Automated Nogood-Filtered Fine-Grained Streamlining: A Case Study on Covering Arrays

Orhan Yiğit Yazıcılar
School of Computer Science, University of St Andrews, St Andrews, Fife, KY16 9SX, United Kingdom

Özgür Akgün
School of Computer Science, University of St Andrews, St Andrews, Fife, KY16 9SX, United Kingdom

Ian Miguel
School of Computer Science, University of St Andrews, St Andrews, Fife, KY16 9SX, United Kingdom

Abstract

We present an automated method to enhance constraint models through fine-grained streamlining, leveraging nogood information from learning solvers. This approach reformulates the streamlining process by filtering streamliners based on nogood data from the SAT solver CaDiCaL. Our method generates candidate streamliners from high-level Essence specifications, constructs a streamliner portfolio using Monte Carlo Tree Search, and applies these to unseen problem instances. The key innovation lies in utilising learnt clauses to guide streamliner filtering, effectively reformulating the original model to focus on areas of high search activity. We demonstrate our approach on the Covering Array Problem, achieving significant speedup compared to the state-of-the-art coarse-grained method. This work not only enhances solver efficiency but also provides new insights into automated model reformulation, with potential applications across a wide range of constraint satisfaction problems.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming

Keywords and phrases Constraint Programming, Constraint Modelling, Constraint Satisfaction Problem

1 Introduction

Constraint Programming (CP) and SAT are ways of solving complex combinatorial problems. Both of these methods require the problem to be formalised in a model that can be understood by the solvers. Even with the help of the modern modelling techniques and software some of these problem classes can be difficult to solve as their search spaces can be too vast to search in a reasonable amount of time. Therefore, to make these hard instances solvable the model can be constrained further so that the solver can strengthen its reasoning. These additional constraints can speed up the search by removing branches from the search space and detecting dead ends earlier.

These additional constraints can be categorised based on their soundness – constraints that retain at least one solution if the instance is solvable – and ones that compromise soundness in order to significantly accelerate the problem-solving. *Implied* constraints are created from the initial model and therefore do not alter the solution set. Implied constraints have been successfully created with Manual [18, 28] and Automated [7, 33, 10] approaches. Other strategies include *symmetry-breaking* [12, 13, 15, 16, 19] and *dominance-breaking* [8, 27] constraints effectively eliminate duplicate solutions within the same equivalence class while ensuring the preservation of at least of solution in each class. These techniques, therefore, align with the sound constraints category. When these techniques do not provide sufficient reduction in search space or are inapplicable, *streamlining* constraints [20] can be employed for satisfiable instances. Streamliners do not guarantee soundness but offer significant speedups by dissecting the search space to focus the search effort in promising areas.

Creating streamlined constraints for a specific model used to be a difficult and time-consuming process. Small instances of a problem class were manually inspected to identify patterns that could be exploited and used to support the creation of streamliners [20, 22]. Spracklen et al. [29] developed a pipeline to automatically generate candidate streamliners from a model specification in the abstract constraint specification language Essence [17, 14] using the automated constraint modeling system Conjure [3, 1, 2]. These candidate streamliners were then applied to training instances using a Multi-Objective Monte Carlo Tree Search (MO-MCTS) [32] with two objectives:

- **Applicability:** The proportion of training instances that yield a solution when the streamliner is applied.
- **Reduction:** The average search reduction on satisfiable training instances.

Upon evaluating these streamliners against the listed objectives, a portfolio of streamliners is constructed. This portfolio holds a collection of the most promising streamliners, optimized for applicability and search reduction. The portfolio is then deployed on unseen instances using various methods.

Apart from the enhancements implemented during the modelling phase, solvers contribute significantly to expediting the search process by learning [11] *nogoods* while navigating through the search space. Nogoods are essentially combinations of variable assignments that are identified as not leading to a solution. Recognizing these can prevent the solver from wasting time exploring useless paths. This learning mechanism is essential to modern SAT and CP solvers, promoting a more informed and thus faster search. Modern SAT solvers, including notable examples like CaDiCaL [5] and Kissat [6], incorporate an advanced technique known as conflict-driven clause learning (CDCL) [24] to optimize their search process. CDCL enables the solver to learn from its encounters with conflicts within the search space. Whenever the solver reaches a contradiction or a dead end – where no variable assignment can satisfy the constraints – instead of backtracking, it analyzes the conflict to derive a new clause. This clause represents a nogood and is added to the problem's constraint set, ensuring that the same conflicting scenario is not revisited. CP solvers like Chuffed [9] use lazy clause generation (LCG) [25, 30], a technique similar to CDCL. In, LCG the solver dynamically generates clauses during the search process, rather than statically before execution. LCG combines the power of SAT solving with the expressiveness and flexibility of CP.

Shishmarev et al. revealed that nogoods inferred by learning solvers such as Chuffed during model execution could be leveraged to enhance model performance [28]. They identified that when model constraints don't propagate strongly enough to prevent search failure, nogoods come into play. Connecting the nogoods to the constraints that produced them, they were able to manually modify existing constraints or introduce new implied constraints, enhancing propagation and overall model performance. Zeighami et al. enhanced the process explained above by turning it into a semi-automatic process rather than a manual one [33].

This paper's primary contribution is to generate fine-grained streamliners by utilising the information derived from the nogoods produced by learning solvers and implementing this knowledge automatically. To accomplish this, we utilized the existing pipeline developed by Spracklen et al. to construct a portfolio of streamliners. Unlike the original approach, which divided the variable domains into two bins, our method splits them into ten bins. While the preexisting pipeline is theoretically capable of handling ten bins, the exponential increase in possible combinations as more bins are added renders the training time computationally infeasible.

Our approach leverages nogoods generated by the SAT solver CaDiCaL to pinpoint areas where extensive search activity occurs, as nogoods indicate locations where the search encounters dead-ends. By identifying these critical areas, we can implement streamlining constraints that enable the solver to avoid these dead-ends. This strategic avoidance significantly accelerates the search process, demonstrating the practical benefits of our refined streamliner generation method. Our approach not only enhances the efficiency of the solver but also sets a precedent for future research in optimizing search algorithms through the application of information gained by nogoods.

