

Constraint

Aayush Bajaj

July 10, 2026

Contents

1	the formalism	1
2	propagation and consistency	2
2.1	ac-3	2
2.2	a worked queue trace	2
3	global constraints	3
3.1	alldifferent	3
3.2	cumulative	4
4	search	4
5	branch and infer vs branch and bound	5
6	code	5
7	see also	7

constraint programming inverts the usual deal: you state *what* a solution must satisfy, and a general-purpose solver figures out *how* to find one. no objective gradient, no simplex tableau — just variables, finite domains, and constraints, attacked by an alternation of *inference* (prune values that cannot appear in any solution) and *search* (guess, propagate, backtrack). this page builds the machinery from the formalism up; the [sudoku solver](#) and [hashiwokakero](#) write-ups on this wiki are the same machinery pointed at actual puzzles.

freuder's slogan: the user states the problem, the computer solves it — the closest computer science has come to the holy grail of programming

1 the formalism

a **constraint satisfaction problem** (CSP) is a triple (X, D, C) :

- **variables** $X = \{x_1, \dots, x_n\}$.
- **domains** $D = \{D_1, \dots, D_n\}$, each D_i a finite set of candidate values for x_i .
- **constraints** $C = \{c_1, \dots, c_m\}$, each $c_j = (S_j, R_j)$ with *scope* $S_j \subseteq X$ and *relation* R_j a subset of the cartesian product of the scoped domains — the tuples the constraint permits.

an **assignment** maps some variables to values in their domains; it is **consistent** if it violates no constraint whose scope it covers, **complete** if it covers all of X . a **solution** is a consistent complete assignment. deciding whether one exists is NP-complete already for binary constraints over three-value domains (graph 3-colouring

is exactly that CSP), so everything below is about making an exponential search collapse in practice, not about escaping it in theory.

structure lives in the **constraint graph**: one node per variable, one edge per binary constraint. constraints of higher arity make it a **hypergraph** — each constraint is a hyperedge covering its scope. the graph is not decoration: tree-structured constraint graphs are solvable in $O(nd^2)$ by one sweep of directional arc consistency, and the general-case parameter that interpolates between “tree” and “hopeless” is the induced width of this graph.

any csp can be binarised — the hidden-variable and dual encodings trade arity for extra variables — so binary theory loses no generality, only efficiency

2 propagation and consistency

inference in CP means enforcing a local *consistency* property, deleting domain values that fail it, and repeating to a fixpoint. the two workhorses:

- **node consistency**: every value in D_i satisfies each unary constraint on x_i . enforced once, in a single pass — fold unary constraints into the domains and forget them.
- **arc consistency**: for every directed arc (x_i, x_j) carrying a binary constraint, every value $v \in D_i$ has a **support** — some $w \in D_j$ with (v, w) permitted. a value without support appears in no solution, so deleting it is safe: propagation never loses a solution, it only shrinks the search space.

2.1 ac-3

mackworth’s AC-3 enforces arc consistency with a worklist of arcs.¹

- **input**: domains, and a queue holding every directed arc.
- **loop**: pop an arc (x_i, x_j) ; **revise** it — delete each value of D_i with no support in D_j .
- **wake-up rule**: if revise deleted anything, the deletion may have destroyed supports elsewhere, so re-enqueue every arc (x_k, x_i) for neighbours $x_k \neq x_j$ (skipping arcs already queued).
- **terminate**: queue empty (arc-consistent fixpoint) or some domain wiped out (no solution — the strongest thing propagation alone can prove).

with e arcs and domains of size at most d , each arc re-enters the queue $O(d)$ times and a revise costs $O(d^2)$, giving $O(ed^3)$ time. AC-4 gets the optimal $O(ed^2)$ by bookkeeping support counts, but its bookkeeping costs that much *always*; AC-2001 reaches the same bound while staying as lazy as AC-3, which is why AC-3’s skeleton is the one everybody actually implements.

