Towards Portfolios of Streamlined Constraint Models: A Case Study with the Balanced Academic Curriculum Problem

Patrick Spracklen, Nguyen Dang, Özgür Akgün, Ian Miguel

School of Computer Science, University of St Andrews, St Andrews, UK
{jlps, nttd, ozgur.akgun, ijm}@st-andrews.ac.uk

Abstract. Augmenting a base constraint model with additional constraints can strengthen the inferences made by a solver and therefore reduce search effort. We focus on the automatic addition of streamliner constraints, derived from the types present in an abstract ESSENCE specification of a problem class of interest, which trade completeness for potentially very significant reduction in search. The refinement of streamlined ESSENCE specifications into constraint models suitable for input to constraint solvers gives rise to a large number of modelling choices in addition to those required for the base ESSENCE specification. Previous automated streamlining approaches have been limited in evaluating only a single default model for each streamlined specification. In this paper we explore the effect of model selection in the context of streamlined specifications. We propose a new best-first search method that generates a portfolio of Pareto Optimal streamliner-model combinations by evaluating for each streamliner a portfolio of models to search and explore the variability in performance and find the optimal model. Various forms of racing are utilised to constrain the computational cost of training. We demonstrate the approach's effectiveness on the Balanced Academic Curriculum Problem, which involves assigning periods to courses in a way that the academic load of each period is balanced.

Keywords: Constraint Programming · Streamliners

1 Introduction

Adding *streamliner* constraints [20] to a model offers a powerful means to improve the performance of a basic constraint model. Streamliners differ from implied constraints [11,14,19], which are inferred from the original model, and symmetry-breaking [18] and dominance-breaking constraints [38], both of which rule out members of equivalence classes of solutions. Streamliners are conjectured rather than inferred from the original model and are hence not guaranteed to be sound. As a result they can very significantly alter the set of solutions to a problem instance. A well chosen streamliner can substantially reduce search effort by focusing search on promising regions of the search space containing at least one solution.

```
given  n_courses ,  n_periods , load_per_period_lb ,  load_per_period_ub ,
       courses_per_period_lb ,  courses_per_period_ub  :  int (1..)

letting  Course  be domain  int (1.. n_courses ) ,
         Period  be domain  int (1.. n_periods )
given  prerequisite  :  relation  of (Course * Course ) ,
       course_load  :  function  (total)  Course  —-->  int (1..)

find  curr  :  function  (total)  Course  —-->  Period

such that
    forAll  c1 , c2  :  Course  .  prerequisite (c1 , c2 )  —>  curr (c1 )  <  curr (c2 ) ,
    forAll  p  :  Period  .
      (sum  c  in  preImage (curr , p ).  course_load (c ))  <=  load_per_period_ub
       /\
      (sum  c  in  preImage (curr , p ).  course_load (c ))  >=  load_per_period_lb ,
    forAll  p  :  Period  .
      |preImage (curr , p )|  <=  courses_per_period_ub  /\
      |preImage (curr , p )|  >=  courses_per_period_lb
```

Fig. 1. Essence specification of the Balanced Academic Curriculum Problem [22]. The goal is to design a balanced academic curriculum by assigning periods to courses in a way that the academic load of each period is balanced, i.e., as similar as possible.

Streamliner generation was originally a laborious manual process, requiring the manual inspection of small instances of a problem class to identify patterns from which to derive candidate streamliners [28,30]. This approach was successful in several domains, such as generating spatially-balanced experimental designs [20] and finding improved bounds for the Erdos discrepancy problem [29].

More recently, it has been shown how this process can be automated to derive candidate streamliners directly from the Essence specification of a problem class of interest [40,41]. Essence [15–17] is an abstract constraint specification language that allows a problem to be described directly in terms of the combinatorial structure to be found, through support for abstract type constructors like set, relation, function and partition, as well as arbitrary nesting of these, such as set of multisets. Figure 1 presents the Essence specification of the Balanced Academic Curriculum Problem (BACP), which we will use throughout. Here, the problem involves finding a total function from courses to periods.

The structure apparent in an Essence specification drives a rule-based system to suggest candidate streamliners, followed by a best-first search to identify the most promising streamliners to apply in practice. For example one of the streamliners automatically generated for BACP enforces that the *Courses* can only be placed into *even* numbered *Periods*.

The existing work [40,41] uses the Conjure automated modelling system [3, 5] to refine a streamlined Essence specification into a constraint model for evaluation. This work is limited to accepting Conjure's default choice for each of the modelling decisions that need to be made, such as how to represent abstract variables as constrained collections of more primitive variables, resulting in a single streamlined model. The hypothesis that motivates the work in this paper is that these default modelling choices are unlikely always to result in the most

effective model, and that the best modelling choices may vary according to which streamliner constraints are added to the original specification.

To test this hypothesis, we use CONJURE to refine each streamlined specification to a set of candidate models, each representing a different set of modelling choices. We augment the best-first search to look for effective streamliner/model pairs to build a portfolio of effective streamlined models for the problem class under consideration. For an unseen instance from the problem class, a model from the portfolio is automatically selected and deployed. For the BACP, our empirical results demonstrate that this often results in a significant improvement in performance over the existing method of focusing on a single default set of modelling choices.

