Balancing Latin Rectangles with LLM-Generated Streamliners

Florentina Voboril
Algorithms and Complexity Group, TU Wien, Austria

Vaidyanathan Peruvemba Ramaswamy
Algorithms and Complexity Group, TU Wien, Austria

Stefan Szeider
Algorithms and Complexity Group, TU Wien, Austria

Abstract

We present an integration of Large Language Models (LLMs) with streamlining techniques to find well-balanced Latin rectangles. Our approach combines LLM-generated streamlining constraints that effectively partition the search space, directing constraint solvers toward structured subspaces containing high-quality solutions. Our methodology extends LLM-generated streamliners, as Voboril et al. (2024) introduced for decision problems, to the optimization context through techniques that incrementally refine the objective function value.

We propose two complementary strategies to orchestrate sets of streamliners: an incremental mechanism that utilizes improving solutions to initialize subsequent search processes, and an evolutionary framework that maintains and refines effective streamliner populations. Our experiments demonstrate that our approach successfully reduces established minimum imbalance values for partially spatially balanced Latin rectangles across multiple problem dimensions. The results validate the efficacy of combining LLMs with constraint programming methodologies for tackling problems characterized by complex global constraints.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Mathematics of computing → Combinatorial optimization; Mathematics of computing → Combinatorial algorithms; Information systems → Language models

Keywords and phrases Balanced Latin Rectangles, Streamliners, Large Language Models, Warm-starts, Evolutionary Search

Digital Object Identifier 10.4230/LIPIcs.CP.2025.36

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.15616074` [19]

Funding Austrian Science Fund (FWF) projects 10.55776/COE12 and 10.55776/P36420.

1 Introduction

Latin rectangles are combinatorial structures consisting of k rows and n columns filled with n symbols, where each symbol appears exactly once in each row and column. When these structures are balanced – meaning the distance between any pair of symbols over all rows is minimized – they become particularly valuable for experimental design, especially in agricultural field trials. *Spatially balanced Latin rectangles (BLRs)* help minimize bias due to spatial correlation in experimental plots, leading to more accurate statistical analyzes and reliable results across fields such as agriculture, drug testing, and psychology [2, 17, 6, 16].

Finding optimally balanced Latin rectangles presents a significant computational challenge. The imbalance of a Latin rectangle is measured as the sum of absolute differences between actual and ideal distances between all pairs of symbols. For many combinations of number of rows and columns, determining whether a given imbalance value is optimal remains an open question. Previous work by Díaz et al. [2] established upper bounds for BLRs of various

sizes, with provably optimal solutions known only for specific dimensions. Despite advances using constraint programming, mixed integer programming, and local search methods, the computational complexity has limited progress beyond rectangles of moderate size [9, 10].

The technique of *streamlining* – adding constraints that focus the search on promising regions of the solution space – was initially introduced by Gomes and Sellmann [3] for related combinatorial design problems, including spatially balanced Latin squares. Streamlining constraints partition the search space, guiding the solver toward structured subspaces likely to contain high-quality solutions. While effective, the manual design of streamliners requires domain expertise and experimentation [7].

In this paper, we push the boundaries of balanced Latin rectangles by combining established streamlining techniques with the novel approach of generating streamliners using *Large Language Models (LLMs)*. We build upon the recent work by Voboril et al. [18], who introduced StreamLLM, a method for using LLMs for generating streamliners for decision problems, and adapt this approach to optimization problems. Our method improves most of the best-known bounds for BLRs in the range of $k \in [3, 12]$ and $n \in [9, 13]$, demonstrating the effectiveness of this hybrid approach.

Our contribution is twofold. First, we extend StreamLLM to address *optimization problems* instead of decision problems, introducing techniques to guide the search toward solutions with better objective values. Second, we present two complementary strategies for orchestrating the generation of a set of streamliners that together help to obtain Latin rectangles with lower imbalance than any known before of the same dimension.

The first strategy is the *incremental warmstart* approach that uses improved solutions as starting points for subsequent searches. The second strategy is an *evolutionary* approach that maintains a population of effective streamliners, combining them to generate increasingly powerful constraints.

Our experimental results show that our method outperforms previous approaches, improving upper bounds on imbalance for 32 out of 44 instances. Table 1 summarizes our results compared to previously known bounds, highlighting the cases where we establish new record values. Our implementation draws on five different LLM models and employs various prompting strategies to generate diverse and effective streamliners, enabling a thorough exploration of the solution space.

Table 1 Final results. The gray cells are already proven optimal and hence excluded from our experiments. The purple cells are where our approach finds a BLR with strictly better imbalance. Of the remaining white cells, the cells with bold text are where our approach matches the previously known imbalance bound.

$n\backslash k$	2	3	4	5	6	7	8	9	10	11	12
8	40.0	36.0	32.0	30.0	**24.0**	**28.0**	0.0				
9	65.3	**56.0**	**56.0**	52.6	**48.0**	51.3	53.3	0.0			
10	92.0	80.0	78.6	66.6	70.0	72.0	82.6	70.0	40.0		
11	124.0	114.0	112.0	116.0	108.0	118.0	114.0	120.0	110.0	0.0	
12	168.0	154.0	154.6	145.3	**120.0**	164.0	148.0	154.0	**174.6**	147.3	0.0
13	218.6	198.0	211.3	210.6	204.0	203.3	220.6	214.0	240.6	224.0	274.0

2 Preliminaries

2.1 Balanced Latin Rectangles

In this section, we define the general notation and background related to balanced Latin rectangles. A *Latin rectangle* is an $k \times n$ grid where $k \leq n$ and each cell contains a symbol from 1 to n, such that no symbol appears more than once within a row or within a column. *Latin squares* (or Magic squares) are Latin rectangles where $n = k$. Latin squares are widely known and have been studied for centuries.