2.2 a worked queue trace

take the chain CSP: variables x, y, z , all domains $\{1, 2, 3\}$, constraints $x < y$ and $y < z$. initial queue: $(x, y), (y, x), (y, z), (z, y)$.

¹mackworth (1977), *consistency in networks of relations*, artificial intelligence 8(1). the $O(ed^3)$ bound is from mackworth and freuder (1985), artificial intelligence 25(1).

pop	revise	domains $x \mid y \mid z$	queue after
(x, y)	$x = 3$ has no support (no $y > 3$): prune 3	$\{1, 2\} \mid \{1, 2, 3\} \mid \{1, 2, 3\}$	$(y, x), (y, z), (z, y)$
(y, x)	$y = 1$ has no support (no $x < 1$): prune 1	$\{1, 2\} \mid \{2, 3\} \mid \{1, 2, 3\}$	$(y, z), (z, y)$
(y, z)	$y = 3$ has no support (no $z > 3$): prune 3	$\{1, 2\} \mid \{2\} \mid \{1, 2, 3\}$	$(z, y), (x, y)$
(z, y)	$z = 1, 2$ lack support (need $y < z, y = 2$): prune both	$\{1, 2\} \mid \{2\} \mid \{3\}$	(x, y)
(x, y)	$x = 2$ has no support ($x < 2$ fails): prune 2	$\{1\} \mid \{2\} \mid \{3\}$	empty

bookkeeping notes, step by step: step 1 revises x , whose only neighbour is y itself, so nothing wakes up. step 2 revises y and would wake (z, y) , but it is already queued. step 3 revises y again and wakes (x, y) — the arc processed back in step 1, back for another pass because its support landscape changed. step 4 revises z ; x is not adjacent to z , so nothing wakes. the fixpoint is the singleton assignment $x = 1, y = 2, z = 3$: on this instance arc consistency alone *solves* the problem. do not be seduced — that is a property of this tiny chain, not of AC-3. the colouring CSP below reaches its arc-consistent fixpoint with a variable still holding two values, and search has to finish the job.

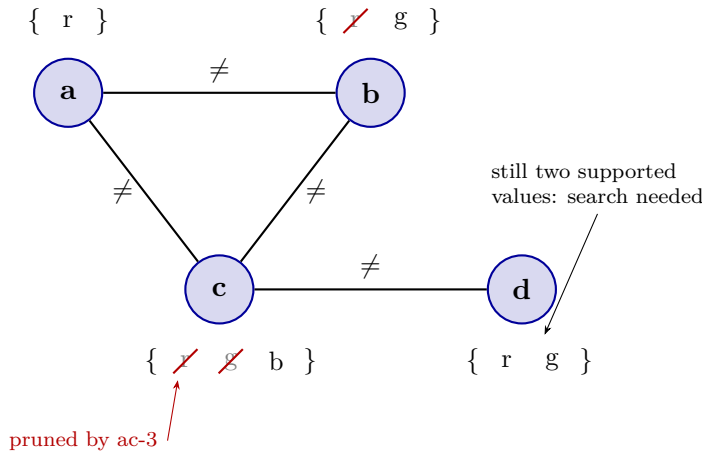


Figure 1: a colouring csp as a constraint graph: nodes are variables, edges are disequality constraints, and each variable’s domain is written beside it. grey struck-out values are exactly what ac-3 prunes; d keeps two supported values, so propagation alone does not finish this instance.

3 global constraints

arity-two thinking hits a wall fast. a **global constraint** is a single constraint over many variables packaged with a dedicated propagation algorithm that exploits its combinatorial structure — the real engine of industrial CP.