2   Candidate Streamliner Generation

An ESSENCE specification like the one presented in Figure 1 comprises the problem class parameters (`given`); the combinatorial objects to be found (`find`); the constraints the objects must satisfy (`such that`); identifiers declared (`letting`); and an optional objective function (`min/maximising`).

The highly structured problem description an ESSENCE specification provides is better suited to streamliner generation than a lower level representation, such as MiniZinc [33] or ESSENCE PRIME [36]. This is because abstract types like function or partition and nested types like multiset of sets must be represented as a constrained collection of more primitive variables in a constraint model, obscuring the structure used to drive streamliner generation.

We employ the same set of streamliner generation rules as [40, 41]. These rules are generic, developed around the structured abstract types provided by ESSENCE. The subset that are applicable to the BACP is summarised in Table 1. Higher-order rules take another rule as an argument and lift its operation onto a decision variable with a nested domain such as the function present in the BACP. This allows for the generation of a rule that enforces that half of the *Courses* are assigned to *Periods* drawn from the lower-half of the *Periods* domain.

Class	Trigger Domain	Name
First-order	`int`	odd{even}, lower{upper}Half
	`function int --> int`	monotonicIncreasing{Decreasing} largest{smallest}First{Last}
Higher-order	`set of X`	all, most*, half, approxHalf*
	`function X --> Y`	range, defined pre{post}fix*, allBut*

Table 1. The subset of rules used to generate streamliners for BACP. The ones with a softness parameter specify a family of rules each member of which is defined by an integer parameter, and are marked with *. Note that the defined and range operators return sets, and the higher order rules that operate on sets are applicable to these sets.

3   From Essence Specifications to Constraint Models

ESSENCE specifications cannot be solved directly by systematic constraint solvers, which lack support for abstract types such as functions and partitions, and nested types. Hence, an ESSENCE specification must be *refined* into a constraint model where these types, and the constraints on them, are modelled in terms of the target constraint language. The automated modelling tool CONJURE [1, 4] refines ESSENCE specifications into constraint models in the solver-independent constraint language ESSENCE PRIME [34, 36].

There are typically many alternative models of an ESSENCE specification, corresponding to alternative modelling choices for the abstract types, and the constraints upon them, present in the specification. Different models can exhibit significantly different performance behaviour, which can also vary according to the particular instance of the problem class modelled. CONJURE is able to enumerate models of an ESSENCE specification by applying the different refinement rules corresponding to the possible modelling choices.

We use the BACP (Figure 1) to illustrate how CONJURE produces alternative models. This specification has one *find* statement with a function domain: *curr*. This can be modelled in four different ways in the current version of CONJURE. As this is a *total* function the first representation uses a 1-D matrix `matrix indexed by [int(1..n_courses)] of int(1..n_periods)`. The second model represents the function as a *total* relation. It utilises a 2-D matrix of Booleans `matrix indexed by [int(1..n_courses), int(1..n_periods)] of bool`. An alternative interpretation models the function as a binary relation first, followed by modelling the relation in two different ways. Both representations use two integer matrices `matrix indexed by [int(1..n_courses * n_periods)] of int(1..n_courses)]` for each component of the relation. They differ in how they represent the cardinality of the function, one using a single integer marker and the other one using a Boolean matrix as flags to denote whether the value is in the relation.

From this one *find* statement there are four possible base representations. However, CONJURE also has to model other types of objects such as *given* parameters, *auxiliary* variables and *quantified* variables each of which could have multiple different representations. CONJURE also has the ability to implement *automated channelling* [12] in which each reference to a variable in a constraint expression can be modelled differently. Each of these components increases the number of modelling choices and as a result a single abstract ESSENCE specification can typically be refined into a large number of constraint models. Figure 2 shows for the number of candidate models of BACP on the unstreamlined ESSENCE and how this changes with the introduction of streamliner constraints. Over 16,384 different models can be generated for the unstreamlined specification alone and with the additional modelling choices streamliners radically increase this number. Even though a large number of these models are small modifications of other models, some models are significantly different from others and they have very different performance characteristics. Moreover, different models are also likely to benefit from the addition of streamliner constraints differently.
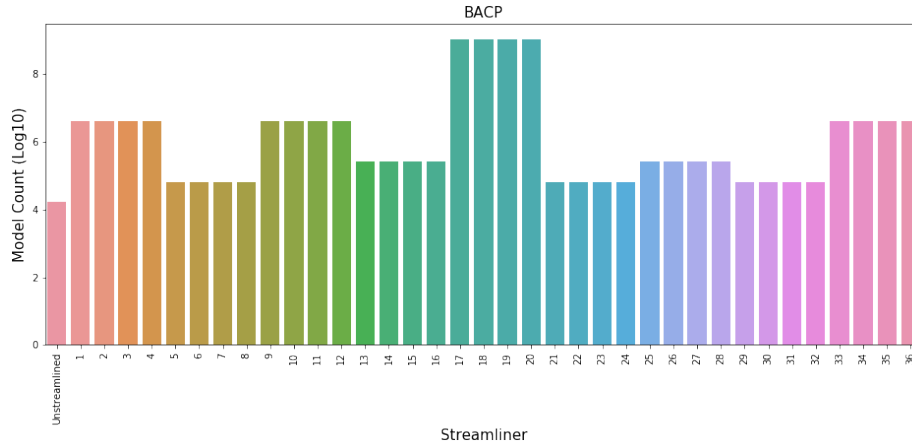
Fig. 2. The number of models (log_{10}) refined from the original ESSENCE specification and with a sample set of single candidate streamliners for BACP. The streamliners are represented in CONJURE via numeric values which are presented.