Díaz et al. [2] introduced the notion of balance in Latin rectangles, minimizing which helps preclude spatial-correlation artifacts and ensures statistical fairness while devising experiments. A *spatially balanced Latin rectangle* (or simply *balanced Latin rectangle*) is a Latin rectangle where the total distance between any pair of symbols is the same. Formally, $\text{dist}_i(r, s)$ denotes the distance between symbols r and s in the ith row. Note that $0 < \text{dist}_i(r, s) < n$. The distance between two symbols is defined as $\text{dist}(r, s) = \sum_{1 \leq i \leq k} \text{dist}_i(r, s)$. A Latin rectangle is *balanced* iff $\text{dist}(r, s)$ is the same for every pair $1 \leq r, s \leq n$. Letting $\text{Z}_{n,k}$ be $k(n+1)/3$, it is easy to see that in a balanced Latin rectangle, $\text{dist}(r, s) = \text{Z}_{n,k}$ for every pair.

However, balanced Latin rectangles only exist when $n = k$ and $n \not\equiv 1 \pmod 3$ [2], which rules out several combinations of n and k. Consequently, Díaz et al. [2] defined the imbalance of a Latin rectangle as a measure of how far it is from being balanced. The imbalance between a pair of symbols r, s is defined as $\mathbb{I}(r, s) = |\text{dist}(r, s) - \text{Z}_{n,k}|$ and the imbalance of a Latin rectangle L is defined as $\mathbb{I}(L) = \sum_{1 \leq r < s \leq n} \mathbb{I}(r, s)$. In the BLR problem, we are given two integers $k \leq n$, and the goal is to find a $k \times n$ Latin rectangle L such that $\mathbb{I}(L)$ is minimized. Note that $\text{Z}_{n,k}$ is always a rational number of the form $p/3$ and if either k or $n + 1$ are divisible by 3, then $\text{Z}_{n,k}$ is an integer. The same holds for $\mathbb{I}(L)$; hence, we always denote the imbalance values as x.0, x.3 or x.6, respectively, to represent x, $\text{x} + \frac{1}{3}$ or $\text{x} + \frac{2}{3}$ where $\text{x} \in \mathbb{Z}$.

1	2	3	4
2	4	1	3
3	1	4	2

Figure 1 Example of a 3×4 optimally balanced Latin rectangle L with imbalance 4. For the pair $(2, 3)$, the distance $\text{dist}(2, 3)$ is the sum of distances over each row, i.e., $1 + 3 + 3 = 7$, and the imbalance $\mathbb{I}(2, 3) = |\text{dist}(2, 3) - k(n + 1)/3| = 2$. By the same method, $\mathbb{I}(1, 3) = \mathbb{I}(2, 4) = 1$ and $\mathbb{I}(1, 2) = \mathbb{I}(1, 4) = \mathbb{I}(2, 4) = 0$. Finally, the imbalance of the Latin rectangle $\mathbb{I}(L)$ is the sum of all the pair-wise imbalances, which is 4.

2.2 LLM-generated Streamliners

Streamlining constraints or *streamliners* are constraints added to a constraint programming model in order to speed up the solving process by pruning the search space and guiding the solver towards more promising subspaces of the solution space. Streamliners have provided significant speedups across numerous problem domains [1, 2, 3, 4, 5, 7, 8, 11, 14].

By definition, they are not required to be *sound*, i.e., they need not preserve the set of feasible solutions and are allowed to remove some or even all of the solutions. Streamliners are a generalization of the following well-known constraints:

- *implied* or *redundant* constraints which do not alter the solution space,

- *symmetry-breaking* constraints which eliminate all but one solution from symmetric equivalence classes
- *dominance-breaking* constraints which eliminate potentially suboptimal solutions such that at least one of the optimal solution remains.

In our work, we use the term "streamliner" in a broader sense to refer to any constraint that can speed up the solving of a constraint programming model. Consequently, the streamliners that we present include but are not limited to the three types listed above.

Formerly, streamliners were generated manually for each problem which was labor-intensive and hard to scale, which motivated research into automating streamliner generation [12, 13, 15, 20]. Typically, these methods involved crafting combinations of atomic constraints which restrict the variable domains and testing them on a large pool of benchmark instances. This process was extremely resource-intensive, taking several days to come up with promising streamliners for a given problem.

Looking for a more scalable and efficient way to come up with high-quality streamliners, we turn our attention to *Large Language Models* or *LLMs*. LLMs are transformer architectures with billions of parameters that have been trained on large datasets and can produce human-like text and code. In recent times, LLMs have seen significant growth in innovation and use. This widespread use is a testament to the versatility and broad applicability of LLMs, such as, chatbots, translators, and coding assistants. Recent advancements have endowed LLMs with the power of reasoning and problem-solving which further enables them to manipulate mathematical expressions and assist in devising proofs. However, LLMs are not infallible and can produce incorrect yet believable output. As a result, independent verification is crucial to harness the power of LLMs.

In this work, we extend the StreamLLM approach developed by Voboril et al. [18]. In contrast to their work, we target optimization problems instead of decision problems, and we use more sophisticated procedures to traverse the space of solutions using the objective values as guidance. Please refer to Listing 1 for the MiniZinc model of the BLR problem that we use to test the performance of the different approaches and also provide to the LLM as context for generating streamlining constraints. Note that in the MiniZinc model: `pos[row,symbol]` denotes the position of a symbol in a row, `rect[row,col]` denotes the symbol at a specific cell of the Latin rectangle, `dist[r,s]` denotes $\mathrm{dist}(r, s)$, and `imbalances[r,s]` denotes $\mathbb{I}(r, s)$.

3 Our Approach

3.1 Framework

We introduce a novel approach to improve the performance of optimization problems in constraint programming. Our approach is based on the automatic generation of streamliners by using LLMs. We use BLR as the target problem to test and demonstrate the performance of our approach. Our completely autonomous procedure prompts the LLM to suggest streamlining constraints for the supplied (unstreamlined) MiniZinc model of the BLR problem. The generated streamliners are tested on the input instance for a short time to evaluate their performance relatively quickly.