3.1 alldifferent

$\text{alldifferent}(x_1, \dots, x_n)$ demands pairwise distinct values. the naive **decomposition** into $\binom{n}{2}$ disequalities is logically equivalent but propagates feebly: a binary $x_i \neq x_j$ only prunes when one side is already a singleton. take three variables with domains $\{1, 2\}$ — every pairwise arc is consistent (each value of each variable has a support on every arc), yet the conjunction is unsatisfiable outright: three variables, two values, pigeonhole (Epp, Susanna S., 2019). arc consistency on the pieces sees nothing; **generalised arc consistency** (GAC) on the global constraint — every value extendable to a full satisfying tuple of the constraint — detects the wipe-out immediately.

regin showed GAC on alldifferent is tractable via matching theory.² build the bipartite *value graph* (variables on one side, values on the other, edges = domain membership); the constraint is satisfiable iff a maximum matching covers every variable, and a value can be pruned from a domain iff its edge lies in *no* maximum matching — decidable for all edges at once from one maximum matching plus a strongly-connected-components pass over the associated directed graph (berge’s alternating-cycle characterisation). the matching itself is a max-flow computation (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009), hopcroft–karp gives $O(\sqrt{n}m)$, and solvers amortise by repairing the previous matching incrementally after each domain change. honest caveat: this is still the priciest propagator in a typical model, so solvers offer weaker bounds-consistency variants of alldifferent when the full filtering does not pay for itself.

3.2 cumulative

cumulative is scheduling’s workhorse: tasks i with start variable s_i , duration d_i and resource demand r_i share a resource of capacity R , and the constraint enforces

$$\sum_{i: s_i \leq t < s_i + d_i} r_i \leq R \quad \text{for every time } t. \quad (1)$$

introduced in the CHIP system for resource-constrained project scheduling.³ here even deciding feasibility is NP-hard, so *no* polynomial propagator can enforce GAC; practical filtering is deliberately incomplete and layered — **timetabling** (accumulate each task’s compulsory part, the interval where it must sit however it slides, and forbid overloads), **edge-finding** (deduce that a task must end before, or start after, a whole set of others because the set’s combined energy leaves no room otherwise), and **energetic reasoning** (compare available area $R \times |I|$ of a time window against the minimum work forced inside it). this is the pattern of the whole field: pick the strongest filtering whose cost per node you can afford.

4 search

propagation prunes; it rarely finishes. the rest is tree search over assignments.

- **chronological backtracking**: depth-first — pick an unassigned variable, try a value consistent with the assignments so far, recurse; on failure undo and try the next value; when values run out, retreat to the previous variable. complete, exponential, and prone to *thrashing*: rediscovering the same dead end under thousands of irrelevant contexts.
- **variable ordering** — **MRV / fail-first**: branch on the variable with the fewest remaining values. a variable with two values left will be wrong half the time — better to be wrong near the root. ties break by *degree*: prefer the variable constraining the most unassigned others.
- **value ordering** — **LCV / succeed-first**: given the variable, try first the value that deletes the fewest values from neighbouring domains. variable choice wants to provoke failure early; value choice wants to dodge it — you must expand every value of a doomed variable anyway, but one good value suffices if a solution lies below.

fail-first is honest pessimism: if a variable is doomed, find out while the tree above it is still small

²regin (1994), *a filtering algorithm for constraints of difference in csps*, proceedings of aaai-94.

³aggoun and beldiceanu (1993), *extending chip in order to solve complex scheduling and placement problems*, mathematical and computer modelling 17(7).

- **forward checking (FC)**: after assigning $x_i := v$, revise each unassigned neighbour's domain against x_i alone — one revise per neighbour and stop. catches immediate wipe-outs and feeds MRV fresh domain counts, but never propagates a deletion's *consequences*: prunings that would only follow through a chain of two or more constraints go unseen.
- **MAC (maintaining arc consistency)**: after the same assignment, seed AC-3's queue with the arcs (x_k, x_i) into the assigned variable and run to fixpoint, so deletions cascade through the whole constraint graph. strictly stronger pruning than FC for strictly more work per node; on hard structured instances MAC's smaller trees win decisively, and it is the default in serious solvers. modern engines stack *conflict-directed backjumping*, nogood recording and restarts on top — the same lessons SAT solvers learned, arriving by convergent evolution.