In existing work [40, 41] this choice is handled by utilising the DEFAULT heuristic of CONJURE. This is a heuristic employed during refinement to commit greedily to promising modelling choices at each point where an abstract type or a constraint expression may be refined in multiple ways [3]. The DEFAULT heuristic is a combination of two other heuristics: COMPACT for decision variables and constraint expressions and SPARSE for parameters. COMPACT favours transformations that produce smaller expressions. For an abstract type, we define an ordering as follows: concrete domains (such as bool, matrix) are smaller than abstract domains; within concrete domains, bool is smaller than int and int is smaller than matrix. Abstract type constructors have the ordering set < multiset (mset) < function < relation < partition. These rules are applied recursively so that COMPACT will select the smallest domain according to this order. The SPARSE heuristic is designed to choose the domain representations that will generate the smallest ESSENCE PRIME parameter files for sparse objects. Some representations represent every *potential* member of a domain explicitly whereas some representations only represent the actual members of a domain.

Figure 3 shows the performance of streamlined models for BACP across 8 different models with two example streamliners. Streamliner 3 enforces that the smallest integer courses must be assigned to the smallest integer periods, while Streamliner 10 enforces that half of the courses must be placed in even numbered periods. Results in Figure 3 indicates that there can be a large variability in the streamliner performance across different model representations. Moreover, the DEFAULT heuristic may not always provide the best performance, as for streamliner 10 the DEFAULT model is only able to achieve around a 20% reduction in time, far inferior to the 60% reduction achieved by the best model. It is

also interesting to note that the best model representation is highly streamliner dependent and changes between streamliner 10 and streamliner 3.
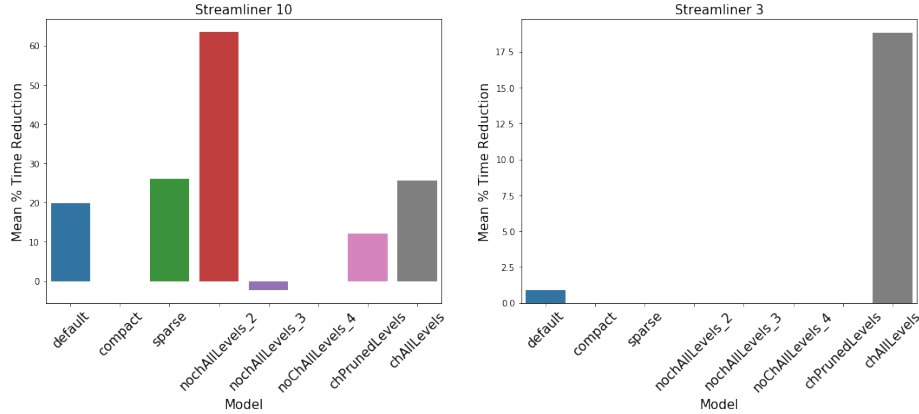
Fig. 3. Performance of two example streamliners across 8 models for BACP. Each model is named with respect to the generating heuristic (table 2)

4   Model Portfolios

As seen from Figure 2 the number of possible models that can be refined, especially from a streamlined specification, means that refining and evaluating all possible models for each streamliner is not feasible. Instead, a sample of N models can be evaluated where N is controlled to balance the computational cost against the exploratory benefit. CONJURE allows for the customisation of the modelling process in several ways. First, different strategies can be employed for the modelling of different types of objects. A distinction can be made among four different forms of declarations, {*find, given, auxiliary, quantified*} depending upon their origin in the specification. *find* and *given* statements define decision variables and problem parameters, *auxiliaries* are decision variables created by CONJURE during model reformulation, and *quantified variables* are defined by quantified expressions like `forAll`, `exists` and `sum`. Second, automated channelling and precedence levels can be activated and deactivated to further customise the modelling heuristic. To produce a portfolio of N models we use a set of heuristics, as listed in Table 2, ranked based on the predicted number of generated models and our perceived effectiveness of the generated models. These heuristics are then invoked in an iterative fashion until a portfolio of N models has been generated.

