

Predictive Algorithms

A short tour through trails, sparse sets, constraint solving, and exact cover

Copyright © 2026 by Streck.ai

Preface

This is a short book about a small family of algorithms with a common shape. Each one explores a space of possibilities, commits tentatively to choices, discovers contradictions, and recovers without losing its footing. This is what I mean by *predictive*: not predicting the future, but trying out a future and being prepared to walk it back.

The book grew out of a real codebase. Every algorithm and data structure described here is implemented in a small C99 library — under two thousand lines total — that you can read, compile, and modify. The code is the primary text; the prose is here to explain why the code is shaped the way it is, and to help you write programs that use the library well.

Four pieces appear in roughly this order:

The **trail** is the recovery mechanism. It is a tiny stack of *(address, old value)* pairs that lets a program walk forward through tentative state changes and walk backward through them in $O(k)$, where k is the number of changes since the checkpoint, regardless of how big the underlying state is. Every interesting predictive algorithm in this book records its mutations on a trail and uses checkpoints to back out of failed branches.

The **sparse set** is a data structure for sets drawn from a small known universe. Two arrays of n integers give $O(1)$ membership, $O(1)$ insertion, $O(1)$ removal, and the ability to iterate over present elements in dense form without skipping holes. It is the right shape for a constraint variable's domain.

CSP — constraint satisfaction — is a framework for building solvers from variables, domains, and constraints. The constraints prune the domains; depth-first search picks branches where pruning leaves work undone. This book shows how to write programs against the library's CSP API for several classic problems, and how to write your own propagator when the built-in set isn't enough.

DLX — Dancing Links — is an alternative encoding for a different shape of problem: when your task is to pick a subset of options such that each thing-that-needs-covering is covered exactly once, DLX represents this directly as a doubly-linked grid in which removing rows and columns is reversible by inspection. The same problems that fit a CSP often fit a DLX too, but the encoding is different and the trade-offs are worth understanding.

The intended reader is a competent C programmer who has not used these tools before. You should be comfortable with pointers, allocation, structs, and basic algorithm analysis. You do not need any background in formal logic or combinatorial optimisation. Where mathematical notation appears it is kept loose and tied to the code.

The library is deliberately small. It is not a competitor to MiniZinc or Knuth's CWEB programs; it is a concrete artefact you can hold in one head. Every function in the public API is shown in this book. Every tutorial program compiles against it.

A note on style. The code listings in this book are taken from the library 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 straight through the first time. Each chapter builds on the one before, and the early chapters establish ideas (the trail, the sparse set) that are quietly used everywhere later.

If you want to skim, the priority order is roughly:

1. Chapters 1–3, which give you the mental model and the data structures.
2. Chapters 4–6, which establish the CSP API.
3. One CSP tutorial of your choice (7, 8, or 9).
4. Chapters 10–12 for DLX.
5. One DLX tutorial (13 or 14).
6. Chapter 15 for guidance on which tool fits a problem.

The tutorials are independent of one another. You can pick 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, modify a constraint, run a benchmark. Both meanings are intended literally.

Part I — Foundations

Chapter 1. The shape of predictive search

Every algorithm in this book follows the same outline. Make a tentative commitment. Propagate the consequences. If the consequences are contradictory, undo the commitment and try a different one. If the consequences are consistent but incomplete, recurse with another tentative commitment on top.

This shape has a name in the literature — *backtracking search* — but the name conceals the interesting question. The interesting question is: *how do you undo?*

The naive answer: don't

A first attempt at solving a constraint problem in C looks something like this:

```
int solve(int depth) {
    if (all_assigned()) return 1;
    var_t v = pick_variable();
    for (int32_t value : domain_of(v)) {
        assign(v, value);
        propagate();
        if (consistent() && solve(depth + 1)) return 1;
    }
    return 0;
}
```

This is wrong. After the recursive call returns without finding a solution, the assignment to *v* and any propagation that followed it are still in effect. The next iteration of the loop tries a new value for *v* against a domain state shaped by the previous failed attempt. Even if you remember to set *v* back, the *other* domain narrowings done during propagation persist.

The bug is not subtle. It just compounds: a sequence of tentative commitments, each one polluting the global state for everyone who follows. By depth ten the program is exploring nonsense.

The slightly less naive answer: copy

If state mutation is the problem, a defensible response is: don't mutate. Copy.

```
int solve(state_t s, int depth) {
    if (all_assigned(s)) return 1;
    var_t v = pick_variable(s);
    for (int32_t value : domain_of(s, v)) {
        state_t s2 = state_copy(s);
        assign(s2, v, value);
        propagate(s2);
        if (consistent(s2) && solve(s2, depth + 1)) return 1;
        state_free(s2);
    }
    return 0;
}
```

This works. It is also slow. A copy at every recursive call is $O(n)$ where n is the size of the state — typically dominated by the variables and their domains. A search tree of depth d with branching factor b does $O(b^{\sup d} \cdot n)$ work just on copies, before any actual reasoning. The asymptotic *time* is fine — backtracking is exponential in the worst case anyway — but the constant is murderous, and worse, the *memory* footprint becomes $O(d \cdot n)$, one full copy per stack frame.

There is a saving grace: most of the state at the next call is the same as the state now. The amount that *changed* between two successive calls is small — perhaps a few domain values removed, perhaps a single assignment recorded. We are paying to copy a great deal that we did not touch.

The right answer: log the changes, replay them in reverse

The trail is the recovery mechanism that follows from this observation. Instead of saving state, save *deltas*. Every time a piece of mutable state is about to change, record (address, old value) on a stack. To undo a series of changes, pop the stack and write each old value back.

```
int solve(int depth) {
    if (all_assigned()) return 1;
    var_t v = pick_variable();
    for (int32_t value : domain_of(v)) {
        size_t mark = trail_mark(t);
        assign(v, value);           /* records changes on the trail */
        propagate();                /* records its changes too      */
        if (consistent() && solve(depth + 1)) return 1;
        trail_restore(t, mark);     /* roll back to before assign */
    }
    return 0;
}
```

The cost of `trail_restore` is exactly the number of mutations made since the mark. If propagation narrowed three domains by one element each, restoration writes three words. The size of the state never enters the calculation.

This is the discipline that makes predictive search affordable. Every chapter that follows builds on it.

Predictive, not prophetic

A word about the title. *Predictive* here means making commitments forward and being prepared to walk them back — a stance, not a forecast. The algorithms in this book do not predict in any statistical sense. They explore.

What they share is a willingness to act on incomplete information, pruning aggressively from the space of possibilities, and a discipline for unwinding cleanly when an action turns out to have been wrong.

Two algorithms in particular dominate the rest of the book.

A *constraint solver* posts predicates over variables, narrows domains by deduction (this variable can no longer be 7, because that would force this other variable to be 12 which is outside its bound), and only branches when deduction has finished narrowing. The trail records every domain narrowing.

An *exact-cover solver*, in the Dancing Links style, treats a search step as removing rows and columns from a matrix and lets the link structure itself act as the trail — the doubly-linked lists remember enough about how a node was unlinked that you can restore it by visiting the same lines in reverse. There is no separate undo log because the data structure *is* the undo log.

Both are predictive in the sense above. Both walk a tree of partial commitments. Both prune. Both unwind cheaply.

What the rest of the book does

Part I — this part — sets up the data structures: the sparse set in Chapter 2, the trail in Chapter 3.

Part II walks through the CSP API, posts a few classic problems, and shows how to write a propagator of your own.

Part III does the same for DLX.

Part IV asks how to choose between them and discusses combining them in a single program.

The library is in C, the smaller for being so. You can read every line of every component in an afternoon, and at the end of this book you will have done so.

Chapter 2. The sparse set

A constraint variable's domain is a set of integers drawn from a known small range. The operations we want on it are:

7. **Membership.** Is value v still in the domain?
8. **Removal.** Take v out of the domain.
9. **Iteration.** Walk the values currently in the domain.
10. **Size.** How many values are left?

These four operations are needed inside the propagator's hot loop. Every constraint we will post asks one of these questions hundreds of times a second. The cost of these operations sets the cost of the solver.

Several familiar data structures answer some of the questions cheaply but botch the others.

A **bitset** answers (1) and (2) and (4) in $O(1)$, but iteration over present elements is $O(\text{range})$, not $O(\text{size})$. When the domain has been narrowed to $\{3, 17, 941\}$ from a range of $[0, 1000]$, iterating over it visits every bit position — a thousand checks for three values.

A **sorted array** iterates in $O(\text{size})$, but membership is $O(\log \text{size})$ and removal is $O(\text{size})$ because it shifts elements.

A **hash set** is $O(1)$ on average for (1) and (2) but the constant is real and iteration order is arbitrary.

What we want is $O(1)$ for *every* operation, *predictable* memory layout (so the trail can rewind a removal cheaply), and dense iteration. The structure that delivers this is the **sparse set**, sometimes attributed to Briggs and Torczon. It uses two arrays of n integers, where n is the size of the universe.

The two-array trick

The universe is the range of representable values, fixed at construction. Call its size r . We allocate two arrays of r integers and a single counter size .

<code>values[0..size)</code>	the dense list of present elements
<code>pos[off]</code>	the index in <code>values</code> where element <code>off</code> lives (valid iff <code>pos[off] < size</code> && <code>values[pos[off]] == off</code>)
<code>size</code>	how many elements are currently in the set

The element at array index `i < size of values` is in the set. Its identity is the value `values[i]`. The array `pos` is the inverse: given a candidate element `off`, `pos[off]` tells you where to look in `values` to see if it's there.

The clever bit is that the two arrays are not initialised in the strict sense: garbage in `pos[off]` is fine, as long as the membership test cross-checks against `values`. This makes the structure cheap to *create empty* (no zero-fill) and cheap to *create full* (a simple identity loop).

In the library the `dom_t` struct is initialised full — domain `[min, max]` starts containing every value:

```
typedef struct {
    uint32_t *values;
    uint32_t *pos;
    uint32_t size;
    uint32_t range;
    int32_t min;
} dom_t;

static void dom_init(dom_t *d, int32_t min, int32_t max) {
    assert(max >= min);
    d->min = min;
    d->range = (uint32_t)(max - min) + 1u;
    d->size = d->range;
    d->values = malloc(d->range * sizeof *d->values);
    d->pos = malloc(d->range * sizeof *d->pos);
    if (!d->values || !d->pos) abort();
    for (uint32_t i = 0; i < d->range; i++) {
        d->values[i] = i;
        d->pos[i] = i;
    }
}
```

A few notes about this listing. The struct stores *offsets* internally — values are kept as `uint32_t` ranging from 0 to `range - 1`, and the user-facing integer value is recovered by adding `min`. This costs nothing and keeps the indexing arithmetic clean. The set is initialised *full*: at construction time every value in the range is a member.

Membership

Given a candidate user-facing integer `v`, the membership predicate is:

```
static int dom_contains(const dom_t *d, int32_t v) {
    if (v < d->min) return 0;
    uint32_t off = (uint32_t)(v - d->min);
    if (off >= d->range) return 0;
    if (d->pos[off] >= d->size) return 0;
    return d->values[d->pos[off]] == off;
}
```

Two range checks, then the cross-check: look up where `pos` claims `off` lives, verify that `values` agrees. If `pos[off]` is past the live region of `values`, or if it points somewhere that holds a different offset, `off` is absent.

The cross-check is what lets us skip the zero-fill. If `pos[off]` happens to hold an out-of-range or stale value left over from initialisation, the predicate still returns 0 correctly. We never have to clear anything.

Removal: swap and pop

This is the operation that motivates the structure. To remove off from the set:

11. Find where it lives: `i = pos[off]`.
12. Find what's at the end: `last = values[size - 1]`.
13. Move `last` into the hole at `i`: `values[i] = last; pos[last] = i;`
14. Decrement `size`.

```
before:  values = [3, 17, 9, 941, 26, 88]    size = 6
```

```

after:  values = [3, 17, 88, 941, 26, _]    size = 5
           ^
           last (88) moved here

```

The cell that previously held 9 is now logically gone — it is past the end of the live prefix. The cell that previously held 88 has changed position from index 5 to index 2, and `pos[88]` was updated to reflect that. Iteration over `values[0..size)` no longer sees 9.

The cost is four word writes regardless of where off was. There is no shifting.

Iteration

Iteration is just a loop:

```
for (uint32_t i = 0; i < d->size; i++) {
    int32_t v = (int32_t)d->values[i] + d->min;
    /* ... do something with v ... */
}
```

This visits every present value exactly once, with no skipping over absent ones, in whatever order they happen to lie in `values`. The order is *not* sorted. It is whatever the swap-and-pop history happens to have produced. For most uses inside a CSP solver this is fine; the value-selection heuristic can sort if needed.

Inserting a new value into a sparse set is the inverse: re-extend `size` and either swap the absent value into the new live slot or rely on the fact that `swap-and-pop` puts removed values at exactly position `size`, so simply incrementing `size` reinstates the most recently removed value.

Why this matches the trail

When the trail rolls back a removal, it does so by writing a saved old value to a saved address. Because the sparse set's removal touched only a fixed small set of addresses — `size`, `values[i]`, and `pos[last]` — those are exactly the addresses that get saved on the trail. There are at most three writes per removal, and therefore at most three writes per restoration.

The trail does not need to know anything about sparse sets. It just records word-sized mutations and replays them in reverse. The sparse set's discipline of mutating exactly three known words per operation is what keeps the trail tight.

This is the first instance of a pattern that recurs through the book: design data structures so that every mutation is local, addressable, and recordable in $O(1)$. The cost of $O(1)$ reversibility is that you accept $O(1)$ mutations per logical operation. For most data structures this is no constraint; it is just good design.