2 Covering Array Problem

The Covering Array Problem was specifically chosen for our experiments due to its demonstrated responsiveness to streamlining techniques. In previous research by Spracklen et al., this problem class showed remarkable speedups when solved using the CDCL Lingeling SAT solver [4] in conjunction with their coarse-grained streamlining approach. Given these promising results, we were particularly interested in exploring how our new fine-grained, nogood-filtered streamlining method would perform on a problem class already known to be highly responsive to streamlining. This choice allows us to not only evaluate the absolute performance of our new method but also to directly compare its effectiveness against a previously successful approach, potentially highlighting the incremental benefits of our fine-grained, nogood-informed streamlining strategy. The Covering Array Problem is a fundamental combinatorial design problem with significant applications in software testing, hardware testing, and experimental design. It involves constructing a matrix with specific coverage properties, which can be used to efficiently test interactions between different components or parameters of a system.

Formally, a covering array $CA(t, k, g)$ of size b and strength t is a $k \times b$ array $A = (a_{i,j})$ over $Z_g = 0, 1, 2, \ldots, g-1$ with the property that for any t distinct rows $1 \leq r_1 < r_2 < \cdots < r_t \leq k$, and any member $(x_1, x_2, \ldots, x_t)$ of Z_g^t, there exists at least one column c such that $x_i = a_{r_i,c}$ for all $1 \leq i \leq t$. The covering array number $CAN(t, k, g)$ is defined as the smallest b such that there exists a $CA(t, k, g)$ of size b [21].

Informally, this definition means that any t distinct rows of the covering array must encode, column-wise, all numbers from 0 to $g^t - 1$, allowing repetitions. This property ensures that all possible t-way interactions between the parameters are tested at least once.

For example, consider a covering array $CA(3, 5, 2)$ over the Boolean alphabet $0, 1$. A solution is shown in Figure 1. In this array, any $t = 3$ rows encode all numbers from 0 (when

0	0	0	0	0	1	1	1	1	1
0	0	0	1	1	0	0	1	1	1
0	0	1	0	1	0	1	0	1	1
0	1	0	0	1	0	1	1	0	1
0	1	1	1	0	1	0	0	0	1

Figure 1 A Solution for $CA(3, 5, 2)$.

the respective elements are $0, 0, 0$) to $2^3 - 1 = 7$ (when the elements are $1, 1, 1$). For instance, the top three rows encode the sequence $0, 0, 1, 2, 3, 4, 5, 6, 7, 7$ from left to right, while the bottom three rows encode $0, 3, 5, 1, 6, 1, 6, 2, 4, 7$.

The ESSENCE specification of the Covering Array Problem is as follows:

```
language Essence 1.3
given t : int(1..)
given k : int(1..)
given g : int(2..)
given b : int(1..)
where k>=t, b>=g**t
find CA: matrix indexed by [int(1..k), int(1..b)] of int(1..g)
such that
    forAll rows : sequence (size t) of int(1..k) .
        (forAll i : int(2..t) . rows(i-1) < rows(i)) ->
        forAll values : sequence (size t) of int(1..g) .
            exists column : int(1..b) .
                forAll i : int(1..t) .
                    CA[rows(i), column] = values(i)
such that forAll i : int(2..k) . CA[i-1,..] <=lex CA[i,..]
such that forAll i : int(2..b) . CA[..,i-1] <=lex CA[..,i]
```

Figure 2 An Essence specification of the Covering Array Problem (Problem 45 at CSPLib.org).

This specification defines the structure and constraints of the Covering Array Problem, forming the basis for our streamlining experiments. By applying our fine-grained streamlining approach to this problem, we aim to demonstrate the effectiveness of our method in reducing search effort and improving solver performance.

3 Architecture

We begin by explaining the architecture of the underlying system that generates, evaluates, and selects streamlining constraints for a problem class of interest. The streamlining implementation has three distinct phases. Firstly, candidate streamliners are generated from an Essence specification using the automated modelling tool Conjure. Secondly, these candidate streamliners are joined together to create a portfolio of streamliners. Lastly, given an unseen instance streamliners are selected from the portfolio and used in solving.

3.1 Phase 1: Generating Candidate Streamliners

Given a problem specification in Essence the automated modelling tool Conjure generates candidate streamliners using a set of encoded rules. There are different rules for each of the possible decision variable types in an Essence specification. Spracklen et al. [29] explain the detailed process of how Conjure derives these candidate streamliners by leveraging the high-level structure of the Essence specification.

Our current approach generates a larger set of candidate streamliners. This is achieved by retaining the original candidates generated by Spracklen et al. and by splitting each dimension of a given decision variable in 10 bins, we create additional candidate streamliners that apply to each of these bins. The new rules are fully described and the generation process is explained in Section 4.

3.2 Phase 2: Creating a Portfolio of Streamliners

Once the candidate streamliners are generated, the next phase involves combining these streamliners to form a robust portfolio. This is achieved by evaluating the effectiveness of each streamliner individually and in combination with others. The objective is to create a portfolio that balances the trade-off between reducing the search space and maintaining a high probability of finding a solution. This involves a search process that explores various combinations of streamliners, using metrics such as applicability and search reduction to guide the selection of the most effective combinations.

Before starting to run the training instances with the streamliners, we run them without streamliners to establish a performance baseline. While running each of these training instances, we store the learnt clauses generated by CaDiCaL [5] for each of them. These learnt clauses are then parsed and binned according to which part of the decision variable they apply. As these learnt clauses are only generated when the search fails, we can identify where most of the search activity happens for a given problem class. An example of the binning process for the Covering Array Problem can be seen in Figure 3. This activity metric is then used to filter the candidate streamliners, retaining streamliners that only affect the bins where most of the search activity occurs. For example, in the Covering Array Problem, the approach by Spracklen et al. generates 144 candidate streamliners. Initially, our approach generates 280 candidate streamliners, which would be extremely costly to create a portfolio from. However, once filtered using the learnt clauses, we are left with only 48 streamliners. The binning process is fully explained in Section 5.

