs to some earlier decision level. Let ℓ be the deepest such level. Jumping back to ℓ—undoing every decision after it, not merely the most recent—makes the learned clause unit, and propagation immediately asserts the negated UIP.

This is *backjumping*, and it is qualitatively different from chronological backtracking. Decisions between ℓ and the conflict that had nothing to do with the failure are discarded wholesale rather than re-enumerated. On formulas with structure—which is to say, on formulas that encode real problems—the difference is often the difference between milliseconds and heat death.

2.7. Decisions: VSIDS

When propagation runs dry, the solver must guess. The library uses the VSIDS heuristic—variable state independent decaying sum. Every variable carries an activity score; variables appearing in a learned clause get a bump; all scores decay geometrically over time; the next decision picks the unassigned variable with the highest score.

The effect is a search that concentrates on the part of the formula currently producing conflicts—the part, in other words, where the action is—and drifts as the action moves. No problem-specific tuning is involved, which is much of why CDCL solvers work well out of the box on formulas from wildly different domains.

2.8. Restarts

CDCL search occasionally commits early to a bad neighbourhood: a few unlucky top-level decisions, and every conflict is being learned in a region that a different opening would never have entered. The remedy is periodic *restarts*: discard the partial assignment, keep the learned clauses and the activity scores, and start deciding afresh.

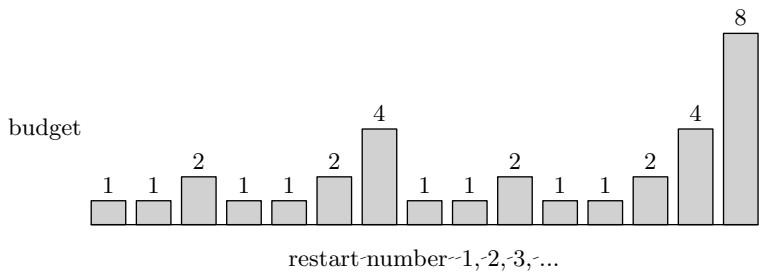

Fig. 4. The Luby sequence 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, ... of restart budgets, in units of a base conflict count. Frequent small restarts are interleaved with progressively rarer long runs.

The library schedules restarts on the Luby sequence, which is provably within a logarithmic factor of the best possible universal restart schedule and, more to the point, works well in practice. The generator is eleven lines of `sat.c`, verbatim:

```
static long luby(int i) {
    int size, seq;
    for (size = 1, seq = 0; size < i + 1; size = 2 * size + 1, seq++)
        ;
    while (size - 1 != i) {
        size = (size - 1) >> 1;
        seq--;
        i = i % size;
    }
    return 1L << seq;
}
```

Because the learned clauses survive the restart, the solver does not repeat itself: the new descent is through a formula that remembers every dead end the old one found.

2.9. What the statistics mean

After solving, five counters summarise what happened:

```
long sat_decisions(s);        /* guesses made                  */
long sat_conflicts(s);        /* learned-clause cycles         */
long sat_propagations(s);     /* unit-propagation steps        */
long sat_restarts(s);         /* Luby restarts                 */
long sat_learned_clauses(s);  /* clauses added by analysis     */
```

These are not curiosities; they are the instrument panel. A high decisions-per-conflict ratio means the solver is mostly guessing and rarely learning—the encoding is not exposing structure it can grip. A high propagations-per-decision ratio means each guess has many consequences—the encoding is propagating well. The pigeonhole formula of Section 3.10 is the textbook pathology: propagations and conflicts both stay modest while decisions blow up exponentially, because no clause-learning argument can express the counting fact a human sees instantly.

Two tuning knobs complete the surface. `sat_set_conflict_limit` caps the number of conflicts before the solver gives up with SAT_UNKNOWN—useful when SAT runs inside a larger loop with a latency budget. `sat_set_verbose` raises the trace level: 0 is silent, 1 prints summary statistics, 2 prints a line per conflict, which is mostly useful for watching the solver thrash on a clause you thought was redundant. That is the entire tuning surface. Production solvers expose dozens of knobs; if you find yourself needing them, the problem has outgrown a tutorial library.

CHAPTER THREE

ENCODING INTO CLAUSES

The solver is fixed; the encoding is where your skill goes. A well-encoded problem solves in milliseconds; a poorly-encoded version of the same problem can take hours. This chapter collects the patterns that cover the great majority of real SAT modelling—each with its clauses written out—then works two complete encodings against the library, and closes with the diagnostics for telling a good encoding from a bad one.

Throughout, clauses are displayed in the library's own literal convention: one clause per line, positive integers for variables asserted true, negative for false. What you see is exactly what `sat_add_clause` receives.

3.1. At least one, at most one, exactly one

The most common structure by far: a group of Booleans of which some number must hold. Take three variables x_1, x_2, x_3 of which *exactly one* must be true. The encoding is four clauses:

```
 1  2  3        at least one of the three
-1 -2           not both x1 and x2  \
-1 -3           not both x1 and x3   >  at most one, pairwise
-2 -3           not both x2 and x3  /
```

At least one is always a single clause over the group:

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

At most one, in the pairwise encoding, posts one binary "not both" clause per pair:

```
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);
    }
```

Exactly one is the two combined, as in the four-clause display above. The idiom propagates well: the moment one variable of the group becomes true, every other is forced false by a binary clause, with no search.

The pairwise count is $\binom{n}{2}$, quadratic, and absolutely fine up to n around 30. Past $n \approx 50$ the next section's encoding wins.

3.2. The sequential counter

For a large group, trade auxiliary variables for clauses. Introduce $n-1$ fresh Booleans $s_1, \ldots, s_{n-1}$, where s_i means "some variable among $x_1..x_i$ is already true". Three clause shapes maintain that meaning and forbid a second true variable:

```
-x1  s1              x1 sets the register
-xi  si              xi sets the register        (1 < i < n)
-s(i-1)  si          the register carries         (1 < i < n)
-s(i-1) -xi          register set: no further xi  (1 < i <= n)
```

That is $3n-4$ binary clauses and $n-1$ auxiliaries—linear where pairwise was quadratic. In code, remembering that the auxiliary ids must be counted into `sat_new`'s total before posting begins:

```
/* At most one of vars[0..n): sequential-counter encoding.
 * Uses the n-1 fresh variables aux0 .. aux0 + n - 2. */
static void at_most_one_seq(Sat *s, const int *vars, int n,
                            int aux0) {
    if (n <= 1) return;
    sat_add_clause(s, (int[]){ -vars[0], aux0 }, 2);
    for (int i = 1; i < n - 1; i++) {
        int si = aux0 + i, sp = aux0 + i - 1;
        sat_add_clause(s, (int[]){ -vars[i], si   }, 2);
        sat_add_clause(s, (int[]){ -sp,      si   }, 2);
        sat_add_clause(s, (int[]){ -sp, -vars[i]  }, 2);
    }
    sat_add_clause(s, (int[]){ -(aux0 + n - 2), -vars[n-1] }, 2);
}
```

The same register idea generalises: a counter k wide instead of one bit wide yields at-most-k, the workhorse cardinality constraint, in $O(nk)$ clauses. The generalisation is a good exercise; MaxSAT's linear search in Chapter 6 leans on exactly this construction.

3.3. Implication and equivalence

$a \Rightarrow b$ is the clause $\neg a \vee b$:

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

Conjunctions in the antecedent and disjunctions in the consequent fold into a single clause; a conjunction in the consequent splits into one clause per conjunct:

```
a & b -> c | d        -a -b  c  d       one clause
a -> b & c            -a  b             two clauses
                      -a  c
a <-> b               -a  b             the two implications
                       a -b
```

These transformations are mechanical, and it is worth performing them mechanically—by a helper function in your code, not by hand at each call site—because a sign error in one clause is the classic source of the "unexpected UNSAT" of Section 3.11.

3.4. Naming a subformula

The patterns above cover formulas that are already conjunctions of simple pieces. For arbitrary propositional structure—a disjunction of conjunctions, a condition used in several places—give the subformula a *name*: a fresh variable defined, by clauses, to be equivalent to it. This is the Tseitin transformation, and its two cases are worth knowing cold. To define $d \Leftrightarrow (a \wedge b)$:

```
-d  a                 d implies a
-d  b                 d implies b
-a -b  d              a and b together imply d
```

and $d \Leftrightarrow (a \vee b)$:

```
-a  d                 a implies d
-b  d                 b implies d
-d  a  b              d implies a or b
```

Three clauses per named subformula, one auxiliary each, and any formula whatever reaches CNF in linear size—where naive distribution of *or* over *and* can blow up exponentially. Name the subformula once, then use the name: in a group of exactly-ones, in an implication, in the objective softs of Part Two.

3.5. Finite domains

A variable taking one of k values becomes k Booleans—one per value—constrained exactly-one. A signal taking one of the three values red, amber, and green, with variables 1, 2, 3:

```
 1  2  3              some colour
-1 -2                 \
-1 -3                  >  only one colour
-2 -3                 /
```

Domain pruning is now a unit clause—ruling out amber is posting `-2` alone—and a rule like "if the signal is red the gate is down" is the implication `-1 g` from Section 3.3. This *direct encoding* is easy to read and good enough for most problems; the dual *order encoding*, $k-1$ Booleans meaning "the value is at most j", is more compact for large domains and is the representation Part Four builds on.

The cost worth watching is variables, not clauses. A domain of size 100 takes 100 Booleans in the direct encoding, and an alldifferent over n such variables

takes $O(n^2)$ pairwise clauses per value. CP solvers handle such domains natively; SAT brute-forces them. When domains grow past the low dozens, Part Four is usually the better tool.

3.6. Permutations

n items into n positions, each item somewhere, each position filled once: use n^2 Booleans x_{ij} meaning "item i in position j", with one exactly-one constraint per row and one per column. This is the SAT rendering of an alldifferent constraint, and it recurs everywhere—colourings, schedules, assignments all contain it. The next section places it on a chessboard.

3.7. A worked encoding: N-Queens

Place N queens on an $N \times N$ board, no two attacking: no shared row, column, or diagonal. The encoding is the permutation pattern plus diagonals, and the complete program is `ex2_queens.c` in the library's examples.

One Boolean per square, and the first decision is bookkeeping:

```
#define X(i, j) ((i) * N + (j) + 1)
```

The `+ 1` is the DIMACS convention biting—variables number from 1, and forgetting the offset gives variable 0 for the corner square and a formula that is subtly about the wrong board. Write the index macro first, once.

Three clause families follow. Each row gets a queen—one at-least-one of width N per row. No two queens share a row or a column—pairwise at-most-ones, as in Section 3.1. And no two share a diagonal, which is the same at-most-one applied to each diagonal's squares; from the example, verbatim:

```
/* Diagonals  (i - j = const). */
for (int d = -(N - 1); d <= N - 1; d++) {
    int sqs[64], k = 0;
    for (int i = 0; i < N; i++) {
        int j = i - d;
        if (j >= 0 && j < N) sqs[k++] = X(i, j);
    }
    for (int a = 0; a < k; a++)
        for (int b = a + 1; b < k; b++) {
            pair[0] = -sqs[a]; pair[1] = -sqs[b];
            sat_add_clause(s, pair, 2);
        }
}
```

(and its mirror for $i + j$ constant). Note what the loop does about short diagonals: nothing special. A diagonal of one square contributes no pairs; the at-most-one of an empty or singleton group is vacuous, and the code simply generates no clauses for it.

The census for $N = 8$: 64 variables; 8 row at-least-ones; $8 \times \binom{8}{2} = 224$ row pairs and 224 column pairs; and 280 diagonal pairs across the thirty diagonals of both slopes—736 clauses in all, matching the program's printout. Every group has $n \leq 8$, far under the pairwise threshold, so Section 3.2's counter stays in the drawer. The solve:

```
decisions   = 37
conflicts   = 16
propagations= 276
```

Sixteen conflicts to place eight queens. The solver knows nothing of rows or diagonals—the only information it has is 736 clauses—and clause learning discovers, conflict by conflict, the same inferences a dedicated queens solver makes by construction. That is the price and the payoff of a generic solver: one uniform language in, one uniform algorithm pulling the structure out.

3.8. Redundant clauses that help

Logical minimality and propagation strength are different virtues, and the library's sudoku example measures the difference.

The canonical encoding has 729 variables x_{rcd}—"the cell at row r, column c holds digit d"—with exactly-one per cell, an at-least-one per (row, digit), (column, digit), and (box, digit), and a unit clause per given clue. Counting shows the corresponding *at-most*-ones for rows, columns, and boxes are logically implied: nine cells, nine digits, each cell exactly one digit, so no digit can repeat without another going missing. Omitting them leaves the solution set untouched at 8829 clauses.

