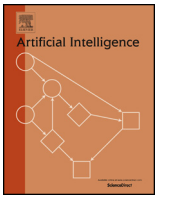
Artificial Intelligence 319 (2023) 103915

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint



Automated streamliner portfolios for constraint satisfaction problems

Patrick Spracklen, Nguyen Dang, Özgür Akgün *, Ian Miguel

School of Computer Science, University of St Andrews, St Andrews, Fife, KY16 9SX, UK

ARTICLE INFO

Article history:
Received 22 January 2022
Received in revised form 20 March 2023
Accepted 24 March 2023
Available online 29 March 2023

Keywords:
Constraint programming
Constraint modelling
Constraint satisfaction problem
Algorithm selection

ABSTRACT

Constraint Programming (CP) is a powerful technique for solving large-scale combinatorial problems. Solving a problem proceeds in two distinct phases: modelling and solving. Effective modelling has a huge impact on the performance of the solving process. Even with the advance of modern automated modelling tools, search spaces involved can be so vast that problems can still be difficult to solve. To further constrain the model, a more aggressive step that can be taken is the addition of streamliner constraints, which are not guaranteed to be sound but are designed to focus effort on a highly restricted but promising portion of the search space. Previously, producing effective streamlined models was a manual, difficult and time-consuming task. This paper presents a completely automated process to the generation, search and selection of streamliner portfolios to produce a substantial reduction in search effort across a diverse range of problems. The results demonstrate a marked improvement in performance for both Chuffed, a CP solver with clause learning, and lingeling, a modern SAT solver.

© 2023 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

1. Introduction

Challenging combinatorial problems, from domains such as planning, scheduling, packing or configuration, often form *problem classes*: families of problem instances related by a shared high-level specification, with a common set of free parameters. Constraint Programming (CP) and Propositional Satisfiability solving (SAT) offer powerful, complementary means to solve these problem classes. For either formalism, a *model* must be formulated, which describes the problem class in a format suitable for input to the intended solver. Since the search spaces involved can be vast, however, sometimes the model initially formulated for a problem class may give instances where it is too difficult for the solver to find a solution in a timely manner

In response, a natural step is to constrain the model further in order to strengthen the inferences the solver can make, therefore detecting dead ends in the search earlier and reducing overall search effort. One approach is to add *implied* constraints, which can be inferred from the initial model and are therefore guaranteed to be sound. Manual [1,2] and automated [3–5] approaches to generating implied constraints have been successful. Other approaches include adding *symmetry-breaking* [6–9] and *dominance-breaking* constraints [10–12], both of which rule out members of equivalence classes of solutions while preserving at least one member of each such class.

* Corresponding author.
E-mail addresses: jlps@st-andrews.ac.uk (P. Spracklen), nttd@st-andrews.ac.uk (N. Dang), ozgur.akgun@st-andrews.ac.uk (Ö. Akgün), ijm@st-andrews.ac.uk (I. Miguel).

https://doi.org/10.1016/j.artint.2023.103915
0004-3702/© 2023 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

```
given n_cars, n_classes, n_options : int(1..)

letting Slots  be domain int(1..n_cars),
        Class  be domain int(1..n_classes),
        Option be domain int(1..n_options)

given quantity : function (total) Class  --> int(1..),
      maxcars  : function (total) Option --> int(1..),
      blksize  : function (total) Option --> int(1..),
      usage    : relation of (Class * Option)

find car : function (total) Slots --> Class

such that
  forAll c : Class . |preImage(car,c)| = quantity(c),
  forAll opt : Option .
    forAll s : int(1..n_cars+1-blksize(opt)) .
      (sum i : int(s..s+blksize(opt)-1) . toInt(usage(car(i),opt)))
      <= maxcars(opt)
```

Fig. 1. Essence specification of the Car Sequencing Problem [14], shown to be NP-complete [15]. A number of cars (`n_cars`) are to be produced; they are not identical, because different classes (`n_classes`) are available (`quantity`) as variants on the basic model. The assembly line has different stations which install the various options (`n_options`) such as air conditioning and sun roof (each class of cars requires certain options, represented by `usage`). A maximum number of cars (`maxcars`) requiring a certain option can be sequenced within a consecutive subsequence block (`blksize`), otherwise the station will not be able to cope.

If these techniques are inapplicable, or improve performance insufficiently, for satisfiable problems a more aggressive step is to add *streamliner* constraints [13], which are not guaranteed to be sound but are designed to focus effort on a highly restricted but promising portion of the search space. Streamliners trade the completeness (i.e. failing to find a solution when there is one) offered by implied, symmetry-breaking and dominance-breaking constraints for potentially much greater search reduction.

Previously, producing effective streamlined models was a difficult and time-consuming task. It involved manually inspecting the solutions of small instances of the problem class in question to identify patterns to use as the basis for streamliners [13,16–18]. For example, Gomes and Sellmann [13] added a streamliner requiring a Latin Square structure when searching for diagonally ordered magic squares.

The principal contribution of this paper is to demonstrate *how a powerful range of streamliners can be generated and applied automatically*. Our approach is situated in the automated constraint modelling system Conjure [19–21]. This system takes as input a specification in the abstract constraint specification language Essence [22,23]. Fig. 1 presents an example specification, which asks us to sequence cars on a production line so as not to exceed the capacity of any station along the line, each of which installs an option such as sun roof. Essence supports a powerful set of type constructors, such as set, multi set, function and relation, hence Essence specifications are concise and highly structured. Existing constraint solvers do not support these abstract decision variables directly. Therefore we use Conjure to *refine* abstract constraint specifications into concrete constraint models, using constrained collections of primitive variables (e.g. integer, boolean) to represent the abstract structure. The constraint modelling assistant tool Savile Row [24,25] is then used for producing solver dependent input. Savile Row supports several solving paradigms, including CP, SAT and SMT (satisfiability modulo theories).

Our method exploits the structure in an Essence specification to produce streamlined models automatically, for example by imposing streamlining constraints on the function present in the specification in Fig. 1. The modified specification is refined automatically into a streamlined constraint model by Conjure. Identifying and adding the streamlining constraints at this level of abstraction is considerably easier than working directly with the constraint model, which would involve first recognising (for example) that a certain collection of primitive variables and constraints together represent a function – a potentially very costly process. Moreover, recovering high-level information from a low-level model expressed in a lower level constraint modelling language like OPL [26], MiniZinc [27] or Essence Prime [28] would be brittle with respect to the exact heuristics and modelling reformulations used inside Conjure and Savile Row. As with automated symmetry breaking during modelling [29,20], automated streamlining therefore motivates the adoption of a higher level language such as Essence and letting automated tools work out the best compilation – just as has happened in general programming languages.

Our streamlining system completely automates the original manual process defined by Gomes and Sellmann [13]. As per their method, it does require an initial investment in generating and testing streamliners for the problem class at hand, but this effort is repaid in two ways. First, we assume a context common across automated algorithm selection [30] and machine learning in general: we expect to solve a large number of instances of the problem class, and so the effort made to formulate the best model that we can is amortised over that substantial solving effort. Second, successful streamlining can result in a vast reduction in search effort, allowing us to solve much harder instances than would otherwise be practically feasible. Our work significantly expands previous work on automated streamliner generation from high-level problem specifications [31] and automatic selection of streamlining constraints [32].

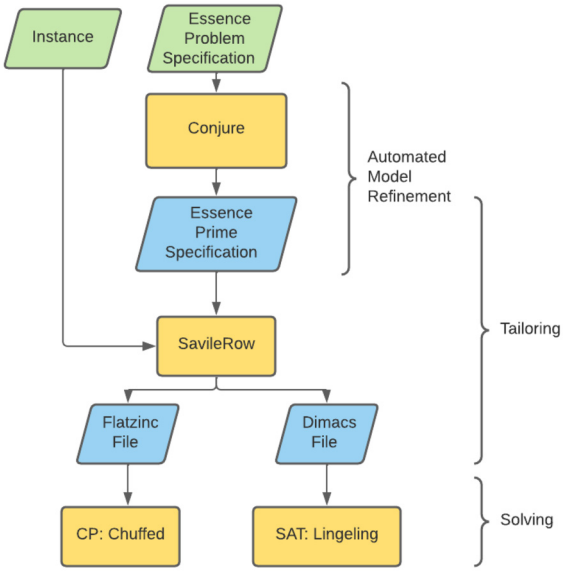


Fig. 2. Solving a problem instance using a streamlined Essence specification. Conjure is run once for a streamlined model, whereas Savile Row and the selected solver is run once per instance. The streamliner that is provided as input to Conjure is generated by a separate call to Conjure as part of Phase 1 (Candidate Streamliner Generation).

We demonstrate the effectiveness of our approach on both the CP and SAT solving paradigms, choosing representative solvers for each. For CP, we use the learning solver chuffed [33], and for SAT, we use lingeling [34]. As presented in Section 8, our method can often produce a substantial reduction in search effort across a diverse range of problems. The automated streamlining system, Essence problem specifications and all the data used in this work for computational evaluation are available at https://www.github.com/stacs-cp/automated-streamliner-portfolios (see [35]).

2. Architecture overview

We begin with an overview of the architecture of our system, before explaining each of its components in detail in the subsequent sections. Given a problem class of interest, our streamlining approach proceeds in three main phases. Firstly, candidate streamliners are generated from an Essence specification (Section 2.1). Secondly, streamliners are combined into a portfolio of streamliner combinations with complementary strengths (Section 2.2). Finally, given an unseen instance of the problem class, one or more streamliner combinations are selected from the portfolio and scheduled for use in solving (Section 2.3).

2.1. Phase 1: candidate streamliner generation

Given an Essence specification of a problem class, several candidate streamliners are automatically derived via a set of prebuilt rules encoded inside Conjure. Those rules define patterns that match against the types of the decision variables in the Essence specification. As an example, a candidate streamliner for the Car Sequencing problem in Fig. 1 can enforce *approximatelyHalf* of the *range* of the *car* function to take *odd* values only, hence reducing the search space. A full description of the rules and the streamliner generation process are presented in Section 3. The performance of a streamlined model on a given problem instance can then be evaluated using the procedure described in Fig. 2.

2.2. Phase 2: portfolio construction

The generated candidate streamliners can be combined to form stronger streamliners [36], i.e., multiple streamliners can be added to the original problem specification at the same time to form a streamlined model. This results in a potentially very large number of possible (combined) streamliners for a given problem class. Each of those streamliners may drastically reduce the search space and lead to large speed up in solving time. However, in contrast to other space-pruning techniques, such as symmetry-breaking or implied constraints, we cannot expect streamliners to be universally applicable. As illustrated in Section 5.2.1, the effectiveness of a streamliner can vary widely among instances of the same problem class. For example, a streamliner can be very useful for solving some instances but impair performance on others or even render them unsatisfiable. Therefore, we need an effective mechanism to search in the large space of streamliner combinations and identify the effective and sound ones.

Given a problem class, our system aims at constructing a *portfolio* of streamliners with *complementary strengths*, i.e., each streamliner is specialised towards a region of the *instance space* (i.e., the high-dimensional space defined by instance features [37]). Our portfolio construction method consists of three stages. Firstly, a set of training instances that is representative of the instance space of the given problem is automatically built (Section 5). Secondly, a Monte Carlo Tree Search method is used for searching in the space of the candidate streamliner combinations (Section 6.1 and Section 6.2). Finally, a portfolio builder approach is applied on top of the search process to enhance the complementary strength of the constructed portfolio (Section 6.3).