5 branch and infer vs branch and bound

CP and **mixed-integer programming** both wrap tree search around per-node reasoning, but the reasoning is philosophically different: MILP *relaxes* (drop integrality, solve the LP, bound the objective), CP *infers* (keep the logic, shrink the domains).⁴

aspect	constraint programming	mixed-integer programming
model	logic + global constraints; native disjunction	linear (in)equalities; some variables integer
per-node inference	propagation to a consistency fixpoint	LP relaxation, tightened by cutting planes
bound on objective	weak: objective is one more variable	strong dual bound; drives the pruning
feasibility logic	strong: propagators see the structure	weak: big-M encodings of logic
optimality proof	exhaust the tree, bounded by the incumbent	close the incumbent-relaxation gap
wins at	scheduling, sequencing, rostering	blending, flows, graded cost trade-offs

the folk theorem holds up: when the difficulty is *finding anything feasible* among tangled combinatorial rules, CP's inference dominates; when the difficulty is *trading off costs* across loosely-coupled decisions, the LP bound is the asset and MILP dominates. an objective enters CP as branch-and-bound over a domain: post $z < z_{\text{incumbent}}$ and let propagation feel it — effective exactly when constraints link z tightly to the decision variables, which in scheduling (makespan) they do and in costing problems they often do not. hybrid decompositions that let an LP bound talk to CP propagators take both columns' strengths, at the price of owning two models of the same problem.

6 code

AC-3 from scratch — a queue of directed arcs, a **revise** that deletes unsupported values, and the wake-up rule; run on the worked chain and then on the colouring CSP from the diagram:

```

1  from collections import deque
2
3  def revise(domains, xi, xj, ok):
4      """prune values of xi with no support in xj. returns the pruned values."""
5      pruned = [v for v in domains[xi] if not any(ok(v, w) for w in domains[xj])]
6      for v in pruned:
7          domains[xi].remove(v)
8      return pruned

```

⁴the framing and the name come from bockmayr and kasper (1998), *branch and infer: a unifying framework for integer and finite domain constraint programming*, informs journal on computing 10(3). the broader canon: russell and norvig's *artificial intelligence: a modern approach* (csp chapter) for the search heuristics, and rossi, van beek and walsh (eds.), *handbook of constraint programming* (2006) for everything else.

```

9
10 def ac3(domains, constraints):
11     """constraints: dict {(xi, xj): ok} listing every directed arc."""
12     queue = deque(constraints)
13     while queue:
14         xi, xj = queue.popleft()
15         pruned = revise(domains, xi, xj, constraints[(xi, xj)])
16         if pruned:
17             print(f"  revise({xi},{xj}) pruned {pruned} -> {xi} in
18                   ↪ {domains[xi]}")
19             if not domains[xi]:
20                 return False # domain wipe-out
21             for (a, b) in constraints: # re-examine xi's other
22                 ↪ neighbours
23                 if b == xi and a != xj and (a, b) not in queue:
24                     queue.append((a, b))
25     return True
26
27 def arcs(edges, ok):
28     """both directed arcs per undirected edge, one shared predicate."""
29     return {(a, b): ok for (x, y) in edges for (a, b) in [(x, y), (y, x)]}
30
31 # --- csp 1: the chain x < y < z, all domains {1,2,3} -----
32 dom = {"x": [1, 2, 3], "y": [1, 2, 3], "z": [1, 2, 3]}
33 cons = {("x", "y"): lambda a, b: a < b, ("y", "x"): lambda a, b: b < a,
34         ("y", "z"): lambda a, b: a < b, ("z", "y"): lambda a, b: b < a}
35 print("chain csp, before:", dom)
36 print("consistent:", ac3(dom, cons))
37 print("chain csp, after: ", dom)
38
39 # --- csp 2: graph colouring, edges a-b, a-c, b-c, c-d -----
40 dom = {"a": ["r"], "b": ["r", "g"], "c": ["r", "g", "b"], "d": ["r", "g"]}
41 cons = arcs([(("a", "b"), ("a", "c"), ("b", "c"), ("c", "d"))],
42             lambda u, v: u != v)
43 print("\ncolouring csp, before:", dom)
44 print("consistent:", ac3(dom, cons))
45 print("colouring csp, after: ", dom)