Rank	Heuristic	Declaration Type				Additional Flags	
		Find	Given	Auxiliaries	Quantifieds	Channelling	Levels
1	DEFAULT	COMPACT	SPARSE	COMPACT	COMPACT	False	True
2	COMPACT	COMPACT	COMPACT	COMPACT	COMPACT	False	True
3	SPARSE	SPARSE	SPARSE	SPARSE	SPARSE	False	True
4	nochPrunedLevels	All	SPARSE	COMPACT	COMPACT	False	True
5	nochAllLevels	All	SPARSE	COMPACT	COMPACT	False	False
6	chPrunedLevels	All	SPARSE	COMPACT	COMPACT	True	True
7	chAllLevels	All	SPARSE	COMPACT	COMPACT	True	False
8	fullPrunedLevels	All	SPARSE	All	All	True	True
9	fullAllLevels	All	SPARSE	All	All	True	False
10	fullParamsPrunedLevels	All	All	All	All	True	True
11	fullParamsAllLevels	All	All	All	All	True	False

Table 2. The ranked set of heuristics we use when generating a portfolio of models.

5   Searching for Streamlined Model Portfolios

5.1   Multi-Objective Monte Carlo Tree Search

Candidate streamliners are often most effectively used in combination [20], forming a lattice of possible combinations [43], and therefore a search allowing good combinations to be identified efficiently is desirable. In reality, streamliner generation has two conflicting goals: to uncover constraints that steer search towards a small and highly structured area of the search space that yields a solution, versus identifying streamliner constraints in training that remain applicable to more difficult instances of the problem class.

To address these two objectives we employ a multi-objective optimisation approach, where each point x in the search space X is associated with a 2-dimensional reward vector r_x in $\mathbb{R}^2$. Our two objectives allow us explicitly to balance considerations of average applicability against how aggressively the streamlined version of each candidate model reduces search: (i) *Applicability* - the proportion of training instances for which the streamlined model admits a solution. (ii) *Search Reduction* - the mean reduction in time to find a solution in comparison with an unstreamlined model. All objectives are transformed such that they can be maximised. With these two objectives for each streamliner combination, the Pareto dominance partial ordering $\succ$ on $\mathbb{R}^2$ is defined as follows. Given $x, y \in X$ with vectorial rewards $r_x = \langle r_x^1, r_x^2 \rangle$ and $r_y = \langle r_y^1, r_y^2 \rangle$:

$$r_x \succ r_y \iff (\forall i \in [1,2], r_x^i \geq r_y^i) \wedge (\exists j \in [1,2], r_x^j > r_y^j) \tag{1}$$

To search the lattice structure for a portfolio of Pareto optimal streamlined models we have adapted the *Dominance-based Multi-Objective Monte Carlo Tree Search (MOMCTS-DOM)* algorithm [42]. The algorithm has four phases:

1. **Selection**: Starting at the root node, the Upper Confidence Bound applied to Trees (UCT) [8] policy is applied to traverse the explored part of the lattice until an unexpanded node is reached.
2. **Expansion**: Uniformly select and expand an admissible child.

3. **Simulation**: For each of a portfolio of N models the collection of streamliners associated with the expanded node are evaluated. The vectorial reward ⟨Applicablity, Search Reduction⟩ across the set of training instances is calculated and returned.

4. **BackPropagation**: The current portfolio, which contains the set of non dominated streamliner/model combinations found up to this point during search, is used to compute Pareto dominance. The reward values of the Pareto dominance test are non stationary since they depend on the portfolio, which evolves during search. Hence, we use the cumulative discounted dominance (CDD) [42] reward mechanism during reward update.

5.2   Training Set Construction

It is necessary to have a training set with which to evaluate the streamliner candidates. We use the automated instance generation system proposed in [2] to generate several satisfiable instances, solvable in a time limit of $[10s, 300s]$. Starting from an Essence specification of a problem class, instances can be generated in a completely automated fashion. Ideally the training set should be diverse and representative of the problem class as a whole, otherwise the streamliner portfolio may not generalise. Hence, three different instance generation runs per problem class are performed to ensure diversity. Instance-specific features are used to judge instance similarity and GMeans clustering [21] is employed to detect the number of instance clusters present. Instances are then selected per cluster to build a representative training set. For instance features, we use the 95 FlatZinc features provided by the `fzn2feat` tool (part of `mzn2feat` [6]).

5.3   Multi-Round Search

One issue with the search method presented thus far is that because of the domination criteria used the resultant portfolio may still be sub-optimal. A streamliner/model combination that has poor average performance across all instances it covers may still be useful, as it might provide very high solving-time reduction within a subset of those instances. However, this information is not considered in the current dominance criteria, and will be discarded.

In order to solve this issue, we adapt our search to incorporate the ideas of Hydra [44], a portfolio builder approach that automatically builds a set of solvers or parameter configurations of solvers with complementary strengths. Instead of performing one lattice search and building one portfolio, we now perform multiple rounds of search. In the first, a normal MOMCTS-DOM is conducted. From the second round onward, the best solving time on each instance obtained from the portfolio created in previous rounds is recorded. Performance of a streamliner/model combination on an instance in the current round is then measured relative to the previously recorded best solving time on that instance. This mechanism allows the portfolio built in the current round to complement the strengths of the combined portfolios built in the prior rounds.

