Logic-Based Binairo Solver Using Constraint Programming in Prolog

Max Dyer and Wynona McWhirter
Rensselaer Polytechnic Institute

December 11, 2025

1 Introduction

Binairo is a logic puzzle in which a board can be filled in with two kinds of item, often 1s and 0s. The positions of these values are found using pre-filled hint values and logical constraints. This project implements a solver for Binairo using Prolog's `clpfd` library. The purpose of this project is to explore how constraint programming can solve deterministic logic puzzles without brute-force search.

2 Problem Description

Binairo is a logic puzzle played on an even-sized grid of 0s and 1s. The rules of the puzzle are as follows:

1. Each row and column must contain an equal number of 0s and 1s
2. No row or column can contain three 0s or 1s in a row
3. No two rows or two columns may be identical

A puzzle starts with some pre-filled cells, and the task is to infer the remaining numbers using logic.

We use Prolog constraint programming to fill in the board. Since Binairo is a logic puzzle, its rules map naturally to arithmetic constraints.

Our solver is designed to handle any even square boards with a few filled in values. If a board is ambiguous in whether a 0 or 1 should be placed in certain spots, our solver will search for multiple solutions. Two puzzles have been given as examples but any board fitting the above requirements will work. If no solution exists, our solver will return false.

3 Approach and Methods

The solver expresses Binairo rules as finite-domain constraints over lists of Prolog variables. The board is represented as a list of lists (Rows), where each row is a list of 0s, 1s, or _s for unfilled values. CLP(FD) allows for unfilled values to be assigned across a defined domain, expanding Prolog's base capabilities of logical evaluation to logical application. This means the program can set unfilled values to whatever values satisfy the specified constraints, which results in solving behaviour. Each rule maps to a constraint.

3.1 Representation

The puzzle is represented as

- **Rows**: a list of integers and variables (1, 0, or _)
- **Cols**: transposed version of **Rows** using **transpose/2**
- **Variables**: unfilled cells represented as _

The domain for all variables is constricted to {0, 1}

3.2 Domain Constraint

The first step of the solver is:

```
append(Rows, Vars), Vars ins 0..1.
```

This makes each entry of Rows into a single variable list and constrains them to be either 0 or 1.

3.3 No-Triplets Constraint

The `no_triplets/1` predicate enforces that no three consecutive values in a row or column are all 0s or all 1s. This is implemented by ensuring the sum of three consecutive numbers is not 000 or 111:

```
A + B + C #\= 0
A + B + C #\= 3
```

By summing the three values, the solver can detect and avoid rule-breaking patterns easily.

3.4 Equal Numbers of 0s and 1s Constraint

To enforce equal numbers in a list:

```
sum(List, #=, Ones),
length(List, L),
Ones #= L // 2.
```

Since the board dimension is always even, each row and column must contain 1s equal to exactly half the length. By default, this results in the remaining half of values being assigned to 0s, resulting in equality.

3.5 Uniqueness Constraint

To ensure no two rows or columns are identical, the solver compares every row/column to every subsequent row/column. For each row column pair, the following is checked:

```
lists_not_equal([], []) :- false.
lists_not_equal([A | As], [A | Bs]) :- lists_not_equal(As, Bs).
lists_not_equal([A | _], [B | _]) :- A #\= B.
```

Each element of both rows are recursively compared. If a match is found, the comparison continues. If either a mismatch or an unfilled value/value pair is found, it is confirmed or enforced that the values are different. If the entirety of both lists are compared and no differences are found, the lists are identical and the constraint fails.

3.6 Labeling Strategy

After all constraints are established, the solver calls:

```
labeling([ffc, bisect], Vars).
```

The **ffc** (first-fail) selects the most constrained variable first to maximize early searching. The **bisect** splits domains to avoid deep search. These search heuristics are used to improve performance on large boards.

4 Implementation

The implementation includes the puzzle input, constraint setup, and solution labeling.

4.1 Top-Level Predicate

The user calls the predicate:

```
binairo(Rows, Dimension)
```

This calls the solver, applies the constraints, and prints the final solved puzzle.

4.2 Solver Logic

The main solving predicate is:

```
binairo_solver(Rows, Dimension)
```

This predicate does domain assignment, dimension checks, row-length validation, grid transposition, the application of all puzzle constraints, and calls labeling to assign variable values.

4.3 Constraint Predicates

Several helper predicates implement the puzzle rules:

- `same_length/2`: ensures consistent row length
- `no_triplets/1`: enforces forbidden patterns (triplets)
- `limit_half/1`: ensures each row/column contains half 1s
- `lists_not_equal/2`: enforces row/column uniqueness
- `unique_list_of_lists/1`: applies uniqueness to elements in list (pairwise)

4.4 Testing

The predicate:

```
binairo_demo(Puzzle Number)
```

loads the given puzzle and calls the solver. Two puzzles are included under `puzzle/2`.

5 Results

After development and optimization, the program can solve any Binairo puzzle commonly played by humans (from 4x4 to 12x12) near instantly. For puzzles with multiple possible solutions, the solver can identify all valid solutions.

6 Discussion

Compared to other solutions such as brute force, our method of constraint programming is far more efficient as it retains high solving speeds at larger grid sizes. We chose to work with CLP(FD), Prolog's Finite Domain constraints, over CLP(B), Prolog's Binary constraints, as the equal number of 1s and 0s constraints proved not fit cleanly into binary constraints. Prolog proved as the best option to develop in because the constraint programming options match the human solving patterns of recursively applying logic rules. This was compared to ShadowProver and Spectra, which have support for intensional and modal logics. Because Binairo is a game with complete information, tools handling logics of belief would have resulted in greater unneeded overhead and development difficulties. Overall, Prolog served to be highly effective for solving Binairo, as evidenced by the ease of matching strategy to the programming structure and speed of the solver.

7 Limitations and Future Work

Potential extensions include:

- Puzzle generator
- Graphic visualization
- Implement Binairo+ rules (inclusion of = and × constraints)

8 Conclusion

In conclusion, the Binairo solver successfully and efficiently models and solves the puzzle using CLP(FD) constraints in Prolog.

9 References

For Binairo rules:

- Wikipedia, "Takuzu" `https://en.wikipedia.org/wiki/Takuzu`

For learning constraint programming:

- Triska, Markus. "Constraint Logic Programming over Integers in Prolog (CLP(Z))." `https://www.metalevel.at/Prolog/clpz`.

- SWI-Prolog Development Team. "CLP(FD) Arithmetic Constraints Documentation." `https://www.swi-Prolog.org/pldoc/man?section=clpfd-arith-constraints`.