A sketch of usage

We will not need to use the sparse set directly — the CSP API hides it behind the variable type. But it is helpful to see what its full surface looks like. The library exposes (internally to `csp.c`) operations along these lines:

```
static void dom_remove(dom_t *d, int32_t v, trail_t *t);
static void dom_assign(dom_t *d, int32_t v, trail_t *t);
static int  dom_contains(const dom_t *d, int32_t v);
static uint32_t dom_size(const dom_t *d);
```

`dom_remove` is the swap-and-pop, with a trail call before each mutation. `dom_assign` is "remove every value except `v`," used when a propagator concludes that `v` is forced. Both of these compose smaller mutations and call `trail_save` before each one; the trail's `len` grows by exactly the number of words mutated.

In the next chapter we will look at the trail itself in detail. For now, the takeaway is that the sparse set is a clean, fast container with one key property — every operation it performs is a sequence of three word writes — and that this property is what makes it the right shape for a CSP variable's domain.

A pause

Read the full `dom_*` functions in `csp.c` if you have not yet — they are about a hundred lines, and they implement everything described in this chapter. The structure is small enough that you can hold all of it in your head simultaneously, and you should, because every constraint propagator in the rest of the book will be reasoning about what those functions do.

The next chapter introduces the trail. After that, we have all the foundations and can start posting actual constraints.

Chapter 3. The trail

The trail is the smallest piece of code in this book and the one with the most leverage. The whole interface is six functions and one struct:

```
typedef struct {
    uint32_t *addr;
    uint32_t old;
} trail_entry_t;

typedef struct {
    trail_entry_t *log;
    size_t        len;
    size_t        cap;
} trail_t;

void trail_init(trail_t *t);
void trail_free(trail_t *t);
void trail_save(trail_t *t, uint32_t *addr);
```

```
size_t trail_mark(const trail_t *t);
void trail_restore(trail_t *t, size_t mark);
```

The implementation is forty lines. We will look at it in full.

What the trail records

A `trail_entry_t` is two words: an address, and the value that lived at that address before some recent change. The trail is an array of these, growing by one entry every time the program is about to mutate a tracked word.

```
void trail_save(trail_t *t, uint32_t *addr) {
    if (t->len == t->cap) {
        size_t new_cap = t->cap ? t->cap * 2 : 256;
        trail_entry_t *new_log = realloc(t->log, new_cap * sizeof *new_log);
        if (!new_log) abort();
        t->log = new_log;
        t->cap = new_cap;
    }
    t->log[t->len].addr = addr;
    t->log[t->len].old = *addr;
    t->len++;
}
```

The pattern of use is: call `trail_save(t, &x)` immediately before every assignment to `x`. The save records `x`'s current value, paired with its address. The next instruction performs the actual write. The trail is now one entry longer.

Note that `trail_save` does not perform the write itself. The caller writes. The trail just photographs the before-state for later use.

Marks and restoration

`trail_mark` is trivially the current length:

```
size_t trail_mark(const trail_t *t) {
    return t->len;
}
```

A *mark* is just an integer, the length of the trail at the moment of marking. There is no allocation, no list of marks, no per-mark structure. The mark is opaque to the caller — just an integer to be passed back to `trail_restore`.

Restoration walks the trail backwards from its current length down to the mark, replaying each saved value:

```
void trail_restore(trail_t *t, size_t mark) {
    while (t->len > mark) {
        t->len--;
        *t->log[t->len].addr = t->log[t->len].old;
    }
}
```

The order matters. If a single word was mutated three times since the mark, the trail has three entries for it, and we must replay them in reverse so the *earliest* saved value is the one that wins. Forward replay would leave the word at the value it had immediately before the most recent change, not the value it had at mark time.

After restoration the trail's length equals the mark, and the world is exactly as it was at the moment of the mark — for every memory location that the program properly trail-saved before mutating.

The discipline

Every reversible mutation must be preceded by a `trail_save`. This is a discipline, enforced by code review and convention rather than by the compiler. The cost of getting it wrong is silent — the program will not crash, it will simply forget to undo something during restoration and produce incorrect results in a later branch of the search.

In practice the discipline is contained because trail-saved mutations live in a small set of helper functions: the sparse-set primitives, the propagator infrastructure, and a handful of search-state variables. As long as those helpers are correct, the rest of the program writes ordinary code and the trail handles the rest.

A short example. Suppose we want to track changes to a single counter:

```
trail_t t;
trail_init(&t);
uint32_t counter = 0;

size_t mark0 = trail_mark(&t);

trail_save(&t, &counter); counter = 1;
trail_save(&t, &counter); counter = 2;
trail_save(&t, &counter); counter = 3;

assert(counter == 3);
trail_restore(&t, mark0);
assert(counter == 0);

trail_free(&t);
```

After three saves and writes, the trail has three entries: (counter, 0), (counter, 1), (counter, 2). `trail_restore(&t, 0)` walks them in reverse: write 2 to counter, write 1, write 0. The final value is 0 — exactly what was there at mark0.

Nested marks

Marks compose. You can take a mark, mutate, take a second mark deeper, mutate more, then restore to either one.

```
size_t m0 = trail_mark(&t);
trail_save(&t, &x); x = 10;

size_t m1 = trail_mark(&t);
trail_save(&t, &y); y = 20;
trail_save(&t, &x); x = 11;

trail_restore(&t, m1);
```

```

/* y is back to its value at m1, x is back to 10. */
assert(x == 10);

trail_restore(&t, m0);
/* x is back to its original value, all changes since m0 are gone. */

```

Each restoration trims the trail back. After `trail_restore(&t, m1)`, the trail's length is exactly `m1`, and `m0` is still a valid mark you can later restore to. Restoring to `m0` would also discard whatever happened between `m0` and `m1`.

What the trail does not do

The trail does not allocate. It does not keep a tree. It does not snapshot. It is a flat append-only log of word-sized writes, with random-access truncation as the only retraction operation.

This means:

- **You cannot trail-save a structure**, only its individual words. To make a multi-word structure reversibly mutable, save each word separately before changing it.
- **You cannot trail-save heap allocations**. If a propagator allocates new memory during its run, that memory is not freed by `trail_restore`. The library handles this by allocating up-front during constraint posting and not allocating during search.
- **You cannot replay forward**. Once a section of the trail has been restored, it is gone; you cannot redo. (You can, of course, repeat the same operations, but the trail entries themselves are discarded.)
- **You cannot save non-uint32_t values without packing**. The trail records `uint32_t` words. For domains, sizes, and indices this is the right granularity. For larger types, pack them into multiple words and save each.

These are not limitations to overcome. They are *design choices* that keep the trail small and predictable. Every limitation is dual to a guarantee: by giving up forward replay we get $O(k)$ restoration; by giving up structure-aware saves we get a flat array with no pointer-chasing in the restore loop.

What it costs

Memory: 8 bytes per saved entry on a 64-bit system (an 8-byte pointer plus a 4-byte value plus padding). The trail grows by doubling. After N saves you have N entries plus up to N unused capacity, so worst-case 16 bytes per save in steady state.

Time: $O(1)$ per save, $O(1)$ per mark, $O(k)$ per restoration where k is the number of saves since the mark. The save cost includes the occasional `realloc` when capacity doubles, amortised to $O(1)$.

Cache behaviour: excellent, because the trail is a contiguous array touched in append order during forward execution and in reverse order during restoration. Both are linear and benefit from prefetch.

Why this is enough

Every reversible mutation in the rest of the book uses this trail. There is no second mechanism. The CSP variable domains use it, the propagator queue uses it, the DLX search uses a slightly different reversibility trick (Knuth's link-removal-by-inspection), but anything outside DLX that needs to be undoable goes through this thirty-line file.

This is the leverage. The trail is small, well-tested, and the only piece of code in the library that the rest of the system has to *trust* to get right. Everything else can use it through `trail_save` and forget that there is anything funny going on. When a bug in propagation is suspected, the trail is rarely the cause; the discipline of "always save before mutating" is straightforward to audit, and the trail itself is too simple to harbour subtle bugs.

The forty-line trail is the foundation for everything that follows. The next chapter introduces the CSP API, and you will see `trail_save` calls quietly underpinning every domain narrowing the propagator does.

Part II — Constraint Satisfaction

Chapter 4. Domains, variables, and propagators

A *constraint satisfaction problem* — CSP — is a set of variables, each with a domain (a set of values the variable may take), together with a set of constraints (predicates that pick out which combinations of values are legal). A solution is an assignment of one value from its domain to each variable such that all constraints hold simultaneously.

The library's CSP type is opaque from the outside:

```
typedef struct csp csp_t;
typedef uint32_t var_t;

csp_t *csp_new(void);
void csp_free(csp_t *csp);

var_t csp_var(csp_t *csp, int32_t min, int32_t max);
uint32_t csp_dom_size(const csp_t *csp, var_t v);
int csp_in_dom(const csp_t *csp, var_t v, int32_t value);
int32_t csp_value(const csp_t *csp, var_t v);
```

`csp_new` constructs an empty problem. `csp_var` introduces a variable with a closed integer interval `[min, max]` as its initial domain, and returns a small handle — the variable's index — that is used to refer to it everywhere else. `csp_dom_size`, `csp_in_dom`, and `csp_value` are read-only inspectors of a variable's current domain state.

A typical opening sequence:

```
csp_t *csp = csp_new();
var_t x = csp_var(csp, 1, 9);
var_t y = csp_var(csp, 1, 9);
var_t z = csp_var(csp, 1, 9);
```

This builds a problem with three variables, each starting with the domain `{1, 2, ..., 9}`. No constraints have been posted, so the problem trivially admits $9^3 = 729$ solutions. The solver does not yet know any of them; it knows only that there are three variables and three domains.

What "narrow" means

The key invariant of the solver is *narrowing only*. Domains shrink, never grow. Once a value has been removed from a variable's domain, no constraint can put it back; the only thing that restores it is `trail_restore` walking back up the search tree to a point before its removal.

This monotonicity is what makes propagation tractable. A constraint can never make the problem more solvable — only less. So the solver can run any constraint at any time, in any order, and the result is consistent: every constraint will eventually have applied all the deductions it can given the current state, and it does not matter in what sequence.

Narrowing is performed by *propagators*. A propagator is a piece of code that reads the current domains, draws inferences, and removes values. The library has propagators built in for several common constraint types; in Chapter 12 we will see how to write our own.

A propagator's contract

Three rules:

15. **Read-only outside its own removals.** A propagator must not allocate, must not mutate any state outside the trail-managed domains, and must not assume that any other propagator has run.
16. **Trail-save before each removal.** Every domain narrowing must be recorded on the trail. The library's domain helpers (`dom_remove`, `dom_assign`) do this for us.
17. **Return failure on infeasibility.** If a propagator concludes that a variable's domain must be empty for the constraint to hold, it returns 0 and the caller treats this as failure for the current branch of the search.

Propagators are *idempotent*: running the same propagator twice on the same state is the same as running it once. This is automatic if rule 1 is followed — the propagator only computes what is consistent with the current domains, and once that has been computed the second run finds the same conclusions and removes nothing further.

Propagators are *monotonic*: running propagator A and then propagator B reaches at least as much narrowing as running B then A then A again. This is because narrowing only ever loses values, and a propagator's deductions are a function of the current domains.

These properties together let the solver schedule propagators in any order without worrying about correctness. The library uses a queue: when one propagator narrows a variable, all other propagators watching that variable are scheduled to run again.

How propagators wake up

Each constraint, at posting time, registers itself on a *watch list* per variable it observes. When a propagator removes a value from a variable, the library examines that variable's watch list and adds every other propagator on it to a propagation queue. The queue is then drained: each propagator runs in turn, possibly adding more entries to its tail, until it is empty.

This is essentially AC-3, the textbook arc-consistency algorithm, with one important detail: the propagator does not have to be a binary arc. It can be over arbitrarily many variables, and it can do bounds reasoning, all-different reasoning, or any other deduction the implementer wants. The watch-list machinery does not care; it just notices that a variable changed and wakes everyone up.

The library's implementation of the queue is hidden inside `csp.c`. The user-facing operations — `csp_solve`, `csp_propagate`, the various `csp_*` constraint posters — take care of the queue. We will not look at it in detail until Chapter 12, where writing a custom propagator forces us to understand the watch-list interface.

Variables are handles, domains are state

The variable type `var_t` is `uint32_t`. It is just a small index into the CSP's array of variables. Two CSPs constructed independently will both number their first variable as 0, and you must not mix them up — the library does not check.

The *domain* is the mutable state that the propagators chase. The library's internal `dom_t` (Chapter 2) holds the sparse set; the public API exposes only inspectors:

- `csp_dom_size(csp, v)` — how many values remain in `v`'s domain.
- `csp_in_dom(csp, v, value)` — does `v`'s domain currently contain `value`?

- `csp_value(csp, v)` — if `v`'s domain has been narrowed to a single value, return it. Asserts on multi-value domains.

`csp_dom_size(csp, v) == 1` is the test for *fixed*. A variable whose domain has one value left is fixed; in a complete solution every variable is fixed.

A three-variable warm-up

To make this concrete, here is a complete program that posts three variables, one disequality, and counts solutions.