6 Model Racing

Evaluating all candidate models for each considered streamliner combination is expensive. Poorly-performing streamliner/model combinations can timeout on several instances and consume a large amount of the training budget. Hence, we employ various *racing* techniques [3,7,9,25] to terminate poorly-performing combinations early. The streamliner search can then allocate more time to evaluate promising streamliners and gain a more accurate estimation of their performance.

Racing is performed at each lattice node of the search, representing a combination of streamliners. We race among the multiple models generated by CONJURE on the given streamliner combination, and try to eliminate the poorly performing models without having to fully evaluate them. We first describe the three racing strategies we used, ρ-*Capping* [3], *racing based on statistical tests* [7, 32] and *adaptive capping* [25] and then discuss how they are combined together in our streamliner search through a multi-level model generation approach.

6.1 ρ-Capping

We conduct a race across multiple models for each training instance. Given a parameter $\rho \geq 1$, a model is ρ-*dominated* [3] on an instance by another model if the solving time of the latter is at least ρ times faster on the given instance. All models enter each race, but a model is terminated as soon as it is $\rho-$dominated by some other model. Hence, the running time of a model on an instance is *capped* by ρ times the best solving time of all models in the race. In our experiments ρ is set as 2.

6.2 Racing using statistical tests

Racing using statistical tests for the automated configuration of parameterised algorithms [23] was first proposed in [7], and later extended and implemented in the automated algorithm configuration tool IRACE [32]. Bad algorithm configurations are discarded from a race as soon as sufficient statistical evidence is observed. We apply the same idea to remove bad models early during a race.

At the beginning of a race, all models generated by CONJURE at the current lattice node are evaluated on a number of instances, denoted by T_{first}, and their solving times are measured. The model with the best average solving time on those instances is identified, and a *paired Student t-test* (with a significance level of 0.05) between that model and each of the other models is conducted. The *Bonferroni correction* is used for adjusting the multiple-comparison statistical tests. Models showing statistically significantly worse performance than the current best-average one are eliminated. The survivors continue to be evaluated on an additional number of T_{next} instances before a new best model is calculated and statistical tests are applied again. This is repeated until all training instances are examined. We use IRACE's default values for T_{first} (10) and T_{next} (5).

6.3   Adaptive Capping

Adaptive capping is another technique in automated algorithm configuration to terminate poorly-performing algorithm configurations early when optimising running time of a parameterised algorithm. It was first proposed in the local search-based automated algorithm configuration tool ParamILS [25], and was later integrated into IRACE [9]. The technique has been shown to significantly speed up the search for the best algorithm configurations in many cases [9,24,25]. Our streamliner search makes use of the adaptive capping mechanism of IRACE.

The main idea is to reduce the time wasted in the evaluation of poorly performing models in the current race by enforcing an upper bound on their running time. Bounds are calculated based upon a set of *elite* models (best-performing models) obtained from a previous race. More specifically, let M denote a set of candidate models in the current race, $I = \{i_1, i_2, .., i_n\}$ a set of instances, M_E a set of elite models and p_k^m the average solving time of a model m on the instance subset $\{i_1, i_2, .., i_k\}$ $(k \leq n)$. Given the fact that we already know the values of $p_k^{m_e}$ for all $m_e \in M_E$ and $k \leq n$ from the previous race, the maximum running time b_{k+1}^m $(k < n)$ for a model $m \in M$ on instance i_{k+1} is calculated as: $median_{m_e \in M_E}\{p_{k+1}^{m_e}\} * (k + 1) - p_k^m * k$. This bound can be thought of as the minimum performance a current model must achieve on an instance subset to be competitive with the elites up to that point. If the running time of a model ever exceeds the calculated bound, it is eliminated from the race. As the bound for an instance is calculated based upon the time taken on preceding instances, adaptive capping can be sensitive to the ordering of the instance set I. Therefore, the training instance set is shuffled at the beginning of every race.

6.4   Multi-Level Model Generation

At each lattice node of the streamliner search, the three forms of racing are combined in a multi-level model generation approach. On each level, a portfolio of models is generated and a new race is started. The size of the portfolio is doubled with each level and the model set created is a strict superset of the models in any preceding level. On the first level a single model is generated (DEFAULT model) and is evaluated on the whole training set. Since it is currently the only model, it becomes an elite for the subsequent level. The second level is then started with a portfolio of size 2, containing DEFAULT and a newly created model. On this level all three racing techniques are utilized to eliminate poorly performing models. Adaptive capping will make use of the elite models from the preceding level to calculate its bound and perform its elimination. Any results from previous levels that have already been calculated are cached and reused. In our experiments, we allow a maximum of four levels, which results in a maximum number of 8 streamlined models at each lattice node.