In each iteration, one of several LLMs and one of three possible prompts is randomly chosen. This capitalizes on the strengths of different LLMs and different prompts. The first prompt `prompt_basic` (shown in Figure 2) asks the LLM to analyze the MiniZinc code and generate five new, creative, and syntactically correct streamliners. The formulation of the prompt is very similar to the prompt used by Voboril et al. [18]. The second

Listing 1 MiniZinc Model.

```
include "alldifferent.mzn";

% Input
int: k;
int: n;

% Alternate representation
array[1..k, 1..n] of var 1..n: pos;
constraint forall(row in 1..k) (
    alldifferent([pos[row, symbol] | symbol in 1..n])
);
constraint forall(symbol in 1..n) (
    alldifferent([pos[row, symbol] | row in 1..k])
);

% Solution representation for convenience
array[1..k, 1..n] of var 1..n: rect;
constraint forall(row in 1..k, col in 1..n, symbol in 1..n) (
    pos[row, symbol] = col <-> rect[row, col] = symbol
);

% Imbalance calculation
array[1..n, 1..n] of var 0..k*(n-1): dist;
array[1..n, 1..n] of var 0..max(k*(n+1)-3, 2*k*(n-2)): imbalances;
constraint forall(r in 1..n, s in 1..n) (
  if r < s then
    dist[r, s] = sum(row in 1..k) (abs(pos[row, r]-pos[row, s]))
  else
    dist[r, s] = 0
  endif
);
constraint forall(r in 1..n, s in 1..n) (
  if r < s then
    imbalances[r, s] = abs(3*dist[r, s]-k*(n+1))
  else
    imbalances[r, s] = 0
  endif
);

% Objective function
solve minimize sum(r in 1..n, s in 1..n where r < s) (
  imbalances[r, s]
);
```

Objective: Analyze the given MiniZinc code and suggest five additional constraints to enhance the problem-solving process. These constraints can include streamlining, implied, symmetry-breaking, or dominance-breaking constraints.

Steps:

1. Analyze Content: Read the provided MiniZinc code. Understand the problem being addressed, including its variables, constraints, and optimization goals.
2. Generate additional Constraints: Based on your analysis, create five unique constraints. These should offer targeted modifications or restrictions designed to reduce the search space effectively.
3. Always return your constraints as a JSON object, adhering to the structure: {"streamliner_1": "constraint <MiniZinc constraint>", ..., "streamliner_5": "constraint <MiniZinc constraint>"}.
   Your final output should exclusively be the JSON object containing the five constraints.

Compliance Rules:

1. Code Quality: All MiniZinc code provided for the constraints must be syntactically correct and functional. For some functions, you may need to include an additional library.
2. Creativity: You're encouraged to be innovative in proposing constraints, keeping in mind their purpose: to narrow down the search space efficiently without oversimplifying the problem.

Figure 2 `prompt_basic`.

prompt `prompt_combinations` asks for five combinations of streamliners (each consisting of multiple constraints) instead of five individual streamliners. The third prompt `prompt_description` is an extension of the first prompt with a detailed description of the BLR problem. In the following sections, we present two different approaches to use the produced streamliners. In both approaches, syntactically incorrect streamliners are simply discarded. All hyperparameter values used in these approaches were fixed on the basis of some preliminary experiments.

3.2 Incremental Warmstart Approach

Our *incremental warmstart* approach utilizes the warmstart annotation in MiniZinc. Warmstarting is a feature of certain solvers for optimization problems, where we can seed the solver's search with an initial feasible solution. This can be useful to guide the solver to find better solutions quicker. In the context of the BLR problem, we warmstart the solver with the best solution obtained so far.

Our approach, as shown in Figure 3, starts with an initial training time $t = 1$ minute, running the original encoding for that time and storing the solution s. Then, one of the LLMs is randomly chosen and asked to produce five new streamliners for the given problem. These streamliners are evaluated in parallel for t minutes. For the evaluation, the currently best solution s is used as warmstart. If a new solution s' is better than s, it replaces s as the current best solution. If there is no new improved solution after 5 iterations, the training time t is increased by 2 minutes. With a longer training time, there is a higher potential for

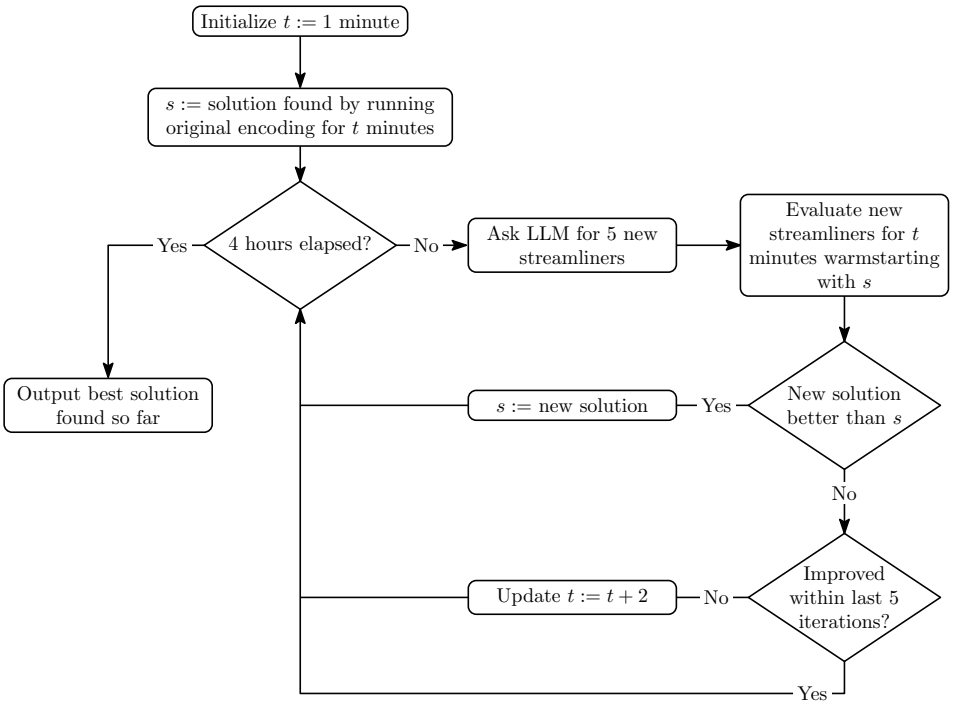

Figure 3 Diagram visualizing the incremental warmstart approach.

future streamliners to find better solutions. The process ends after a total of 4 hours have passed. At the end of the process, we can directly read off the final solution s. One variation of our incremental warmstart approach is to give it an already known solution (potentially from literature) right at the start. This makes the solving process even more efficient and enhances the chances to improve upon previously known bounds.

