

Hashiwokakero (Bridges) Solver

Aayush Bajaj

July 10, 2026

Contents

1	the puzzle	1
2	csp formulation	2
3	the solver	3
3.1	data structures	3
3.2	constraint propagation	4
3.3	backtracking search	4
3.4	complexity	5
4	the c to go rewrite, honestly	6
5	see also	7

hashiwokakero (“build bridges”, nikoli) hands you a grid of numbered islands and asks you to join them with bridges until every number is spent. it is the friendliest possible introduction to constraint satisfaction: the constraints are few and visual, propagation alone solves most human-published puzzles, and when it doesn’t, you get to write a backtracking search. this page documents my solver at [code/private/hashiwokakero](#) — a go rewrite of a uni assignment originally in c — including the debugging session that writing this page forced on it.

the repo’s own README opens with a life lesson: ‘whenever you port code, make sure it is correct before translating otherwise you will introduce a suicidal amount of bugs.’ foreshadowing.

1 the puzzle

- islands are cells holding a number 1..8 (classic rules); everything else is water.
- bridges run **horizontally or vertically** between two islands, over water only.
- at most **two** bridges may join the same pair of islands.
- bridges may not **cross** each other or pass through islands.
- every island’s number equals the count of bridge ends touching it.
- all islands must form one **connected** component.

a real run of the go solver on a 5x5 instance (output verbatim; " and = are double bridges, | and - singles):

```
$ ./hashi_solver -input demo5.txt      input:    3.5.3
3=5-3                                     .....
| " "                                     3.6.4
```

the unsw assignment variant allows up to three planks per pair and island values to 12 — written 1-9 then a, b, c. keep this in mind; it returns later as a plot twist.

3 6=4
 " "
 2 4=2

.....
 2.4.2

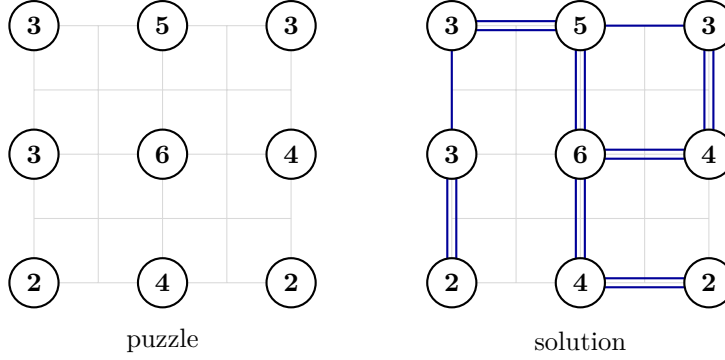


Figure 1: left: the 5×5 instance. right: the solver’s solution — double lines are double bridges. every island’s degree (counting doubles twice) equals its number, nothing crosses, and the graph is connected.

2 csp formulation

islands and their lines of sight form a graph before any bridge is placed: island u sees island v if they share a row or column with only water between them. those visible pairs are the puzzle’s **variables**:

- **variables**: one x_e per visible pair $e = (u, v)$.
- **domains**: $x_e \in \{0, 1, 2\}$ — no bridge, single, double.
- **constraints**:
 1. **degree**: for every island u , $\sum_{e \ni u} x_e = \text{value}(u)$.
 2. **no crossing**: if edges e and f pass through the same water cell (one horizontal, one vertical), then $x_e = 0 \vee x_f = 0$.
 3. **connectivity**: the subgraph $\{e : x_e \geq 1\}$ spans all islands in one component.

constraints 1 and 2 are local and propagate beautifully; constraint 3 is global, which is what stops hashi from being trivially decomposable — and what makes the general decision problem np-complete.¹ the search space is $3^{|E|}$ assignments in the worst case; the entire game of solving hashi well is never enumerating more than a sliver of it.

¹d. andersson (2009), *hashiwokakero is np-complete*, information processing letters 109(19):1145–1146. the reduction preserves planarity — hashi is hard even though its constraint graph is planar.