`ex4_sudoku.c` includes them anyway, at 11745 clauses, because implied is not the same as *propagated*. With the redundant at-most-ones present, placing a 7 immediately unit-propagates "no other 7 in this row" as binary clauses fire; without them, the solver discovers each such fact only by trying the alternative and failing. The formula got bigger and the search got much smaller—on "AI Escargot", a puzzle designed to be adversarial, the redundant encoding solves in 21 decisions, 10 conflicts, and 1488 propagations. Look at the propagation count against the decision count: seventy deductions per guess is the encoding doing the solver's work for it.

The general lesson: when a fact about your problem is cheap to state and would otherwise be learned one conflict at a time, state it. Clauses are nearly free (Section 3.12); conflicts are not.

3.9. Symmetry breaking

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, and the solver does not know they are equivalent; it visits them all. Worse, on an UNSAT instance it must *refute* them all.

The fix is to post extra clauses that pick a canonical representative from each equivalence class. The queens board of Section 3.7 has a left-right mirror symmetry, broken by one clause—an at-least-one over the left half of row 0:

```
X(0,0)  X(0,1)  X(0,2)  X(0,3)      row-0 queen in the left half
```

Every solution is either canonical under this clause or the mirror of one that is, so nothing is lost and half the tree is gone. The same move appears in Chapter 4 as "pin vertex 0 to colour 0". Symmetry breaking is the cheapest large speedup in SAT modelling; look for it before touching anything else.

3.10. What SAT cannot count

The pigeonhole formula—$n+1$ pigeons into n holes, no two sharing—is UNSAT for the counting reason every human sees instantly. Encoded in clauses (an at-least-one per pigeon, pairwise at-most-ones per hole), it is the textbook hard case for CDCL: resolution proofs of pigeonhole are exponentially long, so no sequence of learned clauses can be short, and the solver's decisions blow up exponentially with n while propagations and conflicts stay unhelpfully calm. The library's example `ex1_pigeonhole.c` lets you watch it happen: seven pigeons into six holes is 42 variables and 133 clauses, and the refutation costs 1159 decisions, 949 conflicts, and nine restarts—brutal work for a fact about counting to seven.

The lesson is not that SAT is weak; it is that SAT reasons by resolution, and arguments that are fundamentally about *counting* or *arithmetic* live outside resolution's short reach. When your problem contains such an argument, encode it away (the counter encodings of Section 3.2), hand it to a theory (Part Three), or hand it to arithmetic natively (Part Five).

3.11. Reading the answer

After SAT_SATISFIABLE, every variable has a value: `sat_value(s, v)` returns SAT_TRUE or SAT_FALSE. Variables the formula did not constrain either way get an arbitrary value—typically whatever the decision heuristic last picked, but not contractually so. If you care about the values of unconstrained variables, pin them with unit clauses.

After SAT_UNSATISFIABLE, the formula has no model. The library does not emit an UNSAT proof; production solvers do (DRAT format), and for high-assurance uses that matters—the proof can be checked by a tool much smaller than the solver. For tutorial purposes the verdict alone is enough.

Unsatisfiability is a result, not a malfunction. Sometimes it is the answer you wanted (pigeonhole; the two-colouring of Chapter 4). When it is unintentional, the debugging pattern is relaxation: comment out clause blocks until SAT returns, then narrow to the block—and finally the clause—whose removal flips the verdict. Chapter 6 shows the same idea promoted to an algorithm.

3.12. Tuning knobs

Worth restating as advice rather than API. *The encoding dominates the solve time*—iterate on it, using the statistics, before reaching for solver options. *Watch variables, not clauses*—the watched-literal scheme makes clause count nearly free, while every decision must choose among the variables; the sudoku example added three thousand clauses to save a thousand conflicts, and the trade was right. *Prefer encodings that propagate*—a structural constraint the solver can only discover by search, rather than by unit propagation, is a constraint you have hidden from it.

CHAPTER FOUR

TUTORIAL—GRAPH COLOURING

Graph colouring asks whether the vertices of a graph can be coloured with k colours so that no edge joins two vertices of the same colour. It is the cleanest possible exercise for the encoding patterns of Chapter 3—finite domains, at-most-one, symmetry—and it produces both answers a SAT solver can give: a model, and a proof that none exists.

We colour the Petersen graph: ten vertices, fifteen edges, chromatic number three. (Readers of the companion volume will recognise it; the same graph is coloured there by constraint propagation, and comparing the two programs is instructive.)

4.1. The encoding

One Boolean per (vertex, colour) pair: x_{vc} means "vertex v has colour c". Three constraint families:

1. Every vertex has at least one colour: one clause of k literals per vertex.
2. Every vertex has at most one colour: the pairwise encoding, $\binom{k}{2}$ binary clauses per vertex.
3. No edge is monochromatic: for each edge (u, v) and each colour c, the binary clause $\neg x_{uc} \vee \neg x_{vc}$.

For n vertices, m edges, and k colours: nk variables and $n + n\binom{k}{2} + mk$ clauses. For the Petersen graph at $k = 3$: 30 variables, 85 clauses.

4.2. The program

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

#define N 10
#define K 3

/* x(v, c): "vertex v has colour c"; variables number from 1. */
static int x(int v, int c) { return v * K + c + 1; }
```

```
int main(void) {
    /* The Petersen graph: outer pentagon 0..4, inner
     * pentagram 5..9, spokes joining them. */
    static const int edges[][2] = {
        {0,1}, {1,2}, {2,3}, {3,4}, {4,0},
        {5,7}, {7,9}, {9,6}, {6,8}, {8,5},
        {0,5}, {1,6}, {2,7}, {3,8}, {4,9},
    };
    enum { M = sizeof edges / sizeof edges[0] };

    Sat *s = sat_new(N * K);

    for (int v = 0; v < N; v++) {
        int alo[K];
        for (int c = 0; c < K; c++) alo[c] = x(v, c);
        sat_add_clause(s, alo, K);            /* at least one  */
        for (int c = 0; c < K; c++)
            for (int d = c + 1; d < K; d++) { /* at most one   */
                int amo[2] = { -x(v, c), -x(v, d) };
                sat_add_clause(s, amo, 2);
            }
    }
    for (int e = 0; e < M; e++)               /* edges differ  */
        for (int c = 0; c < K; c++) {
            int ne[2] = { -x(edges[e][0], c), -x(edges[e][1], c) };
            sat_add_clause(s, ne, 2);
        }

    if (sat_solve(s) == SAT_SATISFIABLE) {
        for (int v = 0; v < N; v++)
            for (int c = 0; c < K; c++)
                if (sat_value(s, x(v, c)) == SAT_TRUE)
                    printf("vertex %d: colour %d\n", v, c);
    } else {
        printf("no %d-colouring\n", K);
    }
    sat_free(s);
    return 0;
}
```

Compiled against the library and run, it prints a valid 3-colouring in well under a millisecond. Note what the program does *not* contain: no search code, no propagation code, no undo machinery. The program states; the solver searches.

4.3. Proving two colours impossible

Change `K` to 2 and rerun. The program prints `no 2-colouring`—and this verdict is a theorem, not a failure report. The Petersen graph contains the odd cycle 0–1–2–3–4–0, and odd cycles are not 2-colourable.

It is worth tracing how the solver finds the argument. Suppose the first decision sets $x_{0,0}$ true. The at-most-one clause forces $x_{0,1}$ false; the edge clause on (0,1) forces $x_{1,0}$ false; the at-least-one clause on vertex 1 forces $x_{1,1}$ true. Propagation marches around the pentagon in exactly this way—true, false, true, false—until vertex 4 and vertex 0 are forced to share colour 0, violating the edge clause on (4,0). One conflict; the learned clause rules out the opening; the mirrored opening fails symmetrically; the solver reports UNSAT after a handful of conflicts and essentially no search. The graph-theoretic argument about odd cycles and the trace of unit propagation are the same argument.

The library stages both runs in `ex3_coloring.c`, and the measured statistics confirm the trace: $k = 2$ is refuted with one decision and two conflicts; $k = 3$ finds its model in eight decisions and no conflict at all.

4.4. The chromatic-number loop

To find the chromatic number rather than test a fixed k: solve for $k = 1, 2, 3, \ldots$ until the first SAT. Each k builds a fresh context—the one-shot discipline of Chapter 1—and each UNSAT verdict below the answer is a proof, not a timeout. Symmetry breaking earns its keep here: pinning vertex 0 to colour 0 (a unit clause) and, more aggressively, vertex 1 to colour 0 or 1, removes the factor of $k!$ by which colour-permuted refutations would otherwise be repeated.

4.5. Lessons

1. *Pick variables that match the problem's vocabulary.* The problem is about vertices having colours; the variables are (vertex, colour) pairs.

2. *The encoding is three mechanical patterns.* At-least-one, at-most-one, and an implication per edge; nothing here was invented for colouring.

3. *UNSAT is an answer.* The 2-colouring verdict is the odd-cycle theorem, found by unit propagation.

4. *Break symmetry before scaling up.* The colour classes are interchangeable; tell the solver so, or it will discover each permutation separately.

The same encoding style handles sudoku, scheduling feasibility, and configuration checking without new ideas. What it does not handle is preference—when not everything can be had and some choices are worth more than others. That is Part Two.

PART TWO

PREFERENCES

CHAPTER FIVE

WEIGHTED PARTIAL MAXSAT

Every realistic threat picture is over-constrained. Every realistic schedule has more demands than resources. Every realistic configuration has compromise built in. A SAT solver can tell you that a configuration satisfying everything does not exist; what you then need is the configuration that sacrifices least. That is MaxSAT's question.

5.1. Hard and soft

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, the capacities, whatever cannot be negotiated. *Soft* clauses are preferences. Each carries a weight; the solver returns a model of the hard clauses minimising the summed weight of the soft clauses it leaves unsatisfied.

The distinction is a modelling decision, and getting it wrong in either direction is the most common MaxSAT mistake. A preference declared hard produces MAXSAT_HARD_UNSAT the first time the world over-commits—encode "every threat must be engaged" as hard, and a saturated picture yields no answer at all instead of the best triage. A rule declared soft produces answers that quietly break it when the weights happen to line up. When in doubt: physics is hard, doctrine is expensive.

5.2. A first program

Two variables, one hard mutual exclusion, 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 preference is sacrificed,
     * so x = 1 and y = 0 in the optimal model. */
    printf("cost %ld, x=%d, y=%d\n", cost,
           maxsat_value(m, x), maxsat_value(m, y));

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

Variables are allocated through `maxsat_new_var`; literals are signed integers as in Chapter 2, which is no coincidence—the SAT engine underneath is the same `sat.c`. The unweighted `maxsat_add_soft_clause` is shorthand for weight 1. Weight 0 is a silent no-op; negative weights are invalid and dropped—if you find yourself wanting one, you probably want to negate the clause instead.

`maxsat_add_at_most_one`, the one convenience helper on the posting surface, posts the pairwise at-most-one of Section 3.1 as hard clauses; for at-most-k with $k > 1$ you encode manually, and the sequential-counter pattern lifts directly into user code.

5.3. Weights, not counts

With every weight equal, the solver minimises the *number* of unsatisfied softs, treating every preference as equally important. This is almost never what you want for a real problem. If one unmet preference is a leaked cruise missile and another is a leaked drone, "either is acceptable to miss" is rarely the actual doctrine.

Weight your soft clauses, even crudely, even when unsure of precise values. A weight of 10 against 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 "true" value. Section 7.3 makes the stakes concrete: on the same instance, the unweighted optimum is a tactical disaster that the weighted optimum simply avoids.

5.4. Priorities by weight gaps

Strict hierarchies—"satisfy every priority-1 preference before paying any priority-2 cost"—need no separate solver mode. Give the priority-1 clauses weights large enough that no combination of priority-2 clauses can outweigh a single one of them: with N priority-2 clauses of weight 1, any priority-1 weight of $N{+}1$ or more does it. This emulates lexicographic MaxSAT exactly. Production lex-MaxSAT solvers implement the same semantics more efficiently, but the modelling pattern is identical, and at tutorial scale the gap encoding is entirely adequate.

5.5. Hard-UNSAT is not "everything sacrificed"

If the hard clauses are jointly infeasible, no model exists at any cost, and `maxsat_solve` returns the verdict `MAXSAT_HARD_UNSAT`. Do not conflate this with the very different state "feasible, but every soft was sacrificed"—the latter has a model and a finite cost equal to the total soft weight. Meeting this verdict in practice almost always means a preference was declared hard; the fix is to soften it with a high weight, at which point the solver resumes producing least-bad answers instead of no answers.

CHAPTER SIX

FINDING THE OPTIMUM

MaxSAT is an optimisation problem solved by a feasibility engine. The library implements the two classical ways of bridging that gap. Both produce the same optimum; they approach it from opposite sides, and knowing which side you are on is useful when reading the statistics.

6.1. Linear search

The model-improving approach descends from above. Find any model of the hard clauses; its cost is an upper bound. Post a constraint forcing strictly better cost; solve again; repeat. The first UNSAT proves the incumbent optimal.

Each iteration is one SAT call on a formula augmented with a cardinality constraint over the softs' relaxation variables. The approach is conceptually obvious, easy to trust, and fine when the optimum is close to the initial model—few iterations, each cheap. Its weakness is symmetric: when the optimum is far from the first model, the loop grinds down one cost unit at a time.

6.2. Unsatisfiable cores

The core-guided approach ascends from below, and pivots on a concept worth having for its own sake. An *unsatisfiable core* is a subset of clauses that is already jointly infeasible. When the hard clauses plus all softs are UNSAT, the SAT engine can report a core: a specific set of softs that cannot all be honoured.

A core is diagnosis, not just mechanism. "These five preferences are jointly impossible; every solution sacrifices at least one" is an explanation a human can act on, and it falls out of the solver for free. The over-constrained instances of Chapter 7 produce cores that read directly as tactical statements about which demands collide.

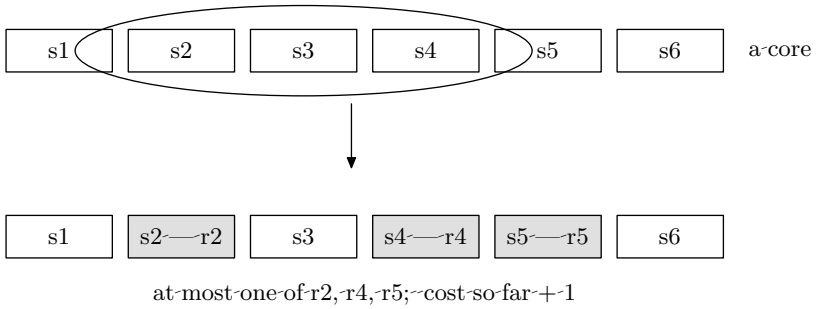

Fig. 5. One Fu–Malik iteration. Solving with all softs asserted is UNSAT; the engine reports a core (here s2, s4, s5). Each core member is relaxed by a fresh variable, at most one relaxation may fire, and the round costs exactly 1. Iteration continues until the relaxed formula is satisfiable.

6.3. Fu–Malik

The Fu–Malik algorithm turns cores into an optimisation procedure:

1. Solve the hard clauses plus all current softs. If SAT, stop: the accumulated cost is the optimum and the model is optimal.

2. Otherwise extract a core. Every soft in it gains a fresh *relaxation variable* r_i (the clause becomes $s_i \vee r_i$); an at-most-one over the new r_i is added as hard; the cost rises by one.

3. Repeat.

The invariant: each round proves that *any* solution must sacrifice at least one soft in the reported core, pays for exactly one, and asks whether that payment suffices. The optimum is reached from below—the accumulated cost is always a valid lower bound—and the number of rounds equals the optimum cost.

This is the architecture of every modern competition MaxSAT solver (with refinements: weight splitting, cardinality networks, core minimisation), and the library's implementation is the honest small version of it.

6.4. Weights by replication

The library handles weights by replication: a soft clause of weight w becomes w identical unit-weight clauses internally. This is conceptually transparent—weight literally equals count—and requires no algorithmic change, but it scales linearly in total weight. Weights up to a few hundred are fine. Weights in the thousands or millions, common in production models wanting fine-grained relative preferences, need a solver with native weight-splitting Fu–Malik or pseudo-Boolean encodings. The library finds the right answer regardless; it just takes its time. Keep your weight scale as coarse as your preferences honestly are.

6.5. Choosing between the algorithms

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

Core-guided is the right default, and the library uses it unless told otherwise. The exception is pathologically small instances, where deletion-based core extraction costs one SAT call per soft clause and that overhead can exceed the algorithmic gain; the example `ex2_core_vs_linear.c` stages exactly this comparison, and on its at-most-3-of-7 instance linear wins 30 SAT decisions to 524. In production, with incremental SAT engines and assumption-based cores, the trade goes the other way decisively.

6.6. The statistics

After solving:

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

For core-guided runs, `cores_found` equals the optimum cost modulo replication—it counts unit relaxations, i.e. total weight relaxed. For linear runs it is always zero, and `iterations` counts the descent steps instead. A SAT decision count implausibly high for the instance size usually means deletion-based core extraction is churning through a formula whose total weight is much larger than its structure; the fix is a coarser weight scale.

CHAPTER SEVEN

TUTORIAL—ASSIGNMENT UNDER SATURATION

The canonical MaxSAT use case in this book's home domain is over-constrained assignment: more demands than the resources can prosecute, every demand carrying a value, minimise what leaks through. The instance here is weapon-target assignment under saturation, but Section 7.4 shows the same shape wearing five other uniforms; substitute freely.

7.1. The problem

Three weapons face six threats. Each threat has a value—10, 5, 8, 3, 7, 2 for T0 through T5—and a compatibility row saying which weapons can engage it. Each weapon can prosecute at most one threat; each threat receives at most one weapon. Three weapons, six threats: at least three threats leak, and the only question is which.

7.2. The model

One Boolean per compatible (weapon, threat) pair, two families of hard at-most-ones, one weighted soft per threat:

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

/* x[w][t] = "weapon w engages threat t", where compatible. */
for (int w = 0; w < N_WEAPONS; w++)
    for (int t = 0; t < N_THREATS; t++)
        if (threat[t].compat[w])
            x[w][t] = maxsat_new_var(m);

/* Hard: at most one engagement per weapon. */
for (int w = 0; w < N_WEAPONS; w++)
    maxsat_add_at_most_one(m, row_of_weapon[w], n_in_row[w]);

/* Hard: at most one weapon per threat. */
```

```
for (int t = 0; t < N_THREATS; t++)
    maxsat_add_at_most_one(m, col_of_threat[t], n_in_col[t]);

/* Soft: each threat engaged, weighted by its value. */
for (int t = 0; t < N_THREATS; t++)
    maxsat_add_soft_clause_weighted(m, col_of_threat[t],
                                    n_in_col[t], threat[t].value);
```