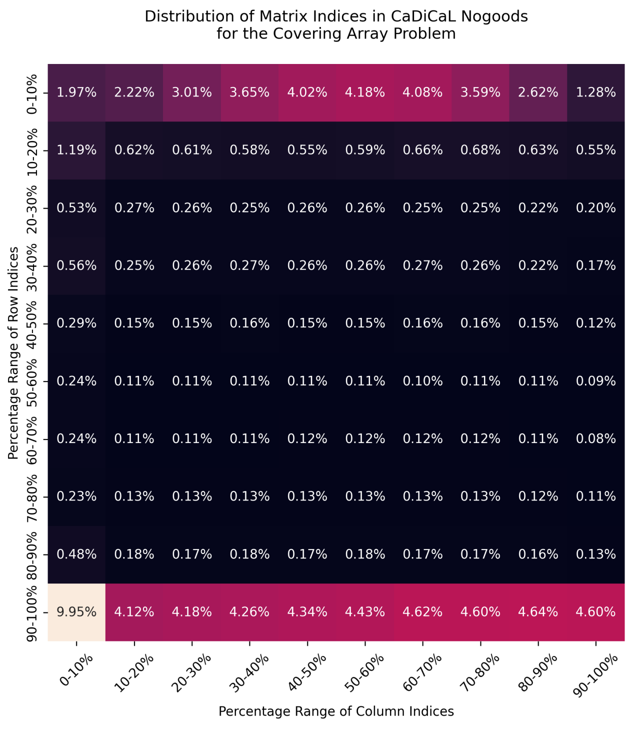

Figure 3 Heatmap showing the distribution of matrix indices in CaDiCaL nogoods for the Covering Array Problem. The values represent the percentage of occurrences within specific ranges of row and column indices. Lighter colors indicate higher percentages of nogood occurrences in those areas.

A Monte Carlo Tree Search (MCTS) method is then used to explore the space of streamliner combinations, evaluating their performance on the training instances. The search aims to identify non-dominated combinations, forming a Pareto front that represents the best trade-offs between applicability and search reduction. This method ensures that the resulting portfolio contains streamliners with complementary strengths, enhancing the overall performance across diverse instances. Portfolio creation is explained in detail in Section 6.

3.3 Phase 3: Streamliner Selection and Application

In the final phase, the constructed portfolio is used to solve unseen instances of the problem class. Several methods can be employed to select the most appropriate streamliner or combination of streamliners from the portfolio. These methods range from simple approaches to more computationally expensive techniques.

One simple method is to select the single best streamliner based on average performance across the training instances. Another approach is the applicability-first method, which chooses the streamliner that maintains the highest applicability (i.e., the proportion of instances for which the streamliner retains at least one solution). Alternatively, the reduction-first method selects the streamliner that achieves the greatest average reduction in solving time for the training instances.

For more complex methods, AutoFolio [23] can be trained on the training instances. AutoFolio uses machine learning techniques to predict which streamliner to apply to test instances based on instance features. This approach leverages the historical performance data of the streamliners and the characteristics of the instances to dynamically select the most effective streamliner for each unseen instance, leading to significant improvements in solving time.

4 Candidate Streamliner Generation

The automatic generation of candidate streamliners is the foundation of our approach. This process enables the exploration of a wide range of potential streamlining constraints without manual intervention, a task that traditionally required substantial human expertise and time investment. Our method capitalises on the rich structure present in ESSENCE specifications to produce streamliners that can significantly reduce the search space while retaining at least one solution. ESSENCE, as a high-level problem specification language, provides abstract type constructors such as set, multiset, function, and relation.

4.1 Streamlining Rules

The generation of candidate streamliners is accomplished through the application of a system of streamlining rules. These rules operate on the domains of ESSENCE terms, which may refer to decision variables or components thereof. Each rule takes such a domain as input and yields a constraint that is then applied to the corresponding term. A key strength of our approach lies in its ability to handle the complex, nested structures that ESSENCE's abstract domains can embody. The language allows for arbitrary nesting of domain constructors, and our streamlining rules are designed to exploit this hierarchical structure fully.

To achieve this, we employ a system of higher-order rules. These allow a rule initially defined for a simple domain D to be "lifted" to operate on more complex domains constructed from D. For instance, a rule can be automatically adapted to work on domains of the form "set of D", "multiset of D", "function of D", "relation of D", and so forth.

This lifting mechanism greatly enhances the versatility and power of our streamliner generation process. It allows a relatively small set of base rules to generate a wide variety of streamliners, capable of capturing intricate patterns and regularities across diverse problem structures. Furthermore, it ensures that our system can handle the full expressive power of Essence, adapting seamlessly to whatever complex domain structures a problem specification might employ. To produce a diverse set of candidate streamliners, we employ rules that are categorised into two classes:

1. **First-order rules:** These rules operate directly on the domains of decision variables, imposing constraints that narrow the search space. For instance, they might restrict integer variables to odd or even values, enforce monotonicity on function variables, or apply specific properties to relation variables. These rules form the foundation of the streamliner generation process, capturing fundamental patterns that may arise across various problem classes.
2. **High-order rules:** These more complex rules take other rules (either first-order or higher-order) as arguments, extending their application to decision variables with nested domains.

This two-tiered approach allows the system to generate streamliners of varying complexity, from simple domain restrictions to intricate constraints on nested structures. The first-order rules are identical to those created by Spracklen et al. [29], and therefore will not be re-explained here. However, to generate the fine-grained streamlining rules, two additional high-order rules were added.

Name matrixByRowBucket
Works on 2D matrices.
Param `R` (another rule)
Param `b` (bucket id, from 0 to 9)
Input `X: matrix indexed by [I, J] of K`
Define `lb` as lower bound of `I`
Define `ub` as upper bound of `I`
Define `s` as the bucket size, `(ub - lb + 1) / 10`
Output
```
forAll r : int(lb+s*b .. min([ub, lb+s*(b+1)]))
    . R(X[r,..])
```

Name matrixByColumnBucket
Works on 2D matrices.
Param `R` (another rule)
Param `b` (bucket id, from 0 to 9)
Input `X: matrix indexed by [I, J] of K`
Define `lb` as lower bound of `J`
Define `ub` as upper bound of `J`
Define `s` as the bucket size, `(ub - lb + 1) / 10`
Output
```
forAll r : I
    forAll c : int(lb+s*b .. min([ub, lb+s*(b+1)]))
        . R(X[r,c])
```

Figure 4 Higher-order streamlining rules for matrix domains implemented for fine-grained streamlining.