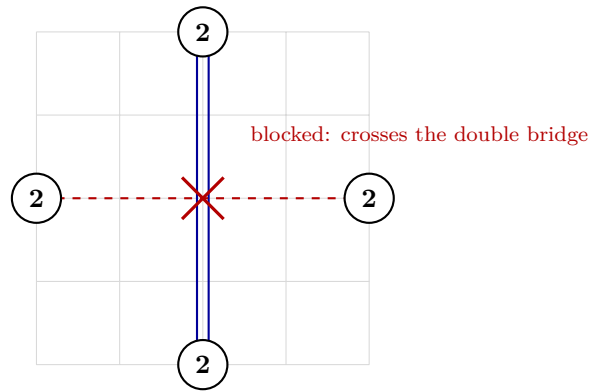


Figure 2: the crossing constraint in action: the top and bottom 2s see only each other, so propagation forces their double bridge (the sole-outlet rule) — which blocks the horizontal 2–2 edge through the shared centre cell. the side 2s are left with no outlet and the solver reports the contradiction.

real run of exactly that instance:

```
$ ./hashi_solver -input cross.txt          # ..2.. / ..... / 2...2 / ..... / ..2..
Error solving puzzle: logical error – node blocked in all directions
```

3 the solver

3.1 data structures

hashisolver/solver.go (1100+ lines) models the board as a dense grid of **Node** pointers — islands carry their value, water carries 0 (and later a negative bridge marker for printing). each island caches its four **line-of-sight neighbours** (computed once at parse), per-direction bridge counts, and per-direction **blocked** flags, which are the solver’s unit of propagation:

```
1 // from hashi/hashisolver/solver.go (fields grouped onto single lines, comments
  ↳ verbatim)
2 type Node struct {
3     Value      int
4     XPos, YPos int
5     UpBridges, DownBridges, LeftBridges, RightBridges int
6     TotalBridges int
7
8     // Neighbor nodes this one MAY connect to
9     UpNeighbor, DownNeighbor, LeftNeighbor, RightNeighbor *Node
10
11     // Blocked directions
12     UpBlocked, DownBlocked, LeftBlocked, RightBlocked bool
13     NumBlocked int
14
15     // Used when traversing nodes to check for potential islands
16     Visited bool
17 }
```

a direction gets blocked when: it has no neighbour; the neighbour is full; two bridges already run there; a perpendicular bridge now crosses the line of sight; or — a classic opening move — both endpoints are 1s (two 1-islands may never be joined in a puzzle with more than two islands, since that pair would form a closed component).

3.2 constraint propagation

`AttemptSpeculativeSolve()` first runs a **fixpoint loop**: sweep every unsatisfied island, apply every rule that fires, repeat until a full sweep changes nothing. the rules the code actually implements:

- **contradiction**: all four directions blocked but bridges still owed — fail this branch.
- **sole outlet**: three directions blocked — everything owed goes into the fourth (one bridge, then a second if exactly one is still owed and the direction can take it).
- **saturation**: if $\text{owed}(u) = \text{capacity}(u)$, every open direction must be filled to its cap. capacity here is `TotalPossibleMoves()`: 2 per line-of-sight *neighbour* (a blocked direction with no planks still counts — a looseness that only ever delays the rule), minus planks already laid, minus one for each direction whose single plank runs to an already-full neighbour. the classic “a 4 with two neighbours”, “an 8 with four” openings fall out of this rule.
- **one-everywhere**: if $\text{owed}(u) = \text{capacity}(u) - 1$, every unblocked direction must carry *at least one* bridge (the code adds the guaranteed first plank in each empty direction).
- **connectivity forcing**: for each unblocked direction d of u , tentatively block d and run a dfs over still-connectable edges (Sedgewick, Robert, 2001); if the island graph falls apart, edge d is a **cut edge** of the potential graph and must carry a bridge — build it. this is the global constraint 3 doing local work.
- two special cases for islands with exactly two open directions: an extra connectivity probe, and — when one direction’s neighbour can accept only one more plank but the island owes at least two — a forced plank in the *other* direction. that second rule re-checks its “owes at least two” premise before every placement; an earlier version evaluated it once, kept firing on the stale premise after its own first plank, and manufactured contradictions on solvable boards (more on this below).

a sharper per-direction accounting — capping directions whose neighbour is one plank from full — exists in `RemainingPossibleMoves()`, which nothing calls. dead code.

each `ConnectNodes()` call re-runs `BlockCheck()` on both endpoints — filling a node blocks it in all directions *and* blocks its neighbours’ views of it — so one placement can cascade through the whole board. this is forward checking in csp terms: domains (blocked flags, remaining capacities) shrink immediately, not lazily. the flip side of that eagerness: every rule that snapshots its open directions and then starts placing planks must re-check the live state before each placement, because its own moves (or their crossing side-effects) may have blocked a direction mid-loop.