(The two bookkeeping arrays gather the eligible variables of each weapon and of each threat; the complete program is `ex3_dwta_weighted.c` in the MaxSAT examples.)

Note the division of labour. The at-most-ones are physics: a weapon cannot be in two engagements. The per-threat softs are doctrine: engaging T0 is worth 10, engaging T5 is worth 2. Nothing anywhere says "engage exactly three"—the arithmetic of saturation emerges from the constraints.

7.3. Weighted versus unweighted

On the saturated scenario the core-guided solver finds the optimum in ten cores: engage the three highest-value threats—T0 (10), T2 (8), T4 (7)—and leak T1, T3, T5. Total leakage 10; engaged value 25 of the 35 on the board. The statistics read: eleven Fu–Malik iterations, ten cores, 3260 SAT decisions in total. The ten cores are the ten units of sacrificed weight, and each intermediate core names a set of engagement demands that cannot all be met.

Now delete the weights—make every soft weight 1—and re-solve the identical compatibility structure. The solver engages T2, T4, and T5: three threats, three weapons, "optimal" by count. Leaked: T0 (value 10) and T1 (value 5) among them—total leakage 18 against the weighted solver's 10, with the single most dangerous threat on the board among the leakers. 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 practical fact about MaxSAT: you almost certainly need weights. If you reach for unweighted MaxSAT on a real problem, check whether you actually mean it or defaulted to 1 because it was easier to type.

7.4. The same shape elsewhere

Once seen, the hard/soft/weight pattern is everywhere. Radar dwell scheduling: beam-time per second is hard; "track i gets a revisit this second" is soft, weighted by priority and error-growth rate. Crew scheduling: qualifications and rest minima are hard; shift preferences and fairness are weighted softs. Maintenance versus sorties: inspection intervals are hard; each sortie and each overdue maintenance task is a weighted soft, and the solver balances the two demand

streams against the airframe pool. Frequency assignment under jamming: the interference graph is hard; "emitter stays on its preferred band" is soft, weighted by the cost of losing it.

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

7.5. Lessons

1. *Hard is for physics, soft is for doctrine.* MAXSAT_HARD_UNSAT on a real picture usually means a preference was declared hard.

2. *Weights carry the tactics.* The unweighted optimum can be the wrong answer at any price; ten-to-one crude weights already fix it.

3. *Cores are explanations.* Each one names a set of demands that cannot coexist—keep them, log them, show them to the operator.

4. *Mind the total weight.* Replication makes weight a resource; spend it on relative order, not on false precision.

MaxSAT optimises over Booleans. When the quantities themselves—positions, times, amounts—need to be first-class, clauses alone stop being enough. Part Three adds arithmetic without giving up the engine.

PART THREE

THEORIES

CHAPTER EIGHT

SATISFIABILITY MODULO THEORIES

Clauses are expressive but blind. A SAT variable is just a Boolean; whatever it means—"$x \le 5$", "$a = b$", "the box is left of the toolbar"—the meaning lives in your encoding, and the solver cannot reason about it. Chapter 3 turned small finite domains into Booleans by brute enumeration; the moment the quantities are real numbers, enumeration is not even an option.

SMT—satisfiability modulo theories—keeps the CDCL engine and gives some of the atoms meaning. An atom may now be an arithmetic statement like $x + y \le 5$ or an equality like $f(a) = f(b)$, and dedicated *theory solvers* check that the set of atoms the SAT engine currently believes is actually consistent under that meaning. The library implements two theories—equality with uninterpreted functions and linear real arithmetic—and the Nelson–Oppen channel that combines them.

8.1. The two-level architecture

There are two solvers inside, stacked.

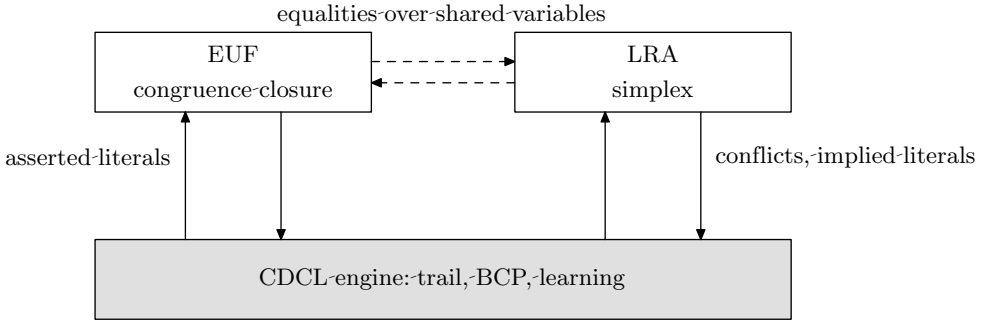

Fig. 6. The DPLL(T) architecture. The CDCL engine assigns truth values to atoms it does not understand; the EUF and LRA theory solvers check each partial assignment for consistency, returning conflict clauses and implied literals through the propagation channel.

At the bottom, the CDCL engine of Chapter 2 sees only literals. It has no idea what the atoms mean; it does unit propagation, clause learning, and backjumping on a flat universe of propositions, exactly as before.

Above it, the theory solvers understand what the atoms say and check consistency of whatever subset is currently true on the trail. When the SAT engine extends its partial assignment, each theory looks at the atoms newly asserted and reports back one of three things: *consistent* (keep searching, or declare SAT if the assignment is complete); *inconsistent* (here is a clause naming a subset of asserted literals that cannot all hold—the engine learns it like any conflict clause); or *implied literals* (here is a literal the theory has derived—assert it on the trail before guessing further).

The third channel, theory propagation, is what powers the Nelson–Oppen combination: each theory advertises what it has learned, the SAT trail carries it, and the other theory sees the consequences. You never thread information between theories by hand. Posting the right atoms and clauses is enough; the channel does the threading.

8.2. Equality with uninterpreted functions

The EUF theory reasons about equalities between terms built from constants and function applications, knowing nothing about the functions except that they are functions: equal arguments give equal results. That single axiom—congruence—plus the reflexivity, symmetry, and transitivity of equality is the whole theory.

The classic three-liner: $a = b$ and $b = c$ together imply $a = c$, so asserting $a \neq c$ alongside is unsatisfiable.

```
SmtCtx *ctx = smt_new();
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(ctx) returns SMT_UNSAT. */
```

The solver finds this without exploring any choices: congruence closure propagates $a = c$ from the chain, the negated equality already on the trail contradicts it, and the search ends on the first theory check.

Function applications make it interesting. Terms made with `smt_mk_app` are hash-consed—`smt_mk_app(f, [a])` returns the same term id every time—so when the solver learns $a = b$, discovering that $f(a)$ and $f(b)$ must be equal is a single union-find merge, not a search. Asserting $f(a) \neq f(b)$ next to $a = b$ is UNSAT by exactly that merge.