3.3 Evolutionary Approach

Evolutionary algorithms are optimization techniques inspired by the biological process of evolution. A set of possible solutions is stored as a population. In every generation, good solutions are selected, combined, and mutated to create potentially better solutions to add to the population.

Our evolutionary approach is shown in Figure 4. At the beginning, we run the original, unstreamlined encoding for three minutes to figure out its performance. Further, we create the set S for potential streamliners. At the beginning, S is empty. So we ask the randomly picked LLM to produce five new streamliners for the given problem. They are evaluated in parallel for 3 minutes. All streamliners that perform better than the original model after three minutes are added to the population, along with their corresponding imbalance values $\mathbb{I}$. This phase is called *exploration*. The first 20 streamliners added to the population are called the *original population*. Then, the *evolutionary* phase starts. In every iteration step, we sample 10 streamliners from our population according to probabilities proportional to $\mathbb{I}^{-4}$, favoring those with lower imbalance values. Then, these 10 streamliners are given to the LLM as reference to derive five new streamliners. This would be the combination and mutation part in terms of evolution. Again, all streamliners that produce better results than the original model are added to the population. We run this process for 5 hours. At the end of

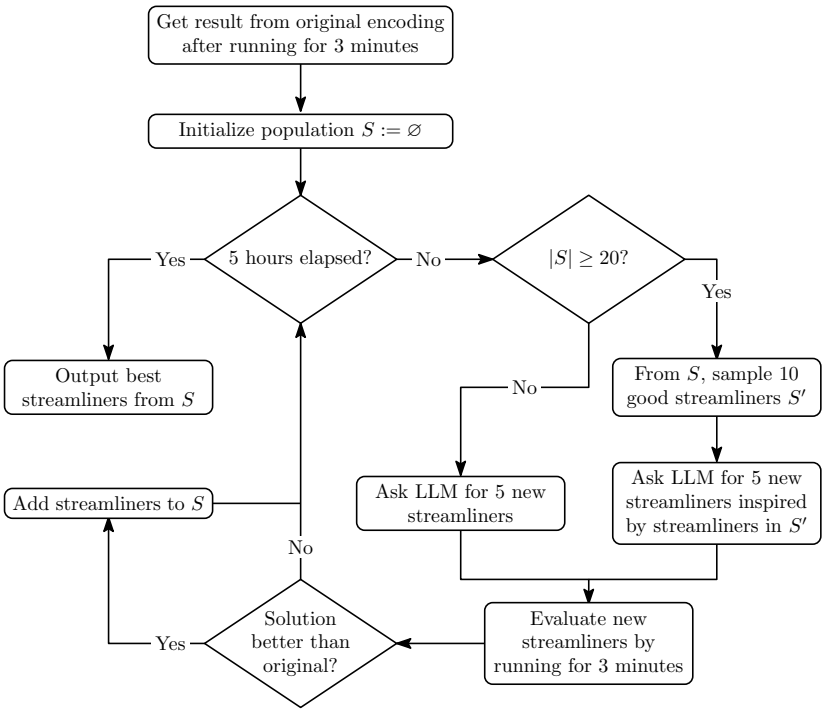

Figure 4 Diagram visualizing the evolutionary approach.

the process, we read off the three best-performing streamliners from the final population. These streamliners can potentially also be used for other instances of the same problem. Finally, we run these three streamliners in parallel for a further 4 hours and then report the best result. In our experiments, we also run a variant of this process called *exploration-only*, where we omit the evolutionary phase and only run the exploration phase for 5 hours. This is again followed by running the three best streamliners in parallel for a further 4 hours. The goal of the exploration-only variant is to assess whether the evolution indeed works well or whether the improvements are only due to the relatively long running time.

4 Experiments

The MiniZinc model, the Python implementation of the incremental warmstart and evolutionary approaches, and the minimum-imbalance Latin rectangle for each instance found by our approaches are available on Zenodo [19].

4.1 Setup and Hardware

We run all our experiments on compute nodes with 2.40GHz, 10-core 2×Intel Xeon E5-2640 v4 processors. We use MiniZinc version 2.9.2 for the incremental warmstart approach[1] and MiniZinc version 2.8.3 for the other experiments. We use Gurobi version 11.0.2 as the backend solver for MiniZinc, since previous work mainly used Gurobi and our preliminary experiments

[1] MiniZinc version 2.9.2 offers more robust support for the warmstart feature

showed it to outperform Chuffed. We use five LLMs, namely GPT-4o (openai/gpt-4o-2024-11-20), GPT-o3 (openai/o3-mini-high), Claude 3.7 Sonnet (anthropic/claude-3.7-sonnet), Deepseek R1 (deepseek/deepseek-r1), and Gemini 2.0 Flash (google/gemini-2.0-flash-001). We access all these LLMs via the unified OpenRouter API in Python 3.11.5.

4.2 Baseline

As a baseline for our experiments, we use the currently best-known results for BLRs from literature [2, 9, 10]. Since several best-known results were computed more than a decade ago, we augment the baseline with results obtained by running Gurobi for 4 hours. We run Gurobi on two variants of the model, one with the symmetry-breaking constraints used by Díaz et al. [2] and one without. These symmetry-breaking constraints enforce: $\text{pos}(s,1) = s$ for $1 \leq s \leq n$ and $\text{rect}(i,1) < \text{rect}(i+1,1)$ for $1 \leq i < k$. Listing 2 shows the MiniZinc code for these constraints. We use the results from Gurobi for instances where it can improve the previous best result or where no results are found in the literature (this includes most of the instances with $n = 13$). The baseline imbalance values are shown in Table 2. In our experiments, we only consider the instances that are yet to be solved optimally, which includes most with $k > 2$ and $n > 7$.

Listing 2 Known Symmetry-Breaking Constraints.

```
constraint forall(col in 1..n) (invrect[1,col]=col);
constraint forall(row in 1..k where row > 1) (
  rectangle[row-1,1]< rectangle[row,1]
);
```

Table 2 Baseline results for our experiments. The gray cells are already known to be optimal. The blue cells indicate instances for which either Gurobi outperforms the known results from literature or no previous results exist. The light blue cells indicate instances where the improvement came from Gurobi without symmetry-breaking, while dark blue cells indicate instances where the improvement comes from Gurobi with symmetry-breaking.