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

int main(void) {
    csp_t *csp = csp_new();
    var_t a = csp_var(csp, 1, 3);
    var_t b = csp_var(csp, 1, 3);
    csp_neq(csp, a, b);                /* a != b */
    size_t n = csp_solve(csp, NULL, NULL);
    printf("%zu solutions\n", n);
    csp_free(csp);
    return 0;
}
```

`csp_neq(csp, a, b)` posts the constraint that `a` and `b` cannot take the same value. With both domains starting at `{1, 2, 3}`, the legal pairs are (1,2), (1,3), (2,1), (2,3), (3,1), (3,2) — six of the nine total. `csp_solve` runs the search; with `cb = NULL` it just counts solutions.

Compiled and run against the library, this program prints `6 solutions`. We have used the trail-based search machinery to confirm a fact we could have computed by hand. In the next chapter we will start posting more interesting constraints.

The point so far

We have introduced the CSP API at the level of building blocks: how to make a problem, how to add variables, how to inspect the domains. We have not yet posted any non-trivial constraint. That is the next chapter.

The mental model to carry forward is: *a CSP is a collection of variable handles, and the interesting state is the domains they refer to. Constraints are written declaratively — you say what must hold, the solver figures out how. Behind the scenes, every domain change is on the trail, and every constraint is woken up when its variables narrow. We do not have to think about the trail or the queue when posting a constraint; we have to think about them only when writing a new propagator from scratch.*

Chapter 5. Posting constraints

The library's constraint set is intentionally small. Eight built-in constraint types cover the great majority of CSP problems we want to encode without writing custom propagators. This chapter walks through each, with the kinds of problems that motivate them.

Disequality and equality

The two binary constraints over a pair of variables:

```
void csp_neq(csp_t *csp, var_t x, var_t y);    /* x != y */
void csp_eq (csp_t *csp, var_t x, var_t y);    /* x == y */
```

The disequality is the workhorse. Most "no two of these can be the same" problems decompose into pairwise `csp_neq` calls — graph colouring, the *n*-queens problem, magic squares. The equality is sometimes called *channelling*: it ties two variables to the same value, which is useful when one variable encodes a position and another encodes a property of that position.

The propagators are straightforward. `csp_neq` watches both *x* and *y*; if either becomes fixed at value *v*, it removes *v* from the other. `csp_eq` is symmetric: if either gets a value removed from its domain, the other has the same value removed.

Unary constraints

```
void csp_eq_c (csp_t *csp, var_t x, int32_t c);    /* x == c */
void csp_neq_c(csp_t *csp, var_t x, int32_t c);    /* x != c */
```

These narrow a single variable. `csp_eq_c` fixes the variable to a specific value; the solver removes every other value from its domain at posting time. `csp_neq_c` removes a single value.

These are useful for hard rejections during rule-style reasoning. A real-time decision system might use `csp_eq_c(csp, E[i], 0)` to mark certain candidate options as ruled out by policy. The constraint runs once at posting time and stays in effect for the life of the CSP.

All-different

```
void csp_alldiff(csp_t *csp, const var_t *vars, uint32_t n);
```

The all-different constraint says that the *n* variables given to it must take pairwise distinct values. Logically it is equivalent to $n(n-1)/2$ individual `csp_neq` calls, and the library's implementation does exactly that decomposition.

This is *not* the strongest possible all-different propagator. A more powerful version, sometimes called *Régin's algorithm* after its inventor, uses bipartite matching to detect inconsistency earlier — for example, it can notice that if the four variables {*x*, *y*, *z*, *w*} all have domain {1, 2, 3}, then no all-different assignment is possible, even though no individual pair has been narrowed to a single value yet.

The library does not implement Régin. The decomposed pairwise version is good enough for the problems we will solve and is dramatically simpler to write. If you find yourself bottlenecked on all-different reasoning, the matching-based propagator is a known engineering target.

Tuple constraints

```
void csp_allowed_pairs (csp_t *csp, var_t x, var_t y,
                       const int32_t (*pairs)[2], uint32_t n);
void csp_forbidden_pairs(csp_t *csp, var_t x, var_t y,
                        const int32_t (*pairs)[2], uint32_t n);
```

When the relation between two variables is too irregular for an arithmetic formula, list it. `csp_allowed_pairs` says "(x, y) may take only one of these listed combinations"; `csp_forbidden_pairs` says "(x, y) may take any combination *except* these." The solver internalises the list and propagates by walking it whenever either variable narrows.

A typical use of `csp_allowed_pairs` is a lookup table — for example, "for each colour, what shapes are compatible?" expressed as a list of (colour, shape) pairs. `csp_forbidden_pairs` is useful for symmetry breaking: "the slot before this one cannot be empty if this one is full" expressed as a list of forbidden (prev, curr) combinations.

The pairs array is copied at posting time, so the caller can free it immediately after the call.

Linear arithmetic

This is the heaviest hammer in the standard set:

```
void csp_linear_le(csp_t *csp,
                  const var_t *vars,
                  const int32_t *coeffs,
                  uint32_t n,
                  int32_t bound);

void csp_linear_eq(csp_t *csp,
                  const var_t *vars,
                  const int32_t *coeffs,
                  uint32_t n,
                  int32_t target);
```

`csp_linear_le` enforces $\sum \text{coeffs}[i] \cdot \text{vars}[i] \leq \text{bound}$; `csp_linear_eq` enforces equality with target. Coefficients can be positive, negative, or zero. Both propagators are *bounds-consistent*: they reason about the minimum and maximum values each variable can take (its domain's min and max), tighten each variable's bounds based on the others, and iterate to fixpoint.

A sample use: encode the constraint that four bits sum to exactly three.

```
var_t b[4];
for (int i = 0; i < 4; i++) b[i] = csp_var(csp, 0, 1);
int32_t coeffs[4] = { 1, 1, 1, 1 };
csp_linear_eq(csp, b, coeffs, 4, 3);
```

A more involved use is bin packing — distributing items of various sizes among bins of fixed capacity. Each bin gets a `csp_linear_le` over the indicator variables for items assigned to it, with coefficients equal to the item sizes and a bound equal to the bin capacity. The bin-packing tutorial in Chapter 9 develops this in detail.

The bounds-consistency limitation is worth noting. The propagator does not reason about *holes* in domains — it cannot, for example, deduce that if a domain is {1, 5, 10} the value cannot be 3, even though 3 is not in the domain, because the propagator only looks at min and max. For most problems this is plenty, and the overhead of full domain consistency on linear constraints is significant.

Posting versus running

A constraint is *posted* once at the start of the problem; from then on the solver re-runs its propagator whenever one of its watched variables narrows. Posting is the act of registering the constraint with the CSP; the propagator runs many times during search.

There is also a one-shot posting mode, where you post a constraint that runs once at posting time and then is gone. The library handles this by simply running the propagator once before returning from the posting call. For most constraints this distinction is invisible because the constraint is always re-watched, but for unary constraints (`csp_eq_c`, `csp_neq_c`) the post-time application is the only thing that matters: once a single value has been removed, there is no future event that will require re-running the propagator.

Initial propagation

After all constraints have been posted, the solver kicks off an *initial propagation*: every constraint runs once on the as-posted domains, with the queue running until it stabilises. This may detect at-the-root infeasibility (some domain became empty) or fix some variables outright (all-but-one of their values were removed). The initial propagation cost is $O(c \cdot d)$ in the worst case where c is the number of constraints and d is the average domain size, but typically much less because many constraints have nothing to do until search begins assigning.

The library handles initial propagation automatically. The user-facing entry points `csp_solve` and `csp_propagate` both run it before doing anything else.

What is missing

The constraint set above is small enough to enumerate but large enough to encode most of the problems in this book. A few common constraint types are deliberately not included:

- **Element:** `y == array[x]`, where `array` is a vector of constants and `x` indexes into it. This is sometimes essential — it makes scheduling problems much cleaner — but it is not in the library and would need to be written as a custom propagator (Chapter 12).
- **Cumulative:** a scheduling constraint that says "at no time may the total resource usage of the running tasks exceed *capacity*." This is the moral equivalent of `csp_linear_le` integrated over time, and writing a strong propagator for it is a research topic.
- **Global cardinality:** "exactly k of these variables take value v ." Decomposes into a `csp_linear_eq` over indicator variables.

The pattern with all of these is that they can be encoded indirectly using the existing constraints, often less efficiently than a dedicated propagator would manage. For the problems in this book, the existing set is enough.

A worked posting

We close with a small posting that uses several of the constraints together. Suppose we want to choose three pairwise-different non-zero digits whose sum is 12.

```
csp_t *csp = csp_new();
var_t x = csp_var(csp, 1, 9);
var_t y = csp_var(csp, 1, 9);
var_t z = csp_var(csp, 1, 9);

var_t vars[3] = { x, y, z };
csp_alldiff(csp, vars, 3);
```

```
int32_t coeffs[3] = { 1, 1, 1 };
csp_linear_eq(csp, vars, coeffs, 3, 12);

size_t n = csp_solve(csp, NULL, NULL);
printf("%zu triples\n", n);
csp_free(csp);
```

This says: three variables, each between 1 and 9, all different, summing to 12. The solver enumerates them. The answer is 42: the unordered sets of three distinct non-zero digits summing to 12 are {1,2,9}, {1,3,8}, {1,4,7}, {1,5,6}, {2,3,7}, {2,4,6}, and {3,4,5} — seven of them — and each contributes $3! = 6$ ordered triples, giving $7 \times 6 = 42$. If you wanted unordered triples you would post a lexicographic constraint, for example $x < y < z$, encoded as two `csp_linear_le` constraints with target -1 (i.e. $x - y \leq -1$ and $y - z \leq -1$). With those added, the count drops to 7.

Try modifying the program to add a callback that prints each solution and to add the ordering constraints. This is the kind of small experiment that solidifies understanding. The next chapter explains the search machinery — how `csp_solve` actually walks the tree of partial assignments — and after that we begin the tutorials.

Chapter 6. Search and propagation

Posting constraints does not solve the problem. It says what *would* be true of a solution. To find one — or to count them, or to enumerate them — we have to search.

The library exposes two ways to drive the solver:

```
typedef int (*csp_solution_cb)(const csp_t *csp, void *ud);
size_t csp_solve(csp_t *csp, csp_solution_cb cb, void *ud);
int csp_propagate(csp_t *csp);
```

`csp_solve` performs a complete depth-first search. For every solution it finds, it calls `cb(csp, ud)`; if `cb` returns 1 the search continues, if 0 it stops. With `cb = NULL` it just counts solutions and returns the count. The user-data pointer `ud` is opaque to the solver and passed through unchanged.

`csp_propagate` is a simpler beast: it runs the propagation queue to fixpoint over the currently posted constraints and returns. No search, no branching. It is the right tool when you want to use the CSP as a *propagator only* — a pure deduction engine that narrows domains without making any choices.

This chapter explains how each works.

Propagation to fixpoint

The propagation engine is a queue. Initially it holds every constraint that has been posted. Each iteration:

18. Take a constraint off the queue.
19. Run its propagator on the current domains.
20. If the propagator removed values from any variable, find every other constraint watching those variables and add them to the queue (skipping ones already in the queue).
21. If the propagator returned failure (a domain became empty), the whole CSP is infeasible — abort and return 0.

The loop terminates when the queue is empty. By that point, every constraint has run on a stable state — a state where no constraint can find anything more to remove.

This is the "AC-3 with watch lists" loop, modulo terminology. It is monotone — domains only ever shrink — so it has to terminate (it cannot remove the same value twice). The number of iterations is bounded by the total domain size summed across all variables.

Inside the loop, every domain narrowing is trail-saved automatically. If propagation succeeds, the trail records every change so a later restoration can undo all of it. If propagation fails, the caller (`csp_solve` or `csp_propagate`) is responsible for restoring the trail to the state before propagation began.

`csp_propagate`: the propagator-only entry point

This is the simplest API:

```
int csp_propagate(csp_t *csp);
```

It runs propagation on the current state and stops. No branching. The return value is 1 if the propagation reached a consistent fixpoint, 0 if any constraint failed.

The use case is rule-style reasoning. You post a set of unary or binary constraints expressing "if this configuration is selected, these consequences must hold," call `csp_propagate`, and read each variable's domain to find out which configurations the rule layer has eliminated. A typical use of this pattern is the rule-driven first stage of a two-stage decision system: hard rejections like `csp_eq_c(E[i], 0)` are posted, propagation runs, and the resulting domain sizes tell the rest of the program which options have been ruled out by the rule layer.

The CSP is *not consumed* by `csp_propagate`. After it returns, you can post more constraints and propagate again, or you can call `csp_solve` to search over the tightened state, or you can simply read the domains and discard the CSP. Whatever you do, the trail records the changes propagation made; if you wanted to restore to the pre-propagation state you would have had to mark the trail before calling propagate, but the API does not currently expose this — `csp_propagate` is intended for terminal use, after which the CSP either solves further or is freed.

`csp_solve`: depth-first search

Search is a depth-first walk over partial assignments. At each step:

22. If every variable is fixed (domain size 1), call the user's callback. If the callback returns 0 (stop), unwind to the top.
23. Otherwise pick a variable to branch on, using the *first-fail* heuristic: the unfixed variable with the smallest domain.
24. For each value remaining in that variable's domain:
 - a. Mark the trail.
 - b. Tentatively assign the value (remove all other values from the domain). This wakes up propagation.
 - c. Propagate to fixpoint.
 - d. If propagation succeeded, recurse.
 - e. After the recursive call returns (whether it found solutions or not), restore the trail to the mark.

The first-fail heuristic — branching on the smallest-domain unfixed variable — is a standard rule of thumb. The intuition: a variable with two values left has two branches; a variable with nine values has nine. By branching on the most constrained variable first, you minimise the breadth of the next level of the tree, and you give the propagators the most material to work with (a freshly fixed variable wakes up its watchers, and a variable with a small domain has more chance of forcing further deductions).

It is not always optimal, but it is robust. For most problems it is at least competitive with hand-tuned alternatives, and it is the default in many constraint solvers.

Value ordering — within a chosen variable, in what order to try its values — is taken from the sparse set's current iteration order. This is not stable across runs, since the iteration order depends on the history of removals. For most problems this is fine; if you need stable output, sort the values yourself in the callback.

The shape of the search tree

A search node is a partial assignment plus the consequences of propagation from it. Because of propagation, the tree is much smaller than a naive brute-force enumeration. A typical example: cryptarithmic with ten letters might have $10! = 3,628,800$ leaves in a brute-force tree, but propagation prunes most of them — the real solver visits perhaps a few thousand internal nodes and finds the unique solution in milliseconds.

The branching factor at each node is the size of the chosen variable's domain *after* propagation — not the original domain size. Propagation does most of the work; the search just resolves the residue.

One useful diagnostic when a CSP runs slowly: instrument the solver to print the depth and branching factor at each level. If most nodes have small branching factor (2 or 3) and the tree is reasonably shallow (10–20 deep), the propagators are doing well. If the branching factor stays high all the way down, propagation is weak and you may need to write a stronger constraint.

Stopping early

The callback's return value lets the caller stop after enough solutions have been found. To find just one:

```
static int first_only(const csp_t *csp, void *ud) {
    /* read the variables and store, then... */
    return 0; /* stop searching */
}
```

The solver returns immediately after the callback returns 0, restoring the trail along the way. The CSP is left in a state somewhere mid-tree — its domains do *not* reflect a solution after `csp_solve` returns, even if a solution was found. You must capture what you need from inside the callback.

Counting

For just-counting use, pass `cb = NULL`:

```
size_t n = csp_solve(csp, NULL, NULL);
printf("found %zu solutions\n", n);
```

The solver still walks the entire tree but skips the per-solution callback. This is faster than counting in a callback (no function call overhead) and sometimes useful in its own right — for combinatorial enumeration problems, the count *is* the answer.

A complete example: the n-queens problem in CSP form

Eight queens on an 8×8 board, no two attacking. Each queen lives on its own row, identified by row index 0..7, and the variable holding the column it sits in.

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

```

#include <stdlib.h>

#define N 8

static int print_cb(const csp_t *csp, void *ud) {
    var_t *Q = ud;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            putchar(csp_value(csp, Q[i]) == j ? 'Q' : '.');
        }
        putchar('\n');
    }
    putchar('\n');
    return 1;
}

int main(void) {
    csp_t *csp = csp_new();
    var_t Q[N];
    for (int i = 0; i < N; i++) Q[i] = csp_var(csp, 0, N - 1);

    /* No two queens in the same column. */
    csp_alldiff(csp, Q, N);

    /* No two queens on the same diagonal. */
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N; j++) {
            int d = j - i;
            int32_t bad[2][2] = { {0, 0}, {0, 0} };
            int n = 0;
            for (int32_t a = 0; a < N; a++) {
                if (a + d < N) { bad[n][0] = a; bad[n][1] = a + d; n++; if (n
== 2) break; }
            }
            /* Generate all (a, a+d) and (a, a-d) pairs explicitly. */
            int32_t (*pairs)[2] = malloc(2 * N * sizeof *pairs);
            int k = 0;
            for (int32_t a = 0; a < N; a++) {
                if (a + d < N) { pairs[k][0] = a; pairs[k][1] = a + d; k++; }
                if (a - d >= 0) { pairs[k][0] = a; pairs[k][1] = a - d; k+
+; }
            }
            csp_forbidden_pairs(csp, Q[i], Q[j], (const int32_t (*)[2])pairs,
k);
            free(pairs);
        }
    }

    size_t n = csp_solve(csp, print_cb, Q);
    printf("%zu solutions\n", n);
    csp_free(csp);
    return 0;
}

```

This is not the most elegant possible encoding — n-queens has a cleaner DLX form, which we will see in Chapter 14 — but it works. `csp_alldiff` handles "no two in the same column"; the nested loop posts a `csp_forbidden_pairs` for each pair of rows, listing the column-pairs that would put the two queens on the same diagonal.

Compiled and run, it finds all 92 solutions to the 8-queens problem in well under a second. The first-fail variable selection finds them in a particular order; the value-ordering within each variable is whatever the sparse set's removal history has produced.

What we have so far

Three chapters of CSP foundation. We can:

- Build a CSP, add variables with bounded integer domains.
- Post unary, binary, all-different, table, and linear-arithmetic constraints.
- Run propagation to fixpoint.
- Run depth-first search with first-fail variable selection and a user callback.

That is enough to solve a great many problems. The next three chapters work three concrete examples in depth: cryptarithmic, graph colouring, and bin packing. Each illustrates a different kind of constraint mix and a different style of encoding. After those, Chapter 12 returns to writing your own propagator from scratch.

Chapter 7. Tutorial — cryptarithmic

Cryptarithmic, sometimes called *alphametic*, is the puzzle form

SEND + MORE = MONEY

where each letter stands for a different decimal digit, leading digits cannot be zero, and the addition must hold when each word is read as a base-10 number. The classic puzzle has exactly one solution: $9567 + 1085 = 10652$.

This tutorial works the puzzle from scratch. By the end you will have a complete program that posts the constraints, runs the search, and prints the solution. Along the way we will see how a constraint problem is modelled — how to choose the variables, how to translate the puzzle's prose into formal constraints, and how to verify the result.

Identifying the variables

The puzzle has eight distinct letters: S, E, N, D, M, O, R, Y. Each must take a value in $\{0, 1, \dots, 9\}$, and each must take a different value from every other letter. So we have eight variables, each with domain $\{0..9\}$.

```
csp_t *csp = csp_new();
var_t S = csp_var(csp, 0, 9);
var_t E = csp_var(csp, 0, 9);
var_t N = csp_var(csp, 0, 9);
var_t D = csp_var(csp, 0, 9);
var_t M = csp_var(csp, 0, 9);
var_t O = csp_var(csp, 0, 9);
var_t R = csp_var(csp, 0, 9);
var_t Y = csp_var(csp, 0, 9);
```


Eight variables \times ten values each is $10^8 = 100,000,000$ candidate assignments before any constraints. With all-different applied, the count drops to $10!/2! = 1,814,400$. We want to drive it lower still.

The arithmetic constraint

The addition SEND + MORE = MONEY is, written out as integers:

$$1000 \cdot S + 100 \cdot E + 10 \cdot N + 1 \cdot D + 1000 \cdot M + 100 \cdot O + 10 \cdot R + 1 \cdot E = 10000 \cdot M + 1000 \cdot O + 100 \cdot N + 10 \cdot E + 1 \cdot Y$$

Move everything to one side:

$$1000 \cdot S + 100 \cdot E + 10 \cdot N + 1 \cdot D + 1000 \cdot M + 100 \cdot O + 10 \cdot R + 1 \cdot E - 10000 \cdot M - 1000 \cdot O - 100 \cdot N - 10 \cdot E - 1 \cdot Y = 0$$

Combine like terms:

$$1000 \cdot S + (100 + 1 - 10) \cdot E + (10 - 100) \cdot N + 1 \cdot D + (1000 - 10000) \cdot M + (100 - 1000) \cdot O + 10 \cdot R - 1 \cdot Y = 0$$

Which simplifies to:

$$1000 \cdot S + 91 \cdot E - 90 \cdot N + 1 \cdot D - 9000 \cdot M - 900 \cdot O + 10 \cdot R - 1 \cdot Y = 0$$

This is a linear equation over the eight letter-variables, with integer coefficients summing to a fixed target (zero). It is exactly the shape `csp_linear_eq` accepts.

```
var_t vars[8] = { S, E, N, D, M, O, R, Y };
int32_t coeffs[8] = { 1000, 91, -90, 1, -9000, -900, 10, -1 };
csp_linear_eq(csp, vars, coeffs, 8, 0);
```

That single call posts the entire arithmetic relationship.

All-different and leading-digit constraints

Each letter is a different digit:

```
csp_alldiff(csp, vars, 8);
```

And no leading digit can be zero. SEND, MORE, and MONEY all start with letters that must be at least 1:

```
csp_neq_c(csp, S, 0);
csp_neq_c(csp, M, 0);
```

(Y is the units digit of MONEY, not a leading digit, so it can be zero.)

Putting it together

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

static int print_cb(const csp_t *csp, void *ud) {
    var_t *v = ud;
    static const char *names = "SENDMORY";
    for (int i = 0; i < 8; i++)
        printf("%c=%d ", names[i], csp_value(csp, v[i]));
    printf("\n");
    return 1;
}
```

```

int main(void) {
    csp_t *csp = csp_new();
    var_t S = csp_var(csp, 0, 9);
    var_t E = csp_var(csp, 0, 9);
    var_t N = csp_var(csp, 0, 9);
    var_t D = csp_var(csp, 0, 9);
    var_t M = csp_var(csp, 0, 9);
    var_t O = csp_var(csp, 0, 9);
    var_t R = csp_var(csp, 0, 9);
    var_t Y = csp_var(csp, 0, 9);

    var_t vars[8] = { S, E, N, D, M, O, R, Y };
    int32_t coeffs[8] = { 1000, 91, -90, 1, -9000, -900, 10, -1 };

    csp_alldiff (csp, vars, 8);
    csp_linear_eq(csp, vars, coeffs, 8, 0);
    csp_neq_c    (csp, S, 0);
    csp_neq_c    (csp, M, 0);

    size_t n = csp_solve(csp, print_cb, vars);
    printf("%zu solutions\n", n);
    csp_free(csp);
    return 0;
}

```

That is the entire program. Compiled against the library and run, it prints

```

S=9 E=5 N=6 D=7 M=1 O=0 R=8 Y=2
1 solutions

```

We can verify by hand: $9567 + 1085 = 10652$. The Y at the end of MONEY is 2; the M at the front is 1; everything checks.

What the propagator did

The variables started with 10 values each. The all-different constraint, when first run, removes nothing — none of the variables is fixed. The linear-equality constraint is more interesting. With eight variables ranging over $[0, 9]$ and the given coefficients, the bounds-consistent propagator computes the minimum and maximum sum, and immediately sees that some bounds are forced.

For example, the coefficient of M is -9000 and the coefficient of S is 1000 . If $S = 9$ and $M = 1$, the contribution to the sum is $9000 - 9000 = 0$, balanced. But if $M \geq 2$, the term $-9000 \cdot M$ is at most -18000 , which the other terms cannot counteract within their ranges. So M is forced to 1 immediately, *at posting time*, before any branching happens. The unary `csp_neq_c(csp, M, 0)` then narrows it from $\{0, 1\}$ to $\{1\}$, and propagation cascades.

Similarly, the coefficient of S is 1000 and M now fixed at 1 means that $1000 \cdot S - 9000 \cdot 1 = 1000 \cdot S - 9000 \geq -1000$, which together with the other terms forces S to be high — eventually pinning it to 9.

You can see this by adding diagnostics. After posting the constraints but before calling `csp_solve`, query the domain sizes:

```

printf("After posting:\n");
printf("  S in dom: size = %u\n", csp_dom_size(csp, S));

```

```
printf(" M in dom: size = %u\n", csp_dom_size(csp, M));
```

The numbers will already be small. The search tree that `csp_solve` explores is consequently shallow — the existing `csp_cryptarithmic` program in the library reports that the puzzle solves in 25 microseconds, exploring only a few dozen nodes.

A second puzzle

The library's existing `csp_cryptarithmic.c` includes a second puzzle:

TWO + TWO = FOUR

with six distinct letters (T, W, O, F, U, R). The arithmetic, after rearrangement, becomes

$$198 \cdot T + 20 \cdot W + 2 \cdot O - 1000 \cdot F - 100 \cdot U - 10 \cdot R - 1 \cdot F - 1 \cdot R = 0$$

(I have left some bookkeeping in to show the structure; the actual coefficient on F when fully simplified is -1001 , and on R is -11 . You can rederive these by writing out the columns.)

This puzzle has multiple solutions because two-digit *plus* two-digit equals at most four-digit, so the shape constrains less. The library's program reports seven distinct (T, W, O, F, U, R) tuples.

Modelling lessons

What the cryptarithmic example teaches:

25. **Pick variables that match the puzzle's vocabulary.** The puzzle is about letters; the variables are letters. We did not introduce per-position variables.
26. **Encode arithmetic as a single linear constraint.** Avoid the trap of writing out the carry digits by hand; the linear-equality propagator is strong enough for this kind of equation, and modelling it directly is cleaner and faster.
27. **Use unary constraints for hard rejections.** Leading-zero rules become `csp_neq_c` calls.
28. **Trust the propagator.** With eight variables and three constraint types, the puzzle solves in microseconds. The branching does almost no work — propagation does it all.

In the next chapter we look at graph colouring, which is structurally simpler — pure pairwise disequality — but introduces an interesting question: how do we know when to give up and conclude no solution exists?

Chapter 8. Tutorial — graph colouring

Graph colouring asks whether the vertices of a graph can be coloured using k colours such that no two adjacent vertices share a colour. The decision version — is k -colouring possible? — is NP-complete in the worst case but solves quickly for moderate sizes via constraint propagation.

This tutorial shows how to encode a colouring problem as a CSP, find one colouring, count all distinct colourings, and detect that $k - 1$ colours are insufficient.