3.3 backtracking search

when the fixpoint stalls short of a solution, the solver **speculates**:

```
1 // from hashi/hashisolver/solver.go (paraphrased: identifiers shortened, error
  ↪ handling condensed)
2 candidateNode := puzzle.FindCandidateNode() // max: 10*owed + (4 - open dirs)
3 for _, dir := range candidateNode.UnblockedNodes() {
4     // hypothesis 1: a single bridge this way
5     spec := puzzle.Clone()
6     ConnectNodes(spec, specNeighbor, dir, true)
7     if solved, err := AttemptSpeculativeSolve(spec, debug); err == nil &&
  ↪ solved.IsComplete() {
```

```

8      return solved, nil
9    }
10    // hypothesis 2: a double bridge (if both endpoints can take it)
11    ...
12    // hypothesis 3: no bridge at all this way
13    spec3 := puzzle.Clone()
14    spec3Node.DirectionBlocked(dir)
15    if solved, err := AttemptSpeculativeSolve(spec3, debug); err == nil &&
        ⇐ solved.IsComplete() {
16        return solved, nil
17    }
18  }
19  return puzzle, errors.New("no solution found with speculation")

```

this is depth-first search over the csp, with the propagation loop re-run inside every hypothetical world — the standard propagate-then-branch architecture (Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford, 2009). three honest observations:

- **state copying, not undo.** each hypothesis deep-clones the entire puzzle (`Clone()` rebuilds the node grid and re-wires neighbour pointers). a chess engine would make/unmake; this solver allocates. for hashi’s search depths (propagation does most of the work, so speculation nests shallowly) the garbage collector shrugs.
- **the branching heuristic is not mrv.** textbook csp says branch on the *most constrained* variable (minimum remaining values). `FindCandidateNode()` scores $10 \times \text{owed} + (4 - \text{open dirs})$ and takes the max — i.e., it prefers islands owing the *most* bridges, tie-broken toward blocked-ness. that is closer to “most constraining variable”: branching on a big island propagates hard in every child. defensible, but nobody benchmarked it against mrv.
- **domain splitting is per-direction, not per-edge-value.** the three hypotheses *cover* edge d ’s domain but overlap rather than partition it: the single-bridge branch does not cap the edge, so deeper recursion may still grow it to a double — that branch really explores $x_d \geq 1$, making hypothesis 2 redundant. completeness holds anyway, because $\{x_d \geq 1\} \cup \{x_d = 0\}$ is exhaustive and within each branch all other directions are revisited recursively.

3.4 complexity

worst case, backtracking over $|E|$ edges with domain 3 is $O(3^{|E|})$, and np-completeness says something exponential is unavoidable in general. in practice the fixpoint loop — each sweep visits n^2 cells, with the connectivity check costing a dfs per open direction on top — solves generated puzzles almost entirely by logic. measured on this machine (real runs, apple m-series; instances from a classic-rules port of bridgen’s backwards method, since the repo’s own generator only emits the variant — the repo’s `40.in` is one of those variant instances and is correctly rejected in milliseconds):

```

30/30 generated classic puzzles (5x5 .. 25x25) solved and
      verified by the repo's own bridgecheck: "Valid Solution."
40x40 classic instance: 0.022s total (including process startup)

```

4 the c to go rewrite, honestly

the commit history reads like a confession: `go port, 10x10 grid solved, ai embarasses me, cpp failure, repo in shambles` (head). the original uni submission was c; the readme admits it was “riddled with bugs”, and the repo’s `c_src/solver.c` is — poetically — an *empty file*. the go port was made from a third-party c++ implementation instead, and the port process (partly llm-assisted, hence `ai embarasses me`) introduced regressions of its own.

writing this page meant running `go test ./...`, and at head it failed — every size, including 3x3. debugging turned up two distinct storylines:

1. **a rules schism.** the test harness generates puzzles with the course’s `bridgen.c` and validates with `bridgecheck.c` — both implement the *unsw variant* (up to three planks per pair, island values to 12). the go solver implements *classic hashi* (two planks, values to 9; its parser maps `a`, `b`, `c` to water!). the suite was structurally incapable of passing reliably: a classic-rules solver being marked against variant-rules generators. the repo’s own sample files need triple bridges too — `puzzle_5x5.txt` has a 6 whose two visible neighbours cap it at 4, and `s56.in`’s published solution contains `E` triples (being non-square and space-padded, `s56.in` also used to *nil-pointer panic* the parser; it is now rejected with a clear error).
2. **three genuine bugs**, fixed in the first debugging pass:
 - `IsComplete()` checked connectivity by walking *unblocked* directions — but every island in a finished puzzle is fully blocked (that is what “finished” means to the propagator), so completion could never be confirmed and the solver speculated itself into “no candidate node” errors on already-solved boards. fix: the dfs now also traverses edges that carry bridges.
 - `CheckForIsland()`’s disconnection test called a dfs helper that *always returns true* — the connectivity-forcing rule was dead code. fix: dfs from the tested node’s component, then scan for unvisited islands.
 - nothing implemented the **crossing constraint**: laying a bridge never blocked the perpendicular lines of sight it cut. fix: `blockCrossings()` blocks the nearest visible island pair across every cell a new plank occupies. (bonus: the neighbour scan also treated *adjacent* islands as connectable — a zero-length “bridge” that the printer cannot even draw; `bridgecheck` caught that one on a 20x20.)

a second pass — cross-checking the solver against a brute-force reference on batches of generated puzzles — found three more, subtler ones:

- the **1–1 opening rule** was applied unconditionally, so a two-island **1.1** board (which `bridgen` happily generates, and whose unique solution *is* the 1–1 bridge) was declared contradictory. the rule now fires only when the puzzle has more than two islands.
- the **two-open-directions rule** evaluated its “owes ≥ 2 ” premise once, then kept forcing planks after its own first placement had already dropped the debt to 1 — an unsound deduction that surfaced as a false contradiction on a solvable generated 10x10. (its sibling rule also forced a plank in the other direction that was never actually implied; that placement is gone.)

- several rules iterated a *snapshot* of their open directions while placing planks, so a direction blocked mid-loop (by a crossing, or a neighbour filling up) could still receive bridges — in the worst case overfilling an edge past two planks and letting the solver “solve” impossible boards (a 6.6 / 5.5 grid came back as 6=6 / 5=5, which **bridgecheck** flatly rejects: “Bridges not Adding Up”). every deferred placement now re-checks the live blocked state and the two-plank cap, and **IsComplete()** refuses any board that ever exceeded it.

after all of that, the solver solves every classic-rules instance thrown at it — the 30/30 batch above, plus random-batch agreement with the brute-force reference — while the bridgen-based suite remains a coin flip by design: **bridgen** seeds from **time(NULL)** and deals a fresh random *variant* puzzle every run, so **TestSolverWithKnownPuzzles** passes only when the 3x3 draw happens to be classic-compatible (measured: 4 of 10 runs), and **TestSolverWithBridgen** fails at *any* size whenever the draw needs triple planks or values above 9 — not just at 8x8 and above. the honest ledger: the port is now a correct *classic* hashi solver with a nondeterministic test harness from a *different game*. the readme’s life lesson stands.

a brief word on the generator, since the tests lean on it: **bridgen.c** works *backwards* — it builds a random valid bridge layout on an empty grid (random endpoints, random plank counts, rejecting crossings), then erases the bridges and prints only the island degrees. every generated puzzle is solvable by construction, though not necessarily uniquely. **bridgecheck.c** is the referee: it re-parses a proposed solution grid and checks degree sums, plank legality and connectedness.

bridgen.c also lives on in code/private/arcade/references/hashiwokakero/ — the same generator that seeds the arcade’s puzzle pages.

5 see also

- **sudoku** — the same propagate-then-backtrack architecture, plus an integer-programming formulation
- **constraint satisfaction problems** — parent section
- **peg solitaire** — dfs over game states rather than assignments
- **classical algorithms** — dfs and connectivity, the two primitives this solver leans on

references

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

Sedgewick, Robert (2001). *Algorithms in C, Part 5: Graph Algorithms*, Addison-Wesley.