8.3. Linear real arithmetic

The LRA theory decides conjunctions of linear inequalities over real-valued variables, using the Dutertre–de Moura simplex—the algorithm at the heart of Z3 and cvc5—over exact rational arithmetic. There is no floating point anywhere: every coefficient and every derived value is an `int64` numerator over an `int64` denominator, and two rationals that are mathematically equal compare equal regardless of how they were computed.

The user-facing vocabulary is small. A linear term is an array of `SmtLinTerm`—variable, coefficient numerator, coefficient denominator—and an atom bounds a sum of them:

```
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 */
```

Only two atom shapes exist: $\leq$ and $<$. There is no "assert greater than"; you build the opposite-facing atom and assert its negation. $x > 3$ is the negation of $x \leq 3$:

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

With $x + y \leq 5$, $x > 3$, and $y > 3$ posted together, the two lower bounds give $x + y > 6$, contradicting the upper bound on the sum; `smt_check` returns UNSAT without a single decision. The four polarities of the two atom shapes cover every inequality you will ever want:

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

This table repays memorising; getting a strict bound where you meant a weak one is the most common LRA modelling slip, and Section 9.7 returns to it.

8.4. The Nelson–Oppen combination

A variable that participates in both theories—the argument of an uninterpreted function *and* a term in a linear constraint—is declared shared:

```
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);
```

An `SmtShared` is a struct of two ids, one per theory; `x.euf` goes wherever a term goes and `x.lra` wherever a variable goes. At `smt_mk_shared` time the library eagerly installs, for the new shared paired with every existing one, three *bridge atoms*

```
eq_ij : (s_i.euf == s_j.euf)        -- an EUF atom
le_ij : (s_i.lra - s_j.lra <= 0)    -- an LRA atom
ge_ij : (s_i.lra - s_j.lra >= 0)    -- an LRA atom
```

and three linking clauses: `eq` implies `le`, `eq` implies `ge`, and `le` with `ge` implies `eq`. That, plus each theory's own propagation, is the entire Nelson–Oppen machinery—the SAT engine's BCP carries the cross-theory implications through the linking clauses. The bridges are pure propagation channels: they cost nothing until they fire, though their count grows quadratically in the number of shareds.

The classic demonstration runs the channel in both directions. Assert $x = y$ and $f(x) \neq f(y)$: EUF merges the classes, congruence gives $f(x) = f(y)$, contradiction. Now instead pin the values numerically:

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

LRA pins both variables to the same value, the bridge atom $x = y$ propagates back to EUF, congruence closes the contradiction. Nothing in your code mentioned the connection; the bridges carried it.

8.5. The statistics

After `smt_check`, five counters tell you which layer did the work:

```
smt_decisions            /* literals the SAT engine guessed     */
smt_conflicts            /* learned-clause cycles               */
smt_theory_conflicts     /* conflicts reported by a theory      */
smt_theory_props         /* literals a theory pushed unasked    */
smt_lra_pivots           /* simplex pivots inside LRA           */
```

A pure-LRA problem with zero decisions is one the tableau dispatched directly, no propositional case analysis at all. Theory propagations above zero mean a theory forced literals the encoding never asserted. Many decisions on a problem you expected to be easy is the signature of disjunctive structure—and the modelling response, reducing disjunctions by making implicit orderings explicit, is the subject of Section 9.5.

8.6. The edges

The solver is one-shot: build a context, assert, call `smt_check` once, read the model, free. Calling `smt_check` twice on one context returns unknown; make a fresh context per question. And the theory set is exactly the two described: no integer arithmetic (LIA is non-convex and needs case splitting inside the theory), no non-linear terms, no quantifiers, no optimisation. Section 10.6 discusses what to do when a problem wants what is missing.

CHAPTER NINE

TERMS, ATOMS, AND CLAUSES

The SMT API asks you to keep three vocabularies straight—terms for EUF, variables for LRA, atoms for the SAT layer—and rewards you with precise control over which theory sees what. This chapter walks the object model and the modelling patterns that recur in every SMT program. The patterns compose; the tutorial in Chapter 10 uses most of them at once.

9.1. The five kinds of object

Contexts. An `SmtCtx` is the solver state: one per problem, created empty, populated, checked once, freed.

Terms (type `SmtTerm`, an opaque integer) belong to EUF: constants from `smt_mk_const`, applications from `smt_mk_app`. Applications are hash-consed, which is what makes congruence detection a merge instead of a search.

Variables (type `SmtVar`, also an integer) belong to LRA, made with `smt_mk_real_var`. The two namespaces do not overlap; a term is not interchangeable with a variable even if the underlying integers happen to match, and the compiler will not catch a mixup.

Shareds (type `SmtShared`) live in both worlds, as Section 8.4 described. If a quantity will ever be both a function argument and a constraint term, make it shared from the start; retrofitting sharedness mid-build is not supported.

Atoms (type `SmtAtom`) are the propositions the SAT layer holds truth values for: `smt_mk_eq_atom` builds an EUF equality, while `smt_mk_lra_le_atom` and `smt_mk_lra_lt_atom` build the two LRA shapes. Atoms are 1-indexed positive integers; a literal is a signed atom; a clause is an array of literals. The whole propositional surface is one function, `smt_assert_clause`.

9.2. The unit assertion

Most constraints are unconditional, and the pattern "make an atom, put its id alone in a clause" is so common that shortcuts bundle the two steps:

```
smt_assert_eq(ctx, a, b);              /* shortcut */

int atom = smt_mk_eq_atom(ctx, a, b);  /* the long form */
int unit = atom;
smt_assert_clause(ctx, &unit, 1);
```

The full shortcut set—`smt_assert_eq`, `_neq`, `_lra_le`, `_lra_lt`, `_lra_eq`, `_shared_eq`, `_shared_neq`—covers every unconditional assertion. Use them by default; reach for the explicit form only when you also need the atom id for a larger clause.

9.3. Negation by sign flip

Disequality is the negated equality atom, asserted as a negative unit:

```
int atom = smt_mk_eq_atom(ctx, a, b);
int unit = -atom;
smt_assert_clause(ctx, &unit, 1);      /* a != b */
```

The same move gives every reversed inequality, per the polarity table of Section 8.3. The design takes getting used to—atoms only ever face one way—but the payoff is a solver whose internals handle strict bounds exactly (via delta-rationals; see the further reading), and after a few programs the sign flip becomes reflex.

9.4. Equality as two inequalities

An LRA equality $e = r$ is the conjunction $e \leq r$ and $e \geq r$; the second is the negation of $e < r$. The shortcut `smt_assert_lra_eq` builds both atoms and both unit clauses. If you need the equality as one disjunct of a larger clause—"$x = 5$ or $y = 7$"—there is no single atom to negate; you build the two atoms per side and write the clause structure by hand. Section 10.4 has a worked instance.

9.5. Case analysis as a clause

A disjunction of theory atoms is just a clause with more than one literal, and it is the whole reason to use SMT rather than a plain LP solver. The canonical example is non-overlap of two width-w rectangles on a shared row: A entirely left of B, or B entirely 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, -w, 1);
int b_left = smt_mk_lra_le_atom(ctx, b_minus_a, 2, -w, 1);
int cl[2] = { a_left, b_left };
smt_assert_clause(ctx, cl, 2);
```

Each branch is one literal. The SAT layer decides which holds; LRA confirms or refutes the choice; a refutation becomes a learned clause that rules the ordering out everywhere.

9.6. Chains beat pairwise disjunctions

When a total order on the placed items is known—B0 leftmost, then B1, then B2—replace the pairwise disjunctions 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, -(w + GAP), 1);
}
```

Every clause is a unit; the SAT layer has nothing to decide; the whole problem lives in the tableau. This is the single most important optimisation in SMT modelling of layout-like problems, and the difference is measurable even at toy sizes: the tutorial's four-buttons-in-a-row solves in zero decisions against the disjunctive form's enumeration of orderings. The price is the commitment—if the order genuinely is not known, the disjunction is the model.

9.7. Reusing atoms

Equality atoms are deduplicated by the constructor: the same pair returns the same atom regardless of argument order—$\mathrm{eq}(a, b)$ and $\mathrm{eq}(b, a)$ are one proposition. LRA atoms are *not*: constructing $x + y \leq 5$ twice yields two atoms the solver treats as independent propositions, with no propagation between them. If a combination recurs across clauses, build the atom once and reuse the id.

9.8. Common pitfalls

Forgetting to assert. Atom construction registers a proposition; it does not constrain anything. An atom in no clause is free, and the SAT layer will happily set it either way. Unexpected SAT nearly always means an atom you built but never asserted.

Strict versus weak. Unexpected UNSAT nearly always means a strict bound where you meant a weak one, or the reverse. The polarity table settles it.

Mixing namespaces. An `SmtVar` cannot be a function argument. Wanted that? The variable should have been shared from the start.

Coefficient overflow. Exact rationals on `int64` can overflow under adversarial pivot chains. Normalise inputs—divide by the gcd, pick small representatives—before posting, and stay within tutorial sizes.

CHAPTER TEN

TUTORIAL—PLACING RECTANGLES

Here is the problem this part has been building toward: given a canvas and a set of UI elements with fixed sizes, find a position for every element satisfying a list of layout constraints. It is a good showpiece for SMT because it sits exactly on the seam between theories: alignment, spacing, and containment are linear arithmetic, while non-overlap is disjunctive. Tools that handle only one side—linear-constraint engines like Cassowary on the one hand, pure combinatorial backtrackers on the other—must either relax the problem or discretise it. SMT takes both sides at once.

Four programs, in rising order of structure. Each is a complete example in the SMT distribution; the text gives the constraint skeleton and reads the statistics.

10.1. Two boxes (ex10)

Two rectangles, each 40 wide, in a canvas 100 wide, must not overlap. Two free variables—the x-coordinates—and five constraints: each box within the canvas (two bounds apiece), plus the non-overlap disjunction of Section 9.5, with $w = 40$.

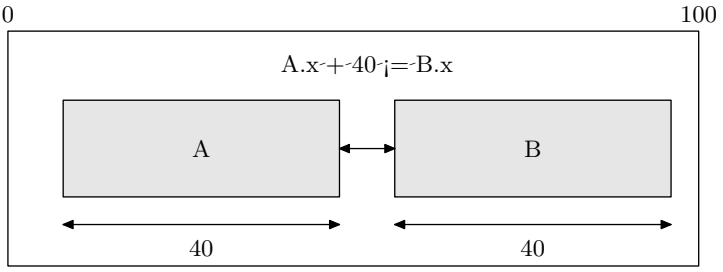

Fig. 7. The smallest disjunctive layout problem: two 40-wide boxes in a 100-wide canvas. The one SAT decision picks which box goes left; the simplex places them.

The output:

```
result       = SAT (valid layout exists)
decisions    = 1
theory conf. = 0
LRA pivots   = 2
```

One decision: the disjunction had to be resolved one way or the other. No theory conflicts: the chosen branch was feasible. Two pivots to position the boxes. This is the baseline profile of a problem with exactly one genuine choice in it.

10.2. Four buttons in a row (ex11)

A row of four buttons whose left-to-right order is fixed. The chain encoding replaces all pairwise non-overlap:

```
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);
}
```

plus first-button-after-zero and last-button-inside-canvas bounds. The profile:

```
decisions    = 0    (pure chain: no disjunction anywhere)
theory conf. = 0
LRA pivots   = 4
```

Zero decisions. Every clause is unit, BCP fills the trail before the first guess, and the problem never leaves the tableau. Compare the disjunctive encoding of the same row: six two-literal clauses whose orderings the SAT layer must enumerate. Same solutions, different machine.

10.3. Four free boxes (ex12)

The general case: four 40-by-30 boxes in a 100-by-100 canvas, no fixed ordering. Each pair contributes a four-literal clause—A left of B, B left of A, A above B, B above A—for six pairs, twenty-four atoms, plus the containment bounds.

```
atoms        = 40
decisions    = 31
conflicts    = 6
theory conf. = 6
```

Thirty-one decisions and six theory conflicts: the SAT layer proposes orderings, LRA refutes the ones that do not fit, each refutation is learned, and the sixth backjump lands on a feasible arrangement. This is DPLL(T) earning its keep—and also the shape of the cost curve to expect, since n boxes mean $n(n-1)/2$ clauses and a worst case exponential in the orderings. Adding any structure you know—quadrant assignments, partial orders—collapses the search back toward the chain profile.

10.4. A real toolbar (ex13)

Three 24-pixel icon buttons in a 240-by-40 toolbar, under the constraints any reasonable toolbar imposes: containment with 8-pixel padding; no overlap; vertical centering; equal horizontal spacing; fixed left-to-right order.

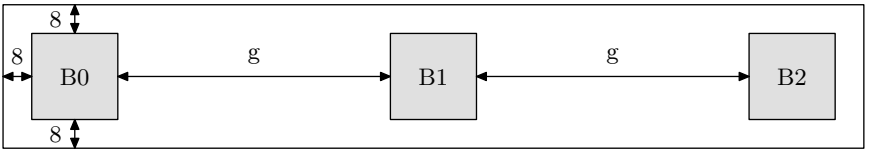

Fig. 8. The toolbar of ex13. Vertical centering and equal spacing are linear equalities; the fixed order turns non-overlap into a chain. Nothing remains for the SAT layer, and the whole layout is five simplex pivots.