$n\backslash k$	2	3	4	5	6	7	8	9	10	11	12
8	40.0	36.0	32.0	30.0	24.0	28.0	0.0				
9	65.3	56.0	56.0	54.0	48.0	64.6	58.6	0.0			
10	92.0	82.0	90.6	66.6	82.0	76.0	95.3	80.0	40.0		
11	124.0	118.0	118.0	118.0	124.0	120.0	128.0	128.0	110.0	0.0	
12	168.0	158.0	162.6	170.6	120.0	165.3	184.6	178.0	174.6	147.3	0.0
13	218.6	210.0	228.6	224.0	234.0	222.6	273.3	248.0	288.6	278.6	352.0

4.3 Incremental Warmstart Approach

We run the incremental warmstart approach described in Section 3.2 twice to see how much the randomness of LLMs influences the experiment's outcome. Overall, the results are fairly similar between the two runs. For about three-quarters of the instances, the results differ by less than 10%. Three outliers differ by more than 20%, with the highest difference being 34%. Of all the LLM responses, around 14% had syntax errors in the produced json or minizinc code and were simply discarded. For each instance, the result of the better run is shown in Table 3. Overall, 24 out of 44 instances show an improvement compared to the baseline.

Table 3 Results of the incremental warmstart approach. The top entry within each cell shows the baseline result (from Table 2), while the bottom entry shows the result from the incremental warmstart approach for each instance. The highlighted cells indicate that the incremental warmstart approach outperforms the baseline.

$n\backslash k$	3	4	5	6	7	8	9	10	11	12
8			30.0	24.0	28.0					
			32.0	24.0	34.0					
9	56.0	56.0	54.0	48.0	64.6	58.6				
	56.0	56.0	52.6	52.0	57.3	60.0				
10	82.0	90.6	66.6	82.0	76.0	95.3	80.0	40.0		
	82.0	84.0	82.6	80.0	82.0	88.0	70.0	76.0		
11	118.0	118.0	118.0	124.0	120.0	128.0	128.0	110.0		
	114.0	120.0	116.0	116.0	122.0	118.0	124.0	118.0		
12	158.0	162.6	170.6	120.0	165.3	184.6	178.0	174.6	147.3	
	158.0	165.3	168.6	152.0	164.0	148.0	170.0	174.6	220.6	
13	210.0	228.6	224.0	234.0	222.6	273.3	248.0	288.6	278.6	352.0
	198.0	214.0	210.6	214.0	212.0	220.6	252.0	242.0	252.0	294.0

Analyzing the different prompts shows that `prompt_basic`, `prompt_description`, and `prompt_combinations` are respectively responsible for generating 37.8%, 36.9%, and 25.3% of all streamliners that lead to an improvement. This shows that providing a detailed problem description does not influence the outcome of the LLM a lot. Further, it shows that asking for combinations of streamliners also does not lead to significantly better results. This might be because there is a higher likelihood that streamliners hinder each other instead of combining their strengths, and the LLM is not fully capable of finding the mutually compatible combinations. When comparing the five different LLMs, the fraction of improvements they contribute are as follows: GPT-o3: 22.6%, GPT-4o: 22.3%, Gemini: 20.7%, Claude: 20.2%, Deepseek R1: 14.2%.

Figure 5 shows the change of imbalance over time for seven curated, representative instances. The initial result that is found after 1 minute is about 25% to 60% worse than the baseline. In the beginning, new improvements are found quite quickly; later, however, it flattens out. After about 2 hours of incremental warmstart, many of the instances can already outperform the baseline. Interestingly, if we ignore the previously known results for the BLR problem and only compare against the results from the 4-hour Gurobi runs, our incremental warmstart approach performs better on 40 of 44 instances.

4.4 Evolutionary Approach

We run the evolutionary approach and the exploration-only approach for 5 hours each. At the end, we evaluate the three best-performing streamliners from each by running them for 4 hours and then report the best result in Table 4. The evolutionary approach could improve 24 out of 44 instances with respect to the baseline, while the exploration-only approach improved 21 instances. For 6 instances, the evolutionary approach failed to assemble the original population of 20 streamliners and thus terminated before starting the evolution phase. In those cases, the result is the same as the result for the exploration-only approach. Overall, the evolutionary approach performs slightly better than the exploration-only approach.

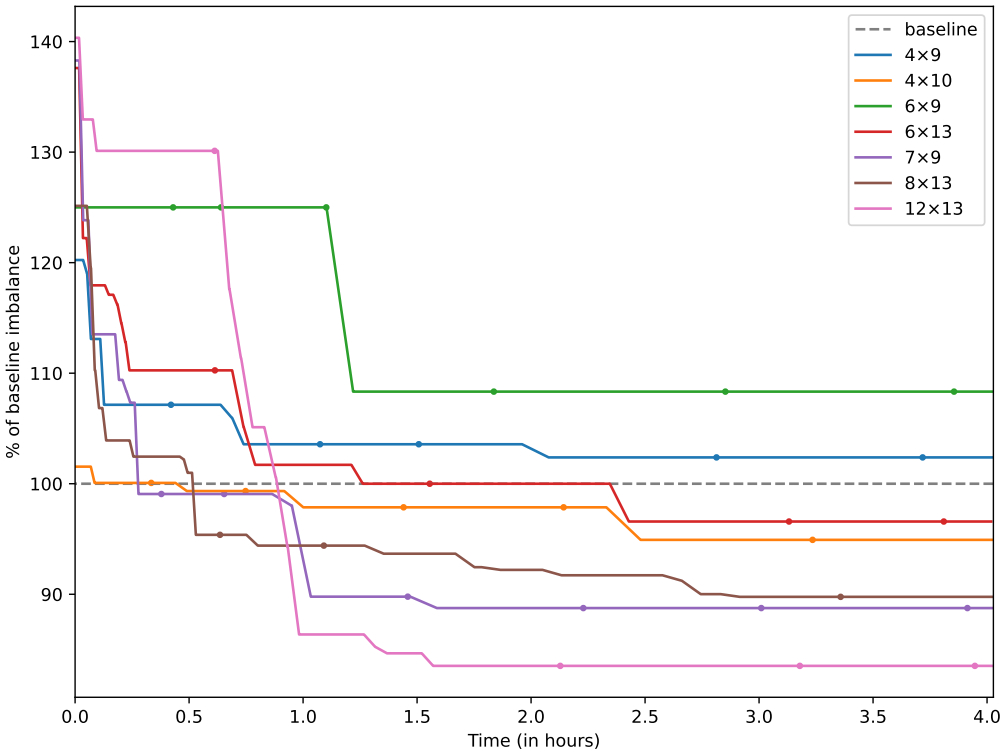

Figure 5 Change of imbalance with time for the incremental warmstart approach shown as percentage of baseline imbalance. The dashed line at 100% denotes the baseline imbalance value, and the dots on each line indicate when the training time is increased.