2.3. Phase 3: streamliner selection and application

Once a streamliner portfolio has been constructed, it can be used for solving any unseen instance of the same problem class. In Section 7, we discuss several methods to select and apply a streamliner portfolio to a new problem instance. We start with the simplest approaches where streamliners are selected based on their average performance across the whole training instance set. We then investigate a learning-based approach [38] where a prediction model is used for deciding the best streamliner based on features of the given instance. Even though the training of such learning-based approach is more computationally expensive, this approach offers significant improvement in performance compared to the other ones in several cases (as shown in Section 8).

3. From streamlining constraints to streamlined specifications

This section presents the methods used to generate streamlined models automatically. The process is driven by the decision variables in an Essence specification, such as the function in Fig. 1. The highly structured description of a problem an Essence specification provides is better suited to streamliner generation than a lower level representation, such as a constraint modelling languages like OPL, MiniZinc and Essence Prime. This is because nested types like multiset of sets must be represented as a constrained collection of more primitive variables, obscuring the structure that is useful to drive streamliner generation. For each variable, the system generates streamlining constraints that capture possible regularities that impose additional restrictions on the values of that variable's domain. Since the domains of Essence decision variables have complex, nested types, these restrictions can have far-reaching consequences for constraint models refined from the modified specification. The intention is that the search space is reduced considerably, while retaining at least one solution.

Candidate streamliners are generated by applying a system of streamlining rules. A streamlining rule takes as input the domain of an existing Essence term (a reference to a decision variable, or parts of it) and produces a constraint posted on this term. Abstract domains in Essence can be arbitrarily nested and streamlining rules take advantage of this nested structure. A rule defined to work on a domain `D` is lifted to work on a domain of the form `set of D` (and other abstract domain constructors mset, function, sequence, relation, partition, tuple, etc) through high-order rules.

We identify groups of streamlining rules and *tag* them appropriately such that multiple streamlining constraints from the same group are not combined. For example a rule that enforces an integer variable to be even uses the same tag as another rule that enforces the same integer to be odd. This simple way of identifying conflicting streamlining constraints allows streamliner selection search to prune some of these combinations without wasting any computational effort (Section 6.1).

3.1. Streamlining rules

Domain attributes can be added to Essence domains annotated to restrict the set of acceptable values. For example, a function variable domain may be restricted to injective functions, or a decision variable whose domain is a partition may be restricted to regular partitions. Hence, the simplest source of streamliners is the systematic annotation of the decision variables in an input specification. This sometimes retains solutions to the original problem while improving solver performance.

The existing Essence domain attributes are, however, of limited value. They are very strong restrictions and so often remove all solutions to the original problem when added to a specification. In order to generate a large variety of useful streamliners we employ a small set of rules, categorised into two classes:

1. First-order rules add constraints to reduce the domain of a decision variable directly.
2. High-order rules take another rule as an argument and lift its operation onto a decision variable with a nested domain, such as the complex set of functions presented in Fixed Length Error Correcting Codes (see Fig. 5). This allows for the generation of a rule such as enforcing that approximately half (with softness parameter) of the functions in the set are monotonically increasing. Imposing extra structure in this manner can reduce search very considerably.

A selection of the first-order rules is given in Fig. 3, and a selection of the higher-order rules is given in Fig. 4. These rules cover all domain constructors in Essence and they can be applied recursively to nested domains. Some of the rules (e.g. approximately half) take a softness argument which can control how strict the generated streamliner constraint is going to be. As a convention, smaller values of the softness parameters produce comparatively strict streamliners (hence potentially

Name	odd (Similarly even)
Input	`X: int`
Output	`X % 2 = 1`
Tag	IntOddEven

Name	lowerHalf (Similarly upperHalf)
Input	`X: int(1..u)`
Output	`X < 1 + (u - 1) / 2`
Tag	IntLowerUpper

Name	monotonicIncreasing (Similarly monotonicDecreasing)
Input	`X: function int --> int`
Output	`forAll i,j in defined(X) . i < j -> X(i) <= X(j)`
Tag	FunctionIncreaseDecrease

Name	smallestFirst (Similarly largestFirst)
Input	`X: function int(1..u) --> int`
Output	`forAll i in defined(X) . X(min(defined(X)) <= X(i)`
Tag	FunctionSmallLarge

Name	commutative
Input	`X: function (D, D) --> D`
Output	`forAll i,j in defined(X) . X(i,j) = X(j,i)`
Tag	FunctionCommutative

Name	nonCommutative
Input	`X: function (D, D) --> D`
Output	`forAll i,j in defined(X) . X(i,j) != X(j,i)`
Tag	FunctionCommutative

Name	associative
Input	`X: function (D, D) --> D`
Output	`forAll i,j,k in defined(X) . X(X(i,j),k) = X(i,X(j,k)))`
Tag	FunctionAssociative

Name	quasiRegular
Param	`k` (softness)
Input	`X: partition from D`
Output	`minPartSize(X, |participants(X)|/|parts(X)| - k) /\` `maxPartSize(X, |participants(X)|/|parts(X)| + k)`
Tag	PartitionRegular

Name	Add binary relation attributes
Input	`X: relation of (D, D)`
Output	`reflexive(X)` Similarly irreflexive, coreflexive, symmetric, antiSymmetric, aSymmetric, transitive, total, connex, Euclidean, serial, equivalence, partialOrder.
Tag	BinaryRelation

Fig. 3. The first-order streamlining rules. For each rule we present the rule name, rule's input and output and the tag. The tags are used to filter trivially contradicting streamliners during streamliner selection. We choose up to 1 streamliner from each tag.

causing greater reductions in the amount of search) and larger values produce more applicable streamliners. The reduction power and applicability are two of the criteria that we use when searching for effective streamliners (see Section 6.2).

The first two set of rules in Fig. 3 operate on a decision variable with an integer domain. They work by adding a unary constraint to limit values to the odd values, even values, values from the lower half of the domain, or upper half of the original domain. The next five sets of rules operate on decision variables with function domains. Monotonically increasing (and decreasing) enforce entries in the function to be monotonically increasing (or decreasing). The smallest first rule is a subset of monotonically increasing: it only enforces the smallest value in the function's defined set to be mapped to the smallest value in the range set (similarly for largest first). Commutative, non-commutative and associative rules enforce the corresponding property on a function variable. All streamlining rules work on partial and total functions. The quasi-regular rule takes a softness parameter and enforces the partition decision variable to be almost regular. A regular partition is one in which all parts of the partition are of equal cardinality. On binary relations, Essence contains common binary relation properties (listed in the figure). These attributes can simply be turned on by adding them to the domain declaration. The last set of streamlining rules generate one of these attributes on decision variables with a binary relation domain.

The tags given in Fig. 3 are used to filter trivially contradicting streamliner rules. For example, since they share the same tag our system would not generate commutative and non-commutative streamliners simultaneously. However, we would generate one of commutative or non-commutative together with associative.

Name all (Similarly half, at most one)
Works on matrices, sets, msets, sequences, and relations.
Param `R` (another rule)
Input `X: set of _`
Output `forAll i in X . R(i)`

Name Approximately half
Works on matrices, sets, msets, sequences, and relations.
Param `R` (another rule)
Param `k` (softness)
Input `X: set of _`
Output
```
(|X|/2-k <= sum i in X . toInt(R(i))) /\
(|X|/2+k >= sum i in X . toInt(R(i)))
```

Name range (Similarly defined)
The range and defined operators return the codomain and the domain sets of a function variable, respectively.
Param `R` (another rule)
Input `X: function _ --> _`
Output `R(range(X))`

Name parts
The parts operator returns a set of sets view of the partition variable.
Param `R` (another rule)
Input `X: partition of _`
Output `R(parts(X))`

Fig. 4. The higher-order streamlining rules. These rules lift existing first-order and higher-order streamlining rules to work on nested domain constructors of Essence. They do not introduce any additional tags, but they propagate the tags introduced by the rule they are parameterised on.

Fig. 4 lists the higher-order streamlining rules implemented in Conjure. These rules are used to apply the first-order streamlining rules to decision variables with nested domains. The first set of rules (all, half and at most one) work on sets, multi-sets, sequences and relations. They take another streamlining rule as an argument and apply it to the relevant members of the domain. The second set of rules applies a given streamlining rule to approximately half of the members of the decision variable's domain. The third set of rules applies a given streamlining rule to the defined set or the range set of a function variable. Finally, the last rule applies a given streamlining rule to the set of parts of a partition (returned by the parts function). The streamlining rule (`R`) that is given as a parameter to these rules can be a first-order or a higher-order rule. Moreover, they can be nested arbitrarily to match the nested domains that can be defined in the Essence problem specification. See Fig. 1 and Fig. 5 for the domains of the decision variables in the ten problem classes we use in this paper.

Although rich, the set of Essence type constructors is not exhaustive. Graph types, for example, are a work in progress [39]. At present, therefore, we might specify such a problem in terms of a set of pairs or a relation domain. The streamliner generator constraints would produce candidate streamliners based on this representation. As further type constructors are added to Essence it is straightforward to extend our streamlining rules to accommodate them.

4. Problem classes

We now introduce the problem classes studied in this paper, in addition to the Car Sequencing Problem presented in Fig. 1. There are 10 problem classes in total. They will be used both to illustrate the remainder of our method and for our empirical evaluation. We selected these problems, presented in Fig. 5, to give good coverage of the abstract domains available in Essence, including matrices, sets, partitions, relations and functions. They also cover various types of problems in practice, such as fundamental combinatorial design problems (Balanced Incomplete Block Design, Social Golfers), scheduling and manufacturing problems (Car Sequencing, Vessel Loading), timetabling problems (Balanced Academic Curriculum Problem), and transportation problems (Transshipment, Tail Assignment). Such problems have been considered in various prior works in both Constraint Programming and Operation Research.

The Balanced Academic Curriculum Problem (BACP) [40] (decision version) is to design a balanced academic curriculum by assigning periods to courses. The constraints include the minimum and maximum academic load for each period, the minimum and maximum number of courses for each period, and the prerequisite relationships between courses. This problem is also specified as finding a function from courses to periods.

The Balanced Incomplete Block Design problem (BIBD) [41] is a standard problem from design theory often used in the design of experiments. It asks us to find an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks. This problem is naturally specified as finding a relation between objects and blocks.

```
$ Balanced Academic Curriculum Problem (BACP)
find curr : function (total) Course --> Period

$ Balanced Incomplete Block Designs (BIBD)
find bibd : relation of (Obj * Block)

$ Covering Array
find CA: matrix indexed by [int(1..k), int(1..b)] of int(1..g)

$ Equidistant Frequency Permutation Arrays (EFPA)
letting String be domain function (total) Index --> Character
find c : set (size numCodeWords) of String

$ Fixed Length Error Correcting Codes (FLECC)
letting String be domain function (total) Index --> Character
find c : set (size numOfCodeWords) of String

$ Transshipment
find amountWT : function (W, T) --> int(1..max(range(stock)))
find amountTC : function (T, C) --> int(1..max(range(demand))

$ Tail Assignment
find route : function (total) Plane --> function int(1..n_flights) -->
    Flight

$ Social Golfers
find sched : set (size w) of
             partition (regular, numParts g, partSize s) from Golfers

$ Vessel Loading
find west,  east : function (total) Container --> X,
     north, south: function (total) Container --> Y
```

Fig. 5. Essence problem specification fragments for the problem classes used for evaluation, in addition to Fig. 1. Here we list the decision variable declaration statements for these problems, since these are what govern the generation of candidate streamlining constraints. The full models we use can be found in our accompanied github repo.

The Covering Array problem [42] requires finding a matrix of integer values indexed by k and b such that any subset of t rows can be used to encode numbers from 0 to g^{t-1}. In addition to the covering constraint, row and column symmetries are broken using the lexicographic ordering constraints [43].