Vertical centering is an equality per button:

```
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 known order turns non-overlap into a two-link chain, and equal spacing—gap(B0,B1) = gap(B1,B2)—rearranges to one linear equality, $2x_1 - x_0 - x_2 = 0$:

```
SmtLinTerm spacing[3] = { { x1, 2, 1 }, { x0, -1, 1 },
                          { x2, -1, 1 } };
smt_assert_lra_eq(ctx, spacing, 3, 0, 1);
```

The profile: zero decisions, zero conflicts, five pivots. That is the take-home of the whole part. Design intent—alignment, ordering, even spacing, centering—is linear structure, and every piece of it you state explicitly removes propositional work. Reach for disjunction only where the problem genuinely is unordered.

10.5. Where SMT sits

Below SMT, pure linear engines: Cassowary and its Auto Layout descendants handle prioritised linear constraints fast, with no case analysis at all. Above it, production solvers: cvc5 and Z3 add integer arithmetic, non-linear theories, quantifiers, optimisation, and industrial scale. The library of this part occupies the pedagogical middle: every constraint in this tutorial ran through code you can read in an evening, and the architecture—CDCL below, theories above, explanations flowing between—is the same one the production tools scale up.

10.6. Lessons

1. *Put structure in the theory, choices in the clauses.* The solver is fast exactly in proportion to how much of the problem is linear.

2. *Known order is free performance.* Chains beat pairwise disjunctions by an exponential margin; state every ordering you know.

3. *The statistics name the bottleneck.* Decisions count disjunctive work, pivots count arithmetic work; the remedy differs.

4. *Atoms are propositions, not constraints.* Nothing holds until a clause says so.

The theories in this part were real-valued. The next part restores integers—not by adding an integer theory, but by encoding bounded integers into the clauses themselves and letting propagators explain their reasoning to the learning machinery.

PART FOUR

INTEGERS

CHAPTER ELEVEN

LAZY CLAUSE GENERATION

Constraint programming and SAT approached the same problems from opposite ends for twenty years. CP has expressive finite-domain variables and global constraints whose propagators exploit structure no clause can see; but classical CP backtracking revisits the same dead end as many times as the tree leads there. SAT has clause learning—meet a dead end once, never meet its relatives—but a flat Boolean vocabulary in which a scheduling constraint explodes into thousands of clauses.

Lazy clause generation is the hybrid. From outside it looks like CP: declare integer variables, post constraints, ask for a model. Inside it is the CDCL engine of Chapter 2, and the propagators are theory solvers in exactly the sense of Part Three, with one refinement that gives the technique its name: when a propagator tightens a bound, it constructs, on demand, a clause explaining *why*. That explanation enters the learnt database and prunes every similar dead end elsewhere in the tree. The eager SAT encoding of the same constraint would post thousands of clauses up front; LCG generates exactly the ones the search actually needed, and none it did not.

11.1. Integer variables and the order encoding

Every LCG variable is a finite-domain integer with a fixed range:

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

Internally the domain is Booleans: one atom per threshold, where b_k means "$x \le k$", with a monotonicity chain $b_k \to b_{k+1}$ keeping the encoding consistent. You never manipulate the encoding directly, but two of its consequences shape every model you will write.

First, domains must be finite and modest. A variable over $[0, 1000]$ allocates a thousand atoms; the library copes, but past some point a coarser model is the better answer.

Second, the encoding is bound-oriented. "$x \le 5$" is one literal; "$x \neq 7$" is a two-literal clause ($x \le 6$ or $x \ge 8$); and constraints that reason about individual interior values are expressible but not native. Bounds, intervals, and time windows are what LCG is for. The literal helpers expose the encoding when you want to write clauses yourself:

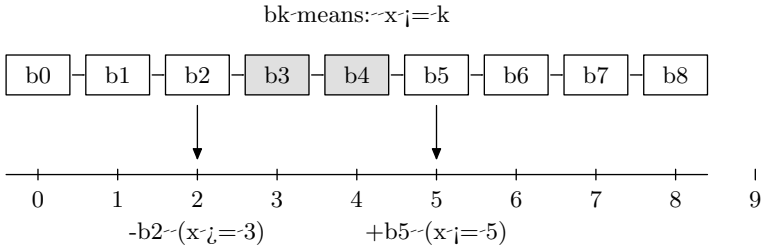

Fig. 9. The order encoding of a domain. One Boolean atom per threshold $x \le k$; the implication chain runs rightward. A bound is one literal; excluding the single value 4 (shaded region of the number line) takes two, $x \le 3 \vee x \ge 5$.

```
int l1 = lcg_lit_le(ctx, x, 5);   /* literal for "x <= 5" */
int l2 = lcg_lit_ge(ctx, x, 3);   /* literal for "x >= 3" */
lcg_post_clause(ctx, &l1, 1);
```

The special returns `LCG_LIT_TRUE` and `LCG_LIT_FALSE` flag relations already trivially settled by the domain, so callers can short-circuit. During and after search, `lcg_lb` and `lcg_ub` read a variable's current bounds and—after `lcg_solve` returns `LCG_SAT`—`lcg_value` reads its model value.

11.2. A first program

Two integers in $[0, 9]$ summing to at most 5, 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;
}
```

The shape of Chapter 1 exactly, with integers in place of Booleans.

11.3. Linear inequalities

The workhorse constraint posts $\sum_i c_i \cdot v_i \leq k$ over integer coefficients:

```
lcg_post_linear_le(ctx, coeff, vars, n_terms, k);
```

There is deliberately no `_ge`, `_eq`, or `_lt` variant; all are sign flips and offsets away, and an equality is two opposite-facing posts:

```
/* x - y == 3, 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);
```

The propagator does bounds reasoning: from the current bounds of every term it computes the extreme values of the sum, reports a conflict when the minimum already exceeds k, and tightens individual variables as the slack shrinks. The explanations are surgical: when it pushes $x \leq 4$ because the other terms leave only that much room, the clause it emits cites exactly the bound literals it used, nothing more—and that clause then works for the learning machinery everywhere.

11.4. Alldifferent

```
lcg_post_alldifferent(ctx, vars, n);
```

enforces pairwise-distinct values, with bounds-consistent propagation: when some k variables have their domains squeezed inside a k-element interval of values—a Hall set—every other variable is pushed out of that interval, and a Hall set of k variables inside fewer than k values is a conflict. What it does not do is remove single values from the middle of a domain; the matching-based domain-consistent algorithm would, but it needs per-value atoms beyond the order encoding, and it is not in this library.

The practical reading: `alldifferent` bites hardest when bounds are already tight against each other, and contributes little while domains overlap loosely. Search narrows the bounds; the Hall sets emerge; the propagator closes in. The pattern to remember from its smallest use—three variables in $[1, 3]$, all different, is a permutation of $\{1, 2, 3\}$—recurs at every scale.

11.5. Auxiliary variables: N-Queens

LCG propagators take variables, not expressions. To say "`col`$[i] + i$ are all different"—the /-diagonal constraint of N-Queens—you introduce an auxiliary variable per offset expression and link it with the two-inequality equality:

```
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 };
    int    c2[2] = { -1,  1 };
    IntVar v [2] = { aux, src };
    lcg_post_linear_le(ctx, c1, v, 2,  offset);
    lcg_post_linear_le(ctx, c2, v, 2, -offset);
    return aux;                      /* aux == src + offset */
}
```

One `col` variable per row, two auxiliary families `d1[i] = col[i] + i` and `d2[i] = col[i] - i`, three alldifferents—columns, /-diagonals, \-diagonals—and the model is complete. The full program, `ex3_nqueens.c`, defaults to $N = 6$ (19 decisions, 11 conflicts) and takes N on the command line; at $N = 8$ it is 30 decisions and 17 conflicts on a fresh search. Globals over auxiliaries linked by linear equalities is the workhorse pattern of LCG modelling; nearly every interesting model ends up shaped like it. (Each equality costs two posts, so where a later constraint could use the source variable with an adjusted constant instead, prefer that.)

CHAPTER TWELVE

CUMULATIVE AND CUSTOM PROPAGATORS

Scheduling is where LCG pulls decisively ahead of everything else in this book, and one constraint carries most of the weight.

12.1. The cumulative constraint

```
lcg_post_cumulative(ctx, starts, dur, demand, n_tasks, capacity);
```

says: n tasks, each with a start-time variable, a fixed duration, and a fixed resource demand, each occupying the half-open interval $[s_i, s_i + d_i)$; at no time may the total demand of the running tasks exceed the capacity. One call captures what a clause encoding would smear across thousands of clauses and a MILP formulation across hundreds of time-indexed binaries.

The propagation is time-table reasoning, built on one elegant observation. However uncertain a task's start still is, part of its execution may already be certain: with $\mathrm{lb}(s) = 2$, $\mathrm{ub}(s) = 4$, and duration 3, the possible executions $[2, 5)$, $[3, 6)$, $[4, 7)$ all contain time 4. The *mandatory part* is $[\mathrm{ub}(s),\ \mathrm{lb}(s) + d)$, non-empty exactly when $\mathrm{ub}(s) - \mathrm{lb}(s) < d$.

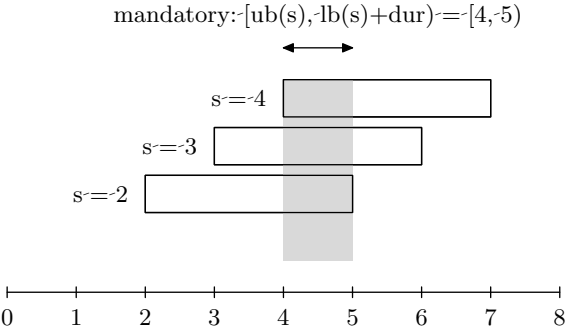

Fig. 10. The mandatory part. A duration-3 task whose start is bounded in $[2, 4]$ might run as any of the three dashed intervals, but every one of them covers time 4: the intersection $[\mathrm{ub}(s), \mathrm{lb}(s) + d)$ is guaranteed load wherever the start lands.

Summing demand over every task's mandatory part gives a per-time profile of guaranteed load. If the profile anywhere exceeds capacity, the constraint is infeasible right now—conflict, with an explanation citing the bound literals that created the mandatory parts. And if placing some task j at a candidate start would push the profile over capacity, j's bounds are tightened to skip past the obstruction, again with a surgical explanation.

The unary special case—every demand 1, capacity 1: one machine, one channel, one human in the loop—has its own entry point:

```
lcg_post_no_overlap(ctx, starts, dur, n_tasks);
```

Time-table is the simplest of the published cumulative propagators; edge-finding and energetic reasoning catch more conflicts sooner at a real cost in code. For problems up to a few dozen tasks—which covers the tutorial and most embedded uses—time-table is enough.

12.2. Writing a propagator

When the built-in vocabulary runs out, the propagator interface the built-ins use is yours too:

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

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

A propagate callback reads the current bounds through `lcg_lb` and `lcg_ub`; assembles an explanation, each call adding one bound literal to the reason; and then emits its deduction, returning one of the three `LCG_PROP` status codes: OK, TIGHTENED, or CONFLICT. The typical shape, from the header's own documentation:

```
lcg_expl_clear(ctx);
lcg_expl_lo(ctx, x, lcg_lb(ctx, x));  /* because x >= lb(x) */
lcg_expl_hi(ctx, y, lcg_ub(ctx, y));  /* because y <= ub(y) */
return lcg_emit_le(ctx, z, 5);        /* therefore z <= 5   */
```

The core converts the buffer to the right clause shape—negated supports, plus the derived literal—and hands it to the SAT engine; `lcg_emit_conflict` does the same with no derived literal. The core also takes ownership of `p.data`, freeing it through `p.destroy` at `lcg_free` time.

The discipline mirrors Part One's contract for theory solvers: read bounds, never mutate anything else, explain every deduction with exactly the literals that justify it. Explanations that cite too much still work but learn weak

clauses; explanations that cite too little are bugs. The built-in cumulative shows what "exactly" means. Its smallest explanation helper, verbatim from `prop_cumulative.c`:

```
/* Add the "task i is mandatorily running at time t" supports
 * to the current explanation buffer. */
static void cum_explain_mandatory(LcgCtx *ctx, IntVar s,
                                  int p, int t) {
    lcg_expl_hi(ctx, s, t);
    lcg_expl_lo(ctx, s, t - p + 1);
}
```

A duration-p task mandatorily runs at t precisely because $s \le t$ and $s \ge t-p+1$; those two bound literals are the whole reason, so those two are what the buffer gets. The distribution's `ex2_custom_prop.c` implements a `not_equal(x, k)` propagator in under fifty lines against the same interface, and the two built-in globals use nothing more private. A project that ends up writing two or three custom propagators—a doctrinal separation rule, a setup-time chain—is using the library the way it is meant to be used.