The encoding

A graph with n vertices becomes a CSP with n variables, each with domain $\{0, 1, \dots, k - 1\}$ (the available colours). For every edge (u, v) in the graph we post `csp_neq(csp, vars[u], vars[v])`. That is the entire constraint set.

```
#include "csp.h"
```

```

#include <stdio.h>

int main(void) {
    /* The Petersen graph: 10 vertices, 15 edges, chromatic number 3.
     *
     * Vertices 0..4 are the outer pentagon (0-1, 1-2, 2-3, 3-4, 4-0).
     * Vertices 5..9 are the inner pentagram (5-7, 7-9, 9-6, 6-8, 8-5).
     * Spokes connect outer to inner: 0-5, 1-6, 2-7, 3-8, 4-9. */
    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 { N = 10, K = 3 };

    csp_t *csp = csp_new();
    var_t v[N];
    for (int i = 0; i < N; i++) v[i] = csp_var(csp, 0, K - 1);
    for (size_t i = 0; i < sizeof edges / sizeof edges[0]; i++) {
        csp_neq(csp, v[edges[i][0]], v[edges[i][1]]);
    }

    size_t n = csp_solve(csp, NULL, NULL);
    printf("%zu colourings with %d colours\n", n, K);
    csp_free(csp);
    return 0;
}

```

For the Petersen graph with $k = 3$ the program prints 120 colourings. Each colouring is counted in every permutation of the colour labels ($3! = 6$ permutations), so the number of *distinct* colourings up to colour relabelling is $120 / 6 = 20$.

Reducing to $*k* = 2$

To check whether the Petersen graph is bipartite — colourable with 2 colours — change K to 2 and rerun. The program now reports 0 colourings. The propagator alone suffices to prove infeasibility, because the cycle 0-1-2-3-4-0 is of odd length (length 5), and odd cycles cannot be 2-coloured.

The proof inside the solver runs as follows. Suppose at the root that $v[0]$ is fixed at colour 0 (we can assume this by symmetry). Propagation from the edge 0-1 forces $v[1] = 1$. From 1-2: $v[2] = 0$. From 2-3: $v[3] = 1$. From 3-4: $v[4] = 0$. But now from 4-0: both $v[4]$ and $v[0]$ are 0, violating csp_neq . The propagator returns failure, and csp_solve reports zero solutions.

The library's first-fail variable selection automatically branches on a free variable — $v[0]$ here — and tries each colour. Both branches fail by propagation alone. The search tree has just two leaves, both infeasible; the run takes a few microseconds.

What if we did not have all-different at our disposal?

The library has csp_alldiff , but even without it, a colouring constraint is just a forest of pairwise csp_neq . The all-different of *all* vertices would not be the right model — we want different colours only on the *adjacent* pairs, not on every pair. Use csp_alldiff when the constraint is "no two of these can share," csp_neq for individual pairs, and choose based on the structure of your problem.

A common mistake: posting `csp_alldiff(csp, vars, n)` on the whole vertex set when only some pairs must differ. The result is a much stronger constraint than the problem demands — it says no two vertices *anywhere* in the graph share a colour, which forces the graph's chromatic number to n , almost certainly making the problem infeasible. The model has to match the problem.

A larger example

Consider a graph with 30 vertices and 100 edges, a typical scale at which you might want to know the chromatic number. The procedure is

29. Try $k = 1$: probably infeasible (works only if there are no edges).
30. Try $k = 2$: feasible iff the graph is bipartite.
31. Try $k = 3$: feasible iff the graph is 3-colourable.
32. Continue until `csp_solve` finds a solution.

At each k we build a fresh CSP — there is no in-place mechanism for changing a variable's domain after construction. Constructing and freeing a CSP is cheap, so this is a reasonable workflow.

A complete chromatic-number search:

```
static int colourable(int N, const int (*edges)[2], int n_edges, int K) {
    csp_t *csp = csp_new();
    var_t *v = malloc(N * sizeof *v);
    for (int i = 0; i < N; i++) v[i] = csp_var(csp, 0, K - 1);
    for (int i = 0; i < n_edges; i++)
        csp_neq(csp, v[edges[i][0]], v[edges[i][1]]);

    int found = 0;
    /* A callback that captures one solution and stops. */
    int saw = 0;
    /* Use NULL: just count, return 1 if any. */
    size_t n = csp_solve(csp, NULL, NULL);
    found = (n > 0);
    free(v);
    csp_free(csp);
    return found;
}

int chromatic_number(int N, const int (*edges)[2], int n_edges) {
    for (int k = 1; k <= N; k++) {
        if (colourable(N, edges, n_edges, k)) return k;
    }
    return N;
}
```

For most graphs this is faster than enumerating every assignment, sometimes dramatically so. The propagator catches infeasibility quickly when k is too small; it finds a colouring quickly when k is high enough.

A subtle point: counting versus stopping early

When you only want to know if a colouring exists, not enumerate them, it is faster to *stop after the first*:

```
static int got_one(const csp_t *csp, void *ud) {
    int *p = ud;
```

```

    *p = 1;
    return 0;    /* stop */
}

int found = 0;
csp_solve(csp, got_one, &found);
return found;

```

This is much faster than counting all colourings in cases where there are many. For the Petersen 3-colouring, counting all 120 takes longer than finding the first one. Decide what you actually need before running.

Summary

Graph colouring is an exemplary "use one constraint type, repeat" problem. The whole encoding is `csp_neq` calls. The interesting work happens in propagation — a particularly satisfying example because the propagator's deductions correspond exactly to the standard graph-theoretic argument about odd cycles being non-bipartite.

If your problem looks like graph colouring (vertices, edges, "no two adjacent X may share Y "), you can almost write the encoding mechanically: introduce a variable per X , post `csp_neq` per edge, run `csp_solve`. Tinker with the domain size to find the chromatic number.

The next tutorial — bin packing — uses linear arithmetic in earnest.

Chapter 9. Tutorial — bin packing

Bin packing asks: given n items of various sizes and m bins of fixed capacity C , can every item be placed in some bin without any bin exceeding its capacity? The decision version is NP-complete, but small instances solve quickly with constraint propagation.

This tutorial models bin packing as a CSP, runs a small instance, and discusses scaling.

The encoding

The natural variable is "which bin does item i go in?". Let $x[i]$ be the bin index of item i , with domain $\{0, 1, \dots, m-1\}$. The variables encode the assignment directly.

For each bin b , the *load* of bin b is the sum of the sizes of items assigned to it:

$$\text{load}(b) = \sum \text{size}[i] \cdot I(x[i] = b)$$

where I is the indicator function. The constraint is $\text{load}(b) \leq C$ for every bin.

To express $I(x[i] = b)$ as a CSP variable, we introduce **indicator variables**: for each (item, bin) pair, a 0/1 variable $y[i][b]$ that is 1 if $x[i] = b$ and 0 otherwise.

```

var_t y[N][M];
for (int i = 0; i < N; i++) {
    for (int b = 0; b < M; b++) {
        y[i][b] = csp_var(csp, 0, 1);
    }
}

```

We need to channel between x and y — each item goes in exactly one bin, so the indicators for item i sum to 1:

```
for (int i = 0; i < N; i++) {
    int32_t coeffs[M];
    for (int b = 0; b < M; b++) coeffs[b] = 1;
    csp_linear_eq(csp, y[i], coeffs, M, 1);
}
```

This says $\sum_b y[i][b] = 1$ for every item.

The capacity constraints

For each bin b , the sum of $(\text{size}[i] \cdot y[i][b])$ over all items must be at most C :

```
for (int b = 0; b < M; b++) {
    var_t bin_vars[N];
    int32_t bin_coeffs[N];
    for (int i = 0; i < N; i++) {
        bin_vars[i] = y[i][b];
        bin_coeffs[i] = sizes[i];
    }
    csp_linear_le(csp, bin_vars, bin_coeffs, N, C);
}
```

Each bin posts one `csp_linear_le` over the indicators for that bin, weighted by item sizes.

A complete program

A small instance: 5 items with sizes $\{3, 4, 4, 5, 6\}$ and 2 bins of capacity 11.

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

#define N 5
#define M 2
#define C 11

int main(void) {
    int32_t sizes[N] = { 3, 4, 4, 5, 6 };

    csp_t *csp = csp_new();
    var_t y[N][M];
    for (int i = 0; i < N; i++)
        for (int b = 0; b < M; b++)
            y[i][b] = csp_var(csp, 0, 1);

    /* One bin per item. */
    for (int i = 0; i < N; i++) {
        var_t vars[M];
        int32_t coeffs[M];
        for (int b = 0; b < M; b++) { vars[b] = y[i][b]; coeffs[b] = 1; }
        csp_linear_eq(csp, vars, coeffs, M, 1);
    }
}
```

```

/* Capacity per bin. */
for (int b = 0; b < M; b++) {
    var_t    vars[N];
    int32_t  coeffs[N];
    for (int i = 0; i < N; i++) { vars[i] = y[i][b]; coeffs[i] =
sizes[i]; }
    csp_linear_le(csp, vars, coeffs, N, C);
}

size_t n = csp_solve(csp, NULL, NULL);
printf("%zu packings\n", n);
csp_free(csp);
return 0;
}

```

The total size is $3 + 4 + 4 + 5 + 6 = 22$, which is exactly $2 \cdot 11$, so every valid packing must use both bins to full capacity. The packings (up to bin permutation) are $\{6,5\}$ and $\{3,4,4\}$, and $\{6,4,?\}$ would have load 10 + something, but to reach 11 we would need a 1 we do not have. The number of *ordered* packings the solver finds is 2 (the two orderings of which bin gets which set), and each contains 5 items so the solver visits $2 \times 1 = 2$ solution leaves.

The actual count printed by the program is 2 — there are exactly 2 distinct (ordered) packings.

What the propagator did

The linear-le propagator on each bin's capacity is doing real work. Consider bin 0 with the first item being size 6. If $y[0][0] = 1$, the bin's current load is at least 6, leaving capacity 5. Item 3 has size 5; if it goes in bin 0 too, that uses the full capacity, and the propagator deduces that $y[1][0]$, $y[2][0]$, and $y[4][0]$ (the other items into bin 0) must all be 0, forcing those items to bin 1.

The deduction chain is rapid. Within propagation, multiple bin constraints feed each other — items forced into one bin tighten the load of that bin, which forces items out, which tightens the *other* bin's load, and so on.

The bounds-consistent propagator works correctly here even though it does not reason about holes in domains, because the indicator domain is $\{0, 1\}$ — only two values — and the linear sum is monotone in each indicator.

Symmetry breaking

The encoding above counts each packing twice: once with item X in bin 0 and item Y in bin 1, and once swapped. For decision problems ("does *any* packing exist?") this is fine; for counting distinct packings it overcounts.

A standard technique is **symmetry breaking**: post a constraint that forces a canonical orientation. For bin packing, one easy choice is to insist that item 0 goes in bin 0:

```
csp_eq_c(csp, y[0][0], 1);
```

This rules out exactly half the assignments, leaving 1 distinct packing.

Symmetry breaking is a deep topic — for m bins instead of 2, the symmetry group is the full permutation group on bins, and breaking it cleanly requires more thought. For the purposes of this tutorial, the simple "fix item 0 to bin 0" works.

Scaling up

For larger instances, the encoding stays the same but the variable count grows: n items \times m bins indicator variables. With 50 items and 10 bins that is 500 indicators, and 60 linear constraints. The library handles this in well under a second for most instances.

The bottleneck for large bin packing is often *which order to try the items in*. The first-fail heuristic picks the smallest-domain unfixed variable, which after the initial propagation is typically the indicator for the largest item — the largest item has the most bins it cannot fit into, and so the propagator has narrowed its options furthest. This is a *good* default order: the largest items are placed first, which is what classical bin-packing heuristics like First Fit Decreasing do.

For very large instances (hundreds of items) you may want to write a more sophisticated propagator that does cardinality-based reasoning, but for the size that fits naturally on a piece of paper, the above is enough.

What we have shown

- How to encode an integer assignment problem (which bin?) using indicator variables ($y[i][b]$ is 1 if item i is in bin b).
- How to channel from indicators to a sum-equals-one constraint.
- How to express capacity using `csp_linear_le` with item sizes as coefficients.
- How symmetry breaking reduces the count of "essentially the same" solutions.

The general pattern — *integer variable plus indicator variables plus linear sums* — recurs throughout combinatorial optimisation. Whenever you have a "where does this go?" decision and "how much fits in here?" capacity, this is the shape.

The three CSP tutorials in this part have shown the API across three different problem shapes. The next chapter — Chapter 12 of the full book — will return to the API and show how to write a custom propagator when none of the built-in constraints capture what you need. But first, in Part III, we change tack entirely and look at exact cover via Dancing Links.

Part III — Exact Cover

Chapter 10. Exact cover

We change topic now. The three previous chapters were tutorials in the CSP framework: model the problem with variables and constraints, post the constraints, run the solver. Constraint solving is general but it is not always the cleanest fit. Some problems have a different shape: not "what value does each variable take?" but "which subset of options should I pick to cover everything exactly once?".

This chapter introduces that shape. The chapters after it show the data structure that makes searching it efficient, the API the library exposes, and worked examples.

The exact cover problem

Given a finite set U of items and a collection S of subsets of U , the **exact cover** problem asks: is there a sub-collection $S' \subseteq S$ such that every element of U is contained in exactly one set of S' ?

A small example. Take $U = \{1, 2, 3, 4, 5\}$ and the rows

	1	2	3	4	5
A	•			•	
B		•			•
C			•		
D	•	•			
E				•	•
F			•		•

Each row is a subset of U , marked by the columns it contains. The exact cover question is: which rows do we pick so that every column has exactly one mark in the chosen rows?

Try $\{A, B, C\}$: A covers $\{1, 4\}$, B covers $\{2, 5\}$, C covers $\{3\}$. The union is $\{1, 2, 3, 4, 5\}$, every column once. This is an exact cover.

Try $\{A, B, F\}$: A gives $\{1, 4\}$, B gives $\{2, 5\}$, F gives $\{3, 5\}$. Column 5 is covered twice. Invalid.

Try $\{D, C, E\}$: D gives $\{1, 2\}$, C gives $\{3\}$, E gives $\{4, 5\}$. Each column once. Another exact cover.

This small instance has at least these two solutions; in fact those two and no others.

The decision question and the enumeration question

The exact cover decision question is whether at least one cover exists. The enumeration question is to list them all. Both are NP-complete in general; the practical question is how small the search tree is on a given instance, which depends entirely on the structure of the rows.

Many combinatorial problems reduce to exact cover with a clean encoding:

- **Sudoku.** Each cell-value choice covers four "slots" (the cell itself; the row-value, column-value, and box-value uniqueness slots). A complete solution covers every slot exactly once.

- **N-queens.** Each queen covers a row, a column, and (with care) a diagonal. A complete placement covers every row and column exactly once and no diagonal twice.
- **Polyomino tiling.** Each piece-position covers a set of cells. A complete tiling covers every cell exactly once.
- **Crossword filling.** Each word-slot choice covers the slot itself and uses specific letters at specific cells. With letters as *colours* on cells, two word choices that share a cell must agree on the letter — a multi-cover where the cell is allowed to be shared but only with consistent values.

The encoding for each is mechanical once you have seen one or two; the exercise is identifying *what is the universe and what are the rows*.

Why a special-purpose solver?

You could encode exact cover as a CSP. Introduce a 0/1 variable for each row, post a `csp_linear_eq` for each column saying "the sum of row indicators that include this column equals 1", and run `csp_solve`. This works.

But it works less well than a dedicated exact-cover solver. The reason is that a CSP encoding loses an important structural fact: exact cover's branching can be much smarter than first-fail variable selection. At each search node, the *most-constrained column* — the one that is in the fewest remaining rows — is the right column to branch on. The CSP propagator does not see columns; it sees variables and linear sums.

The Dancing Links representation — the subject of the next chapter — gives $O(1)$ access to each column's *current* row count, after all the rows that would have conflicted with the current partial cover have been removed. Branching on the column with the fewest rows is then trivial. For sudoku and tiling problems the speedup over the equivalent CSP encoding is one to two orders of magnitude.

There is another, more aesthetic, reason. Exact cover encodes very directly. You list options ("here is a thing I could do"), each option lists what it covers ("if I do this, these things happen"), and the solver finds combinations of options that cover everything once. This is closer to many problem statements than the CSP form, and the resulting code is shorter.

What's coming next

Chapter 11 explains the data structure: a doubly-linked grid in which removing a row or column is reversible by inspection — you do not need a trail because the link relinking happens to invert the removal exactly.

Chapter 12 introduces *primary* and *secondary* items (the latter being "may be covered at most once" rather than "must be covered exactly once") and *colours*, a refinement that makes XCC strictly more expressive than plain exact cover.

Chapters 13 and 14 work the sudoku and n-queens problems in detail, end to end.

Chapter 15 gives some guidance on choosing between CSP and DLX for a given problem and shows how to combine them in one program (a real production setup).

A modelling exercise

Before you go on, try this. Take a 3×3 grid; place three rooks on it so that no two attack. The "exact cover" formulation is

- Universe: the three rows $\{R0, R1, R2\}$ and the three columns $\{C0, C1, C2\}$.

- Rows of the cover: every (row, column) pair, nine in total. Each row covers two universe items: the row R_i and the column C_j .

How many exact covers are there? Each one corresponds to a valid placement. The answer is $3! = 6$ — a permutation of three rooks across three rows.

Re-encode for the more realistic 8×8 case. The universe has 16 items (8 rows + 8 columns), the option set has 64 (every cell), and the answer is $8! = 40,320$ placements. We are not yet handling the diagonal constraint — that comes in the n-queens chapter, where secondary items handle "may be covered at most once".

If you've followed that, you have the gist. The next chapter shows the data structure that makes the search efficient.

Chapter 11. Dancing links

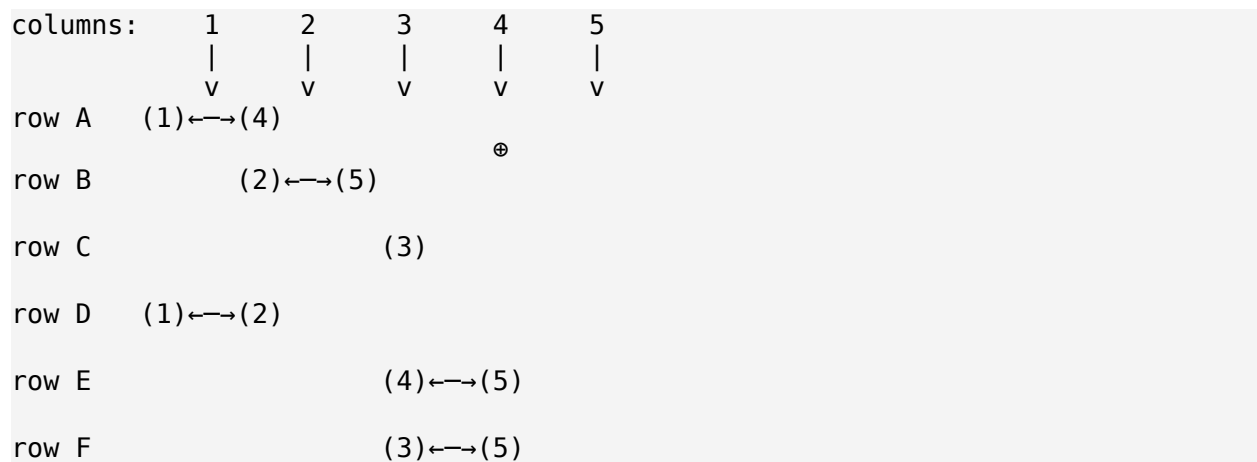
The exact cover problem can be solved with depth-first search: at each level pick an uncovered column, branch on the rows that cover it, recurse with that row's contribution accounted for. The naive implementation needs to keep track of which rows and columns are still in play and how to restore them when a branch fails.

Knuth's contribution, in the paper "Dancing Links" (2000), is to represent the problem as a doubly-linked structure in which removing a row or a column is one short loop of pointer rewiring, and **restoring** the same row or column is one short loop *in the reverse direction* of the same pointers. The data structure remembers how to undo itself.

This chapter explains the trick.

The structure

A row of an exact-cover matrix becomes a circular doubly-linked list of nodes, one per column the row covers. A column becomes a vertical circular doubly-linked list of nodes, one per row that covers the column. Each node lives at the intersection of a row and a column and has four pointers — left, right, up, down — that link it to its row neighbours and its column neighbours.



In this picture each (c) is a node holding a back-reference to its column header, and the arrows are left/right links. The vertical (up/down) links are not drawn but go between every two (c) cells in the same column.

There is also a sentinel column-header for each column (sitting "above" the topmost row) and a single root sentinel that joins all the column headers into another circular list — letting us iterate over the still-uncovered columns.

Covering a column

To pick row R as part of the cover, we must remove from the matrix every column that R covers (because they are now done) and every row that intersects any of those columns (because picking another such row would cover those columns again, violating exact cover).

The library does this in two pieces. First, *cover* the column header itself:

```
unlink the header from the horizontal sentinel ring:
    header.left.right = header.right
    header.right.left = header.left
```

After this, the column header is no longer reachable from the root sentinel by walking right. Searches for "uncovered columns" will skip it.

Second, *for each row that has a node in this column*, unlink that row from its other columns:

```
for each node n in this column (top to bottom):
    for each node m in n's row (n.right, then m.right, ...):
        unlink m from its column:
            m.up.down = m.down
            m.down.up = m.up
            m.column.size -= 1
```

After this, every row that touched this column is gone from the matrix as far as column membership is concerned.

The cost is proportional to the number of *cells* removed — every cell rewires four pointers and decrements one counter.

The dancing trick

Here is the magic. To *restore* the column, run the same loops in reverse order with reversed link operations:

```
for each row that touched the column (in reverse order):
    for each cell of the row (in reverse order):
        relink:
            m.up.down = m
            m.down.up = m
            m.column.size += 1
relink the column header:
    header.left.right = header
    header.right.left = header
```

The relinking works because **the unlinked node still holds its old left/right and up/down pointers, even after being unlinked**. Its neighbours have been told to skip over it, but the node itself still remembers who its neighbours used to be. To restore it, we just tell its neighbours to point *back* to it. The data structure carries its own undo information in the unlinked node's stale pointers.

This is the central observation. Knuth phrased it as the data structure "dancing" — pointers flying out and back. Mechanically, every operation has an obvious inverse that runs in the same time as the operation itself.

What about the trail?

There is no trail in DLX. Cover and uncover operations are inverses by construction. The search algorithm does:

33. Pick a column to branch on (heuristic: smallest column).
34. For each row in that column:
 - a. Add the row to the partial solution.
 - b. Cover the column.
 - c. For every other column that the row also touches, cover it.
 - d. Recurse.
 - e. After the recursive call, uncover everything in reverse: uncover the other columns (rightmost first), then uncover the branched column, then remove the row from the partial solution.

The "uncover everything in reverse" step is the only undo machinery, and it is just the inverse loop. There is no separate log of changes. This is what makes DLX so memory-efficient and so cache-friendly: no log to write, no separate restore phase, just a backward walk of the same pointers we used going forward.

A consequence is that DLX cannot easily share state with another piece of code that uses a different undo mechanism. If you want to combine DLX search with CSP propagation in the same recursion, you have a choice: either run them serially (CSP propagates first to narrow the candidate set, then DLX searches), or write a hybrid that maintains both undo mechanisms in lockstep. The library uses the serial pattern, and Chapter 15 will show why this is the right architectural choice.

Choosing a column to branch on

Knuth's standard heuristic is *S-heuristic*: at each branching step, choose the column with the smallest current count of remaining rows. This is the column whose remaining options are most constrained, and it is the analogue of first-fail variable selection in CSP.

Each column header carries a `size` field, decremented whenever a node in it is unlinked. To find the smallest column, walk the header sentinel ring:

```
column_t *best = root.right;
for (column_t *c = root.right; c != &root; c = c->right) {
    if (c->size < best->size) best = c;
}
```

The walk is $O(\text{number of columns})$, which is small compared to the cost of each search step. For sudoku with 324 columns, this is 324 comparisons per branching decision — negligible.

If the smallest column has zero remaining rows, the current partial cover cannot be extended — the column will never be covered. Backtrack.

If the smallest column has exactly one remaining row, that row is forced. Branch and recurse.

If it has more, branch on each in turn.

What this looks like in code

The library's DLX implementation lives in `dlx.c`. It is about five hundred lines, dominated by the various walk functions. The key cover/uncover pair is short:

```

static void cover_column(dlx_t *d, item_t col) {
    /* Unlink the column header from the horizontal ring. */
    nodes[nodes[col].right].left = nodes[col].left;
    nodes[nodes[col].left].right = nodes[col].right;

    /* For every row in this column, unlink the row from its other columns.
    */
    for (uint32_t i = nodes[col].down; i != col; i = nodes[i].down) {
        for (uint32_t j = nodes[i].right; j != i; j = nodes[j].right) {
            nodes[nodes[j].down].up = nodes[j].up;
            nodes[nodes[j].up].down = nodes[j].down;
            nodes[nodes[j].column].size -= 1;
        }
    }
}

static void uncover_column(dlx_t *d, item_t col) {
    /* The reverse of cover_column. Walk in reverse order. */
    for (uint32_t i = nodes[col].up; i != col; i = nodes[i].up) {
        for (uint32_t j = nodes[i].left; j != i; j = nodes[j].left) {
            nodes[nodes[j].column].size += 1;
            nodes[nodes[j].down].up = j;
            nodes[nodes[j].up].down = j;
        }
    }

    /* Relink the column header. */
    nodes[nodes[col].right].left = col;
    nodes[nodes[col].left].right = col;
}

```

(The actual library code is slightly different — it uses index-based "node IDs" rather than pointers, for cache density and serialisability, and it handles secondary items and colours which we have not yet introduced. But the shape is the same.)

The cover and uncover loops are mirror images of each other. The amount of work done is identical. There is no separate undo log because the data structure carries the undo information in its dangling pointers.

A note on memory layout

The library stores all nodes in a single contiguous array, with neighbour links as indices into the array. This is dramatically more cache-friendly than a malloc-per-node layout. For sudoku, the entire problem fits in a few kilobytes; for harder problems it scales linearly with the number of (option, item) cells.

The use of indices instead of pointers is a small but important detail. With pointers, every cover step would visit pointers that, having been freshly malloc'd, might be anywhere in the heap. With indices, every step visits an index that is a small integer, the array is laid out in posting order, and most accesses hit the same few cache lines.

What we have

A data structure that:

- Represents an exact-cover problem as a doubly-linked grid.

- Removes (covers) a column and all rows touching it in $O(\text{size of cells removed})$.
- Restores (uncovers) a column by running the inverse loop on the same pointers.
- Identifies the smallest still-active column in $O(\text{number of columns})$ for branching.

The next chapter introduces *primary* versus *secondary* items, and *colours*, which together raise the data structure from solving plain exact cover to solving the more expressive XCC. Then we begin the tutorials.

Chapter 12. Primary, secondary, and colour

The plain exact cover problem says every item must be covered exactly once. Many practical problems weaken this in two directions:

- Some items must be covered exactly once (the **primary** items).
- Some items may be covered at most once — they may be left uncovered, but no more than one option may include them (the **secondary** items).

This is the **exact cover with secondaries** problem, and it is what the library's DLX type handles. It is strictly more expressive than plain exact cover: any plain instance is a secondary-less instance.

A further refinement uses **colours**. When an item is secondary, multiple options may "claim" it as long as they all agree on a colour value attached to that claim. This is what lets crosswords work: the cell at row 3, column 5 might be claimed by several word-options, but they must all assign it the same letter. Without colours, those options would be in conflict; with colours, they coexist as long as their colour at that cell matches.

This combination — exact cover with secondaries and colours — is what Knuth calls **XCC** (exact cover with colours). The library's API uses the term *items* for both primary and secondary because that is how Knuth describes it.

The API

