

Spatially Balanced Latin Square Project

Decision Intelligence

Student Name: Eya Ben Moulehem

Academic Year: 2025–2026

1 Constraints Used

1.1 Decision Variables (The Grid)

Constraint: For each plot (i, j) in the $n \times n$ field:

$$X_{i,j} \in \{1, \dots, n\}$$

Each cell of the field contains exactly one fertilizer type, represented by an integer value between 1 and n .

1.2 Row and Column Uniqueness (Latin Square)

Constraint: For every row i and column j :

$$\text{AllDifferent}(X_{i,1}, \dots, X_{i,n})$$

$$\text{AllDifferent}(X_{1,j}, \dots, X_{n,j})$$

These constraints enforce the Latin Square property, ensuring that each fertilizer appears exactly once in every row and every column.

1.3 Dual Variables and Channeling

Instead of using reified variables, the model introduces auxiliary coordinate variables to efficiently locate the positions of each fertilizer.

Definition:

- $R_{c,k}$: row index of the k -th occurrence of color c
- $C_{c,k}$: column index of the k -th occurrence of color c

Channeling Constraint: For every color c and occurrence index k :

$$X_{R_{c,k}, C_{c,k}} = c$$

This constraint links the grid variables to their corresponding coordinate representations, allowing the solver to directly retrieve the position of any fertilizer without scanning the entire grid. This significantly improves the efficiency of distance computations.

1.3.1 Occurrence Ordering Constraint

Constraint: For each color c and occurrence index $k > 1$:

$$n \cdot R_{c,k-1} + C_{c,k-1} < n \cdot R_{c,k} + C_{c,k}$$

This enforces a strict ordering of occurrences in row-major order, preventing duplicate positions and eliminating symmetric permutations of occurrences.

1.4 Distance Definition

Definition: The Manhattan distance between two plots is defined as:

$$d((r_1, c_1), (r_2, c_2)) = |r_1 - r_2| + |c_1 - c_2|$$

1.5 Pairwise Distance Sum (Spatial Balance Core)

Constraint: For every pair of distinct fertilizers $a < b$, the sum of all pairwise Manhattan distances between their occurrences is constrained to equal a shared constant K :

$$\sum_{k_a=1}^n \sum_{k_b=1}^n d((R_{a,k_a}, C_{a,k_a}), (R_{b,k_b}, C_{b,k_b})) = K$$

The total spatial interaction between two fertilizers is computed by summing the Manhattan distances between every occurrence of color a and every occurrence of color b , using the dual coordinate variables directly.

1.6 Symmetry Breaking

To reduce redundant symmetric solutions, the following constraints are applied.

Constraints:

- **First Row Fixed:**

$$X_{1,j} = j \quad \forall j \in \{1, \dots, n\}$$

- **First Column Ordering (for $n > 2$):**

$$X_{1,2} < X_{2,1}$$

Fixing the first row removes symmetries due to permutations of fertilizer labels. The ordering constraint further eliminates reflection and transpose symmetries, significantly reducing the search space.

2 Filtering Algorithms for SBLS Constraints

2.1 Row & Column Uniqueness (AllDifferent)

The "AC" string explicitly requests Arc Consistency (Regin's Algo)

```

model.allDifferent(grid[i], "AC").post();
model.allDifferent(colVars, "AC").post();

```

Since the code specifies "AC", it enforces **Generalized Arc Consistency (GAC)** using Régin's algorithm, which detects Hall sets and prunes values that cannot participate in a valid matching.

2.2 2. The "Bridge" Constraints (Channeling)

The 'element' constraint implements the channeling logic

```
model.element(model.intVar(c), flatGrid, index, 0).post();
```

Choco's element constraint propagator enforces **Bounds Consistency (BC)** between the index variable and the array. While not full arc consistency, this ensures that domain bounds of the value and index variables remain consistent, which is sufficient for the channeling logic in the SBLS model.

2.3 Distance Calculations (Absolute Difference)

```
'absolute' creates a propagator enforcing |X| = Y
model.absolute(rDiff, rDelta).post();
```

This is efficient and sufficient for propagating Manhattan distances in the SBLS problem.

2.4 The "Sum of Distances" (Global Sum)

```
'sum' creates a linear propagator for Sum(vars) = K
model.sum(distances, "=", K).post();
```

The **sum** constraint enforces **Bounds Consistency (BC)**: maintains the range $[\sum \min, \sum \max]$ for all distance variables. If K falls outside this range, the solver prunes the search branch immediately. This makes sum propagation both fast and effective.

2.5 Symmetry Breaking (Arithmetic)

Simple arithmetic constraints

```
model.arithm(grid[0][j], "=", j + 1).post();
model.arithm(grid[0][1], "<", grid[1][0]).post();
```

BC is sufficient and effectively equivalent to arc consistency, as the domains are pruned according to the minimum and maximum values of the involved variables.