Figure 4 presents the two high-order streamlining rules designed for fine-grained streamlining of 2D matrices: matrixByRowBucket and matrixByColumnBucket. Both streamliners operate on similar principles but focus on different dimensions of the matrix. They take two parameters: `R`, which is another rule to be applied, and `b`, a bucket identifier ranging from 0 to 9. Both of these streamliners work on a 2D matrix as shown by the input field. These streamliners divide the matrix into 10 buckets along either the rows or columns.

The matrixByRowBucket streamliner focuses on rows. It calculates a bucket size by dividing the number of rows by 10. The streamliner then applies the rule `R` to a subset of rows determined by the bucket identifier `b`. For example, if `b = 3`, the streamliner would apply `R` to rows in the fourth decile of the matrix. For each row `r` that is inside the selected bucket, rule `R` is applied to the entire row `X[r,..]`.

The matrixByRowBucket streamliner operates on columns. It calculates a bucket size by dividing the number of columns by 10. The streamliner then applies the rule `R` to a subset of columns determined by the bucket identifier `b`. This streamliner applies `R` to individual elements within the selected column bucket for each row. Specifically, for each row `r` in the matrix and each column `c` that is inside the selected bucket, rule `R` is applied to the individual value `X[r,c]`.

Both of these streamliners use the minimum function to ensure that the last bucket, which might not contain the same number of rows/columns as other buckets is handled appropriately. For example, if we had a matrix with 97 rows, each bucket would have $\lceil \frac{97}{10} \rceil = 10$ rows, except for the last one. The last bucket would contain rows 91 to 97. However, without the minimum function, we would get the following range for the last bucket: `int(1+10*9..1+10*10)`, which simplifies to `int(91..101)`. With the minimum function, we get `int(91..min(97, 101))` so the range is correctly set to `int(91..97)`, accurately representing the boundaries of the streamlined matrix.

The division of each dimension into 10 bins in these new streamlining rules was an arbitrary choice in this initial implementation. We acknowledge that this selection was not based on any specific theoretical or empirical optimization. The number 10 was chosen as a starting point that seemed to offer a reasonable balance between granularity and computational feasibility. It's important to note that the optimal number of bins may vary depending on the problem domain and instance size. While 10 bins allow us to demonstrate the concept and potential benefits of this more granular approach to streamlining, we recognize there is room for optimization. In future work, we plan to explore the impact of different bin sizes (e.g., 5, 8, or 15) on both the quality of streamlining and computational efficiency.

5 Binning Learnt Clauses to Filter Candidate Streamliners

A key innovation in our approach is the utilisation of learnt clauses from the SAT solver CaDiCaL to filter and refine the set of candidate streamliners. This process allows us to focus on the most promising areas of the search space, significantly reducing the number of streamliners that need to be evaluated whilst maintaining or even improving overall performance.

5.1 Binning Process

During the initial solving phase, we run each training instance without any streamliners to establish a baseline performance. Concurrently, we collect the learnt clauses generated by CaDiCaL throughout the solving process. These clauses represent combinations of variable assignments that have been proven to lead to conflicts, effectively capturing the solver's "knowledge" about the problem structure and difficult areas of the search space. Once the learnt clauses are collected, we parse and categorise them based on which parts of the decision variables they affect. For matrix-based problems like the Covering Array, this involves mapping each clause to specific rows and columns of the matrix. We divide each dimension of the decision variable into ten bins. For instance, in a 100x100 matrix, each bin would represent a 10x10 submatrix. The learnt clauses are then assigned to these bins based on the variables they contain. This process creates a heatmap, as shown in Figure 3, indicating where the solver encountered the most difficulties during the search. This approach offers several key benefits:

1. **Enhanced Portfolio Construction:** By significantly decreasing the number of candidate streamliners, we can explore streamliner combinations more thoroughly within the same computational budget. This allows the portfolio construction phase to uncover more complex and potentially synergistic combinations of streamliners, leading to more robust and effective streamlining strategies.
2. **Focused Streamlining:** The retained streamliners are more likely to address the most challenging aspects of the problem, as identified by the concentration of learnt clauses affecting the area. This targeted approach potentially leads to greater search reductions and more efficient problem solving.
3. **Problem-Specific Adaptation:** The binning process adapts to the specific characteristics of each problem class, as reflected in the distribution of learnt clauses. This flexibility allows our method to be effective across a wide range of problem types and instances.

These benefits collectively contribute to a more efficient and effective streamlining process, leveraging the power of modern SAT solvers to guide the streamlining process and improve overall constraint model performance.

5.2 Finer-grained Streamlining

In our current implementation, bins are associated with some number of rows or columns, as the candidate streamliners we generate affect entire rows or columns. This approach allows us to focus on the most problematic areas of the matrix while keeping the number of streamliners manageable. This method could be made finer-grained, leading to more precise streamlining, although at the cost of significantly increasing the number of candidate streamliners and therefore presenting a greater computational challenge. The increased number of streamliners and their combinations would require substantially more processing time and resources to evaluate effectively. Furthermore, overly specific streamliners might risk overfitting to the training instances, potentially reducing their effectiveness on unseen problems.

6 Creating the Streamliner Portfolio

The creation of an effective streamliner portfolio is a crucial component of the approach. The process involves exploring the space of possible streamliner combinations to identify those that provide the best balance between applicability and search reduction. The method has evolved significantly, building upon the original approach by Spracklen et al. [29] while introducing several key enhancements.

In the original implementation by Spracklen et al., a single-threaded Multi-Objective Monte Carlo Tree Search (MO-MCTS) was employed to traverse the lattice of streamliner combinations. This search aimed to build a portfolio of non-dominated streamliners, where domination was defined across two objectives: applicability (the proportion of training instances for which the streamlined model admits a solution) and search reduction (the mean search reduction achieved on satisfiable instances).