The Equidistant Frequency Permutation Arrays problem (EFPA) [44,45] is to find a set (optionally of maximal size) of codewords, such that any pair of codewords are a certain Hamming distance apart. The decision version we consider here works for a given number of codewords. In comparison to Fixed Length Error Correction Codes problem which only has a minimum distance requirement, this problem requires the distances to be equal to an exact value. In addition each codeword must include each symbol a certain number of times.

The Fixed Length Error Correction Codes problem (FLECC) [46,47] asks us to find a set of code words of a uniform length such that each pair of code words are at least a specified minimum distance from each other, as computed by a given distance metric (e.g. hamming distance).

The Transshipment problem [48] considers the design of a distribution network, which includes a number of warehouses and transshipment points to serve a number of customers. The cost of delivering items from each warehouse to each transshipment point and from each transshipment point to each customer, and the amount of stock available at each warehouse are given. We are asked to find a delivery plan that meets customer demand within a cost budget. This is specified as a pair of functions describing the amount of demand supplied between each warehouse and transshipment point, and each transshipment point and customer.

The Tail Assignment problem [49] is the problem of deciding which individual aircraft (identified by its tail number) should cover which flight. The problem is represented using a nested function variable, where the outer function is total and the inner one is potentially partial. There are several constraints on this nested function variable ensuring a sensible sequence of flights and frequent enough visits to a maintenance depot.

The Social Golfers problem [50] is concerned with finding a schedule for a number of golfers over w weeks. The schedule for each week is a partitioning of the golfers such that over the course of the entire schedule no golfer plays in the same group as any other golfer on more than one occasion.

The Vessel Loading decision problem [51] is to determine whether a given set of containers can be positioned on a given deck, without overlapping, and without violating any of the separation constraints. The problem is modelled in Essence using four total functions capturing the location of each container.

```
given n_warehouses_middle: int(1..100)
given n_warehouses_delta: int(0..49)
find n_warehouses: int(1..100)
such that
  n_warehouses >= n_warehouses_middle - n_warehouses_delta,
  n_warehouses <= n_warehouses_middle + n_warehouses_delta
given n_transshipment_middle: int(1..100)
given n_transshipment_delta: int(0..49)
find n_transshipment: int(1..100)
such that
  n_transshipment >= n_transshipment_middle - n_transshipment_delta,
  n_transshipment <= n_transshipment_middle + n_transshipment_delta
given costWT_range_middle: int(1..100)
given costWT_range_delta: int(0..49)
find costWT: function (int(1..100), int(1..100)) --> int(1..100)
such that
  and([q1[1] >= 1 /\ q1[1] <= n_warehouses /\ (q1[2] >= 1 /\ q1[2] <=
      n_transshipment) <-> q1 in defined(costWT)
             | q1 : (int(1..100), int(1..100))]),
  and([q1[2] >= costWT_range_middle - costWT_range_delta | q1 <- costWT
      ]),
  and([q1[2]<=costWT_range_middle + costWT_range_delta | q1 <- costWT])
```

Fig. 6. A snippet of the instance generator model for the Transshipment problem. For brevity, we omit the specification for `n_customer`, `costTC`, `stock`, `demand` and `maxCost`, as they are rewritten in exactly the same way as other parameters of the same types.

5. Generating and selecting training instances

The core idea behind our methodology is that we can build a portfolio of streamliners on a training set and then employ that portfolio to solve unseen instances from the same problem class with substantially less effort than the unstreamlined model. This method relies upon the automatic evaluation of model candidates in order to construct a high quality portfolio of streamlined models. In this section, we describe how a set training instances that are representative of the problem instance space is generated automatically.

5.1. Generating candidate training instances

As the first step, we generate a large number of candidate training instances for each pair of problem class and solver via AutoIG[1] [52,54,55], a constraint-based automated instance generation tool. AutoIG allows users to describe the generation of instances for a given problem class in a declarative way as a constraint model, and supports the automated generation of new instances with certain properties required by the users. Fig. 6 shows (part of) an example generator model for the Transshipment problem. For this work, we use AutoIG to find satisfiable instances that are solvable by a chosen solver within the solving time range of $[10, 300]$ seconds. The lower bound of 10 seconds is imposed to avoid trivially solvable instances, as the gain when applying streamliners on such instances are often negligible.

The internal working mechanism of AutoIG is as follows. Starting from a description of the problem and an instance generation model (either created by users or generated via an automated generator approach [52,54]), AutoIG generates new instances by searching in the parameter configuration space of the generator using the automated algorithm configuration tool IRACE [56], and by sampling a new instance via solving each instance of the generator model (provided by IRACE) via the constraint solver MINION [53]. Every time a new instance is generated, it is evaluated using the chosen solver and its quality (in term of satisfying the properties specified by users) is given to IRACE as feedback to update its sampling model. The instance generation process stops once a given tuning budget is exhausted and all instances satisfying the required properties will then be returned.

5.2. Training set construction

For some problems, the number of instances found by the automated instance generation process can be quite large. For example, we got 4647 instances for FLECC problem with CHUFFED as the target solver (Table 1). Using all instances during the streamliner portfolio construction phase would require a significant amount of computation. In this section, we propose a method to select a small representative subset of training instances from the ones obtained in previous step.

[1] https://github.com/stacs-cp/AutoIG.

Table 1
For each problem class, the table shows the following fields: the number of candidate streamliners automatically generated by Conjure, the total number of training instances generated by the automated instance generation procedure, and the number of clusters detected by GMeans. The instance-related fields are different per solver, as instance generation is done separatedly for each solver.

Problem	#Candidate Streamliners	#Instances		#Clusters	
		Chuffed	Lingeling	Chuffed	Lingeling
BACP	108	235	133	< 50	< 50
BIBD	200	427	272	< 50	< 50
CoveringArray	64	4301	1641	153	54
Car Sequencing	36	4376	5651	171	149
EFPA	312	1233	1215	57	62
FLECC	144	4647	6930	128	162
Transshipment	68	1534	3889	< 50	96
Tail Assignment	336	1515	1627	90	96
Social Golfers	260	709	340	< 50	< 50
Vessel Loading	208	2764	3430	65	80

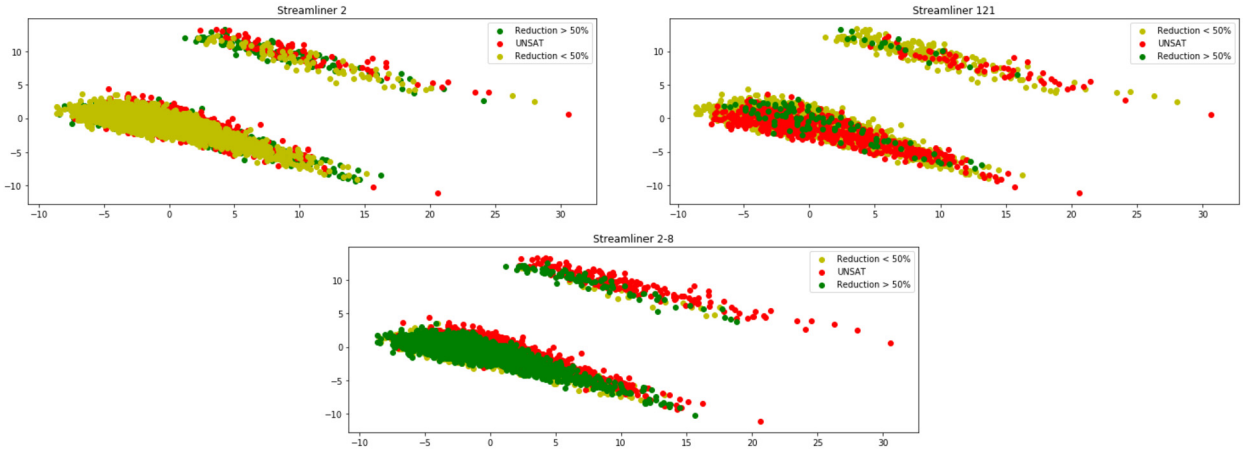


Fig. 7. Performance of three example streamliners on the training set of 4647 instances generated for the Fixed Length Error Correcting Code problem with Chuffed. Each two dimensional plot is a projection of the original multi-dimensional instance feature space via Principal Component Analysis. (For interpretation of the colours in the figure(s), the reader is referred to the web version of this article.)

5.2.1. Streamliner performance footprint analysis

The aim of the instance selection process is not only to reduce the size of the training instance set, but also to make sure that the selected instances are as diverse as possible. The motivation behind the latter objective is to ensure the generalisation ability of the constructed portfolio.

We illustrate our motivation via a visualisation analysis on streamliner performance across the instance space. The visualisation is similar to the algorithm footprint analysis method [57]. An algorithm footprint is simply the part of the instance space where the algorithm performs well. In our context, a streamliner is an algorithm. We will show that the footprints of different streamliners can cover different parts of the instance space, which suggests the necessity of selecting the training instances with good coverage across the space.

To visualise the instance space, we extract FlatZinc instance features via the `fzn2feat` tool (part of `mzn2feat` [58]). There are 95 features grouped into 6 categories (variables, constraints, domains, global constraints, objective, and solving features) [58]. The feature space is then projected into a 2-D space using Principal Component Analysis [59]. For each streamliner, we mark its performance on all generated instances with different colours. For simplicity, the performance is divided into three categories: (i) UNSAT (the streamliner eliminates all solutions of the instance), (ii) the streamliner offers less than 50% reduction in solving time for the given instance, and (iii) the streamliner achieve more than 50% reduction in solving time.

Fig. 7 shows performance of three example streamliners on the FLECC problem with Chuffed. Looking at the group of instances at the bottom of each plot, we see that the performance achieved by different streamliners varies drastically. Streamliner 2 is mostly satisfiable across the group but generally it only achieves a reduction of less than 50%. Streamliner 121 on the other hand occasionally achieves good reductions but in general it seems to be too strict and renders most instances infeasible. Note also that we can vastly modify the footprint achieved by applying the combination of 2 and 8. For most instances in the bottom group it seems that this is a more effective combination as the general reduction has drastically increased to $\geq 50\%$. However in other parts this combination makes the resulting instance too tight and as such negatively affects its feasibility. Therefore, during training set construction, it is important that we take this diversity

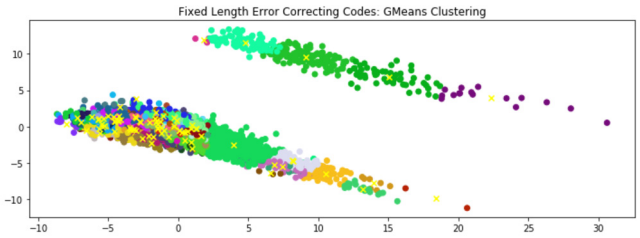


Fig. 8. GMeans clustering results on the instance feature space (projected to 2-dimensional space by PCA) for the Fixed Length Error Correcting Codes problem with CHUFFED. Each colour represents a cluster.

into account. If our training set only include instances from this particular group, this will directly limit the ability of our streamliner portfolios to generalize across the problem class.

5.2.2. Building a compressed training instance set via clustering

To select a diverse subset of instances for the training phase, the GMeans [60] clustering method is used on the instance feature space to detect the number of instance clusters (column 4 of Table 1). An example of clustering results for the FLECC problem with CHUFFED is shown in Fig. 8. With those clusters, we can build a compressed version of the original training set by selecting a subset of instances per cluster.

To make sure that we have a sufficient number of representative training instances, we define a minimum number of 50 instances to comprise our compressed training set. This value was chosen based on the computational resources available for our experiments. If the number of clusters detected by GMeans is larger than this minimum size, one representative instance per cluster is selected. In the scenarios where the number of detected clusters is less than 50, instances are chosen from each cluster until the minimum number (50) is met, i.e., the number of instances selected per cluster is proportional to the size of the cluster. In order to take into account the information regarding instance difficulty in the selection of representative instances, for each cluster, instead of using purely random selection or selection of the instances closest to the centroid, we perform sampling without replacement of the median instance in terms of its corresponding solving time by the unstreamlined model.