12.3. The statistics

```
lcg_decisions        /* guesses                            */
lcg_conflicts        /* learned-clause cycles              */
lcg_propagations     /* bound tightenings by propagators   */
lcg_explanations     /* explanation clauses generated      */
```

Many decisions with few conflicts means the propagators are not pruning—the search is degenerating toward plain DFS, and the model wants tighter windows or a stronger constraint. Conflicts with explanations growing apace is LCG working as designed: the lazy clauses are being generated and the learning machinery is consuming them.

CHAPTER THIRTEEN

TUTORIAL—SCHEDULING UNDER TIME WINDOWS

The assignment tutorial of Chapter 7 decided *which* demands to serve when there were too many. This one decides *when*. The two are consecutive layers of the same operational pipeline, and Chapter 17 assembles the pipeline; here we take the allocation as given and schedule it.

13.1. The problem

Six inbound threats are tracked. Each has an arrival time (when it enters the engagement envelope), a deadline (when it impacts if not engaged), and an engagement duration (launch through intercept, during which one fire-control illumination channel is held). The radar has two channels. Doctrine: every threat engaged, entirely within its window, channels never over-committed.

```
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 },
};
```

13.2. The model

One start variable per threat, its domain doing the time-window work, and a single cumulative over the lot:

```
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 entire model. The domain bounds encode "within the window"—a start can be no earlier than arrival and no later than deadline minus duration—and the cumulative encodes the channel budget. On the six-threat picture the solver lands on a schedule in seven decisions:

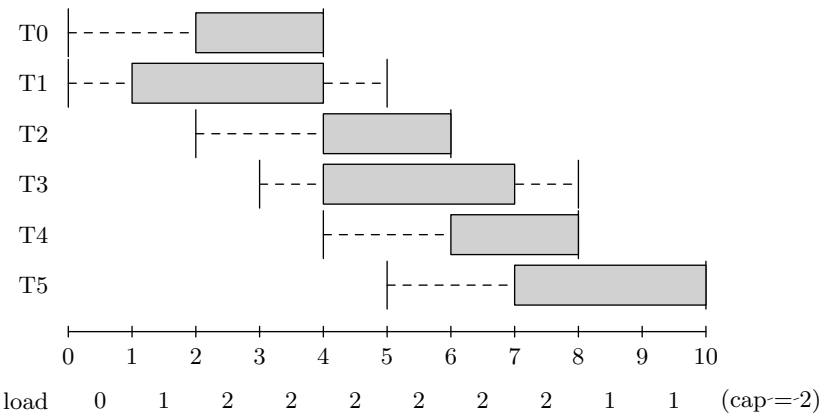

Fig. 11. The schedule found for the six-threat picture. Each bar is an engagement occupying one illumination channel for its duration inside its window; the bottom row is the channel load, riding at the capacity of 2 through the middle of the timeline.

Every threat is engaged inside its window, and the load rides at the cap for most of the timeline—the defence fully committed, never over-committed. The statistics are almost embarrassing:

```
decisions      = 7
conflicts      = 0
propagations   = 2
explanations   = 3
```

Seven guesses, no conflict. The windows are tight enough that the time-table propagator steers every guess into place.

13.3. Reading infeasibility

Drop the capacity to 1 with the same picture and the solver returns UNSAT at the root: one conflict, zero decisions. The time-table propagator sums the mandatory parts—T0 confined to $[0, 4)$ with duration 2 and T1 to $[0, 5)$ with duration 3 already have overlapping guaranteed load—and the profile exceeds a single channel before any guess is made. That immediacy is worth pausing on: the solver did not search and fail to find a schedule, it *proved* none exists, and the explanation names the windows that collide. In an operational loop, that proof arriving in microseconds is what lets the layer above react—shed a demand, relax a window—within the same decision cycle.

13.4. What is not in this model

Kill probabilities, weapon-target affinity, magazine depth, optimisation of anything: absent, deliberately. The cumulative model answers one question—given that these engagements must happen, when do they happen?—and answers it completely. Which layer of a system that question belongs to, and what the neighbouring layers look like, is Chapter 17's subject.

13.5. The same shape elsewhere

Strip the uniforms and the model is: tasks with release times, deadlines, durations, unit demands; a resource of capacity k; find start times. That is bay scheduling in a repair shop, operating theatres in a hospital wing, berths at a quay, test rigs in a lab, interview slots across a panel. If your problem statement contains the words "window", "deadline", or "at most k at once", this chapter's twenty lines are its model.

13.6. Lessons

1. *Domains are constraints.* The time windows cost nothing to post; they are the variables' bounds.

2. *One global beats a thousand clauses.* The cumulative sees structure—mandatory parts, load profiles—that no clause-level encoding exposes.

3. *UNSAT is an answer.* Root-level infeasibility with an explanation is actionable intelligence about the resource budget.

4. *Feasibility, then preference.* LCG has no objective; it tells you what can be scheduled at all. Optimisation wraps it, or precedes it in another solver.

The last remark points forward. When the question is not "can it be done" but "what is it worth", and the quantities are continuous, clauses stop being the right substrate altogether. Part Five changes substrate.

PART FIVE

NUMBERS

CHAPTER FOURTEEN

LINEAR PROGRAMS AND THE SIMPLEX

Everything so far has been clauses at the bottom. This part's solver has no clauses at all. A mixed-integer linear program states an objective—a linear function of the decision variables—and constraints—linear inequalities or equalities—and asks for the assignment that optimises the objective among all that satisfy the constraints. Some variables range over the reals; some are required integer. It is the paradigm for problems about *how much*: budgets, flows, expected values, fractions of a resource.

Two layers do the work, and this chapter covers the inner one: the linear program that remains when integrality is ignored, and the simplex method that solves it.

14.1. The model surface

The library's API is index-based: variables are $0..n-1$, constraint rows $0..m-1$, both counts fixed at creation.

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

milp_set_objective(m, MILP_MAX, c);       /* or MILP_MIN     */
milp_set_row(m, i, a, MILP_LE, rhs);      /* also _EQ, _GE   */
milp_set_var_type(m, j, MILP_BINARY);     /* also _INTEGER,  */
                                          /* _CONTINUOUS     */
milp_set_var_bounds(m, j, lo, hi);        /* default [0,inf) */
```

Row i becomes $\sum_j a_j x_j$ `op` r. Variables default to continuous and non-negative; `MILP_BINARY` is shorthand for integer with bounds $[0, 1]$. Every row and the objective must be populated before solving—the library does not error on a forgotten row, it just leaves the row all zeros and the constraint trivially loose, which is worth a pre-solve assertion in your own code.

A complete program, solving a three-item knapsack:

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

int main(void) {
    MilpModel *m = milp_new(3, 1);
```

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

    /* 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);

    for (int j = 0; j < 3; j++)
        milp_set_var_type(m, j, MILP_BINARY);

    double x[3], obj;
    if (milp_solve(m, x, &obj) == 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;
}
```

It prints an objective of 16: items 0 and 1, sizes $4 + 6$ filling the capacity exactly. Five calls built the model; the library did the relaxation, the tree, and the bookkeeping.

14.2. The feasible polytope

Drop the integrality requirement for a moment. The constraints of a linear program carve out a convex polytope in n-dimensional space, and the linear objective attains its optimum at a vertex of it. Dantzig's simplex method walks from vertex to adjacent vertex, always improving, until no improving neighbour exists—and convexity makes that local stop a global optimum.

The library implements a two-phase revised simplex in `milp.c`'s 495 lines: phase one finds any vertex of the polytope (or proves the polytope empty), phase two walks to the best one. The figure also shows the complication that fills the next chapter: the LP optimum is a vertex of the polytope, and the integer points—the assignments a binary or integer variable is actually allowed—are a lattice inside it. The vertex is rarely on the lattice.

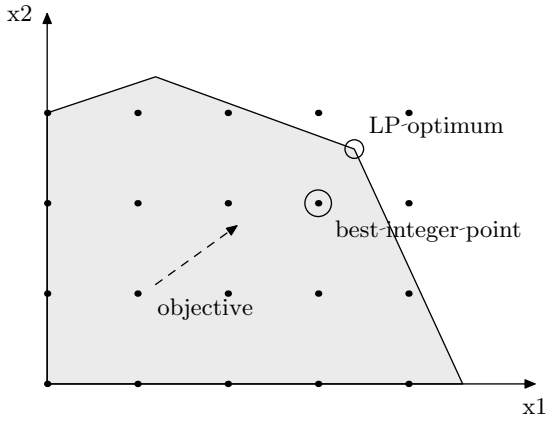

Fig. 12. An LP's feasible region is a convex polytope; the objective's level lines sweep across it, and the optimum sits at a vertex. The simplex walks the boundary from vertex to vertex; an integer program restricts attention to the lattice points inside, where the LP vertex (circled) may not lie.

14.3. Solving the relaxation alone

The LP relaxation is useful enough by itself that the library exposes it as a separate entry point:

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

This treats every variable as continuous and returns the LP optimum: for maximisation an upper bound on the integer optimum, for minimisation a lower bound. Three uses recur. As a formulation sanity check—an infeasible LP means an infeasible MIP, an unbounded LP means a modelling bug, both discovered at a fraction of the cost. As a cheap feasibility filter inside a loop, before committing to a full integer solve. And as the bound that Chapter 17 lends to other solvers: "no integer solution beats this" is a certificate any search can use.

14.4. Reading the verdict

```
MILP_OPTIMAL      /* x and obj hold the optimum           */
MILP_INFEASIBLE   /* no assignment satisfies the rows     */
MILP_UNBOUNDED    /* objective improves without limit     */
MILP_NODE_LIMIT   /* budget exhausted; x may hold a       */
                  /* feasible incumbent, unproven         */
MILP_ERROR        /* internal error; check stderr         */
```

UNBOUNDED is almost always a modelling mistake—a variable you forgot to bound above, a sign error in the objective, a missing constraint—and the unbounded direction the LP reports points at it. INFEASIBLE is more interesting: sometimes the request truly is impossible, sometimes the model is over-specified. The debugging pattern is to relax rows one at a time until feasibility returns; the last relaxation names the conflict. (Production solvers automate this as irreducible-infeasible-subsystem detection; at tutorial sizes the manual loop takes minutes.)

Two numerical cautions round out the LP layer. The library does not scale your matrix; if coefficients span many orders of magnitude—kilometre ranges beside fractional probabilities—normalise the rows yourself or the arithmetic suffers. And integrality is judged to a tolerance near 10^{-6}; formulations that put meaningful values at that boundary will confuse it.

CHAPTER FIFTEEN

BRANCH AND BOUND

The LP relaxation solves in polynomial time and lands, usually, on a fractional vertex. The integer answer must come from somewhere else, and the somewhere else is a search tree with the LP as its oracle.

15.1. The tree

At each node, solve the node's LP relaxation. Three outcomes. If the LP is infeasible or its bound cannot beat the best integer solution found so far—the *incumbent*—prune the node; nothing below it can win. If the LP optimum happens to be integer-feasible, it is a candidate: update the incumbent. And if it is fractional, pick a fractional variable $x_j = f$ and branch: one child adds $x_j \le \lfloor f \rfloor$, the other $x_j \ge \lceil f \rceil$, and no integer point is lost between them.

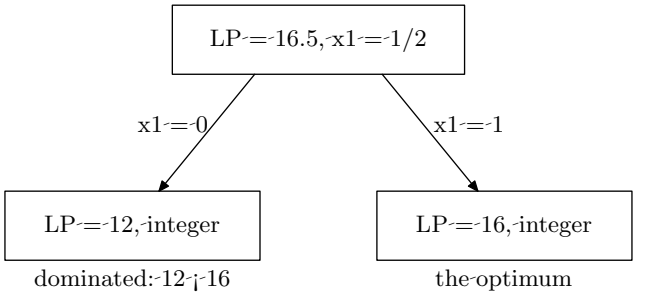

Fig. 13. A branch-and-bound node. The root LP is fractional at $x_1 = 1/2$ with bound 16.5; branching on x_1 gives two children whose LPs are both integer. The right child's 16 becomes the incumbent, and no open node's bound exceeds it, so 16 is optimal.

The bound is the engine of the whole method. A tight LP prunes almost everything and the tree stays shallow; a loose LP prunes almost nothing and the tree approaches enumeration. Which of these happens is a property of your formulation, not of the solver—and it is the single most useful thing to know about a MILP model before running it.

15.2. Three kinds of relaxation

Category 1: totally unimodular. Some constraint matrices—bipartite assignment, transportation, basic network flow—have the property that every vertex of the polytope is already integer. The LP at the root *is* the answer; branch-and-bound never branches. If your problem is an assignment problem, you are here, and the MIP costs one LP.