The original process involved four main phases:

1. **Selection:** Starting from the root node, the Upper Confidence Bound (UCT) policy was applied to traverse the explored part of the lattice until reaching an unexpanded node.
2. **Expansion:** A random admissible child was selected and expanded.
3. **Simulation:** The collection of streamliners associated with the expanded node was evaluated, calculating the applicability and reduction across the set of training instances.
4. **Back Propagation:** The new combination was tested for Pareto dominance against the current portfolio. If non-dominated, it was added to the portfolio, potentially replacing dominated combinations. The result updated reward values throughout the lattice, guiding future search.

To improve upon this foundation, several enhancements have been introduced. Firstly, a multi-threaded approach to portfolio construction has been implemented. This allows for the simultaneous evaluation of multiple streamliner combinations, significantly reducing the time required to build a comprehensive portfolio. The number of threads is configurable, enabling optimal utilisation of available computational resources. Secondly, a new method for pruning the search space of streamliner combinations has been introduced. In addition to the existing pruning strategies, a technique that skips combinations if they don't have any common satisfiable instances is now used. This is achieved by maintaining a set of satisfiable instances for each streamliner and only evaluating combinations where the intersection of these sets is non-empty. This approach drastically reduces the number of unnecessary evaluations, particularly for more aggressive streamliners that tend to render many instances unsatisfiable.

To understand how the pruning technique for the streamliner combinations works in practice, let's consider a simplified example. Consider a scenario with 5 instances $\{A, B, C, D, E\}$ and 2 streamliners $\{S1, S2\}$. Initially, the original model (empty set of streamliners) is satisfiable on all instances. Streamliner $S1$ is satisfiable on instances $\{A, C, E\}$, while $S2$ is satisfiable on $\{B, D, E\}$. When evaluating the combination $S1 + S2$, we first check the intersection of their satisfiable instances: $\{A, C, E\} \cap \{B, D, E\} = \{E\}$. Since this intersection is non-empty, we proceed to evaluate $S1 + S2$, but only on instance E. If $S1 + S2$ is satisfiable on E, its set of satisfiable instances becomes $\{E\}$; if unsatisfiable, $S1 + S2$ would be pruned. This approach significantly reduces evaluations: instead of testing $S1 + S2$ on five instances, we only evaluated it on 1. Moreover, if $S1$ and $S2$ has no common satisfiable instances, we would have skipped evaluating their combination entirely. This pruning technique is especially effective for aggresive streamliners that ofter render many instances unsatisfiable, as it quickly identifies and avoids overly restrictive combinations.

7 Experimental Results

To evaluate our fine-grained streamliner generation approach, we conducted experiments on the Covering Array Problem. This section details the results, comparing our method against the coarse-grained approach and examining the impact of filtering streamliners using learnt clause information.

7.1 Experimental Setup

We utilised the Covering Array Problem as our benchmark due to its inherent complexity and the potential benefit from streamlining techniques. The experiments followed the pipeline developed by Spracklen et al. [29], with significant modifications to integrate our fine-grained streamliner generation and filtering approach. The instances were executed using the CaDiCaL SAT Solver.

All experiments were performed on compute nodes equipped with two 2.1 GHz, 18-core Intel Xeon E5-2695 processors, ensuring a robust and consistent hardware environment. The streamliner portfolio construction phase utilised 30 cores and was allocated a maximum walltime of 2 days, allowing for extensive exploration and generation of streamliners.

Three groups of streamliners were evaluated: Coarse, Fine, and Fine Filtered. The Coarse group utilised the original coarse-grained streamliners developed by Spracklen et al. The Fine group consisted of our novel fine-grained streamliners, generated by dividing each dimension of a decision variable into 10 bins, thus providing a more detailed approach. The Fine Filtered group included fine-grained streamliners that were further refined by filtering using learnt clause information from the CaDiCaL SAT Solver.

For each group, three selection strategies were applied: Reduction First, Applicability First, and Oracle. The Reduction First strategy involved selecting the streamliner with the greatest average reduction in solving time for the training instances, prioritising performance gains. The Applicability First strategy focused on choosing the streamliner with the highest applicability across training instances, ensuring broad usability. The Oracle strategy, an idealised approach, always selected the best-performing streamliner for each instance, representing the theoretical upper bound of performance and serving as a benchmark for the other strategies.

To ensure a meaningful evaluation, we carefully selected problem instances with base runtimes between 300 and 3,600 seconds. This range was chosen to focus on moderately difficult to challenging problems, excluding trivial instances that might not benefit significantly from streamlining techniques and extremely hard instances that could skew our results. The selected instances were divided into five folds for robust cross-validation. Each fold was used as a test set once, with the remaining folds serving as training data. This division allowed for a comprehensive evaluation of our streamliner generation and selection strategies across a variety of instance subsets.

7.2 Performance Comparison

The results demonstrate the effectiveness of our fine-grained streamliner generation, particularly when combined with learnt clause information filtering. Figure 5 shows speedup evolution across five folds, and Table 1 summarises the average speedups for different strategies. A speedup of less than 2 times is considered a loss, as running the streamlined model alongside the unstreamlined model effectively doubles computational resources used. Thus, a speedup of at least 2 times is necessary to justify the additional resource consumption.

Problem Class	Streamliner Group	Reduction First	Applicability First	Oracle
Covering Array	Coarse	5.45	5.41	5.69
	Fine	6.81	6.38	7.56
	Fine Filtered	10.03	8.96	10.21

Table 1 Average Speedups for Different Strategies (Maximum Common Iteration Between All Folds)

A critical factor in the performance of each approach is the number of streamliners generated: 144 for the Coarse approach, 280 for the Fine approach (unfiltered), and 48 for the Fine Filtered approach. The large number of streamliners in the Coarse and Fine