6. Identifying effective combinations of streamliners

It has previously been observed that applying several streamlining constraints to a model simultaneously can result in larger performance gains than any of the constraints in isolation [17]; and we give an example of this behaviour using the FLECC problem in Fig. 7. In order to find such combinations of constraints we must consider the power set of candidate streamliners, which form a lattice: the root is the original ESSENCE specification and an edge represents the addition of a streamliner to the combination associated with the parent node. Finding effective streamliner combinations involves search in this lattice and evaluating the combinations at each node at which the search arrives. In order to keep the size of the set of streamliners (and hence the search) manageable, we used a small number of softness parameter values for each rule that requires a softness parameter.

6.1. Pruning the streamliner lattice

For many of the problems considered a large number of singleton streamliners are generated (see Table 1), resulting in a space of streamliner combinations too large to be explored exhaustively in practice. Two forms of pruning are used to reduce the number of combinations to be considered:

1. If a set of streamliners fails all supersets are excluded from consideration. To be considered failed a streamliner must have zero applicability across the instance space, i.e. it removes all solutions for all instances. The effectiveness of this pruning strategy is largely dependent on the ordering of the traversal of the streamliner configurations. For instance in the example from Fig. 9 it can be seen that the streamliner *CD* is unsatisfiable, which allows for the supersets of that configuration (*ABCD, BCD*) to be immediately pruned. However, this pruning can only occur if the fact that *CD* is unsatisfiable is discovered before *BCD* and *ABCD* are evaluated. Thus different traversal orderings can impact on the amount of pruning that can be performed.
2. Trivially conflicting streamliners are not combined, such as streamliners A and B in the figure. For example, we avoid forcing a set simultaneously to contain only odd numbers and contain only even numbers. We associate a set of tags with each of the rules in order to implement this pruning. Rules applied to the same variable that share tags are not combined. This also removes the possibility of combining two different streamliners that differ only in the values of their softness parameters.

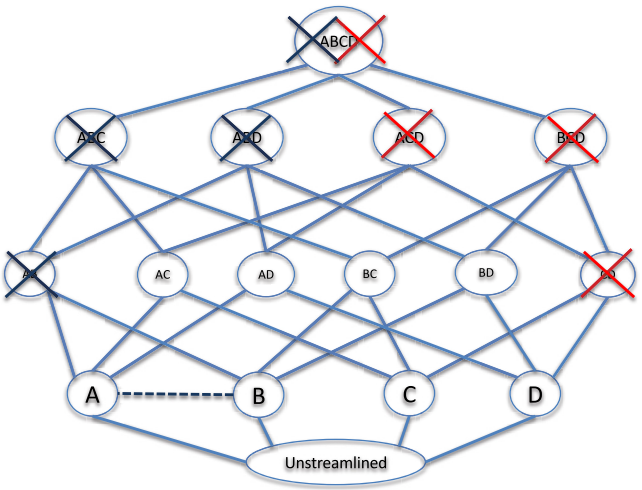


Fig. 9. The power set of singleton candidate streamliners is explored to identify combinations that result in powerful streamlined specifications. Starting from an empty set of streamliners (the unstreamlined model), new streamliners are gradually added. If small sets of streamliners that fail to retain solutions are identified, such as CD, all supersets can be pruned from the search, vastly reducing the number of vertices to be explored. Streamliners A and B are tagged (Section 3) mutually exclusive, and so no streamliner combinations containing both are evaluated.

6.2. Searching for a streamliner portfolio

The two pruning rules described above only remove combinations that are sure to fail, or are equivalent to a smaller set of streamliners. Therefore, even after pruning, the number of combinations to consider is still typically too large to allow exhaustive enumeration. A traversal of the lattice allowing good combinations to be identified rapidly 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 generalise to as many instances as possible. These goals conflict as generally the search reduction a streamliner achieves is related to its tightness. The tighter a streamliner constraint the more propagation it can achieve at each node of search resulting in a more restricted search space; this is the reason that combining different candidate streamliners can provide superior results as with the addition of each streamliner the search space is further restricted. With two competing objectives, it is no longer feasible to find a single "best" streamlined specification: a streamliner combination may be optimal in relation to one objective, but at the expense of compromising the other.

To address these problems we adopt a multi-objective optimisation approach, where each point x in the search space X is associated with a 2-dimensional (following the number of objectives) reward vector r_x in R^2. Our two objectives:

1. **Applicability**. The proportion of training instances for which the streamlined model admits a solution.
2. **Search Reduction**. The mean search reduction in solving time achieved by the streamliner on the satisfiable instances

With these two objectives for each streamliner combination we define a partial ordering on R^2 and so on X using the Pareto dominance definition in multi-objective optimisation. Given $x, x\prime \in X$ with vectorial rewards $r_x = \langle r_1, r_2 \rangle$ and $r_{x\prime} = \langle r_{1\prime}, r_{2\prime} \rangle$:

$$r_x \text{ dominates } r_{x\prime} \iff (\forall i \in [1,2]\, r_i \geq r_{i\prime}) \wedge (\exists j \in [1,2]\, r_j > r_{j\prime}) \tag{1}$$

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 [61]. The algorithm has four phases, as summarised below. An illustration example of the four phases is presented in Fig. 10.

1. **Selection**: Starting at the root node, the Upper Confidence Bound applied to Trees (UCT) [62] policy is applied to traverse the explored part of the lattice until an unexpanded node is reached.
2. **Expansion**: Uniformly select a random admissible child and expand on it (see the simulation step below).
3. **Simulation**: The collection of streamliners associated with the expanded node are evaluated. The vectorial reward ⟨Applicability, Search Reduction⟩ across the set of training instances is calculated and returned.

   Simulation is the most expensive phase. To constrain the computational cost and improve iteration speed we perform *instance filtering* to avoid wasting time on evaluating runs known to be UNSAT. More specifically, if a streamliner combination is proved to be UNSAT on an instance, we know that any of its descendants will also be UNSAT on that same instance without having to evaluate them. We make use of this knowledge to reduce the number of instances being

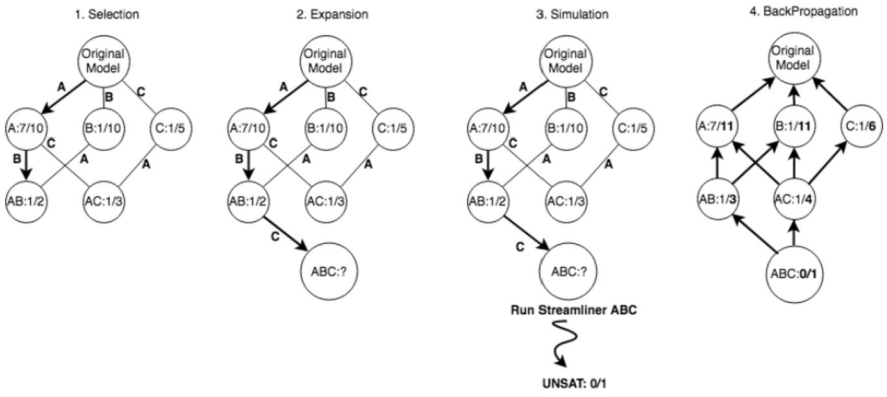


Fig. 10. MOMCTS-DOM operating on the streamliner lattice. A, B and C refer to single candidate streamliners generated from the original Essence specification. As MOMCTS-DOM descends down through the lattice the streamliners are combined through the conjunction of the individual streamliners (AB, ABC). The nodes are labelled with CDD reward value divided by the number of times visited.

unnecessarily evaluated at each lattice node. When we arrive at a given node in the lattice, the intersection of the sets of satisfiable instances from all available parents in the lattice is used to construct the evaluation set. For example, given three streamliners A, B, and C and a set of five instances. If we know AB is only satisfiable on instances $\{1, 2, 3\}$ while AC is only satisfiable on instances $\{2, 3\}$, then for ABC we only need to evaluate the streamliner combination on instances $\{2, 3\}$. This reduces the computation on the current node by 60%.

4. **Back Propagation**: The current portfolio of non-dominated streamliner combinations is used to compute the Pareto dominance test. 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)* [61] reward mechanism during reward update. If the current vectorial reward is not dominated by any streamliner combination in the portfolio then the evaluated streamliner combination is added to the portfolio and a CDD reward of 1 is given, otherwise 0. Dominated streamliner combinations are removed from the portfolio. The result of the evaluation is propagated back up through all paths in the lattice to update CDD reward values, as shown in Fig. 10.

6.3. Improving portfolio complementary strength

In our initial implementation the multi-objective search of the lattice is performed until either the computational budget is reached or the lattice is fully explored. During this time one portfolio of non-dominated streamliners is built where domination is defined across the two objectives defined in Section 6.2. There are two deficiencies with this method that can be highlighted through an example. Consider a setting with three instances {A,B,C} and two singleton streamliners {S1, S2}. S1 retains satisfiability on instances {A,C} with {50%, 25%} reduction percentage respectively but renders instance {B} unsatisfiable, yielding *{AvgReduction:37.5%, AvgApplicability: 66.6%}* in terms of our two objectives. {S2} renders instance *{A}* unsatisfiable but retains satisfiability on instances {B,C} with {30%, 35%} reduction respectively, resulting in *{AvgReduction:32.5%, AvgApplicability: 66.6%}*. The resulting portfolio at the end of search will only ever contain S1 as S2 is always Pareto dominated and so will be disregarded. However S2 actually possesses some interesting qualities that we might not want to overlook. Firstly, it manages to cover instance B which is not covered by the current portfolio and it also manages to achieve a higher reduction on instance C. Given this in our ideal setting we would like to retain streamliner S2 as part of our portfolio. By averaging the performance of a streamliner and maintaining just one Pareto front it makes it difficult to distinguish cases like this and will often mean that the resultant portfolio will be suboptimal.

An alternative would be to create an objective per instance that records the performance of the streamliner on that individual instance. This would retain full information and allow us to distinguish the situation where a streamliner works on a complementary set of instances, or manages to attain a higher reduction on one particular instance. The difficulty with this approach is that with more objectives the size of the portfolio will grow exponentially as a result and it would become impractical to schedule the streamliners from the portfolio in any meaningful way. The other difficulty is if each streamliner produces small improvements to one or more instances then every path traversed in the lattice will produce a reward which will make it very difficult for our best first search procedure to focus and would essentially degrade it into random search.

In order to solve this issue, we adapted our search to incorporate elements of Hydra [63], a portfolio builder approach that automatically builds a set of solvers or parameter configurations of solvers with complementary strengths by iteratively configuring (a set of) algorithms. Instead of performing just one lattice search and building one portfolio, we now perform multiple rounds of search. In each round a portfolio is built to complements the strengths of the combined portfolios built in the prior rounds.

More specifically, in the first round, an MO-MCTS search with our original performance metric, which tries to optimise both applicability and solving-time reduction on the training set, is done and a portfolio of streamliners is constructed. In each subsequent round, a new MO-MCTS search is started using a modified performance metric. For each instance i, the

best streamliner p for i (the one that has the highest solving-time reduction on i) in the combined portfolios from previous rounds is identified if it existed. For any new streamliner q being evaluated in the search of the current round, if it has better reduction than p on i, or if i was not yet solved by any streamliner in the current combined portfolio, performance of q on i is used, otherwise, the reduction value of p on i is used instead. This means that the new streamliner q will not be penalised for its poor performance on an instance if the instance is already efficiently solved by the combined portfolio in the previous rounds. Therefore, the MO-MCTS can focus on trying to improve performance in regions of instance space where the current portfolio is weak. The final result is a combined portfolio with complementary strengths that can perform well on all parts of the training instance set.