Iterative deepening of the portfolio size allows us to initially evaluate the models that we believe are most likely to perform best based upon the ranking in Table 2. The use of elite models derived from previous levels helps the adaptive capping mechanism to speed up the evaluations in the next levels substantially. This allows the search to explore larger portfolio sizes within a reasonable

amount of time. It is especially useful where the truly best model for a particular streamliner combination is actually located in a lower ranked heuristic.

7   Streamliner/Model Selection

The generated portfolio for a problem class contains streamliner/model combinations with complementary strengths. For an unseen instance the question arises as to which streamlined model should be used. This is a common situation in many AI domains. Algorithm Selection [39] uses instance characteristics to select from a set of algorithms the ones expected to solve a given problem instance most efficiently, and has been shown empirically to improve the state-of-the-art in several domains [26, 27]. We exploit existing Algorithm Selection techniques to effectively select from our streamliner portfolios for unseen test instances.

We use Autofolio [31], an automatically self-configured algorithm selection framework. Given a particular problem instance the goal is to have Autofolio predict, based upon the features of the instance, which streamlined models from the generated portfolio will most efficiently solve the instance. For instance features, we again use the FlatZinc features provided by the `fzn2feat` tool [6]. Autofolio also supports a *pre-solving schedule*, a static schedule built from a small subset of streamlined models. This schedule is run for a small amount of time. If it fails to solve an instance, the model chosen by the prediction model is applied. Autofolio chooses whether to use a pre-solving schedule during its tuning phase. For each problem class, Autofolio was run ten times in tuning mode with a one day CPU budget. 10-fold cross validation was used on the training sets and the model with the best average cross-validation score was used for prediction.

8   Experimental Results

Experiments were run on a cluster of 280 nodes, each with two 2.1 GHz, 18-core Intel Xeon E5-2695 processors. MOMCTS-DOM was run with leaf parallelisation [10] using up to 30 cores with a maximum budget of 4 wall-time days per problem class. Difficult test instances, with solving time in $[300s, 3600s]$, were generated automatically using the same instance generation system [2] used for training instance construction. We will show that although the portfolios of streamlined models are trained on easy instances (solved in $[10s, 300s]$), their large improvement in solving time transfers to difficult, unseen test instances.

Previous automated streamlining approaches have been limited to using only a single DEFAULT model for each streamlined specification. To demonstrate the positive impact of adding multiple-model exploration during streamliner search, two independent searches were conducted for each problem class: one with the single default model, and one with the multiple-model approach proposed in this paper. The generated portfolio of streamlined models from each search was then given to Autofolio to build a streamlined-model selector using a set of training instances. Performance of each streamliner search is measured as the performance of the selector over the test instances. Both searches were given

the same training time budget for a fair comparison. Our hypothesis is that even though each round of a multiple-model search is generally more expensive than its single-model counterpart, the opportunity of exploring multiple models and our enhanced search method using model-racing will be able to produce higher-quality portfolios of streamlined models within the same time budget.

The CP learning solver CHUFFED [13] was used with its default parameter setting as the target solver. Each streamlined model was generated by CONJURE and further tailored for the target solver by SAVILE ROW [35,37]. SAVILE ROW was also used to produce the FlatZinc inputs for the `fzn2feat` feature extraction tool.

The evaluation of a test instance proceeds as follows: Initially the features are extracted (by `fzn2feat`) and used by Autofolio to predict the best streamliner for that part of the instance space. The selected streamliner is then evaluated for a maximum time period of one hour. As streamliners are not necessarily sound and no Algorithm Selection system is perfect in its predictions there is no guarantee that the streamliner will find a solution and can either time out or be proven unsatisfiable by the solver. In either case after the streamliner is finished we revert back to running the original model until a solution is found. The overall time for a streamliner is then calculated as follows:

$$TotalTime = \begin{cases} \text{SAT:} & \text{streamlined solving time} \\ \text{UNSAT/timeout:} & \text{streamlined + original model solving time} \end{cases}$$

Table 3 shows our experimental results of the single-model and multiple-model settings on the 25 difficult test instances generated. We report results of Autofolio and the Virtual Best Solver (VBS) for each setting using three measurements. *%imp* and *%red* represent the percentage of test instances where the solving time is improved by adding streamliners and the average solving-time reduction percentage on those improved instances respectively. *speedup* is calculated as the average solving time of the original model divided by the average streamlined *TotalTime* across all test instances.

Experiment	#inst	VBS			Autofolio		
		%imp	%red	speedup	%imp	%red	speedup
BACP (D)	25	92.0	95.06	8.31	72.0	96.0	1.23
BACP (MM)	25	**96.0**	98.7	**14.6**	**92.0**	92.0	**3.6**

Table 3. Results of the Virtual Best Solver (VBS) and Autofolio on 25 test instances for BACP, under the DEFAULT model setting (D) and the multi-model (MM) setting. *%imp* and *%red* represent the percentage of test instances where the solving time is reduced by adding streamliners and the average solving-time reduction percentage on those improved instances respectively. *speedup* represents the relative performance between the average solving time of the unstreamlined model in comparison to the two streamlined approaches.