Another striking observation is that some of the streamliners found by our approach perform so well that running the streamlined model for 3 minutes yielded better results than running the unstreamlined model for 4 hours. Running the streamliners found in the evolutionary process for 4 further hours improved the imbalance by 10% on average compared to the result from initial 3-minute run.

Comparing the five different LLM models shows that GPT-o3, GPT-4o, Deepseek R1, Claude, and Gemini respectively found 26.0%, 25.2%, 17.9%, 17.5%, and 13.4% of the three best streamliners across all instances and variants. It is also worth noting that there are only two streamliners that appear in the list of top three streamliners for more than two instances. This demonstrates the great diversity of LLM-generated streamliners.

We showcase some interesting streamliners that were generated by our evolutionary approach in Listing 3 and provide a brief explanation of each of them below:

A This streamliner is generated for the instances 8×10, 8×11, and 3×9. It enforces each row to be strictly lexicographically smaller than the next row. However, since every entry within a row must be different, this constraint amounts to enforcing lexicographically ascending entries in the first cell of each row. Thus, the LLM rediscovered the already known symmetry-breaking constraint from literature. Interestingly, the formulation found by the LLM performed a bit better on the corresponding instances than the original formulation by Díaz et al. [2].

B This symmetry-breaking constraint works similarly to the one above. It fixes the order of the rows according to the position of the symbol 1 in each row. Instead of the values of the variable `rect`, the values of the variable `pos` are increasing with every row. This

Listing 3 Selected Streamliners.

```
A: constraint forall(r in 1..k-1) (
     lex_less(
       [rect[r,c] | c in 1..n],
       [rect[r+1,c] | c in 1..n]
     )
   );

B: constraint forall(row in 1..k-1) (
     pos[row,1] <= pos[row+1,1]
   );

C: constraint forall(row in 1..k, col in 1..(n div 2)) (
     rect[row, col] + rect[row, n-col+1] = n+1
   );

D: constraint forall(i in 1..n, j in 1..n where i < j) (
     dist[i,j] != 0
   );

E: include "global_cardinality.mzn";
   array[1..n] of var 0..k: val_counts;
   constraint global_cardinality(
     [pos[i,j] | i in 1..k, j in 1..n],
     [j | j in 1..n],
     val_counts
   );
   constraint forall(i in 1..n) (val_counts[i] >= k-1);
   constraint forall(i in 1..n, j in 1..n where i<j) (
     imbalances[i,j] <= max(k*(n+1)-3,2*k*(n-2)) div 2
   );
```

Table 4 Results of the exploration-only and evolutionary approaches. Within each cell, the top entry within each cell shows the baseline result, the middle entry shows the result from the exploration-only approach, and the bottom entry shows the result from the evolutionary approach. Green-colored cells indicate that the corresponding variant outperformed the baseline. For those instances where the baseline could be improved upon, bold text indicates which variant performed better

$n\backslash k$	3	4	5	6	7	8	9	10	11	12
8			30.0	24.0	28.0					
			32.0	28.0	28.0					
			32.0	28.0	28.0					
9	56.0	56.0	54.0	48.0	64.6	58.6				
	56.0	56.0	54.6	52.0	**55.3**	60.0				
	56.0	56.0	54.6	52.0	56.0	60.0				
10	82.0	90.6	66.6	82.0	76.0	95.3	80.0	40.0		
	80.0	**84.6**	82.0	**76.0**	78.6	**87.3**	92.0	77.3		
	80.0	**84.6**	82.0	78.0	76.0	89.3	94.0	88.0		
11	118.0	118.0	118.0	124.0	120.0	128.0	128.0	110.0		
	116.0	**112.0**	118.0	118.0	122.0	**124.0**	136.0	138.0		
	116.0	**112.0**	**116.0**	**112.0**	130.0	126.0	132.0	144.0		
12	158.0	162.6	170.6	120.0	165.3	184.6	178.0	174.6	147.3	
	160.0	**154.6**	169.3	160.0	174.0	182.0	**160.0**	186.0	204.0	
	156.0	**154.6**	**145.3**	120.0	174.0	**177.3**	176.0	199.3	219.3	
13	210.0	228.6	224.0	234.0	222.6	273.3	248.0	288.6	278.6	352.0
	204.0	220.0	219.3	**204.0**	**218.6**	**227.3**	270.0	**252.6**	283.3	332.0
	204.0	**213.3**	**216.6**	220.0	227.3	**227.3**	**228.0**	**252.6**	**224.0**	**314.0**

constraint is generated for the instances 5×8, 8×10. Further, this constraint was also part of a combination of constraints for three instances, namely, 5×11, 5×8, and 6×11.

C This streamliner is decisive to achieve the improved imbalance result for the 4×12 instance. It enforces that for every row, cells that are horizontally mirrored about the center column, add up to $n + 1$. The resulting Latin rectangle can be found in Figure 6c.

D This constraint enforces that for every pair of symbols, the distance must not be 0. Since all elements in a row are anyway defined to be different, this is an implied constraint. Nonetheless, it helped the solver find a 4×11 Latin rectangle with better imbalance.

E This combination of constraints ensures that each value $i \in [1, n]$ appears at least $k - 1$ times in each column of `pos` and that each pairwise imbalance is less than or equal to half of the maximum possible imbalance. It showcases one of the most complex combinations of constraints we obtained from the LLMs. It is not only able to combine multiple constraints but also add an `include` statement and introduce a new array of decision variables. This code is generated for the 6×8 instance. It outperforms the results from the 4-hour Gurobi runs but does not manage to beat the previously known best result.