Performance of all evaluated streamliners are cached and re-used across rounds. In each round, at least M iterations (not including iterations using cached results) have to be completed. After that, the round is stopped if it spends N consecutive iterations without finding anything to add to the current portfolio, as it is an indication that we might have reached the point of diminishing returns for the current round. The whole Hydra search is terminated if the current combined portfolio remains unchanged after a round. In our experiments M and N are set as 10 and 5 respectively. The values were chosen based on a small manual tuning experiment, which aims at having a good balance between the number of rounds being done and the amount of resources given to each round within the available computational budget.

6.4. Independent solver search

For each problem class, we perform a streamliner search per solver (CHUFFED or LINGELING). It might be expected that this is unnecessary and that the streamliners that work well for CHUFFED will also work well for LINGELING and vice-versa, as we are generating the streamliners from the ESSENCE specification of a problem class, which is solver independent. However, the intricacies of solvers such as heuristics, propagation mechanisms and restarts can be so different that the performance of a constraint can vary wildly. Also, the streamliners are defined in ESSENCE and how they are represented in a constraint or SAT model can be very different. One streamliner that can be efficiently represented in a constraint model might be very verbose in the SAT encoding, which may result in substantial overhead during search and as such affect performance. We illustrate that via Fig. 11. The performance of a streamliner when tested on the same instance sets can drastically differ between CP and SAT. In Transshipment the streamliners that comprise the portfolio found via CHUFFED search do elicit reductions in LINGELING albeit not as strong as their CHUFFED counterpart. In CarSequencing, however, almost all of the CHUFFED portfolio produces a negative reduction in solving time when applied in LINGELING.

7. Applying a streamliner portfolio on unseen instances

Having constructed a streamliner portfolio for a particular problem class using our streamliner search on the training instance set, the next question is how to apply the given portfolio on an unseen instance of the same problem class. In this section, we describe different streamliner portfolio application methods used in this work, ranging from simple instance-oblivious approaches (Section 7.1 and Section 7.2) to instance-specific methods taken from the literature of automated algorithm selection [64,30] (Section 7.3).

7.1. Single best streamliner (SBS)

The most basic streamliner application approach, namely the *Single Best Streamliner (SBS)*, is to choose from the portfolio the streamliner that results in the lowest average solving time across all training instances, and applying that chosen one for any unseen future instance. The deficiency of this approach is that streamliners that do not perform well on average across the instance space are neglected even if they may exhibit good performance on a subset of instances.

7.2. Lexicographic selection methods

It is possible to order the streamlined models in a portfolio lexicographically by, for example, prioritising Applicability, then Search Reduction. Given two objectives, there are two such orderings to consider. Thus two lexicographic selection methods are used herein: *AppFirst*, which prioritises applicability over search reduction, and *ReducFirst*, which has the reverse priority.

The selection process involves traversing the portfolio (using the defined ordering) for a given time period and applying each streamliner in turn to the given instance. The schedule is static in that it only moves to the next streamlined model when the search space of the current one is exhausted.

When traversing a schedule it is possible to dynamically filter streamliners based upon prior results. If for a given instance we are evaluating the schedule containing the streamliners {*S-1, S-3, S-1-2*} in their respective ordering. When evaluating this on a given instance if the first streamliner *S-1* renders the test instance unsatisfiable, and this is proven within the given time limit, then this allows us to filter the rest of the schedule and remove *S-1-2* since any superset of *S-1* is guaranteed also to render the test instance unsatisfiable.

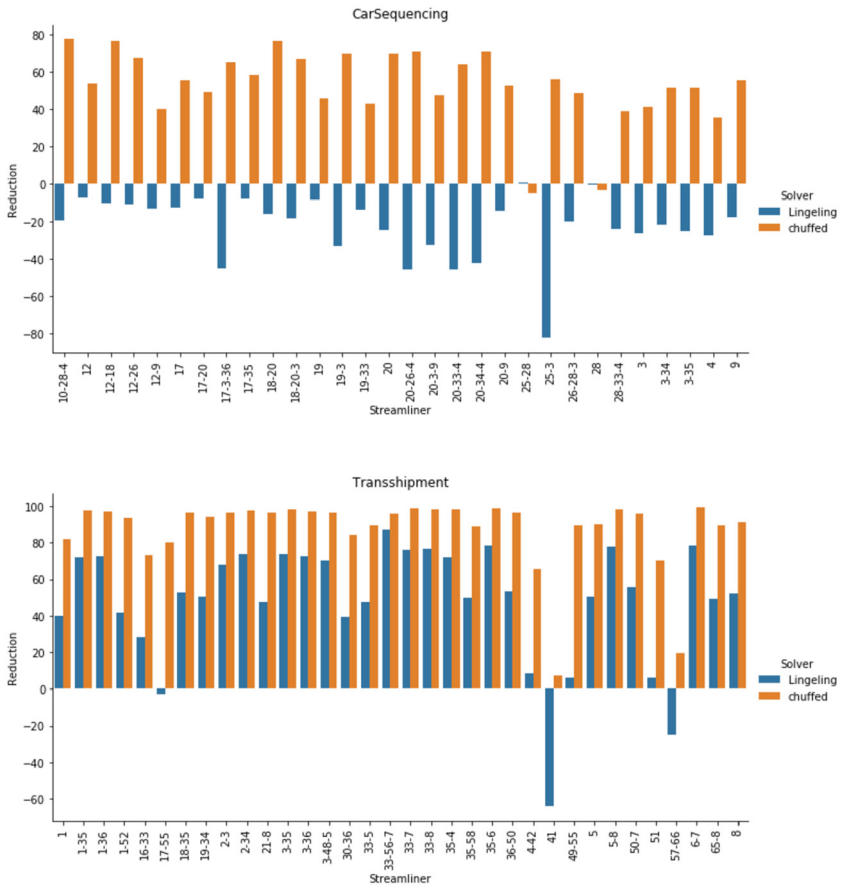


Fig. 11. For Transshipment and CarSequencing the portfolios generated during the CHUFFED streamliner search are tested on LINGELING on the same set of test instances. The Average Reduction across the two portfolios is represented for both paradigms. The same set of test instances are used so that any variation in the reductions of the streamliners is purely due to the different setting.

7.3. Automated algorithm selection methods

In many paradigms for solving combinatorial problems, such as SAT, CP, ASP (Answer Set Programming) there are multiple available algorithms or search strategies, all with complementary solving strengths. Automated Algorithm Selection (AS) techniques [65,64,30] aim to exploit this fact by utilising instance characteristics to select from a set of algorithms the one(s) expected to solve a given problem instance most efficiently. Algorithm selectors have had great success and have been shown empirically to improve the state of the art for solving heterogeneous instance sets [38,66].

This is a very similar setting to our portfolio of streamliners, with complementary solving strengths and no single dominating streamliner. In this work we employ the algorithm selection system AutoFolio [38]. Given a particular problem instance the goal is to have AutoFolio, based upon the features of the instance, predict which streamliner from the generated portfolio will most efficiently solve the instance.

When applying algorithm selection to a new domain a number of questions arise. First, there are multiple different algorithm selection techniques and there is the question of which particular AS technique is best for the current domain. Second, AS approaches generally contain several parameters and there is the need to set these effectively to obtain good performance. The AS framework *AutoFolio* [38] addresses these questions by integrating several AS techniques and automatically choosing the best one as well as configuring their hyper-parameters using the automatic algorithm configuration tool SMAC [67]. 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 configuration phase by SMAC.

8. Experimental results

In the preceding sections we have presented a completely automated approach to the generation and selection of streamliner constraints, hitherto a laborious manual task. In this section we present two experiments to evaluate the efficacy of this approach. The first one is designed to measure the frequency with which streamlining results in a reduction in search,

Table 2
Performance comparison between ApplicFirst, ReducFirst, AutoFolio the Oracle and the SBS using the overall speedup on *Distribution A*, containing instances with unstreamlined solving times in [10, 300] seconds.

Solver	Problem	# Instances	Oracle	SBS	ApplicFirst	ReducFirst	Autofolio
Chuffed	BACP	100	25.61	**2.84**	2.01	2.81	2.48
	BIBD	98	1.63	1.05	1.05	1.03	**1.15**
	CarSequencing	100	11.21	3.77	3.35	3.66	**4.46**
	CoveringArray	100	1.73	1.45	1.05	1.05	**1.57**
	EFPA	98	3.21	**2.02**	**2.02**	1.86	1.99
	FLECC	100	2.42	**1.90**	1.88	1.71	**1.90**
	SocialGolfersProblem	100	1.12	1.02	1.07	1.07	**1.09**
	TailAssignment	35	3.20	**3.20**	**3.20**	1.21	**3.20**
	Transshipment	100	9.19	2.44	2.58	**2.63**	2.17
	VesselLoading	100	5.92	**3.56**	1.51	1.09	1.89
	Geometric-Mean Speedup		4.06	2.11	1.81	1.63	**2.01**
Lingeling	BACP	82	3.22	**1.32**	**1.32**	1.16	1.19
	BIBD	99	1.03	**1.01**	**1.01**	**1.01**	**1.01**
	CarSequencing	100	1.04	**1.01**	**1.01**	1.00	1.00
	CoveringArray	100	3.73	**1.60**	1.01	1.01	1.46
	EFPA	98	1.32	**1.16**	1.05	1.03	1.05
	FLECC	100	4.06	2.72	2.79	2.40	**3.08**
	SocialGolfersProblem	91	1.10	**1.05**	1.01	1.01	1.04
	TailAssignment	100	2.47	**2.47**	**2.47**	1.04	**2.47**
	Transshipment	99	6.13	1.84	1.95	3.20	**4.15**
	VesselLoading	100	2.24	1.03	1.03	1.44	**1.49**
	Geometric-Mean Speedup		2.19	1.42	1.35	1.31	**1.57**

and the magnitude of that reduction (Section 8.2). The second one situates our approach in the simplest practical setting, where an unstreamlined model is solved in parallel with a streamlined model, in order to measure the overall speedup obtained when solving a given set of instances (Section 8.3).

8.1. Experimental setup

The effectiveness of our streamliner approach is demonstrated on a wide range of 10 different problem classes (as described in Section 4) with two solving paradigms (CHUFFED for CP and LINGELING for SAT). All experiments were run on compute nodes with two 2.1 GHz, 18-core Intel Xeon E5-2695 processors. The streamliner portfolio construction phase was run on a single core with a maximum time budget of 4 CPU days for each pair of problem class and solver.

The generated streamliner portfolios were evaluated on two different instance distributions, distinguished by their comprising instance difficulty. The first one, denoted *Distribution A*, consists of instances with similar difficulty to those used during the portfolio construction phase (Section 6), i.e. satisfiable instances with a solving time within [10, 300] seconds by the unstreamlined model. We use this to analyse the generalisation performance of the streamliner portfolios on similarly difficult instances unseen by the portfolio construction. The second test set, denoted *Distribution B*, includes instances generated by the same method (Section 5.1), but drawn from a different distribution with a solving time limit of (300, 3600] seconds (by the unstreamlined model). *Distribution B* allows us to study the ability of the portfolio to generalise to instances of greater difficulty.

Similar to the generation of training instances, the automated instance generation tool AutoIG [55] is used for generating instances of both distributions. Each AutoIG run is given a wall-time limit of 24 CPU hours. The number of instances generated for each problem class are listed in Table 2 and Table 3).

Our system makes use of CONJURE[2] (for generating streamliners and for producing streamlined models), SAVILE ROW[3] (for producing solver-specific inputs, including the FlatZinc input for the `fzn2feat`[4] feature extraction tool), and the two solvers CHUFFED[5] and LINGELING[6] for the evaluation. All of those softwares were run with their default parameter settings.