3 Variable and Value Selection Strategies

The “Smart SBLS” solver utilizes a **Chained Search Strategy**. Rather than applying a single heuristic for all variables, it splits the problem into two phases:

- **Phase 1: Solving the Core (Grid)** — Focuses on the main $N \times N$ Latin Square.
- **Phase 2: Verifying the Details (Coordinates & K)** — Instantiates auxiliary variables to complete the constraints.

The strategies are defined in this Choco block:

```
solver.setSearch(  
    Search.domOverWDegSearch(flatGrid),  
    Search.inputOrderLBSearch(ArrayUtils.flatten(rowsOfColor)),  
    Search.inputOrderLBSearch(ArrayUtils.flatten(colsOfColor)),  
    Search.inputOrderLBSearch(K)  
) ;
```

Phase 1: The “Smart” Strategy (Grid Variables)

The main $N \times N$ grid where the Latin Square is constructed.

Variable Strategy: Domain Over Weighted Degree (DomOverWDeg)

- **Principle:** “Conflict Learning” prioritizes the most problematic cells first.
- **Score Calculation:**
$$\text{Score}(X) = \frac{\text{Size of Domain}(X)}{\text{Weighted Degree}(X)}$$
 - **Numerator (Domain Size):** Cells with fewer options are prioritized (Fail-First principle).
 - **Denominator (Weighted Degree):** Each constraint that caused previous failures increases its weight, making “hard” cells more likely to be selected.
- **Decision:** Pick the variable with the smallest score.
- **Effect:** Cells involved in difficult constraints are tackled first, improving solver efficiency.

Value Strategy: Minimum Domain Value (IntDomainMin)

- **Rule:** Try values in ascending order: 1, 2, 3, ...
- **Rationale:** In Latin Squares, the choice of specific numbers matters less than placement; a simple ascending order is sufficient.

Phase 2: The “Fast” Strategy (Auxiliary Variables)

Variable Strategy: Input Order

- **Rule:** Pick variables strictly in the order they appear in the array (Index 0, Index 1, Index 2, ...)
- **Rationale:** After Phase 1, the Grid is complete and the coordinates are effectively determined. No sophisticated strategy is needed; the solver just “instantiates” remaining variables.

Value Strategy: Lower Bound (LB)

- **Rule:** Assign the variable to its current Lower Bound (smallest value in its domain)
- **Effect:** Since most coordinate variables will have singleton domains (e.g., [3, 3]), assigning the lower bound effectively locks them in, completing the constraints quickly.

4 Definition of the Simple Method (Baseline)

To evaluate the performance gain provided by the Constraint Programming (CP) model, we implemented a baseline approach referred to as the “**Simple Method**”. This implementation follows a classic **Generate-and-Test** paradigm.

Unlike the “Smart” method, which actively uses spatial constraints to prune the search tree during construction, the Simple Method decouples the problem into two distinct phases: **Generation** and **Verification**.

Phase 1: Generation (The “Blind” Solver)

In this phase, the CP solver is configured with a relaxed model that contains only the fundamental requirements of a Latin Square.

- **Variables:** The model creates the $N \times N$ integer grid.
- **Constraints Applied:**

- **Row Uniqueness:** `model.allDifferent(rows, "AC")`
- **Column Uniqueness:** `model.allDifferent(cols, "AC")`
- **Basic Symmetry Breaking:** Fixing the first row ($1..N$) and enforcing order on the first column to avoid trivial permutations.
- **Constraints Omitted:** This model contains zero constraints related to **distance, coordinates, or spatial balance (K)**.

The solver iterates through the search space, finding valid Latin Squares one by one. To the solver, an unbalanced square and a balanced square look identical.

Phase 2: Verification (The Post-Check)

Once the solver finds a valid Latin Square (a complete grid instantiation), the execution pauses, and the solution is passed to a manual Java function:

```
checkSpatialBalance(grid, n)
```

This function performs a brute-force verification:

- **Iterate Pairs:** Loop through every unique pair of colors (c_1, c_2).
- **Calculate Distance:** Scan the entire grid to find the coordinates of these colors and calculate the sum of Manhattan distances between them.
- **Check Consistency:** The first pair establishes a **Target K**. If any subsequent pair has a total distance different from the Target K, the grid is immediately rejected.