4.5 Combinations of Approaches

Our incremental warmstart approach can not only be started from scratch but also from an already known solution. Hence, with an aim of finding the best Latin rectangles without concern for fair comparison, we run the incremental warmstart approach using the results from the evolutionary approach and with some previously published results from literature.

Table 1 summarizes the final best results we found from all approaches considered. Overall, we can improve 32 out of 44 instances. The list below shows which improving result is found by which approach. If two approaches arrive at the same result for a particular instance, then it is included in both approaches.

- **incremental warmstart**: 5×9, 9×10, 3×11, 5×11, 7×12, 8×12, 3×13, 5×13, 8×13
- **exploration-only**: 3×10, 4×11, 4×12, 6×13
- **evolution**: 3×10, 4×11, 5×11, 4×12, 5×12, 11×13
- **combination of approaches**: 7×9, 8×9, 4×10, 6×10, 7×10, 8×10, 6×11, 7×11, 8×11, 9×11, 3×12, 9×12, 4×13, 7×13, 9×13, 10×13, 12×13

In Figure 6 we present some of the Latin rectangles generated by our approach which significantly improved upon previously known results (by at least 14%) as well the 4×12 Latin rectangle for illustration purposes.

5 Discussion and Conclusion

In this paper, we present two strategies for using LLM-generated streamlining constraints for the optimization problem BLR, namely the incremental warmstart approach and the evolutionary approach. While the incremental warmstart approach uses streamliners directly on the last-found best solution, the evolutionary approach discovers a high variety of new promising streamliners. Both approaches show strong potential. An interesting finding is that, when running an optimization problem, the biggest improvements are often found in the beginning. Our approach exploits this by running many streamlined versions of the original encoding for only a few minutes, and thus can select good streamliners quickly and efficiently. Thereby, we successfully improve the upper bounds for many instances of the BLR problem, outperforming state-of-the-art methods in 32 out of 44 instances. For the other instances, it is important to consider that some of the previously known results might already be optimal. This demonstrates the potential of using LLMs to generate structural constraints that significantly enhance the solver performance.

It would be interesting to see whether it works just as effectively for other problems, particularly novel problems that are unlikely to be in the training corpus of LLMs. Concerning this, our approach has the advantage that it is very flexible. One can easily adapt it to other optimization problems and other solvers or use different LLMs. The only limitation is that it must be possible to find the first feasible solution rather quickly. However, it is also crucial to note that our method cannot prove optimality. Although we may not always find the optimal solution, our approach often finds better solutions in shorter time frames. This trade-off between theoretical guarantees and practical performance is acceptable in many real-world applications, particularly when improved solutions are more valuable than guarantees.

Looking ahead, we see several potential avenues. One particularly promising vision is that future constraint solvers might integrate our incremental warmstart approach with LLM-generated streamliners in their solving procedure. Performance improvements could be substantial. In summary, LLM-generated streamlining constraints offer a practical and powerful way to enhance the solving performance of optimization problems. Our method not only contributes new best results for the BLR problem, but also opens the door for more intelligent and adaptive solving frameworks in the future.

1	2	4	5	3	6	7	9	8	10
3	4	9	7	2	8	1	10	6	5
8	7	2	6	4	10	9	1	5	3
7	1	10	3	8	5	4	2	9	6
9	10	5	2	7	3	8	6	1	4
4	5	7	8	9	1	6	3	10	2

(a) 6×10 instance with imbalance 70.0.

8	9	5	4	6	1	2	3	7
7	4	2	6	9	3	5	1	8
5	2	4	1	7	8	9	6	3
3	5	8	7	4	6	1	2	9
4	1	3	2	8	9	7	5	6
1	6	7	8	5	2	3	9	4
9	7	1	5	3	4	6	8	2

(b) 7×9 instance with imbalance 51.3.

5	7	2	3	1	4	9	12	10	11	6	8
9	1	11	7	8	3	10	5	6	2	12	4
6	2	1	9	3	8	5	10	4	12	11	7
3	11	5	6	4	1	12	9	7	8	2	10

(c) 4×12 instance with imbalance 154.6.

5	11	3	6	4	1	12	9	7	10	2	8
7	3	1	8	11	4	9	2	5	12	10	6
1	4	10	5	7	2	11	6	8	3	9	12
9	10	11	1	6	8	5	7	12	2	3	4
2	1	6	3	9	5	8	4	10	7	12	11

(d) 5×12 instance with imbalance 145.3.

1	8	9	4	11	10	6	5	12	2	3	7
12	1	6	2	9	3	11	10	7	4	8	5
3	10	4	1	2	5	8	7	6	9	12	11
11	7	1	3	5	9	4	12	10	6	2	8
5	12	8	6	3	11	10	1	2	7	4	9
7	6	10	12	1	4	5	9	3	8	11	2
4	2	11	5	6	7	1	8	9	12	10	3
6	4	3	11	12	8	7	2	1	5	9	10

(e) 8×12 instance with imbalance 148.0.

1	11	4	8	2	7	12	13	5	6	10	9	3
8	2	10	11	6	13	7	3	1	9	5	12	4
10	12	13	2	1	3	9	8	6	4	11	5	7
6	1	7	10	9	4	2	5	8	12	13	3	11
9	13	8	4	11	5	10	6	12	2	3	7	1
7	8	12	5	10	9	1	11	4	3	6	13	2
2	9	3	7	4	10	11	12	13	1	8	6	5
12	6	11	9	7	2	8	1	3	5	4	10	13

(f) 8×13 instance with imbalance 220.6.

11	12	2	3	7	10	5	6	1	8	4	13	9
6	7	13	5	8	9	2	4	12	10	11	1	3
13	10	7	4	12	2	8	3	6	5	9	11	1
4	5	11	13	3	8	7	12	9	1	2	6	10
8	3	6	7	4	1	11	9	2	13	10	12	5
10	9	3	12	5	6	4	1	8	7	13	2	11
12	6	4	1	2	5	13	11	10	3	8	9	7
2	4	9	8	6	11	10	5	7	12	1	3	13
5	8	1	10	11	13	6	2	3	9	7	4	12
7	1	5	2	9	3	12	13	4	11	6	10	8
9	11	12	6	13	7	1	10	5	4	3	8	2

(g) 11×13 instance with imbalance 224.0.