```
dlx_item_t dlx_primary (dlx_t *d, const char *name);
dlx_item_t dlx_secondary(dlx_t *d, const char *name);

dlx_opt_t  dlx_option_begin(dlx_t *d);
void       dlx_option_add  (dlx_t *d, dlx_opt_t o,
                           dlx_item_t item, uint32_t colour);
void       dlx_option_end  (dlx_t *d, dlx_opt_t o);

#define DLX_NOCOLOUR 0u
```

`dlx_primary` and `dlx_secondary` register an item, returning a handle. The optional `name` is purely for debugging; the solver never inspects it.

To create an option (a row of the matrix), call `dlx_option_begin` to start, `dlx_option_add` once per item the option covers, and `dlx_option_end` to finalise. Each `dlx_option_add` takes a colour tag — `DLX_NOCOLOUR` (which is just zero) for the common case where the option does not use colours.

The classic exact-cover examples — sudoku, polyominoes, n-queens — use primary items only and `DLX_NOCOLOUR` everywhere. We will work them in the next two chapters. Crosswords use colours; we will not work that example here, but the library has `dlx_crossword.c` if you want to read it.

Why secondaries

A motivating example: n -queens. Each queen covers a row, a column, and two diagonals (one positive-slope, one negative-slope). Rows and columns must be covered *exactly once* — every row needs a queen, every column needs a queen. But diagonals must be covered *at most once* — a board has $2n - 1$ diagonals of each slope, more than the n queens we are placing, so most diagonals will be empty.

If we treated diagonals as primary, the cover problem would demand that every diagonal be covered, which is impossible. If we treated diagonals as primary but had only some of them, we would be solving a different problem.

Secondaries are the right model: every diagonal is registered as a secondary item, and an option (a queen placement) covers two diagonals. The exact-cover constraint says no two queens may share a diagonal — covered at most once — but unused diagonals are fine.

How secondaries work in the search

The search algorithm chooses primary items to branch on, never secondaries. A secondary item is "covered" implicitly when a row that includes it is selected; the row's contribution to the secondary's column is the same as for a primary, but the search loop does not consider the secondary as a candidate for branching.

This works because the *only* role of an item in branching is to force the search to pick *some* row covering it. Primary items must be covered, so the search picks one row per primary; secondary items may be left uncovered, so the search has no obligation to pick a row for them. The covered-at-most-once constraint is enforced just like a primary's exactly-once: if two rows both touch the same secondary, picking one removes the other from contention.

In the library's code, the distinction between primary and secondary is one bit per item, examined when looking for the next column to branch on:

```
/* Find the next primary item to branch on. */
for (item_t i = root.right; i != ROOT; i = nodes[i].right) {
    if (!is_primary(d, i)) continue;
    /* ... compute size, track minimum ... */
}
```

If the loop completes with no primary item found, the search has solved the problem — every primary is covered. The current partial cover, plus whatever rows were taken to cover them, is an exact cover.

Colours

Colour tags are 32-bit integers. The convention is that 0 means "no colour" (the option claims the item without imposing any colour constraint), and non-zero means "this option claims the item with this colour value, and any other option also claiming this item must use the same colour."

Mechanically: when an option is selected in the search, for each item it covers with a colour tag, the column's "colour" field is set to the option's tag (if it was previously `DLX_NOCOLOUR`) or checked against the option's tag (if it was already set). If the check fails, the option cannot be selected — backtracking must consider the next option.

Two options claiming the same secondary with different non-zero colours are in conflict; they cannot both be in the cover. Two options claiming with the same non-zero colour are compatible.

Colours are most useful when the "secondary item" represents a shared resource that has a value, and options that share the resource must agree on the value. The crossword example: a cell shared by two word slots holds a single letter, and both words must agree on the letter.

When to use secondaries

A useful rule of thumb: if the item represents a *constraint* that must hold (every row of a sudoku must contain each digit), make it primary. If the item represents an *allowance* — at most one of these is allowed but it is fine to have none — make it secondary.

If you find yourself wanting to say "exactly one or zero of these," you have a secondary. If you want to say "exactly one of these," you have a primary. If you want to say "all of these must be the same value," you have a coloured secondary.

What this enables

The exercises that follow in Chapters 13 and 14 stay within plain exact cover, but you should be aware that the library can handle the more expressive forms. If you find a problem that does not quite fit plain exact cover, ask whether the missing constraint is "at most once" — secondary — or "all options choosing this must agree" — colour.

For example: a scheduling problem where each task takes one machine-hour and there are m machines and t hours. The primary items are the tasks (each task must be scheduled). The secondary items are (machine, hour) pairs (each pair may run at most one task). Each option is a (task, machine, hour) assignment, covering the task primary and the (machine, hour) secondary.

The same problem in CSP form is more elaborate — you would introduce a (machine, hour) variable per task and post mutual-exclusion constraints. The DLX form is closer to the natural statement.

We are now ready for the tutorials.

Chapter 13. Tutorial — Sudoku

Sudoku is the canonical demonstration of exact cover. The puzzle's structure is unambiguous, every constraint is a "covered exactly once" requirement, and the resulting DLX solver is dramatically faster than alternatives.

This tutorial encodes 9×9 sudoku from scratch and runs it on a published puzzle.

The four constraint families

A sudoku solution is a 9×9 grid of digits 1..9 such that:

- **Cell:** every cell contains exactly one digit.
- **Row:** every row contains each digit exactly once.
- **Column:** every column contains each digit exactly once.
- **Box:** every 3×3 box contains each digit exactly once.

In exact-cover form, each of these is a family of primary items:

- 81 **CELL** items, one per (row, col) pair: "(r, c) holds some digit."
- 81 **ROW** items, one per (row, digit) pair: "row r contains digit v."
- 81 **COL** items, one per (col, digit) pair: "column c contains digit v."
- 81 **BOX** items, one per (box, digit) pair: "box b contains digit v."

Total: 324 primary items.

The options are the candidate placements: for each (row, col, digit) triple, an option that says "put v at (r, c)." Each such option covers four items: CELL[r][c], ROW[r][v], COL[c][v], BOX[box(r,c)][v]. That is 729 options before any givens are applied.

A given digit at (r, c) eliminates 8 of the 9 options at that cell; we model this by simply *not* generating the conflicting options.

The encoding code

```
#include "dlx.h"
#include <stdio.h>
#include <stdlib.h>

#define N 9
#define BOX(r, c) (((r) / 3) * 3 + (c) / 3)

/* The puzzle. 0 means empty. */
static const int GIVENS[N][N] = {
    {5,3,0, 0,7,0, 0,0,0},
    {6,0,0, 1,9,5, 0,0,0},
    {0,9,8, 0,0,0, 0,6,0},
    {8,0,0, 0,6,0, 0,0,3},
    {4,0,0, 8,0,3, 0,0,1},
    {7,0,0, 0,2,0, 0,0,6},
    {0,6,0, 0,0,0, 2,8,0},
    {0,0,0, 4,1,9, 0,0,5},
    {0,0,0, 0,8,0, 0,7,9},
};

/* Held in user-data on each option: the (row, col, value) it represents. */
typedef struct { int r, c, v; } placement_t;

/* Build the DLX problem. */
static dlx_t *build_sudoku(void) {
    dlx_t *d = dlx_new();
    dlx_item_t cell[N][N], row[N][N], col[N][N], box[N][N];
    char name[32];

    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++) {
            snprintf(name, sizeof name, "cell_%d_%d", r, c);
            cell[r][c] = dlx_primary(d, name);
        }
    for (int r = 0; r < N; r++)
        for (int v = 0; v < N; v++) {
            snprintf(name, sizeof name, "row_%d_v%d", r, v);
            row[r][v] = dlx_primary(d, name);
        }
    for (int c = 0; c < N; c++)
        for (int v = 0; v < N; v++) {
            snprintf(name, sizeof name, "col_%d_v%d", c, v);
            col[c][v] = dlx_primary(d, name);
        }
}
```

```

for (int b = 0; b < N; b++)
    for (int v = 0; v < N; v++) {
        snprintf(name, sizeof name, "box_b%d_v%d", b, v);
        box[b][v] = dlx_primary(d, name);
    }

/* Generate options. Skip those incompatible with the givens. */
int n_options = 0;
for (int r = 0; r < N; r++) {
    for (int c = 0; c < N; c++) {
        for (int v = 0; v < N; v++) {
            /* Rule out options that would conflict with any given. */
            if (GIVENS[r][c] != 0 && GIVENS[r][c] - 1 != v) continue;
            /* Also rule out options that conflict with a given in
             * the same row, column, or box. */
            int conflicts = 0;
            for (int k = 0; k < N && !conflicts; k++) {
                if (GIVENS[r][k] - 1 == v && k != c) conflicts = 1;
                if (GIVENS[k][c] - 1 == v && k != r) conflicts = 1;
                int br = (r / 3) * 3 + k / 3;
                int bc = (c / 3) * 3 + k % 3;
                if (GIVENS[br][bc] - 1 == v && (br != r || bc != c))
                    conflicts = 1;
            }
            if (conflicts) continue;

            placement_t *p = malloc(sizeof *p);
            p->r = r; p->c = c; p->v = v;

            dlx_opt_t opt = dlx_option_begin(d);
            dlx_option_add(d, opt, cell[r][c], DLX_NOCOLOUR);
            dlx_option_add(d, opt, row[r][v], DLX_NOCOLOUR);
            dlx_option_add(d, opt, col[c][v], DLX_NOCOLOUR);
            dlx_option_add(d, opt, box[BOX(r,c)][v], DLX_NOCOLOUR);
            dlx_option_end(d, opt);

            dlx_option_set_userdata(d, opt, p);
            n_options++;
        }
    }
}
fprintf(stderr, "%d options\n", n_options);
return d;
}

static int solve_cb(const dlx_t *d, void *ud) {
    int grid[N][N] = {0};
    size_t n = dlx_chosen_count(d);
    for (size_t i = 0; i < n; i++) {
        dlx_opt_t o = dlx_chosen(d, i);
        placement_t *p = (placement_t *)dlx_option_userdata(d, o);
        grid[p->r][p->c] = p->v + 1;
    }
    /* Fill in the givens (which had no options to win, so no userdata). */
    for (int r = 0; r < N; r++)

```

```

        for (int c = 0; c < N; c++)
            if (grid[r][c] == 0) grid[r][c] = GIVENS[r][c];

    /* Print. */
    for (int r = 0; r < N; r++) {
        if (r % 3 == 0) printf("+---+---+---+\n");
        for (int c = 0; c < N; c++) {
            if (c % 3 == 0) printf("|");
            printf("%d", grid[r][c]);
        }
        printf("|\n");
    }
    printf("+---+---+---+\n");
    return 0;    /* stop after first solution */
}

int main(void) {
    dlx_t *d = build_sudoku();
    dlx_solve(d, solve_cb, NULL);
    /* Free user-data... omitted for brevity. */
    dlx_free(d);
    return 0;
}

```

This is the entire program. It posts 324 primary items, generates options skipping ones that conflict with givens, runs the search, prints the first solution.

For the Wikipedia "easy" puzzle above, the library reports 183 options after pruning by givens (down from 729) and finds the unique solution in well under a millisecond.

What the search did

The S-heuristic — branch on the smallest column — is doing real work here. Given the puzzle, several columns may have only one row left after the givens have been processed. For example, if a row already has all of digits 1-8 placed, then the ROW[r][9] item has only one (row, col, 9) option remaining — the cell that does not yet hold a digit. The search picks this column first, makes the forced choice, and propagates.

This is exactly what a human solver does — find the cell or row that has only one possibility and fill it in. The search continues until either the puzzle is solved or a forced choice fails (in which case backtracking restores the previous state).

For "easy" puzzles, the search makes no choices that need backtracking — it just walks down the chain of forced moves. For "hard" or "evil" puzzles, some level of backtracking is required, and the search may visit hundreds or thousands of nodes. The library still solves them in milliseconds.

Why the DLX encoding wins for sudoku

A sudoku CSP encoding (with 81 variables and 27 alldiff constraints) is correct but slow. The library's `csp_sudoku.c` and `dlx_sudoku.c` solve the same puzzle, and the DLX version is faster by roughly an order of magnitude.

The reason is that DLX's branching heuristic — smallest column — is exactly the right granularity for sudoku. Picking the most-constrained *cell-or-row-or-column-or-box-slot* is finer than picking the most-constrained *cell* (which is what CSP first-fail does over an 81-variable model). DLX sees that a particular

row has only one place left for the digit 9 even before any cell becomes uniquely determined; CSP has to wait until propagation discovers this through alldiff narrowing.

This is the central trade-off between CSP and DLX for problems that fit both. CSP is more general — you can post any constraint you want. DLX is more specific — your problem has to be exact cover — but when it is, DLX's tighter branching often wins.

A note on performance

The library reports timings for both solvers. On the Wikipedia easy puzzle:

- `csp_puzzles` (the CSP encoding): a few tens of microseconds.
- `dlx_sudoku`: about ten microseconds.

On harder puzzles (the world's hardest 17-clue puzzles, for example), the gap widens — DLX stays in the millisecond range, while CSP can take tens of milliseconds.

Both are fast enough for any practical use. The point of measuring is not to pick a winner but to understand the structure of the problem and the strengths of each tool.

What we have shown

- How to recognise sudoku as exact cover (four families of primary items).
- How to build the DLX problem mechanically from the puzzle structure.
- How to handle givens by pruning the option set.
- How to read back a solution by walking the chosen options.

The next chapter does the same for *n*-queens, which has a different mix — primary items for rows and columns, secondaries for diagonals — and a much smaller option set. Then we close Part III.

Chapter 14. Tutorial — N-queens

Eight queens on a chessboard, no two attacking. The classic. We saw a CSP encoding in Chapter 6; the DLX encoding is shorter, faster, and more revealing.

The encoding

For an $N \times N$ board:

- **ROW** items, one per row: each row gets exactly one queen. Primary.
- **COL** items, one per column: each column gets exactly one queen. Primary.
- **D+** items, one per NE-SW diagonal (indexed by $r + c$, range 0 to $2N - 2$): each such diagonal gets at most one queen. Secondary.
- **D-** items, one per NW-SE diagonal (indexed by $r - c + (N - 1)$, range 0 to $2N - 2$): at most one queen. Secondary.

Each option is a (row, col) placement. It covers four items: `ROW[r]`, `COL[c]`, `D+[r+c]`, `D-[r-c+N-1]`. The first two are primary, the latter two secondary.

The total: $2N$ primary + $2(2N - 1)$ secondary items, and N^2 options.

For $N = 8$ that is 16 primary + 30 secondary = 46 items, and 64 options.

The code