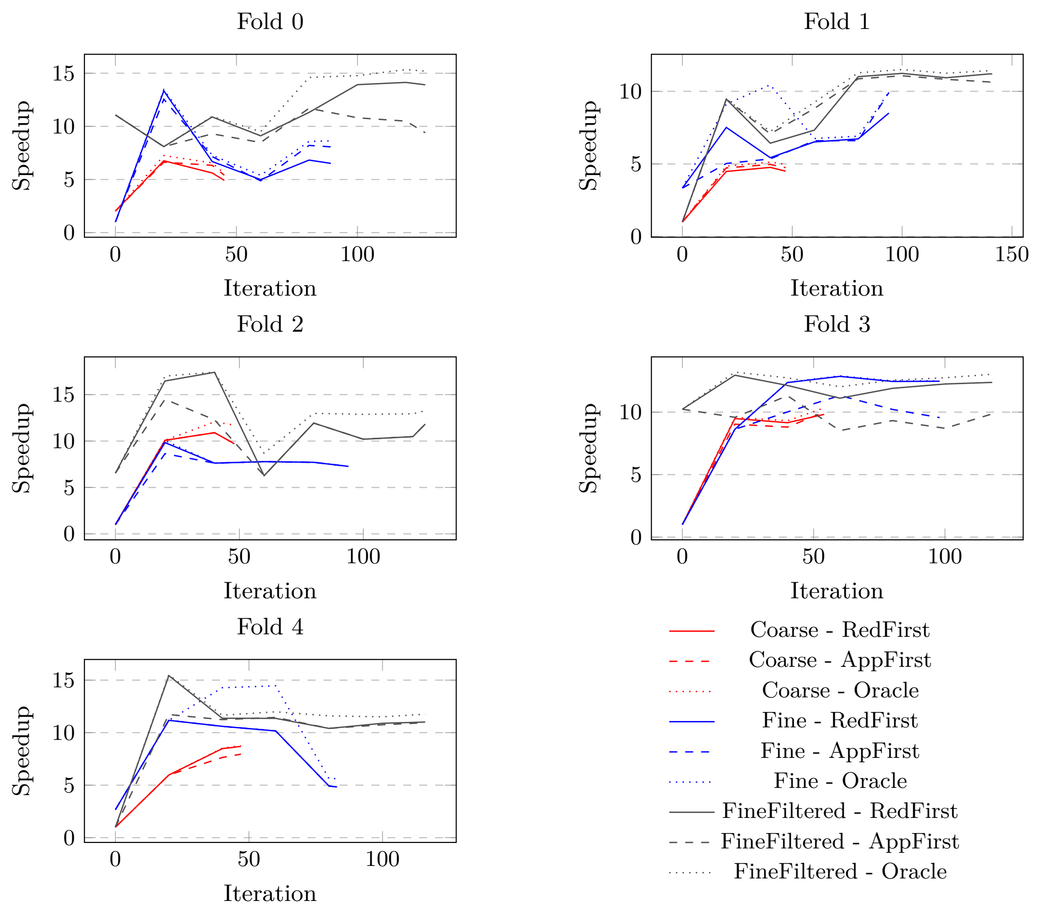

Figure 5 Speedup Evolution for each Fold (Samples Were Taken Every 20 Iterations)

approaches made it computationally infeasible to explore many combinations within the time budget, limiting these approaches to using individual streamliners only. In contrast, the Fine Filtered approach, with a more manageable set of 48 streamliners, could evaluate and combine streamliners, discovering synergistic effects and significantly better performance.

The Fine Filtered - Reduction First strategy achieves the highest average speedup of 10.03x, a significant improvement over the best Coarse approach performance of 5.45x. The Fine Filtered approach completes more iterations within the same time budget compared to the Coarse and Fine approaches. This efficiency stems from targeting areas where the solver encountered the most deadends, as identified by learnt clause information. The smaller streamliner set allows for quicker evaluation and combination, further reducing the search space and enabling faster solutions. This targeted and efficient approach maintains superior performance across all folds, suggesting its robustness and potential for generalisation to unseen instances. To illustrate the effectiveness of our Fine Filtered - Reduction First strategy, consider a Covering Array instance that typically requires one hour (3600 seconds) to solve using the original, unstreamlined model. The Coarse approach, with its average speedup of 5.45x, would reduce this solving time to about 11 minutes (660 seconds). However, our Fine Filtered - Reduction First strategy, achieving an average speedup of 10.03x, further cuts the solving time to just 6 minutes (359 seconds).

Table 2 compares the final streamliner portfolios for each group on Fold 0 of the Covering

Streamliner Group	Streamliner	Avg Applicability	Overall Reduction
Coarse	295	88.2%	70.0%
	423	88.9%	54.8%
	299	91.0%	45.9%
Fine	47	92.4%	77.4%
	43	96.5%	51.9%
	20	93.1%	68.8%
	19	95.1%	61.5%
Fine Filtered	240	97.2%	80.0%
	24	96.5%	82.8%
	13-239	95.8%	84.6%
	13-230	93.1%	84.1%
	23-229	93.8%	84.0%

Table 2 Comparison of the final streamliner portfolios for each group on Fold 0. The table presents the streamliner identifiers which are used by CONJURE to know which streamliner to apply, average applicability (percentage of instances where the streamliner maintains at least one solution), and overall reduction in search time for satisfiable instances. Each row represents a streamliner in the final portfolio. The Fine Filtered group shows a larger and more diverse portfolio with generally higher applicability and reduction rates compared to the Coarse and Fine groups.

Array Problem, highlighting the effectiveness of our fine-grained, nogood-filtered streamlining approach. The Coarse group shows moderate performance with applicability ranging from 88.2% to 91.0% and reduction rates of 45.9% to 70.0%. The Fine group demonstrates improvement, with higher applicability (92.4% to 96.5%) and better reduction rates, peaking at 77.4%. The Fine Filtered group, however, exhibits the most promising results, with high applicability rates above 93% and reduction rates from 80.0% to 84.6%. This group's portfolio includes combinations of streamliners, indicating synergistic effects. The superior performance of the Fine Filtered group can be attributed to the precision of fine-grained streamlining and the targeted focus provided by nogood-based filtering, which effectively narrows streamliner generation to problematic areas of the search space. Although the Fine approach would theoretically discover the same effective candidates as the Fine Filtered approach given sufficient time and resources, the filtering process significantly accelerates discovery, making it feasible within practical time constraints and with more modest computational requirements. The remaining tables corresponding to the other folds can be found in Appendix A.