1	2	11	10	12	6	5	13	9	4	3	7	8
13	11	3	6	8	5	12	9	10	2	1	4	7
9	10	4	11	3	8	13	1	12	6	7	2	5
3	6	1	12	9	10	7	4	13	8	5	11	2
11	12	8	7	1	3	10	2	5	9	13	6	4
4	8	6	2	10	13	1	5	3	7	12	9	11
6	7	5	9	11	2	8	3	4	12	10	1	13
7	13	10	8	6	9	2	11	1	5	4	3	12
10	5	7	4	13	12	11	6	2	3	9	8	1
5	9	13	1	4	11	3	8	7	10	2	12	6
8	4	12	5	2	1	9	7	6	13	11	10	3
2	3	9	13	7	4	6	12	11	1	8	5	10

(h) 12×13 instance with imbalance 274.0.

Figure 6 Examples of Latin rectangles generated by our approach that improve upon the previous best imbalance values.

References

1 Md. Masbaul Alam, M. A. Hakim Newton, and Abdul Sattar. Constraint-based search for optimal Golomb rulers. *J. Heuristics*, 23(6):501–532, 2017. doi:10.1007/s10732-017-9353-x.

2 Mateo Díaz, Ronan Le Bras, and Carla P. Gomes. In search of balance: The challenge of generating balanced Latin rectangles. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, pages 68–76. Springer, 2017. doi:10.1007/978-3-319-59776-8_6.

3 Carla P. Gomes and Meinolf Sellmann. Streamlined constraint reasoning. In Mark Wallace, editor, *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2004. doi:10.1007/978-3-540-30201-8_22.

4 Aditya Grover, Tudor Achim, and Stefano Ermon. Streamlining variational inference for constraint satisfaction problems. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 10579–10589, 2018. URL: https://proceedings.neurips.cc/paper/2018/hash/02ed812220b0705fabb868ddbf17ea20-Abstract.html.

5 Marijn J. H. Heule, Manuel Kauers, and Martina Seidl. Local search for fast matrix multiplication. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 155–163. Springer, 2019. doi:10.1007/978-3-030-24258-9_10.

6 Marcus Jones, Richard Woodward, and Jerry Stoller. Increasing precision in agronomic field trials using Latin square designs. *Agronomy Journal*, 107(1):20–24, 2015. doi:10.2134/agronj14.0232.

7 Ronan LeBras, Carla P. Gomes, and Bart Selman. From streamlined combinatorial search to efficient constructive procedures. In Jörg Hoffmann and Bart Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*, pages 499–506. AAAI Press, 2012. doi:10.1609/aaai.v26i1.8147.

8 Zhenjun Liu, Leroy Chew, and Marijn J. H. Heule. Avoiding monochromatic rectangles using shift patterns. In Hang Ma and Ivan Serina, editors, *Proceedings of the Fourteenth International Symposium on Combinatorial Search, SOCS 2021, Virtual Conference [Jinan, China], July 26-30, 2021*, pages 225–227. AAAI Press, 2021. doi:10.1609/socs.v12i1.18591.

9 Renee Mirka, Laura Greenstreet, Marc Grimson, and Carla P. Gomes. A new approach to finding 2 x n partially spatially balanced Latin rectangles (short paper). In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada*, volume 280 of *LIPIcs*, pages 47:1–47:11. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.47.

10 Vaidyanathan Peruvemba Ramaswamy and Stefan Szeider. Proven optimally-balanced Latin rectangles with SAT (short paper). In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada*, volume 280 of *LIPIcs*, pages 48:1–48:10. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.48.

11 Casey Smith, Carla P. Gomes, and Cèsar Fernández. Streamlining local search for spatially balanced Latin squares. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1539–1540. Professional Book Center, 2005. URL: http://ijcai.org/Proceedings/05/Papers/post-0460.pdf.

12 Patrick Spracklen, Özgür Akgün, and Ian Miguel. Automatic generation and selection of streamlined constraint models via Monte Carlo search on a model lattice. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 362–372. Springer, 2018. doi:10.1007/978-3-319-98334-9_24.

13 Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. Automatic streamlining for constrained optimisation. In Thomas Schiex and Simon de Givry, editors, *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 of *Lecture Notes in Computer Science*, pages 366–383. Springer, 2019. doi:10.1007/978-3-030-30048-7_22.

14 Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. Towards portfolios of streamlined constraint models: A case study with the Balanced Academic Curriculum Problem. *CoRR*, abs/2009.10152, 2020. doi:10.48550/arXiv.2009.10152.

15 Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. Automated streamliner portfolios for constraint satisfaction problems. *Artificial Intelligence*, 319:103915, 2023. doi:10.1016/J.ARTINT.2023.103915.

16 Nseobong Peter Uto and RA Bailey. Balanced semi-Latin rectangles: properties, existence and constructions for block size two. *Journal of Statistical Theory and Practice*, 14(3):51, 2020.

17 H.M. van Es and C.L. van Es. Spatial nature of randomization and its effect on the outcome of field experiments. *Agronomy Journal*, 85(2):420–428, 1993.

18 Florentina Voboril, Vaidyanathan Peruvemba Ramaswamy, and Stefan Szeider. Stream-LLM: Enhancing constraint programming with large language model-generated streamliners. In *2025 IEEE/ACM 1st International Workshop on Neuro-Symbolic Software Engineering (NSE)*, pages 17–22, Los Alamitos, CA, USA, May 2025. IEEE Computer Soc. URL: https://doi.ieeecomputersociety.org/10.1109/NSE66660.2025.00010, doi:10.1109/NSE66660.2025.00010.

19 Florentina Voboril, Vaidyanathan Peruvemba Ramaswamy, and Stefan Szeider. Supplementary material for paper - Balancing Latin Rectangles with LLM-generated Streamliners, June 2025. doi:10.5281/zenodo.15616074.

20 James Wetter, Özgür Akgün, and Ian Miguel. Automatically generating streamlined constraint models with Essence and Conjure. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*, volume 9255 of *Lecture Notes in Computer Science*, pages 480–496. Springer, 2015. doi:10.1007/978-3-319-23219-5_34.