Category 2: tight, with a real gap. The 0/1 knapsack is the canonical case: Dantzig's classical analysis shows the LP optimum has at most one fractional variable, so the gap is bounded by one item's value and the tree stays shallow. The library's `ex1_knapsack.c` measures it on an eight-item instance: the LP relaxation is 23.5 with exactly one fractional variable, the integer optimum is 23, and branch-and-bound closes the gap in seven nodes. The relaxation dominates the runtime; branching cleans up.

Category 3: loose. Set covering is the canonical case: the LP returns a fractional cover far from any integer point, the bound is weak, and the tree can grow exponentially. Production solvers fight back with cutting planes that tighten the relaxation; this library does not, so a category-3 problem at scale wants either a stronger formulation or a production solver.

Diagnosing your category is cheap: `milp_lp_relaxation` returns the root LP value, and its distance from the integer optimum—the *integrality gap*—is the measurement. Half of MILP modelling skill is knowing your category; the other half is reformulating toward a better one.

15.3. Modelling patterns

Indicator variables. A binary y meaning "this happens", its objective coefficient the value or cost of it happening. The whole of Chapter 16 is indicators.

Big-M implication. "If $y = 0$ then $\sum a_j x_j \le 0$" becomes $\sum a_j x_j \le My$ for a large constant M: vacuous when $y = 1$, binding when $y = 0$. Big-M is numerically delicate—too small excludes valid solutions, too large turns the relaxation to mush—and choosing it needs domain knowledge. Use sparingly, and prefer structural reformulations where they exist. Disjunctions encode the same way, one binary steering which of two constraint bodies is active, with the same cautions doubled.

Budget rows. $\sum c_j x_j \le B$ over binaries. One such row is what separates category 1 from category 2—it destroys total unimodularity but only gently, as Chapter 16 demonstrates.

Covering rows. $\sum_{s \ni e} x_s \ge 1$ per element e: the set-cover shape, category 3, gap logarithmic in the worst case. Partitioning (equalities instead) is tighter—the LP cannot double-cover.

Time-indexed scheduling. A task becomes T binaries "running at time t", with duration and contiguity rows. The variable count scales with the horizon,

the relaxation is loose, and the whole construction is MILP at its weakest—which is why Part Four exists. Reach for it only when the horizon is short and something else forces MILP.

15.4. Knobs and statistics

Branching picks the most-fractional variable; the node budget defaults to 2^{20} (`milp_set_node_limit` adjusts it); `milp_set_verbose` at 1 prints a tree summary and at 2 a line per node. Diagnosis reads two counters against each other: `milp_nodes_explored` high with cheap LPs means the tree is the cost (look at the formulation's gap); nodes few but slow means the LPs are the cost (look at the matrix scaling). If you hit the node limit at these sizes, the formulation—not the budget—is the thing to fix.

CHAPTER SIXTEEN

TUTORIAL—BUDGETED ASSIGNMENT

The tutorial problem is the one this book's home domain calls static weapon-target assignment, and operations research calls the assignment problem with a side budget. It is chosen because it walks the category boundary of Chapter 15 in a single model.

16.1. The problem

Weapons i, targets j. Engaging target j with weapon i destroys it with probability p_{ij}; target j is worth v_j; the engagement costs c_{ij} in some budgeted resource—rounds, fuel, illumination seconds. Each weapon fires at most once, each target is engaged at most once, total cost within budget B. Maximise expected damage.

16.2. The model

The instance in `ex4_wta.c`: four weapons, three targets of values 10, 8, and 6, a 4×3 kill-probability matrix, a cost matrix, and a budget of 7. One binary x_{ij} per pair, indexed as `i*NT + j`. The objective is where the probabilities enter—and where MILP earns its place in this book, because no clause-based solver in Parts One through Four takes an expected value natively:

```
/* Objective: maximise expected damage. */
for (int i = 0; i < NW; i++)
    for (int j = 0; j < NT; j++)
        c[i * NT + j] = v[j] * p[i][j];
milp_set_objective(m, MILP_MAX, c);
```

Three families of rows, verbatim from the example:

```
/* Each weapon used at most once. */
int r = 0;
for (int i = 0; i < NW; i++) {
    for (int k = 0; k < n_vars; k++) row[k] = 0.0;
    for (int j = 0; j < NT; j++) row[i * NT + j] = 1.0;
    milp_set_row(m, r++, row, MILP_LE, 1.0);
}
/* Each target hit at most once. */
```

```
for (int j = 0; j < NT; j++) {
    for (int k = 0; k < n_vars; k++) row[k] = 0.0;
    for (int i = 0; i < NW; i++) row[i * NT + j] = 1.0;
    milp_set_row(m, r++, row, MILP_LE, 1.0);
}
/* Budget. */
for (int k = 0; k < n_vars; k++) row[k] = 0.0;
for (int i = 0; i < NW; i++)
    for (int j = 0; j < NT; j++)
        row[i * NT + j] = cost[i][j];
milp_set_row(m, r++, row, MILP_LE, B);
```

Twelve binaries, eight rows $(4 + 3 + 1)$, and the model is complete.

16.3. The category boundary

Delete the budget row and the remaining matrix is the bipartite incidence matrix of the assignment problem: totally unimodular, category 1, every LP vertex integer, the whole MIP one LP solve.

Restore the budget row and unimodularity is destroyed. The LP can now split a weapon fractionally across targets—committing 0.6 of a shot here and 0.4 there—to spend the budget with an efficiency no integral assignment can match. The relaxation is a strict upper bound; branch-and-bound must close the gap. But the damage is mild: one row among $W + T + 1$, a small gap, a shallow tree—textbook category 2.

16.4. The numbers

Running the example prints both layers. First the relaxation:

```
=== LP relaxation ===
  expected damage (UB) = 17.6000
  fractional engagements:
    W1 -> T1 : 0.500
    W2 -> T1 : 0.500
```

The LP splits target 1 between two weapons, half a commitment each, precisely the budget-sharing no shooter can execute. Then the integer plan:

```
=== Integer engagement plan ===
  expected damage = 16.6000
  fire orders:
    W0 -> T1   p=0.70  v=8   cost=3
    W1 -> T0   p=0.80  v=10  cost=3
    W3 -> T2   p=0.50  v=6   cost=1
  cost spent  = 7 / 7

LP bound       17.6000
```

```
MIP optimum    16.6000
Integrality gap 1.0000  (6.02%)
B&B nodes      13
```

Two readings are worth taking. The gap is 6 per cent and the tree is thirteen nodes—textbook category 2, one loose row's worth of damage. And the integer plan is *not* a rounding of the LP: the fractional solution had W1 and W2 sharing T1, while the optimum sends W0 to T1 and W1 elsewhere entirely. Branch-and-bound found a plan the relaxation never hinted at, which is exactly why the rounding heuristics of folklore are not a substitute for the search. Delete the budget row and re-run: the LP is integral at the root and the node count drops to one.

16.5. Modelling notes

Damage or leakage? Maximising expected damage and minimising expected leakage are the same static problem ($\sum v_j$ minus one is the other); the maximisation reads more cleanly, with no constant offset. In dynamic settings where *when* a threat leaks matters, leakage becomes the natural objective.

Multiple shots per target. The combined kill probability $1 - \prod_i (1 - p_{ij})^{x_{ij}}$ is not linear. The standard repair is logarithmic: with $u_{ij} = -\log(1 - p_{ij})$, the exponent $\sum_i u_{ij} x_{ij}$ is linear and monotone in the combined probability. This model sidesteps the issue by allowing one shot per target; the log-space transform is the first extension to reach for when that assumption goes.

Re-solving. A real system solves this continuously as the picture changes. The one-shot library rebuilds the model each cycle, which at these sizes is fine; production solvers warm-start the simplex from the previous basis and cut adjacent-solve times by an order of magnitude. Re-solve friendliness is a genuine MILP advantage over the clause-based solvers, whose learned state is harder to carry across runs.

16.6. Lessons

1. *Know your category.* One diagnostic LP tells you whether integrality will cost one node or a tree.

2. *The objective is native.* Values and probabilities go straight into coefficients; nothing in Parts One through Four does this.

3. *Structure is worth protecting.* Every row you add to a TUM core weakens the relaxation; add the ones the problem demands and no more.

4. *Fractional solutions are information.* The LP's split commitments show where the budget binds—worth reading even though you cannot execute them.

Five solvers are now on the bench. The last chapter decides among them.

PART SIX

CHOOSING

CHAPTER SEVENTEEN

WHICH SOLVER, AND HOW TO COMBINE THEM

Five solvers, one temperament, different strengths. This closing chapter puts them side by side on one problem family, extracts the rules of thumb, and describes the patterns by which real systems use several at once. The family is the one that has recurred through the tutorials—allocating and scheduling defensive resources against inbound demands—but every conclusion transfers; substitute your own nouns.

17.1. The axes that matter

Four questions locate a problem on the map.

Whether, or how much? Purely combinatorial decisions—this or that, present or absent—are clause territory. Continuous quantities, weighted sums, expected values are MILP territory. Bounded integers with structure sit between, which is where LCG lives.

Feasibility, or optimality? SAT, SMT, and LCG answer feasibility natively and optimise only by repeated solving. MaxSAT optimises over clause weights; MILP over linear objectives. Wrapping a feasibility solver in an optimisation loop works—Part Two is exactly that wrap, done well—but a native objective is worth seeking when the objective is the point.

How much arithmetic in the constraints? None: SAT. Linear over reals with case analysis: SMT. Linear over bounded integers, plus globals: LCG. Linear with an objective: MILP. Logic-rich constraints encode awkwardly in MILP—big-M for every implication, loose relaxations—exactly where clauses are at home; the reverse holds for weighted sums.

Time windows and shared resources? LCG, almost regardless of the other answers. One cumulative replaces the time-indexed binaries that make MILP scheduling formulations blow up, and replaces the clause avalanche that makes SAT encodings of the same thing unreadable.

17.2. The family verdicts

On the running problem family, the assignments fall out cleanly.

Doctrinal rule checking—compatibility, exclusion, configuration—is SAT: pure propositional structure, feasibility the only question, cores as explanations when rules collide.

Assignment under saturation is MaxSAT: hard physics, soft preferences, weights carrying value, the optimum a least-bad compromise (Chapter 7).

Geometric and mixed logical-numeric feasibility—placement, alignment, disjunctive design rules—is SMT (Chapter 10).

Engagement scheduling through limited channels inside time windows is LCG (Chapter 13).

Static optimal allocation with probabilities, values, and a budget is MILP (Chapter 16).

The classical backtracking toolkit of the companion volume slots in beside these: exact cover via dancing links remains the tool of choice when the inner subproblem is literally a cover—each demand covered exactly once, magazine slots as secondary items—and especially when *all* solutions are wanted, since enumeration is DLX's natural mode and sits awkwardly on every CDCL-based solver.

17.3. Two-phase decomposition

Real systems rarely pick one. The workhorse pattern splits allocation from scheduling:

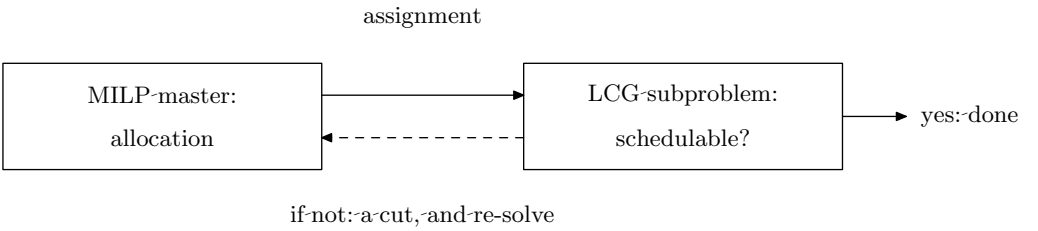

Fig. 14. Logic-based Benders decomposition. The MILP master proposes an allocation optimising the numeric objective; the LCG subproblem checks it against time windows and channel capacity. An infeasible proposal returns as a cut—a constraint excluding it, and ideally its relatives—and the master re-solves.

The master is a MILP because the allocation objective is numeric—expected damage, weighted values, budget. The subproblem is LCG because schedulability is time windows and a cumulative. The contract between them is the cut: at minimum "not this exact allocation", better "no allocation using these three weapons inside this window", and the tighter the cut, the fewer round trips. This is logic-based Benders decomposition, and it solves instances neither paradigm handles alone—the master never enumerates schedules, the subproblem never reasons about value.

Chapter 13's root-level UNSAT is what makes the loop fast in the infeasible direction: the subproblem's verdict, with its explanation naming the colliding windows, arrives in microseconds and becomes the cut almost verbatim.