The final streamliner portfolios for Fold 0 of the Covering Array Problem, as shown in Table 2, comprise from various streamliners for the Coarse, Fine, and Fine Filtered approaches. To better understand the performance differences observed in the table, let's examine the specific constraints imposed by each of these streamliners in Fold 0:

The Coarse approach streamliners (295, 423, and 299) represent different strategies for imposing broad constraints on the covering array CA. Streamliner 295 enforces a near-balanced distribution in each row of CA, requiring that the number of cells with values less than or equal to the midpoint falls within a narrow range: between $(b/2) - 1$ and $(b/2) + 1$, allowing for only minor deviations from an exact 50-50 split in each row. Streamliner 423 focuses on the rows of CA, requiring that at most half of all rows $(k/2)$ have at most half of their cells containing values not exceeding the midpoint, ensuring that a significant portion of the rows have a majority of higher values. Streamliner 299 sets a minimum threshold for lower values in every row of CA, stating that in each row, at least half of the cells must contain values less than or equal to the midpoint.

The Fine approach streamliners (47, 43, 20, and 19) focus on specific sections of the covering array CA, applying constraints to targeted ranges of rows. Streamliner 47 ensures that in the second 10% of rows, at most half of the cells in each row have values less than or equal to the midpoint. Streamliner 43, applied to the same range of rows, requires that at least half of the cells in each row have values less than or equal to the midpoint. Streamliner 20 targets the first 10% of rows, specifying that at least half of the cells in each row have values greater than the midpoint. Streamliner 19, also focusing on the first 10% of rows, requires that at least half of the cells in each row have values less than or equal to the midpoint.

The Fine Filtered approach streamliners (240, 24, 13, 239, 230, 229, and 23) apply targeted constraints to specific portions of the covering array CA, focusing on either the first 10% or last 10% of rows. Streamliners 24 and 240 ensure that at most half of the cells in each row have values greater than the midpoint in the first and last 10% of rows, respectively. Streamliner 23 requires that at most half of the cells in each row have values less than or equal to the midpoint in the first 10% of rows, while Streamliner 239 applies the same constraint to the last 10% of rows. Streamliners 13 and 229 balance the number of odd-valued cells around the midpoint in the first and last 10% of rows, respectively, ensuring this count falls between $(b/2) - 1$ and $(b/2) + 1$. Streamliner 230 applies a similar balancing constraint for even-valued cells in the last 10% of rows.

8 Comparison to Sampling-Based Methods

While our approach focuses on streamlining to find a single solution efficiently, it shares conceptual similarities with recent sampling-based methods in constraint programming. Two notable works in this area are particularly relevant: Vavrille et al.[31] proposed adding randomly generated table constraints to reduce the solution space, aiming to improve sampling randomness while maintaining efficiency. Their method divides the search space by adding random hashing constraints until only a small, tractable number of solutions remain. Similarly, Pesant et al.[26] introduced linear modular equality constraints to partition the solution space into roughly equal-sized cells, aiming for near-uniform sampling with probabilistic guarantees.

Despite these similarities, our approach differs in several key aspects. Firstly, our primary objective is to find a single solution efficiently, rather than sampling uniformly from the solution space. Secondly, we generate streamliners based on the problem structure specified in ESSENCE, allowing for more tailored search space reduction, whereas the sampling methods use more generic constraints. Thirdly, our approach leverages nogood information to guide the application of streamliners, potentially focusing on more promising areas of the search space, while the sampling methods aim for a more uniform exploration. Lastly, the sampling methods provide probabilistic guarantees about the uniformity of their samples, which our method does not aim to do.

The effectiveness of these sampling approaches in partitioning the search space suggests potential avenues for future work in our streamlining framework. For instance, incorporating linear modular equality constraints as streamliners might offer an interesting alternative to our current approach, potentially providing stronger theoretical properties. Additionally, adapting our use of nogood information to guide search space partitioning for sampling purposes could lead to a hybrid approach that combines the strengths of both streamlining and sampling methods. While our current focus remains on finding a single solution quickly, these ideas from sampling methods could potentially enhance the versatility and effectiveness of our streamlining approach in future iterations.

9 Conclusion

In conclusion, our proposed method of utilizing nogood information from learning solvers to generate fine-grained streamliners has demonstrated significant improvements in solver efficiency for constraint satisfaction problems. By splitting variable domains into multiple bins and filtering streamliners based on learnt clause information, we have achieved a substantial reduction in search space and solving time.

Our experimental results on the Covering Array Problem show that the fine-grained, filtered approach outperforms the coarse-grained method, achieving higher average speedups and more efficient solver performance. However, there are several areas for future improvement. Firstly, our current implementation has been primarily tested on the Covering Array Problem. To ensure the robustness and generalisability of our approach, it is essential to test it on a wider range of problem classes. This will help identify any limitations and further refine the methodology to enhance its applicability across various CSPs. Secondly, our approach currently supports only matrices as decision variables. For broader applicability, it is crucial to extend support to all ESSENCE datatypes. This will enable the generation of streamliners for a more diverse set of problems, enhancing the versatility and utility of our approach in real-world scenarios. By addressing these issues, we can further improve the effectiveness and applicability of our fine-grained streamliner generation method, paving the way for more efficient and scalable solutions to complex constraint satisfaction problems.

References

1 Özgür Akgün. *Extensible automated constraint modelling via refinement of abstract problem specifications*. PhD thesis, University of St Andrews, 2014.

2 Özgür Akgün, Alan M Frisch, Ian P Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Conjure: Automatic generation of constraint models from problem specifications. *Artificial Intelligence*, 310:103751, 2022.

3 Ozgur Akgun, Ian P Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Breaking conditional symmetry in automated constraint modelling with conjure. In *ECAI*, pages 3–8, 2014.

4 Armin Biere. Lingeling essentials, a tutorial on design and implementation aspects of the the sat solver lingeling. *POS@ SAT*, 27:88, 2014.

5 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

6 Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*, pages 10–11. University of Helsinki, 2022.

7 John Charnley, Simon Colton, and Ian Miguel. Automatic generation of implied constraints. In *ECAI*, volume 141, pages 73–77, 2006.

8 Geoffrey Chu and Peter J Stuckey. Dominance breaking constraints. *Constraints*, 20:155–182, 2015.

9 Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed, a lazy clause generation solver, 2018. URL: `https://github.com/chuffed/chuffed/`.