Algorithm Logic

The process can be summarized by the following pseudocode:

```
While (Solver finds next Latin Square):
    1. Extract the grid.
    2. Calculate distances manually in Java.
    3. IF (All pairwise distances are equal):
        RETURN Success (Solution Found)
    ELSE:
        CONTINUE (Force solver to backtrack and find the next square)
```

5 Experimental Results: Smart CP vs. Simple Generate-and-Test

To quantify the efficiency of our Constraint Programming model, we performed a direct comparison between two approaches:

- **Smart Method (CP):** The proposed model using DomOverWDeg variable ordering, dual-variable channeling, and active spatial constraints.
- **Simple Method (Baseline):** A naive approach that generates valid Latin Squares using lexicographical search and checks the spatial balance (K) only after the grid is fully filled.

1. Performance Data

The following table summarizes the solving time (in seconds) and search space exploration (Nodes/Checks) for orders $N = 2$ to $N = 29$.

1. Performance Data

The following table summarizes the solving time (in seconds) and search space exploration (Nodes/Checks) for orders $N = 2$ to $N = 29$.

N	Smart M (T)	Nodes	Simple M (T)	Simple Status
2	0.004s	2	0.003s	OK (1 Check) - Tie (Simple faster due to overhead)
3	0.005s	6	0.004s	OK (1 Check) - Simple Wins (Lucky first guess)
4	0.012s	14	0.001s	OK (1 Check) - Simple Wins
5	0.022s	26	0.001s	OK (1 Check) - Simple Wins
6	0.033s	42	0.002s	OK (1 Check) - Simple Wins
7	0.049s	61	> 10.0s	FAIL - Smart Wins (Simple method collapses)
8	0.086s	86	-	FAIL - Smart Only
9	0.100s	114	-	FAIL - Smart Only
10	0.150s	142	-	FAIL - Smart Only
11	0.214s	174	-	FAIL - Smart Only
12	0.291s	214	-	FAIL - Smart Only
13	0.410s	259	-	FAIL - Smart Only
14	0.708s	304	-	FAIL - Smart Only
15	1.209s	355	-	FAIL - Smart Only
16	1.230s	406	-	FAIL - Smart Only
17	1.819s	468	-	FAIL - Smart Only
18	3.145s	530	-	FAIL - Smart Only
19	3.904s	595	-	FAIL - Smart Only
20	10.11s	671	-	FAIL - Smart Only
21	18.09s	734	-	FAIL - Smart Only
22	7.96s	817	-	FAIL - Smart Only
23	9.85s	905	-	FAIL - Smart Only
24	12.91s	992	-	FAIL - Smart Only
25	17.76s	1069	-	FAIL - Smart Only
26	23.44s	1169	-	FAIL - Smart Only
27	54.56s	1268	-	FAIL - Smart Only
28	53.56s	1389	-	FAIL - Smart Only
29	54.28s	1482	-	FAIL - Smart Only

2. Analysis of Results

The benchmark reveals two distinct behavioral phases:

Phase 1: The Trivial Zone ($N \leq 6$)

- The Simple Method appears superior in this range, solving problems in under 2 milliseconds, whereas the Smart Method takes \sim 20-30 ms.
- **Reason:** For small N , the search space is tiny. The lexicographically first Latin

Square generated (often a cyclic shift square) happens to be spatially balanced due to inherent symmetry.

- **Overhead:** The Smart Method builds a complex constraint graph and initializes conflict histories (`DomOverWDeg`), which exceeds the solving time for trivial problems.

Phase 2: The Combinatorial Cliff ($N \geq 7$)

- **Failure of Simple Method:** At $N = 7$, there are approximately 16.9 million Latin Squares. Simple symmetries break down, causing the Simple Method to hit the time limit without finding a balanced square.
- **Success of Smart Method:** The CP model efficiently solves the problem without enumerating millions of squares. At $N = 7$, it explores only 61 nodes and finds the solution in 0.049 seconds.

Phase 3: Scalability ($N = 8$ to $N = 29$)

- **Linear Node Growth:** As N increases, the number of nodes required grows almost linearly (from 86 nodes at $N = 8$ to 1482 nodes at $N = 29$), indicating that the channeling constraints and `DomOverWDeg` heuristics guide the solver efficiently.
- **Time Growth:** The solving time increases due to the propagation cost of distance calculations on large grids (e.g., 29×29 with 841 cells and thousands of distance pairs), not due to backtracking inefficiency.

3. Conclusion

The comparison conclusively demonstrates that Constraint Programming is essential for the SBLS problem. The Simple Method is strictly limited to trivial instances ($N \leq 6$), whereas the Smart CP model is robust and solves complex instances up to $N = 29$, where the search space size exceeds the number of atoms in the universe.

Machine Information

1. Hardware Specifications

- **CPU (Processor):** 12th Gen Intel i5-12450H
- **RAM (Memory):** 16 GB DDR4

2. Software Specifications

- **Operating System (OS):** Windows 11 Home (64-bit)
- **Java Version:** java version 1.8.0_471
- **Choco-Solver:** 5.0.0-beta.1