```
#include "dlx.h"
```

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define N 8

typedef struct { int r, c; } pos_t;

static int print_first_cb(const dlx_t *d, void *ud) {
    (void)ud;
    int board[N][N];
    memset(board, 0, sizeof board);
    for (size_t i = 0; i < dlx_chosen_count(d); i++) {
        const pos_t *p = dlx_option_userdata(d, dlx_chosen(d, i));
        board[p->r][p->c] = 1;
    }
    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++) putchar(board[r][c] ? 'Q' : '.');
        putchar('\n');
    }
    return 0;
}

int main(void) {
    dlx_t *d = dlx_new();
    char name[32];

    /* Primary: rows and columns. */
    dlx_item_t row[N], col[N];
    for (int i = 0; i < N; i++) {
        snprintf(name, sizeof name, "R%d", i);
        row[i] = dlx_primary(d, name);
    }
    for (int i = 0; i < N; i++) {
        snprintf(name, sizeof name, "C%d", i);
        col[i] = dlx_primary(d, name);
    }

    /* Secondary: diagonals. */
    dlx_item_t dpos[2*N - 1], dneg[2*N - 1];
    for (int i = 0; i < 2*N - 1; i++) {
        snprintf(name, sizeof name, "D+%d", i);
        dpos[i] = dlx_secondary(d, name);
    }
    for (int i = 0; i < 2*N - 1; i++) {
        snprintf(name, sizeof name, "D-%d", i);
        dneg[i] = dlx_secondary(d, name);
    }

    /* Options: every (r, c) placement. */
    pos_t *positions = malloc(N * N * sizeof *positions);
    int n_pos = 0;
    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++) {
            positions[n_pos] = (pos_t){ r, c };

```

```

        dlx_opt_t opt = dlx_option_begin(d);
        dlx_option_add(d, opt, row[r],          DLX_NOCOLOUR);
        dlx_option_add(d, opt, col[c],          DLX_NOCOLOUR);
        dlx_option_add(d, opt, dpos[r + c],      DLX_NOCOLOUR);
        dlx_option_add(d, opt, dneg[r - c + N - 1], DLX_NOCOLOUR);
        dlx_option_end(d, opt);
        dlx_option_set_userdata(d, opt, &positions[n_pos]);
        n_pos++;
    }
}

dlx_solve(d, print_first_cb, NULL);

free(positions);
dlx_free(d);
return 0;
}

```

That is the entire program. Compiled and run, it prints

```

Q.....
...Q.
...Q....
...Q...
...Q..
...Q
.Q.....
...Q...
..Q.....

```

— a valid 8-queens solution.

To enumerate all solutions, change the callback to return 1 (continue) and count. The library reports 92 solutions for $N = 8$, found in well under a millisecond.

Why secondaries are essential here

If diagonals were primary, the search would require *every* diagonal to be covered. There are 15 NE-SW diagonals and 15 NW-SE diagonals on an 8×8 board, but only 8 queens, so at most 8 of each type can be covered. The encoding would be infeasible.

Making them secondary expresses the right thing: covered at most once. The search picks 8 row primaries and 8 column primaries (each forced once it has been the unique remaining option), and the diagonal secondaries are covered at most once as a side effect of which (row, col) pairs end up selected.

The branching heuristic — smallest primary column — picks the row or column with the fewest remaining placements at each step. As queens are placed, more rows and columns are eliminated, and the search rapidly narrows.

Comparing to the CSP version

In Chapter 6 we showed an n -queens CSP encoding using `csp_alldiff` for "no two queens in the same column" and a forest of `csp_forbidden_pairs` for the diagonals. That works but is uglier — the diagonal constraint requires building a forbidden-pairs table for each row pair, with $O(N^2)$ tables and $O(N)$ entries each.

The DLX encoding has N^2 options, each with four items. The total memory and time per option is constant. The diagonal constraint is expressed by literally identifying the diagonal item the placement covers; no tables, no nested loops.

For $N = 12$, the CSP version takes a few milliseconds; the DLX version is dramatically faster. For larger N the gap widens. By $N = 16$ (with 14,772,512 solutions) DLX is the only practical choice.

Why this works so well

The shape of the n -queens problem maps cleanly to exact cover. We have things that must happen exactly once (queens per row, queens per column) and things that may happen at most once (diagonal sharing). DLX expresses both directly with primary and secondary items, and the search heuristic naturally discovers forced placements as branches close.

For comparison with sudoku in the previous chapter: sudoku used only primary items because *every* constraint family is "exactly once." N -queens used both. As you encounter more problems, you will get a feel for which structure they have.

Summary

The DLX framework expresses combinatorial problems with a particular structure — pick options to cover requirements — directly and concisely. For sudoku and n -queens, the encoding is short, the search is fast, and the program is easy to read.

We close Part III here. There are more examples in the library — `dlx_pentomino.c` for polyomino tiling, `dlx_crossword.c` for crossword filling with colours — and you should read them as reference. Each demonstrates the same shape: identify items, identify options, list which items each option covers, run the solver.

Part IV, just one chapter, brings the two strands together and discusses how to choose between CSP and DLX for a given problem and how to combine them when neither alone is the right fit.

Part IV — Choosing

Chapter 15. CSP versus DLX, and combining them

The two solvers in this book are general-purpose tools that approach combinatorial problems from different angles. CSP is more flexible: any predicate over variables and domains can be a constraint, and the API supports arithmetic, table lookups, all-different, and arbitrary user propagators. DLX is more focussed: any problem expressible as exact cover (with secondaries and colours) is fast, but problems outside that shape do not fit at all.

This closing chapter is about how to choose, and about a hybrid pattern that uses both in the same program.

When CSP is the right tool

CSP is the right tool when the problem has a strong arithmetic flavour, when constraints are over more than one or two variables in irregular combinations, or when the propagator's deductions correspond to natural reasoning about the problem.

Some examples:

- **Cryptarithmic** is purely arithmetic; the linear-equality propagator is exactly the right reasoning engine.
- **Bin packing** uses linear-le over indicator variables; nothing in DLX expresses "the sum of selected weights is \leq capacity" naturally.
- **Scheduling with cumulative resource constraints** is CSP territory.
- **Rule-based decision making** (run propagation, read which options have been ruled out) uses the CSP as a propagator-only system; DLX has no analogue.

CSP is also the right tool when the solver must run a *user* propagator that cannot be expressed in the existing constraint set. The library's CSP architecture supports this directly through the watch-list interface (which we have not opened up at this depth, but which is documented in `Csp.c`).

When DLX is the right tool

DLX is the right tool when the problem is genuinely about choosing a subset of options to cover a fixed universe, and when the items in the universe naturally divide into "must be covered" and "may be covered."

Some examples:

- **Sudoku, polyominoes, tilings** — the classical exact-cover applications.
- **N-queens and other constraint-satisfaction problems with secondary "at-most" constraints.**
- **Crossword filling**, where the colour mechanism handles letter agreement at shared cells.
- **Set-cover-style problems** more generally, when the relationship is "this option supplies these requirements; pick a covering set."

DLX shines at branching efficiency for these structures because the smallest-column heuristic — picking the most constrained item to branch on — is exactly the right granularity.

When to suspect one over the other

Some heuristics for diagnosis:

- If your problem statement contains arithmetic ("the sum of these is...", "twice this plus three times that..."), suspect CSP.
- If your problem statement contains the word *cover* or "exactly one" naturally ("each row gets a queen", "each cell holds a digit"), suspect DLX.
- If the problem combines both — a cover problem with arithmetic side constraints — you may need a hybrid; see below.
- If you are not sure, try the cleaner encoding and see if it works. Both solvers are fast enough on small problems that you can afford to prototype.

When the encoding is constrained

Some problems can be expressed in either form, but one is more natural. Sudoku is a cover problem (cell-row-column-box uniqueness); the CSP encoding works but is verbose and slower. Bin packing is an arithmetic problem; you could try to express it in DLX but the encoding is unwieldy.

The choice is not always obvious. Polyomino tiling can be CSP — variables for each cell, domain is "which polyomino piece occupies this cell, in which orientation, at which offset", with all-pairs same-piece constraints — but the DLX form is shorter and faster. Crossword filling can be CSP — each slot is a variable over the dictionary of words — but the colour mechanism in DLX makes letter sharing cleaner.

In practice the choice gets easier with experience. If neither feels natural, try both prototypes; the better fit is usually obvious within a few hundred lines of code.

The hybrid pattern

There is a pattern that uses both solvers in the same program, in sequence rather than nested. The shape is:

35. **Rule layer (CSP, propagator only).** Post hard rules using `csp_eq_c`, `csp_neq_c`, and `csp_linear_le`. Run `csp_propagate`. Read each variable's domain to discover which options have been ruled out.
36. **Assignment layer (DLX, full search).** Build an exact-cover problem over the surviving options. Each option that the rule layer did not eliminate becomes a row in the cover. The cover constraints encode the assignment problem (e.g., each track gets at most one weapon, each weapon's magazine is bounded). Run `dlx_solve`.
37. **Read back.** The DLX solution identifies the chosen options. Apply them.

Why split it? The rule layer's job is to express what *must* hold and let propagation figure out the consequences. CSP's `csp_propagate` is exactly the right tool for this — fast, declarative, and extensible. The assignment layer's job is to optimise: given the candidates, which subset maximises some score? DLX with a "best so far" callback handles this directly.

Trying to do both in one solver is awkward. A pure CSP solver would have to encode the optimisation as an objective variable plus search, which is correct but slow. A pure DLX solver would have to express every rule as constraints on the cover, losing the modularity of the rule layer.

The serial pattern keeps each layer doing what it is best at. It also makes each layer testable in isolation — you can verify the rule layer's narrowing without running search, and you can verify the assignment layer's optimisation on a fixed candidate set without re-running rules.

A word about state sharing

The trail in the CSP and the link-rewiring undo in DLX are different mechanisms. They cannot be interleaved in a single recursion without considerable engineering. The serial pattern works because the rule layer runs to completion (`csp_propagate` returns) before the assignment layer starts; the trail is then no longer needed, and DLX can build its own world from scratch.

If you ever do need them interleaved — for example, a propagator that uses DLX as part of its propagation — you have to extend one of the data structures to play nicely with the other. This is a research project, not a weekend's work, and you almost never actually need it.

What this book has not covered

A few notable omissions:

- **Optimisation in CSP.** The library counts solutions and finds them, but it does not maximise an objective directly. To find the best solution, run `csp_solve` with a callback that compares each found solution to a running best and stops when an upper bound is reached. This is a standard technique; the library's design supports it, but no built-in helper does it for you.
- **Strong all-different.** The library's `csp_alldiff` decomposes to pairwise disequality. For problems where matching-based reasoning would help, the strong version (Régin's algorithm) is a known engineering target.
- **Soft constraints and weighted CSP.** When some constraints can be violated at a cost, the resulting problem (MaxSAT, weighted CSP) is a different beast. The library does not address it.
- **Distributed and parallel search.** Both solvers are single-threaded. Parallelising backtracking search is well-studied; nothing in the library precludes it, but nothing supports it either.
- **The full theory of exact cover.** Knuth's *Art of Computer Programming, Volume 4 Fascicle 5* covers Dancing Cells in depth; this book is the gentler introduction.

These are good directions for further reading and for extending the library. The concrete code base is small enough that adding a new propagator, a new constraint type, or a new search heuristic is a matter of a few hundred lines.

Closing

The four ideas this book has developed — the trail, the sparse set, CSP, and DLX — are a coherent set of tools for a particular shape of problem: predict, prune, recover, repeat. None of them is novel; all of them have been refined for decades in the literature. The contribution of this book is to present them small, with working code, and to show that you can hold them all in one head.

The library is in front of you. It is under two thousand lines of C99. It has tests, benchmarks, and worked examples for half a dozen problems. Read it. Modify it. Break it and put it back together. The best way to internalise these techniques is to build something with them.

Good luck.