```

chain csp, before: {'x': [1, 2, 3], 'y': [1, 2, 3], 'z': [1, 2, 3]}

```

  revise(x,y) pruned [3] -> x in [1, 2]
  revise(y,x) pruned [1] -> y in [2, 3]
  revise(y,z) pruned [3] -> y in [2]
  revise(z,y) pruned [1, 2] -> z in [3]
  revise(x,y) pruned [2] -> x in [1]

```

consistent: True

chain csp, after: {'x': [1], 'y': [2], 'z': [3]}

colouring csp, before: {'a': ['r'], 'b': ['r', 'g'], 'c': ['r', 'g', 'b'], 'd': ['r', 'g']}

```

  revise(b,a) pruned ['r'] -> b in ['g']
  revise(c,a) pruned ['r'] -> c in ['g', 'b']
  revise(c,b) pruned ['g'] -> c in ['b']

```

consistent: True

colouring csp, after: {'a': ['r'], 'b': ['g'], 'c': ['b'], 'd': ['r', 'g']}

the chain run reproduces the hand trace line for line — five revisions, (x, y) processed twice. the colouring run confirms the diagram: three prunings, then a fixpoint with d still holding two values. finishing the job takes search — chronological backtracking with MRV, checking each candidate value only against already-assigned neighbours (ac3 and arcs from the block above are assumed in scope):

```

1 def solve(domains, edges):
2     neighbours = {v: set() for v in domains}
3     for x, y in edges:
4         neighbours[x].add(y); neighbours[y].add(x)
5     stats = {"tried": 0, "deadends": 0}
6
7     def backtrack(assign):
8         if len(assign) == len(domains):
9             return dict(assign)
10        # mrv: unassigned variable with the fewest remaining values
11        var = min((v for v in domains if v not in assign),
12                  key=lambda v: (len(domains[v]), v))
13        for val in domains[var]:
14            stats["tried"] += 1
15            if all(assign.get(n) != val for n in neighbours[var]):
16                assign[var] = val
17                if (sol := backtrack(assign)) is not None:
18                    return sol
19                del assign[var]
20            stats["deadends"] += 1
21        return None
22
23    return backtrack({}), stats
24
25 edges = [("a", "b"), ("a", "c"), ("b", "c"), ("c", "d")]
26
27 raw = {"a": ["r"], "b": ["r", "g"], "c": ["r", "g", "b"], "d": ["r", "g"]}
28 print("raw domains:      ", solve(raw, edges))
29
30 pruned = {"a": ["r"], "b": ["g"], "c": ["b"], "d": ["r", "g"]} # post ac-3
31 print("after propagation:", solve(pruned, edges))

```

```

raw domains:      ({'a': 'r', 'b': 'g', 'd': 'r', 'c': 'b'}, {'tried': 7, 'deadends': 0})
after propagation: ({'a': 'r', 'b': 'g', 'c': 'b', 'd': 'r'}, {'tried': 4, 'deadends': 0})

```

same solution both ways, but propagation first means four value trials — one per variable, zero wasted — against seven on the raw domains. on a four-variable toy that saving is cute; on a thousand-task **cumulative** model it is the difference between milliseconds and heat death, because every value propagation deletes at the root would otherwise be refuted separately under exponentially many partial assignments. running AC-3 *inside* the search loop after every assignment is exactly the MAC strategy from the search section.

7 see also

- **integer programming** — the branch-and-bound rival; same trees, different inference
- **linear programming** — the relaxation engine that rival runs on
- **multi-objective programming** — when “the” objective is several
- **sudoku solver** — alldifferent earning its keep on a 9x9 grid
- **hashiwokakero** — bridges, degrees and connectivity as a CSP

references

Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford (2009). *Introduction to Algorithms*, MIT Press.

Epp, Susanna S. (2019). *Discrete Mathematics with Applications*, Cengage Learning.