10 Simon Colton and Ian Miguel. Constraint generation via automated theory formation. In *Principles and Practice of Constraint Programming—CP 2001: 7th International Conference, CP 2001 Paphos, Cyprus, November 26–December 1, 2001 Proceedings 7*, pages 575–579. Springer, 2001.

11 Rina Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.

12 Pierre Flener, Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Symmetry in matrix models. In *Proceedings of SymCon*, volume 1. Citeseer, 2001.

13 Alan Frisch, Ian Miguel, Zeynep Kiziltan, Brahim Hnich, and Toby Walsh. Multiset ordering constraints. In *IJCAI*, volume 3, pages 221–226. Citeseer, 2003.

14 Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. Essence: A constraint language for specifying combinatorial problems. *Constraints*, 13:268–306, 2008.

15 Alan M Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Propagation algorithms for lexicographic ordering constraints. *Artificial Intelligence*, 170(10):803–834, 2006.

16 Alan M Frisch, Chris Jefferson, Bernadette Martinez-Hernandez, and Ian Miguel. Symmetry in the generation of constraint models. In *Proceedings of the international symmetry conference*, 2007.

17 Alan M Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The rules of constraint modelling. In *IJCAI*, pages 109–116, 2005.

18 Alan M Frisch, Christopher Jefferson, and Ian Miguel. Symmetry breaking as a prelude to implied constraints: A constraint modelling pattern. In *ECAI*, volume 16, page 171, 2004.

19 Ian P Gent, Tom Kelsey, Steve A Linton, Iain McDonald, Ian Miguel, and Barbara M Smith. Conditional symmetry breaking. In *International Conference on Principles and Practice of Constraint Programming*, pages 256–270. Springer, 2005.

20 Carla Gomes and Meinolf Sellmann. Streamlined constraint reasoning. In *International Conference on Principles and Practice of Constraint Programming*, pages 274–289. Springer, 2004.

21 Alan Hartman and Leonid Raskin. Problems and algorithms for covering arrays. *Discrete Mathematics*, 284(1):149–156, 2004. Special Issue in Honour of Curt Lindner on His 65th Birthday. URL: `https://www.sciencedirect.com/science/article/pii/S0012365X0400130X`, `doi:10.1016/j.disc.2003.11.029`.

22 Michal Kouril and John Franco. Resolution tunnels for improved sat solver performance. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 143–157. Springer, 2005.

23 M. Lindauer, H. Hoos, F. Hutter, and T. Schaub. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research*, 53:745–778, 2015.

24 Joao Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In *Handbook of satisfiability*, pages 133–182. ios Press, 2021.

25 Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14:357–391, 2009.

26 Gilles Pesant, Claude-Guy Quimper, and Hélène Verhaeghe. Practically uniform solution sampling in constraint programming. In Pierre Schaus, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 335–344, Cham, 2022. Springer International Publishing.

27 Steven Prestwich and J Christopher Beck. Exploiting dominance in three symmetric problems. In *Fourth international workshop on symmetry and constraint satisfaction problems*, pages 63–70. Citeseer, 2004.

28 Maxim Shishmarev, Christopher Mears, Guido Tack, and Maria Garcia de la Banda. Learning from learning solvers. In *Principles and Practice of Constraint Programming: 22nd Interna-*

tional Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings 22, pages 455–472. Springer, 2016.

29 Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. Automated streamliner portfolios for constraint satisfaction problems. *Artificial Intelligence*, 319:103915, 2023. URL: `https://www.sciencedirect.com/science/article/pii/S0004370223000619`, `doi:10.1016/j.artint.2023.103915`.

30 Peter J Stuckey. Lazy clause generation: Combining the power of sat and cp (and mip?) solving. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 5–9. Springer, 2010.

31 Mathieu Vavrille, Charlotte Truchet, and Charles Prud'homme. Solution sampling with random table constraints. *Constraints*, 27(4):381–413, 2022.

32 Weijia Wang and Michele Sebag. Multi-objective monte-carlo tree search. In *Asian conference on machine learning*, pages 507–522. PMLR, 2012.

33 Kiana Zeighami, Kevin Leo, Guido Tack, and Maria Garcia de la Banda. Towards semi-automatic learning-based model transformation. In *Principles and Practice of Constraint Programming: 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings 24*, pages 403–419. Springer, 2018.

A Portfolios of Remaining Folds

Streamliner Group	Streamliner	Avg Applicability	Overall Reduction
Coarse	287	84.0%	66.8%
	400	97.9%	65.5%
	299	88.2%	64.5%
Fine	271	0.7%	91.0%
	48	97.9%	84.3%
Fine Filtered	240	97.9%	82.4%
	24	97.2%	84.1%
	13-23	94.4%	86.5%

Table 3 Comparison of the final streamliner portfolios for each group on Fold 1.

Streamliner Group	Streamliner	Avg Applicability	Overall Reduction
Coarse	295	90.3%	67.8%
	296	93.1%	68.4%
Fine	24	97.9%	86.7%
Fine Filtered	24	97.9%	86.1%
	13-239	96.5%	83.2%

Table 4 Comparison of the final streamliner portfolios for each group on Fold 2.

Streamliner Group	Streamliner	Avg Applicability	Overall Reduction
Coarse	295	88.2%	65.5%
	400	96.5%	64.3%
	296	93.8%	67.5%
	287	81.4%	67.9%
Fine	72	97.2%	79.2%
	24	96.5%	84.5%
Fine Filtered	240	97.2%	79.2%
	24	96.5%	84.5%
	230-240	92.4%	86.5%
	13-24	94.4%	84.6%
	7-14	95.1%	82.8%

Table 5 Comparison of the final streamliner portfolios for each group on Fold 3.

Streamliner Group	Streamliner	Avg Applicability	Overall Reduction
Coarse	295	90.3%	65.8%
	391	88.2%	66.5%
Fine	96	97.9%	79.3%
	47	92.4%	76.7%
	263	61.8%	78.6%
Fine Filtered	240	98.6%	78.6%
	24	97.9%	82.1%
	13-23	95.8%	86.2%
	23-229	93.8%	85.1%

Table 6 Comparison of the final streamliner portfolios for each group on Fold 4.