Both streamlined approaches provide large improvement in solving time to the original unstreamlined model. Compared to the single-model approach, the multi-model search results in higher percentages of instances with improvement in search time (*%imp*) and higher speedup values. These results are achieved both on the VBS and with Autofolio which shows that the multi-model portfolios not only performing better in theory (VBS) but also that these theoretical gains can substantially be realised in practice by an Algorithm Selector (Autofolio).

Figure 4 presents the composition of the portfolio of models produced for BACP. It is immediately apparent that Conjure's default heuristic, which is the only one employed in previous work [40], is not the one mostly used by the portfolio. In what follows we discuss the composition the portfolio in more details and its connection to the performance improvement observed.
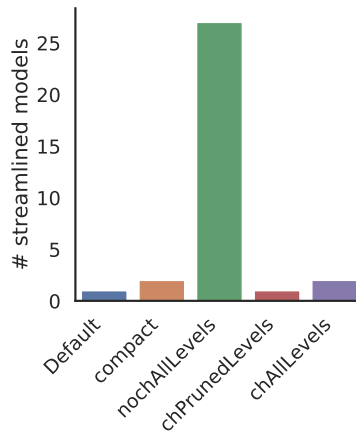
Fig. 4. Composition of the portfolio generated by our multi-model search for BACP. The x-axis represents heuristic rules (Table 2) used by Conjure. The y-axis shows the number of streamlined models in the portfolio generated by the corresponding heuristics rules. Note that models refined according to the same set of rules can still differ due to the additional modelling choices introduced by the added streamliners.

The generated portfolio is comprised of various models generated from five different heuristics. Some models have both *channelling* and *precedence levels* for representations enabled whereas others have both disabled. Those modelling choices interact with the candidate streamliners and enable efficient propagation during the search, leading to the improvement in all performance measurements as shown in Table 3.

To give a concrete example Streamliner 1 in BACP enforces that the *curr* function is monotonically increasing. For this streamliner the best model enables *channelling*. The default model chooses only one representation for the *curr* function `matrix indexed by [int(1..n_courses)] of int(1..n_periods)`

whereas in the dominating model a second representation is also used `matrix indexed by [int(1..n_courses), int(1..n_periods)] of bool` and CONJURE will automatically add channelling constraints between these two representations. Having these two representations present in the model makes a large difference in the reduction achieved by Streamliner 1. Another example is for Streamliner 10 which restricts half of the values of the range of the *curr* function to be even. In this case, representing the function as a 2-dimensional Boolean matrix (as above) instead of a 1-dimensional integer matrix allows the streamliner to again achieve a higher overall average reduction.

9   Conclusion

We have demonstrated that searching for the right model to pair with a streamliner can improve performance significantly over a fixed, default model using a case study with the Balanced Academic Curriculum Problem. Racing can be used to constrain the computational search cost by removing inferior models early. We have also shown for the first time that Algorithm Selection tools can be used to effectively select streamliners on a per-instance basis. Future work is to extend the experiments to other problems, to increase selection performance to close the gap on the VBS, and to investigate additional modelling choices such as when and how it is beneficial to encode streamliners to different representations such as SAT or SMT.

References

1. Akgün, Ö.: Extensible automated constraint modelling via refinement of abstract problem specifications. Ph.D. thesis, University of St Andrews (2014)
2. Akgün, Ö., Dang, N., Miguel, I., Salamon, A.Z., Stone, C.: Instance generation via generator instances. In: International Conference on Principles and Practice of Constraint Programming. pp. 3–19. Springer (2019)
3. Akgun, O., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in conjure. In: International Conference on Principles and Practice of Constraint Programming. pp. 107–116. Springer (2013)
4. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Breaking conditional symmetry in automated constraint modelling with Conjure. In: ECAI. pp. 3–8 (2014)
5. Akgün, Ö., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence. pp. 4–11. AAAI Press (2011)

6. Amadini, R., Gabbrielli, M., Mauro, J.: An enhanced features extractor for a portfolio of constraint solvers. In: SAC 2014: Proceedings of the 29th Annual ACM Symposium on Applied Computing. pp. 1357–1359. ACM (2014). https://doi.org/10.1145/2554850.2555114, code available from `https://github.com/CP-Unibo/mzn2feat`

7. Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K., et al.: A racing algorithm for configuring metaheuristics. In: Gecco. vol. 2 (2002)

8. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P.I., Tavener, S., Perez, D., Samothrakis, S., Colton, S., et al.: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI (2012)

9. Cáceres, L.P., López-Ibáñez, M., Hoos, H., Stützle, T.: An experimental study of adaptive capping in irace. In: International Conference on Learning and Intelligent Optimization. pp. 235–250. Springer (2017)

10. Cazenave, T., Jouandeau, N.: On the parallelization of uct (2007)

11. Charnley, J., Colton, S., Miguel, I.: Automatic generation of implied constraints. In: ECAI. vol. 141, pp. 73–77 (2006)