17.4. Bounds across paradigms

The LP relaxation travels well. Solve it once and hand the value to whatever search actually runs the integer problem—a MaxSAT loop, an LCG optimisation wrap, a bespoke branch-and-bound: "no solution beats this" prunes in any of them. The reverse gift, from clauses to MILP, is the unsatisfiable core: when the MILP's constraints came from a clause layer, a core names the smallest colliding subset far more legibly than a relax-rows-one-at-a-time loop.

17.5. Column generation

When the space of candidate plans is astronomically large but the optimum uses only a few, invert the decomposition. The master becomes an LP or MILP choosing among whole candidate schedules—columns—and a pricing subproblem, naturally an LCG or DLX search, generates new candidates whose dual prices say they would improve the master. Crew scheduling, vehicle routing, and cutting stock run on this pattern at industrial scale. On top of the tutorial libraries it is a moderate project; the point here is to recognise the shape, because when the two-phase pattern stalls, this is usually the next move.

17.6. Rolling horizons

Operational problems are dynamic: demands arrive, engagements complete or fail, the picture changes every few seconds. The standard container is the rolling horizon—solve the next-K-seconds problem, execute the first decisions, re-plan with fresh state. Every library in this book is one-shot per context, which fits the container exactly: a fresh context per cycle, stale state impossible by construction. Which solver runs inside the horizon is decided by everything above; the horizon itself is indifferent.

17.7. One solver to rule them all?

The integrated end of the road exists: CP-SAT in Google's OR-Tools is essentially a productionised LCG with native optimisation, a rich constraint library, and an LP layer consulted for bounds—the paradigms of this book fused into one engine. The commercial MILP solvers approach the same fusion from the other side, with callbacks admitting combinatorial reasoning into the tree. When a problem outgrows these libraries, those are the destinations, and everything in this book is the map: the statistics read the same, the modelling patterns transfer verbatim, and the decomposition seams remain visible inside the fused engines.

For a system built from scratch, start with the two-phase pattern. It is simple, the contract between layers is explicit and testable, and each layer uses the tool this book has shown to be right for it.

17.8. What this book has not covered

Incremental solving under assumptions, the mechanism production SAT solvers use to re-solve cheaply; proof logging and certified UNSAT; presolve and cutting planes on the MILP side; stronger propagation (edge-finding, domain-consistent alldifferent) on the LCG side; MaxSMT and optimisation modulo theories, which would fuse Parts Two and Three. Each is a known engineering target from where the libraries stand, and each library is small enough that adding one is a project, not a career.

17.9. Closing

The five solvers of this book—and the backtracking toolkit of its companion—are one instrument with interchangeable heads: state the problem declaratively, let search meet structure, read back an answer with a guarantee attached. None of the implementations is novel; all of them are honest, small enough to hold in one head, and shaped like the production tools they introduce. The libraries are in front of you, under four thousand lines all told. Read them. Modify them. Break them and put them back together. The best way to internalise these techniques is to build something with them.

Good luck.

APPENDIX A

BUILDING THE LIBRARIES

Each library builds with one command and no dependencies beyond libc. All commands are run from the library's distribution root, with sources in `src/` and worked examples in `examples/`. Every distribution also ships a `build.zsh` that compiles the core into `build/`, runs the sanity tests, and builds every example; the one-line equivalents below are for linking the solver into a program of your own.

A.1. SAT

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

A.2. MaxSAT

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

A.3. SMT

The SMT core is five translation units—the SAT engine, the exact rationals, the two theories, and the DPLL(T) glue:

```
gcc -std=c99 -O2 -Isrc your_program.c src/sat.c src/rat.c \
    src/euf.c src/lra.c src/smt.c -o your_program
```

A.4. LCG

```
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
```

A.5. MILP

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

The `-lm` is the only external library anywhere in the book.

APPENDIX B

API SUMMARIES

The five public interfaces, one section each. Every function below appears in the body of the book; the section references point back to the primary discussion.

B.1. SAT (Chapters 2–4)

```
Sat *sat_new(int n_vars);          /* vars 1..n exist         */
void sat_free(Sat *s);
int  sat_add_clause(Sat *s, const int *lits, int n);
                                   /* 1 if now trivially UNSAT */
int  sat_solve(Sat *s);            /* SAT_SATISFIABLE (10) /   */
                                   /* SAT_UNSATISFIABLE (20) / */
                                   /* SAT_UNKNOWN (0)          */
int  sat_value(Sat *s, int v);     /* SAT_TRUE / SAT_FALSE     */

void sat_set_conflict_limit(Sat *s, long limit);   /* -1 = off */
void sat_set_verbose(Sat *s, int level);           /* 0, 1, 2  */

long sat_decisions(Sat *s);        long sat_conflicts(Sat *s);
long sat_propagations(Sat *s);     long sat_restarts(Sat *s);
long sat_learned_clauses(Sat *s);
```

B.2. MaxSAT (Chapters 5–7)

```
MaxSatCtx *maxsat_new(void);       void maxsat_free(MaxSatCtx *m);
int  maxsat_new_var(MaxSatCtx *m);
void maxsat_add_hard_clause(MaxSatCtx *m, const int *lits, int n);
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);
void maxsat_add_at_most_one(MaxSatCtx *m, const int *lits, int n);

void maxsat_set_algorithm(MaxSatCtx *m, MaxSatAlgorithm alg);
         /* MAXSAT_ALG_LINEAR or MAXSAT_ALG_CORE_GUIDED     */
MaxSatAlgorithm maxsat_get_algorithm(const MaxSatCtx *m);
```

```
long maxsat_solve(MaxSatCtx *m);   /* optimum cost, or       */
                                   /* MAXSAT_HARD_UNSAT (-1); */
                                   /* once per context        */
int  maxsat_value(const MaxSatCtx *m, int var);
                                   /* 0 or 1; 0 if unknown    */

long maxsat_iterations     (const MaxSatCtx *m);
long maxsat_cores_found    (const MaxSatCtx *m);
long maxsat_total_decisions(const MaxSatCtx *m);
```

B.3. SMT (Chapters 8–10)

```
SmtCtx *smt_new(void);             void smt_free(SmtCtx *ctx);

/* EUF terms */
int smt_mk_const(SmtCtx *ctx, const char *name);
int smt_mk_app(SmtCtx *ctx, const char *f, const int *args, int n);

/* LRA variables; SmtLinTerm is { var, num, den } */
int smt_mk_real_var(SmtCtx *ctx, const char *name);

/* shared variables: struct { euf, lra } */
SmtShared smt_mk_shared(SmtCtx *ctx, const char *name);

/* atoms; literals are signed atom ids; eq atoms dedupe */
/* regardless of argument order; LRA atoms do not       */
SmtAtom smt_mk_eq_atom(SmtCtx *ctx, SmtTerm a, SmtTerm b);
SmtAtom smt_mk_lra_le_atom(SmtCtx *ctx, const SmtLinTerm *t,
                           int n, int64_t rhs_num, int64_t rhs_den);
SmtAtom smt_mk_lra_lt_atom(SmtCtx *ctx, const SmtLinTerm *t,
                           int n, int64_t rhs_num, int64_t rhs_den);
void smt_assert_clause(SmtCtx *ctx, const int *signed_atoms, int n);

/* unit-assertion shortcuts */
void smt_assert_eq (SmtCtx *ctx, SmtTerm a, SmtTerm b);
void smt_assert_neq(SmtCtx *ctx, SmtTerm a, SmtTerm b);
void smt_assert_lra_le(SmtCtx *ctx, const SmtLinTerm *t, int n,
                       int64_t rhs_num, int64_t rhs_den);
void smt_assert_lra_lt(SmtCtx *ctx, const SmtLinTerm *t, int n,
                       int64_t rhs_num, int64_t rhs_den);
void smt_assert_lra_eq(SmtCtx *ctx, const SmtLinTerm *t, int n,
                       int64_t rhs_num, int64_t rhs_den);
void smt_assert_shared_eq (SmtCtx *ctx, SmtShared a, SmtShared b);
void smt_assert_shared_neq(SmtCtx *ctx, SmtShared a, SmtShared b);
```

```
int smt_check(SmtCtx *ctx);        /* SMT_SAT / SMT_UNSAT;     */
                                   /* one shot per context     */

/* model inspection and term printing */
int smt_model_eq(SmtCtx *ctx, SmtTerm a, SmtTerm b);
int smt_term_print(const SmtCtx *ctx, SmtTerm t,
                   char *buf, int buf_sz);

int  smt_n_atoms(const SmtCtx *ctx);
int  smt_n_terms(const SmtCtx *ctx);
long smt_decisions       (const SmtCtx *ctx);
long smt_conflicts       (const SmtCtx *ctx);
long smt_theory_conflicts(const SmtCtx *ctx);
long smt_theory_props    (const SmtCtx *ctx);
long smt_propagations    (const SmtCtx *ctx);
long smt_lra_pivots      (const SmtCtx *ctx);
void smt_set_verbose(SmtCtx *ctx, int v);
```

B.4. LCG (Chapters 11–13)

```
LcgCtx *lcg_new(void);             void lcg_free(LcgCtx *ctx);
IntVar lcg_new_int_var(LcgCtx *ctx, int lo, int hi,
                       const char *name);

int  lcg_lit_le(LcgCtx *ctx, IntVar x, int k);
int  lcg_lit_ge(LcgCtx *ctx, IntVar x, int k);
        /* may return the LCG_LIT_TRUE / LCG_LIT_FALSE     */
        /* sentinels for trivially settled relations       */
void lcg_post_clause(LcgCtx *ctx, const int *lits, int n);
        /* handles sentinels: a true literal removes the   */
        /* clause, a false literal is dropped from it      */

void lcg_post_linear_le(LcgCtx *ctx, const int *coeff,
                        const IntVar *vars, int n_terms, int k);
void lcg_post_alldifferent(LcgCtx *ctx, const IntVar *vars, int n);
void lcg_post_cumulative(LcgCtx *ctx, const IntVar *starts,
                         const int *dur, const int *demand,
                         int n_tasks, int capacity);
void lcg_post_no_overlap(LcgCtx *ctx, const IntVar *starts,
                         const int *dur, int n_tasks);

void lcg_register_propagator(LcgCtx *ctx, LcgPropagator p);
        /* the core owns p.data; freed via p.destroy       */
void lcg_expl_clear(LcgCtx *ctx);
void lcg_expl_lo(LcgCtx *ctx, IntVar v, int lo);
        /* cite the support "v >= lo"                      */
```

```
void lcg_expl_hi(LcgCtx *ctx, IntVar v, int hi);
        /* cite the support "v <= hi"                        */
int  lcg_emit_le(LcgCtx *ctx, IntVar v, int k);
int  lcg_emit_ge(LcgCtx *ctx, IntVar v, int k);
int  lcg_emit_conflict(LcgCtx *ctx);
        /* each returns LCG_PROP_OK / _TIGHTENED /         */
        /* _CONFLICT                                       */

int  lcg_solve(LcgCtx *ctx);   /* LCG_SAT / LCG_UNSAT /       */
                               /* LCG_UNKNOWN                 */
int  lcg_lb   (const LcgCtx *ctx, IntVar x);
int  lcg_ub   (const LcgCtx *ctx, IntVar x);
int  lcg_value(const LcgCtx *ctx, IntVar x);

long lcg_decisions    (const LcgCtx *ctx);
long lcg_conflicts    (const LcgCtx *ctx);
long lcg_propagations (const LcgCtx *ctx);
long lcg_explanations (const LcgCtx *ctx);
void lcg_set_verbose(LcgCtx *ctx, int level);
```

B.5. MILP (Chapters 14–16)

```
MilpModel *milp_new(int n_vars, int n_constraints);
void milp_free(MilpModel *m);

void milp_set_objective(MilpModel *m, int sense, const double *c);
        /* MILP_MIN or MILP_MAX */
void milp_set_row(MilpModel *m, int row, const double *a,
                  int op, double rhs);
        /* MILP_LE, MILP_EQ, MILP_GE */
void milp_set_var_type(MilpModel *m, int var, int type);
        /* MILP_CONTINUOUS, MILP_INTEGER, MILP_BINARY */
void milp_set_var_bounds(MilpModel *m, int var,
                         double lo, double hi);

int milp_solve   (MilpModel *m, double *x_out, double *obj_out);
int milp_solve_lp(MilpModel *m, double *x_out, double *obj_out);
        /* MILP_OPTIMAL, MILP_INFEASIBLE, MILP_UNBOUNDED, */
        /* MILP_NODE_LIMIT, MILP_ERROR                    */

int    milp_nodes_explored(const MilpModel *m);
int    milp_lps_solved    (const MilpModel *m);
double milp_lp_relaxation (const MilpModel *m);  /* root LP  */
void   milp_set_verbose(MilpModel *m, int level); /* 0, 1, 2 */
void   milp_set_node_limit(MilpModel *m, int limit);
                                          /* default 1<<20 */
```

INDEX