We report the performance of all streamliner scheduling/selection approaches described in Section 7, including the Single Best Streamliner (SBS), the two simple streamliner scheduling methods ApplicFirst and ReducFirst, and the automated algorithm selection approach AutoFolio[7] [38]. A training of AutoFolio on a pair of problem class and solver is given a tuning budget of one CPU day. We also report as a reference point the theoretically best performance, namely the *Oracle*, where

[2] https://github.com/conjure-cp/conjure.
[3] https://savilerow.cs.st-andrews.ac.uk/.
[4] https://github.com/CP-Unibo/mzn2feat.
[5] https://github.com/chuffed/chuffed.
[6] https://github.com/arminbiere/lingeling.
[7] https://github.com/automl/AutoFolio.

Table 3
Performance comparison between ApplicFirst, ReducFirst, AutoFolio the Oracle and the SBS using the overall speedup on *Distribution B*, containing instances with unstreamlined solving times in [300, 3600] seconds.

Solver	Problem	# Instances	Oracle	SBS	ApplicFirst	ReducFirst	Autofolio
Chuffed	BACP	16	53.47	1.47	2.48	4.71	**46.56**
	BIBD	59	2.25	1.13	1.15	1.04	**1.71**
	CarSequencing	52	8.77	1.91	1.88	2.19	**6.77**
	CoveringArray	46	3.36	2.20	1.26	1.26	**3.20**
	EFPA	121	4.86	1.02	1.93	1.79	**2.53**
	FLECC	192	3.95	2.18	2.02	1.68	**2.24**
	SocialGolfersProblem	19	2.53	1.28	1.00	1.00	**2.53**
	TailAssignment	35	3.20	**3.20**	**3.20**	1.21	**3.20**
	Transshipment	216	16.21	2.77	2.89	2.93	**5.39**
	VesselLoading	322	4.72	1.64	1.21	1.02	**2.12**
	Geometric-Mean Speedup		5.59	1.80	1.79	1.65	**3.74**
Lingeling	BACP	15	5.92	2.20	2.20	1.88	**4.91**
	BIBD	25	2.26	1.25	**1.39**	1.04	1.30
	CarSequencing	69	3.32	1.06	1.34	1.19	**2.95**
	CoveringArray	34	16.65	2.19	1.63	1.63	**10.81**
	EFPA	158	1.39	1.00	1.18	1.03	**1.20**
	FLECC	166	5.89	1.62	1.79	1.42	**3.39**
	SocialGolfersProblem	17	2.23	1.14	1.09	1.09	**1.89**
	TailAssignment	36	2.97	**2.95**	**2.95**	1.18	**2.95**
	Transshipment	68	12.42	3.59	3.55	3.60	**5.25**
	VesselLoading	78	2.51	1.29	1.11	1.80	**2.34**
	Geometric-Mean Speedup		4.07	1.66	1.68	1.47	**2.99**

we assume that the best solving model (either unstreamlined or the streamlined ones from the constructed portfolio) for each instance is used.

The construction of our streamliner portfolios is similar to previous works on manual streamlining [13,17], we show that streamliners found using small instances can be highly effective when solving larger and more difficult instances. As described in Section 5.1 and Section 6, for each pair of problem class and solver, the lattice search is done using instances with the solving times similar to distribution A. By limiting the search to small (yet non-trivial) instances, we can not only examine a large amount of streamliners within a reasonable time budget, but also evaluating each streamliner on several instances, which allows the search to identify regularities and structures that are common among various instances.

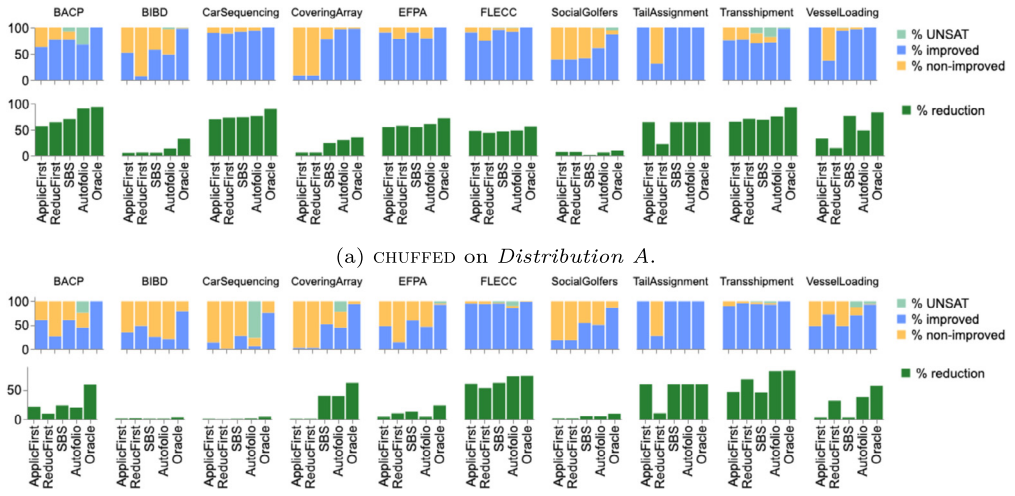
The training cost of building a streamliner selection model on the constructed streamliner portfolio (i.e., AutoFolio) is generally much lower since it only focuses on a limited number of streamliner combinations. Therefore, to ensure the selection is effective on larger and more difficult instances, we add a small number of instances drawn from distribution B (separated from the test instances used for the evaluation) to the training of AutoFolio.[8] This AutoFolio model is used for the evaluation on distribution B. Note that when evaluating on distribution A, we do not need to add those extra instances, i.e., the AutoFolio model is trained on instances from distribution A only (again separated from the test instances used for the evaluation). As shown in the subsequent section, AutoFolio achieves the best overall performance in several cases, especially on the larger and more difficult instances (Table 3), which demonstrates the importance of effective portfolio-based streamliner selection.

8.2. Frequency and magnitude of search reduction

The effectiveness of our streamliner method in term of improvement frequency (how often the selected streamlined models wins over the original model) and the magnitude of the improvement (in term of search reduction) are presented in Fig. 12 and Fig. 13. We begin by considering the setting where the instances are solved with the CP solver CHUFFED. The high improvement frequency of the Oracle on all problem classes demonstrates that there is almost always a streamliner in our portfolio that can be used to reduce search for a given unseen instance. As might be expected, the magnitude of the search reduction does vary with problem class. On Distribution A, for BACP, Car Sequencing, and Transshipment it is most pronounced, approaching one hundred percent, which would indicate a solution obtained with little or no search efforts, while for other problems such as BIBD, CoveringArray and Social Golfers, the reduction is generally much smaller.

Performance across all problem classes significantly improves for Distribution B, suggesting that the impact of streamlining grows with the difficulty of the problem instance. This is as expected: the size of the search space typically increases

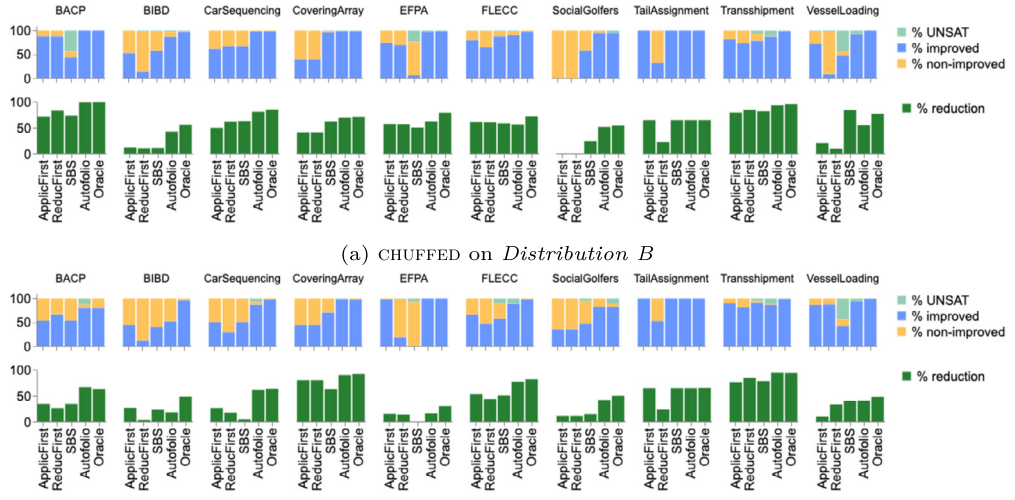
[8] In preliminary experiments, we also tried training AutoFolio on instances from distribution A only and evaluating the trained selection model on more difficult instances. We did not achieve good results, which suggested the lack of necessary information for the selection model to work properly on instance type that it never saw during the training.



(a) CHUFFED on *Distribution A*.

(b) LINGELING on *Distribution A*.

Fig. 12. Results with CHUFFED and LINGELING on *Distribution A*. The top of each pair of charts shows how frequently the associated approach produces an improvement (% improved), and also indicates the reason for failure to improve on the remainder of the instances: the instance was rendered unsatisfiable (% UNSAT), or the search completed more slowly than the unstreamlined model (% non-improved). The bottom of each pair of charts shows the magnitude of the solving time reduction on those instances where an improvement was obtained (% reduction). Hence, care must be taken when comparing approaches, since an infrequently applicable approach may do well on the few instances it does improve (e.g. the lexicographic scheduling approaches on the Social Golfers Problem). The best approaches are both frequently applicable and result in a large search reduction.



(a) CHUFFED on *Distribution B*

(b) LINGELING on *Distribution B*

Fig. 13. Results with CHUFFED and LINGELING on *Distribution B*. Detailed on meaning of the plots are described in Fig. 12.

with that of the instance, providing the opportunity for the selected streamliner to prune larger parts of the search space and reduce search further.

In several cases, our automated streamliner selection approaches are able to deliver a substantial fraction of the performance of the oracle in terms of the percentage of instances improved. The two simple scheduling approaches sometimes perform well, but for problem classes such as BIBD, CoveringArray (on both instance distributions) and SocialGolfers (on distribution B), their performance is relatively weak. The Single Best Streamliner approach, which requires no further training following the streamliner search, offers a fairly good compromise between performance and cost before solving unseen instances, especially on the easy instances (distribution A). However, the most robust performance comes from AutoFolio.