12. Cheng, B., Choi, K.M.F., Lee, J.H.M., Wu, J.: Increasing constraint propagation by redundant modeling: an experience report. Constraints **4**(2), 167–192 (1999)

13. Chu, G., de la Banda, M., Stuckey, P.: Exploiting subproblem dominance in constraint programming. Constraints **17**(1), 1–38 (2012). https://doi.org/10.1007/s10601-011-9112-9, code available from `https://github.com/chuffed/chuffed`

14. Colton, S., Miguel, I.: Constraint generation via automated theory formation. In: International Conference on Principles and Practice of Constraint Programming. pp. 575–579. Springer (2001)

15. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The Essence of Essence. Modelling and Reformulating Constraint Satisfaction Problems p. 73 (2005)

16. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The design of ESSENCE: A constraint language for specifying combinatorial problems. In: IJCAI. vol. 7, pp. 80–87 (2007)

17. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008)

18. Frisch, A.M., Jefferson, C., Martinez-Hernandez, B., Miguel, I.: Symmetry in the generation of constraint models. In: Proceedings of the international symmetry conference (2007)

19. Frisch, A.M., Miguel, I., Walsh, T.: Cgrass: A system for transforming constraint satisfaction problems. In: Recent Advances in Constraints, pp. 15–30. Springer (2003)

20. Gomes, C., Sellmann, M.: Streamlined constraint reasoning. In: International Conference on Principles and Practice of Constraint Programming. pp. 274–289. Springer (2004)

21. Hamerly, G., Elkan, C.: Learning the k in k-means. In: Advances in neural information processing systems. pp. 281–288 (2004)

22. Hnich, B., Kiziltan, Z., Walsh, T.: CSPLib problem 030: Balanced academic curriculum problem (bacp). `http://www.csplib.org/Problems/prob030`

23. Hoos, H.H.: Automated algorithm configuration and parameter tuning. In: Autonomous search, pp. 37–71. Springer (2011)

24. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: International conference on learning and intelligent optimization. pp. 507–523. Springer (2011)

25. Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: Paramils: an automatic algorithm configuration framework. Journal of Artificial Intelligence Research **36**, 267–306 (2009)

26. Kerschke, P., Hoos, H.H., Neumann, F., Trautmann, H.: Automated algorithm selection: Survey and perspectives. Evolutionary computation **27**(1), 3–45 (2019)

27. Kotthoff, L.: Algorithm selection for combinatorial search problems: A survey. In: Data Mining and Constraint Programming, pp. 149–190. Springer (2016)

28. Kouril, M., Franco, J.: Resolution tunnels for improved sat solver performance. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 143–157. Springer (2005)

29. Le Bras, R., Gomes, C.P., Selman, B.: On the erdős discrepancy problem. In: International Conference on Principles and Practice of Constraint Programming. pp. 440–448. Springer (2014)

30. LeBras, R., Gomes, C.P., Selman, B.: Double-wheel graphs are graceful. In: IJCAI. pp. 587–593 (2013)

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

32. López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L.P., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives **3**, 43–58 (2016)

33. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: International Conference on Principles and Practice of Constraint Programming. pp. 529–543. Springer (2007)

34. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in savile row. Artificial Intelligence **251**, 35–61 (2017)

35. Nightingale, P., Akgün, O., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. Artificial Intelligence **251**, 35–61 (2017). https://doi.org/10.1016/j.artint.2017.07.001

36. Nightingale, P., Rendl, A.: Essence' description. arXiv preprint arXiv:1601.02865 (2016)

37. Nightingale, P., Spracklen, P., Miguel, I.: Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. In: CP. pp. 330–340. LNCS 9255, Springer (2015)

38. Prestwich, S., Beck, J.C.: Exploiting dominance in three symmetric problems. In: Fourth international workshop on symmetry and constraint satisfaction problems. pp. 63–70. Citeseer (2004)

39. Rice, J.R.: The algorithm selection problem. In: Advances in computers, vol. 15, pp. 65–118. Elsevier (1976)

40. Spracklen, P., Akgün, Ö., Miguel, I.: Automatic generation and selection of streamlined constraint models via monte carlo search on a model lattice. In: International Conference on Principles and Practice of Constraint Programming. pp. 362–372. Springer (2018)

41. Spracklen, P., Dang, N., Akgün, Ö., Miguel, I.: Automatic streamlining for constrained optimisation. In: International Conference on Principles and Practice of Constraint Programming. pp. 366–383. Springer (2019)

42. Wang, W., Sebag, M.: Hypervolume indicator and dominance reward based multi-objective monte-carlo tree search. Machine learning **92**(2-3), 403–429 (2013)
43. Wetter, J., Akgün, Ö., Miguel, I.: Automatically generating streamlined constraint models with Essence and Conjure. In: International Conference on Principles and Practice of Constraint Programming. pp. 480–496. Springer (2015)
44. Xu, L., Hoos, H., Leyton-Brown, K.: Hydra: Automatically configuring algorithms for portfolio-based selection. In: Twenty-Fourth AAAI Conference on Artificial Intelligence (2010)