Performance in the SAT domain is generally less strong than for Constraint Programming. Although the performance of the oracle again indicates that there is almost always a streamliner in our portfolio that can improve search, in Distribution A on 4 of 10 problem classes (BIBD, CarSequencing, EFPA and SocialGolfers), the magnitude of this reduction is small. On the remaining problem classes, the performance is stronger and in some cases (CoveringArray and FLECC) exceeds the

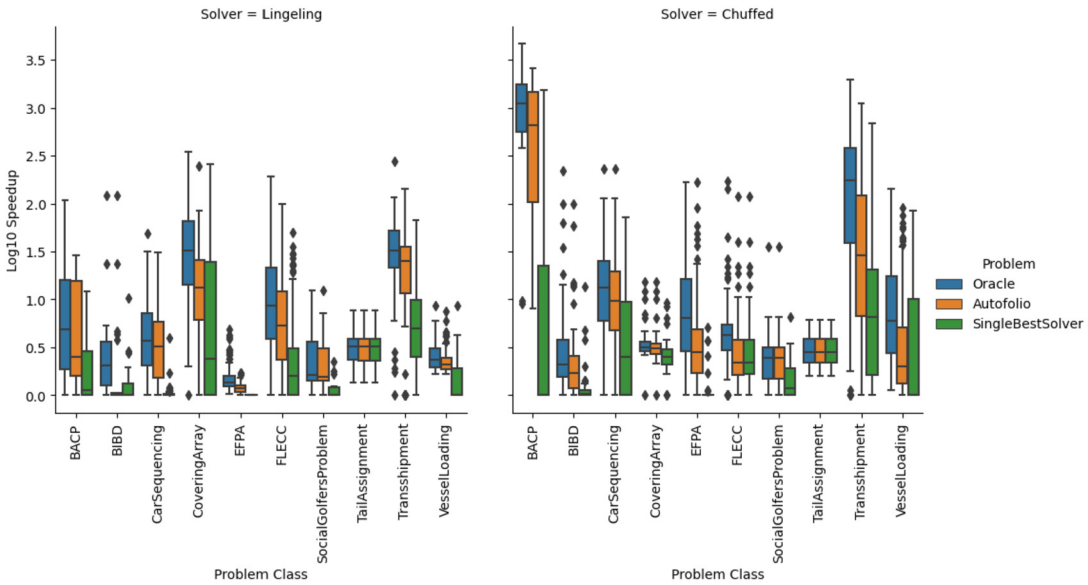


Fig. 14. Distribution of speedup values (in base 10 logarithmic scale) for the Oracle, AutoFolio and SingleBestSolver on *Distribution B*.

improvement delivered in the corresponding CP setting. Once again performance clearly improves on Distribution B relative to Distribution A, suggesting that for SAT the impact of streamlining also grows with difficulty.

8.3. A practical setting

By employing the algorithm selection techniques described in Section 7 our aim is to maximise the occasions on which streamlining produces a reduction in search effort. However, we cannot expect an aggressive technique such as streamlining to be universally applicable: in particular, the selected streamliner may render the instance under consideration unsatisfiable. Therefore, we envisage a practical setting in which a streamliner portfolio is a constituent of a wider portfolio containing other more conservative approaches. The simplest such setting, which we employ here, is to run the streamliner portfolio in parallel with the unstreamlined model, which will produce a solution if the selected streamliner renders a satisfiable instance unsatisfiable.

We evaluated this parallel configuration on both Distribution A and B. Our results are summarised in Table 2 and Table 3, which present the *Overall Speedup* of each approach across the ten different problem classes with two solving paradigms, CP and SAT. *Overall Speedup* represents the total time of the original model divided by the total streamlined time across all instances. This metric gives an indication of the overall reduction in search effort across each instance distribution.

Across both distributions, AutoFolio is clearly the best performing among our different streamliner selection approaches. It achieves geometric mean speedups of 2.01× and 1.57× for Distribution A and 3.74× and 2.99× for Distribution B, with maximum speedups on Distribution A of over 4× for both CP and SAT, and on Distribution B of over 40× for CP and over 10× for SAT. As in the results presented in Section 8.2, there is a pronounced increase in the speedups achieved for the more difficult *Distribution B* instances. Savile Row introduces a cost for translating models to solver input, which can increase in the presence of streamlining constraints. Since many of the instances in *Distribution A* are relatively easy to solve, this limits the speedup obtainable through streamlining. As difficulty increases in *Distribution B*, the benefits of streamlining become clear.

Overall Speedup is an aggregation metric, which can obscure the individual instance speedups being obtained. For example, if a streamliner is evaluated on 10 instances and for half its application makes them trivial reducing the solving time to near zero but for the other half it is unsatisfiable the *Overall Speedup* will be $\approx 2\times$. This does not provide a good indication that on half of the instance distribution the streamliner is decimating the search space to such a high degree. In order to better visualize this Fig. 14 shows the distribution of speedup values across instance distribution B for the Oracle, SingleBestSolver and AutoFolio methods. The minimum speedup obtained for a streamliner in this setup is 1 ($\log_{10}(0)$) as these are the cases where the chosen streamliner is not satisfiable and as such is solved by the original model providing no speedup. If we restrict our scope to individual problem classes it can be seen that the application of streamliners can provide substantial speedups.

Using the Chuffed CP solver, BACP and Transshipment are two problem classes where AutoFolio is able to obtain large speedups for a majority of the instance distribution. In the case of BACP the speedups range from $\approx 8\times$ to $\approx 2568\times$ with the 1st and 3rd quartiles having values at $\{q_1 \approx 103\times, q_3 \approx 1458\times\}$. For Transshipment the speedups range from $\approx 1\times$ to $\approx 1120\times$ with the 1st and 3rd quartiles having values at $\{q_1 \approx 6\times, q_3 \approx 120\times\}$. These large speedups are not just restricted

to *CP* and under the *SAT* paradigm CoveringArray and Transshipment are also examples where this search space decimation can occur. For Transshipment the speedups range from $\approx 1\times$ to $\approx 141\times$ with the 1st and 3rd quartiles having values at $\{q_1 \approx 25\times, q_3 \approx 35\times\}$. For CoveringArray the speedups range from $\approx 1\times$ to $\approx 244\times$ with the 1st and 3rd quartiles having values at $\{q_1 \approx 6\times, q_3 \approx 25\times\}$.

8.4. Cumulative CPU consumption

We must be mindful that all results presented thus far in this section are in terms of the reduction/speedup in time of the portfolio approach versus the *original* model. Since the portfolio method utilises two cores, one for the *original* model and one for the portfolio, this consumes more resources to arrive at a solution. We present a further analysis in Fig. 15 to illustrate the cumulative time spent by the scheduling methods across the instances comprising Distribution B. Here the cumulative time of the portfolio approach to solve all instances, taking into account both cores, is compared against that of the original model. For the CHUFFED solver on 9 of the 10 problems, BIBD being the exception, AutoFolio is still able to reduce the cumulative time with, in some cases, a substantial reduction. Speedups of BACP: $\approx 42.98x$, CarSequencing: $\approx 3.10x$, TailAssignment: $\approx 1.59x$ and Transshipment: $\approx 3.88x$ are attained. For LINGELING, on every problem except BIBD and EFPA positive speedups in cumulative time are achieved again with substantial results for some problems. Speedups of BACP: $\approx 2.37x$, CarSequencing: $\approx 1.66x$, CoveringArray: $\approx 5.96x$, FixedLengthErrorCorrectingCodes: $\approx 2.13x$, TailAssignment: $\approx 1.47x$ and Transshipment: $\approx 2.7x$ are attained.

It is useful to note that the cumulative time speedup of the portfolio approach is not always half that of the elapsed time speedup. The reason for this is that the portfolio approach does not always use twice the time of the *original* model. Let us use an example to illustrate this point. We have an instance which takes 100 s under the original model. If during evaluation the selected streamliner is proven to be unsatisfiable at $T = 50\,s$ then from $T = 50$ s to $T = 100$ s only one core will be utilised running the *original* model. The cumulative time speedup of the portfolio approach is then $\frac{2}{3}$ compared with a speedup of 1 in elapsed time.

9. Related work

Improving solving performance has been a major research goal in Constraint Programming. Streamlining is a very powerful method towards this goal, among methods like adding implied constraints, symmetry breaking constraints and dominance constraints. Typically each of these methods are initially explored on a small number of problem classes manually by experts. Once their efficacy is shown, research has moved to automating their application. Symmetry breaking has been most successfully automated among these methods, implied constraints and dominance breaking constraints comparatively less so. Streamlining has been successfully applied manually prior to our work, we present the first substantial step towards automating the application of streamlining in this paper. This section provides the necessary context in prior work in streamlining and the related research areas.

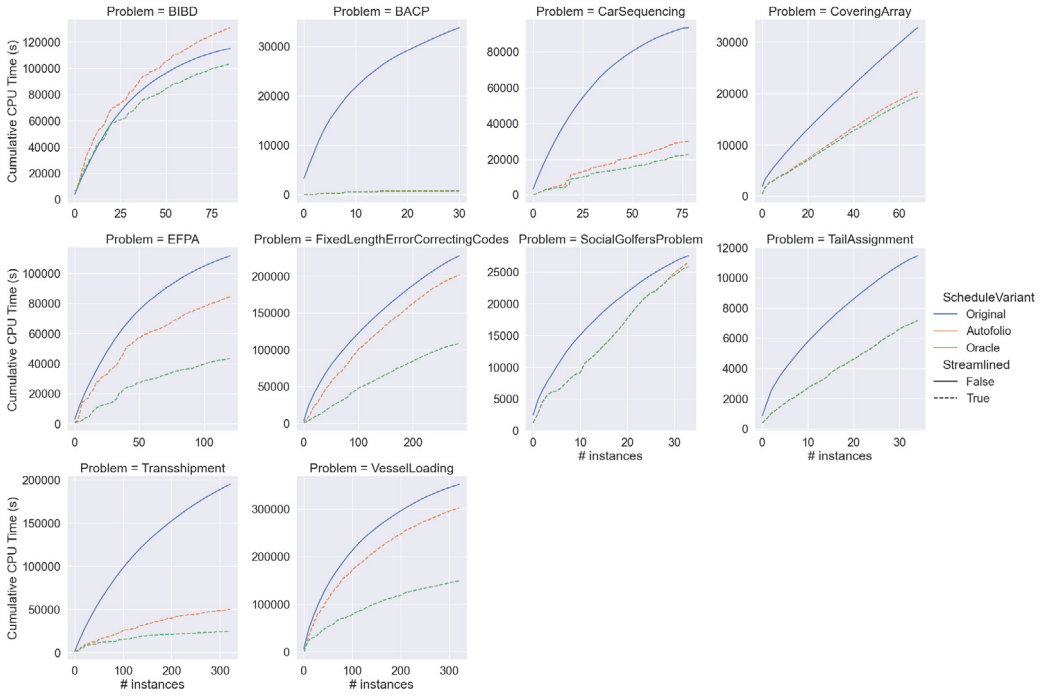
9.1. Manual streamlining

Gomes and Sellmann in their introductory work on *Streamlining* worked on problems from the field of Combinatorial Design [13]. One of the problems that they looked at was the construction of *Diagonally Ordered Magic Squares* (DOMS). They note that even for small sizes finding solutions to this problem was difficult, their base model could only find solutions up to size 9. They noticed a regularity in the solutions to the small instances: numbers within the magic square are quite evenly distributed. Intuitively this makes sense since numbers on each row, column, and diagonal have to sum to the same number and placing several large numbers on the same row is unlikely to lead to a solution. Once they identified this regularity, they posted a *streamliner* constraint that disallows similar numbers from appearing near each other. The streamlined model was then able to solve instances up to size 18.

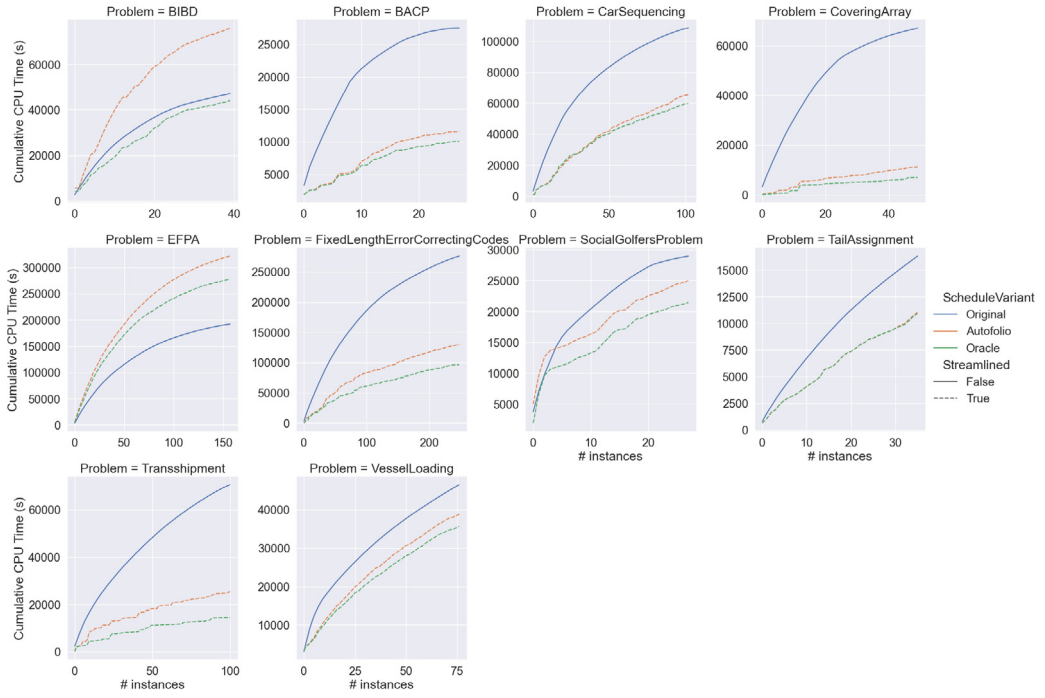
Using a similar methodology to Gomes and Sellmann [13], Kouril et al. applied streamlining constraints to the Van der Waerden numbers problem [68]. After observing patterns in the solutions to small instances, they added simple constraints that force or disallow certain sequences of values to occur in the solutions. Again, this led to a dramatic improvement in run time of the solver, allowing much tighter bounds to be computed.

Le Bras et al. used streamlining to help construct graceful double wheel graphs [17]. Constraints forcing certain parts of the colouring to form arithmetic sequences allow for the construction of colourings for much larger graphs. These constraints led to the discovery of a polynomial time construction algorithms for such colourings, and eventually to a proof that all double wheel graphs are graceful.

Finally Le Bras et al. made use of streamlining constraints to compute new bounds on the Erdős discrepancy problem [18]. Here the streamlining constraints were used to enforce periodicity in the solution, the improved Walters sequence to occur in the solution and a partially multiplicative property. Thanks to these streamlining constraints new bounds were discovered for this problem.



(a) CHUFFED



(b) LINGELING

Fig. 15. The cumulative CPU time of AutoFolio in comparison to that of the Oracle and the Original model, totalling across the test instances of Distribution B. The x-axis shows the total number of instances, sorted in descending order of instance difficulty (solving time) according to the original model. The y-axis shows the corresponding total cumulative CPU.

Streamlining proved to be valuable in all of these cases. Despite their success, their application was limited to mathematical and combinatorial design problems. The main reason preventing their widespread adoption has been the need for a laborious manual component that requires domain expertise to analyse solution patterns and derive common patterns. This motivated the automation of streamlining in this paper.

Streamlining is related to implied constraints, symmetry breaking constraints and dominance breaking constraints. All of these methods can be applied manually or (semi-)automatically to a given base model. Streamlining is orthogonal to these methods: it can fruitfully coexist with them in the same model.

9.2. Relation to model counting via XOR constraints

Gomes et al. [69] employ XOR constraints in order to obtain good quality lower bounds for model (meaning solution in this context) counting in SAT problems. Their approach involves repeatedly adding randomly chosen XOR constraints on the problem variables. The central idea of the approach is that each random XOR constraint cuts the search space approximately in half, which means that approximately half of the solutions remain satisfiable after the addition of an XOR constraint. By adding several XOR constraints they aim to bring the SAT problem to the boundary of being unsatisfiable. They provide a formal proof that with high probability, the number of XOR constraints required in this process determines the solution count. They also empirically study the relationship between the number of XOR constraints and the solution count. They report promising results including a good bound on the number of solutions.

The randomly chosen XOR constraints in this work are streamlining constraints, since they aim to remove some but not all solutions. However the focus is very different from ours, specifically they focus on counting solutions rather than efficiency of finding the first solution. In their empirical analysis they compare their XOR-based approximation method with exact solution counting methods. Exact solution counting methods require repeatedly solving the same problem, typically by adding solution blocking clauses to avoid enumerating the same solution twice. The XOR-based method is computationally faster than exact solution counting overall, however it is unlikely to be faster for finding a single solution. Indeed, as the authors note, adding large XORs can make a SAT solver inefficient. Nonetheless, this scheme might work as an option in our current framework, although we would need to consider in future work how best to integrate it with our method of generating streamliners from the structure present in an Essence specification.

9.3. Relation to implied constraints

Streamlining constraints are most closely related to *implied constraints*. An implied constraint is an additional constraint that is added to the model to increase the solving performance. They are also called *redundant* constraints, since the correctness of the model does not depend on them. There is a key difference between implied constraints and streamlining constraints: implied constraints are sound (they do not change the set of solutions).

Some previous work focuses on manually adding implied constraints to a model to increase the solving performance [1,2]. The scope for adding implied constraints is narrower due to the soundness requirement. Automated approaches ([3–5]) focus on deriving simple facts from the model constraints and using forward-chaining to generate more complex constraints that still hold for a set of instances. Automatically proving the soundness of these implied constraints is a challenging task in general, which limits the impact of these approaches.

9.4. Relation to symmetry breaking constraints

Breaking modelling symmetry has been shown to have a substantial impact in improving solving performance. There are two main approaches: dynamic approaches that generate symmetry breaking constraints during search [70], and static approaches that work by adding constraints to the model to break symmetries [71,43]. Static symmetry breaking constraints may allow the derivation of additional implied constraints [1]. Symmetry breaking constraints are either detected from the data structures in the model [72] or detected automatically by applying graph automorphism detection methods on the constraint graph [73]. Conjure generates static symmetry breaking constraints thanks to its high-level variable domains [74,75] and our streamlining approach works in conjunction with the symmetry breaking constraints.

9.5. Relation to dominance breaking constraints

A generalisation of symmetry is *dominance* between solutions. In the context of optimisation problems, dominance breaking constraints can result in dramatic speed ups [10–12]. A dominance breaking constraint disallows solutions that are known to be sub-optimal, as well as some optimal solutions, as long as at least one optimal solution remains. This is achieved by identifying a mapping between solutions and a condition under which this mapping will improve a solution. Preliminary work on automating parts of the derivation of dominance constraints shows promising results [76].

9.6. Portfolio approaches

In several contexts, no single algorithm has been seen to dominate all others on all instances of a computational problem. Algorithms often show complementary strengths, and the ideas of making use of algorithm portfolios have been investigated in various fields with great success [30]. For SAT, perhaps the most successful examples of portfolio approaches is SATzilla [77,78,66], the first portfolio approach that was shown to outperform stand-alone SAT solvers, and has exhibited strong performance in several SAT competitions. For CP, the portfolio approaches CPHydra [79] and SUNNY-CP [80,81] won

the Second International Constraint Solver Competition [82] and MiniZinc challenges 2015–2017 (open track), respectively. We refer to [64,30] for an overview of portfolio approaches and their achievements. To the best of our knowledge, this work is the first one that applies portfolio approaches in the context of streamliners for constraint modelling.

10. Conclusions and future work

Streamliner generation has been the exclusive province of human experts, requiring substantial effort in examining the solutions to instances of a problem class, manually forming candidate streamliners, and then testing their efficacy in practice. In this work we have presented the first completely automated method of generating effective streamliners, achieved through the exploitation of the structure present in abstract constraint specifications written in Essence, a best-first search among streamliner candidates, and a procedure to select and apply streamliners when faced with an unseen problem instance. Our empirical results demonstrate the success of our approach.

Streamlining constraints are typically used for constraint satisfaction problems. Streamlining in the context of optimisation is a challenge, because rather than rendering an instance unsatisfiable, an over-aggressive streamliner might disallow the best solution(s). Recently, the effective use of streamlining constraints for optimisation problems has been investigated in a preliminary study in [83] In this work, the authors present a way of producing automatically a portfolio of streamliners for optimisation problems. Each streamlined model in the portfolio represents a different balance between three criteria: how aggressively the search space is reduced, the proportion of training instances for which the streamliner admitted at least one solution, and the average reduction in quality of the objective value versus the unstreamlined model. This third criteria is important for optimization problems because the goal is to find a solution with an optimal objective value, not just any solution as in satisfaction problems.

In support of our new method, we present an automated approach to training and test instance generation, and provide several approaches to the selection and application of the streamliners from the portfolio. Empirical results demonstrate drastic improvements both to the time required to find good solutions early and to prove optimality on three problem classes.

A further important item of future work is to exploit the ability of Conjure to refine several alternative models from an Essence specification. Herein we have employed the default heuristic in Conjure in order to refine a single model from each streamlined specification, but although known generally to select effective models [19], this is not necessarily the best model among the set that Conjure can produce. We will add an extra degree of flexibility to the search for effective candidate streamliners by measuring the performance of several models for each streamlined specification. Selecting the best model of a streamlined specification among those available will lead to still greater performance gains. For SAT this approach could be extended further by searching over the possible SAT encodings of the Essence Prime models refined by Conjure. Preliminary results on a single problem class are promising [84].

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Ozgur Akgun reports financial support was provided by Engineering and Physical Sciences Research Council. Nguyen Dang reports financial support was provided by Leverhulme Trust. Ian Miguel reports financial support was provided by Engineering and Physical Sciences Research Council.

Data availability

We have shared a link to our code and all other data used in this paper.

Acknowledgements

This work is supported by the EPSRC grants EP/P015638/1 and EP/P026842/1, and Nguyen Dang is a Leverhulme Early Career Fellow. We used the Cirrus UK National Tier-2 HPC Service at EPCC (http://www.cirrus.ac.uk) funded by the University of Edinburgh and EPSRC (EP/P020267/1).

References

[1] A.M. Frisch, C. Jefferson, I. Miguel, Symmetry breaking as a prelude to implied constraints: a constraint modelling pattern, in: ECAI, vol. 16, 2004, p. 171.
[2] A.M. Frisch, I. Miguel, T. Walsh, Symmetry and implied constraints in the steel mill slab design problem, in: Proc. CP01 Workshop on Modelling and Problem Formulation, 2001.
[3] J. Charnley, S. Colton, I. Miguel, Automatic generation of implied constraints, in: ECAI, vol. 141, 2006, pp. 73–77.
[4] S. Colton, I. Miguel, Constraint generation via automated theory formation, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2001, pp. 575–579.
[5] A.M. Frisch, I. Miguel, T. Walsh, CGRASS: a system for transforming constraint satisfaction problems, in: Recent Advances in Constraints, Springer, 2003, pp. 15–30.

[6] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh, Symmetry in matrix models, in: Proceedings of SymCon, vol. 1, Citeseer, 2001.
[7] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh, Propagation algorithms for lexicographic ordering constraints, Artif. Intell. 170 (10) (2006) 803–834.
[8] A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, I. Miguel, Symmetry in the generation of constraint models, in: Proceedings of the International Symmetry Conference, 2007.
[9] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh, Filtering algorithms for the multiset ordering constraint, Artif. Intell. 173 (2) (2009) 299–328.
[10] S. Prestwich, J.C. Beck, Exploiting dominance in three symmetric problems, in: Fourth International Workshop on Symmetry and Constraint Satisfaction Problems, Citeseer, 2004, pp. 63–70.
[11] G. Chu, P.J. Stuckey, Dominance breaking constraints, Constraints 20 (2) (2015) 155–182.
[12] J.H. Lee, A.Z. Zhong, Exploiting functional constraints in automatic dominance breaking for constraint optimization, in: 28th International Conference on Principles and Practice of Constraint Programming (CP 2022), Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
[13] C. Gomes, M. Sellmann, Streamlined constraint reasoning, in: Principles and Practice of Constraint Programming - CP 2004, Springer, 2004, pp. 274–289.
[14] B. Smith, CSPLib problem 001: car sequencing, http://www.csplib.org/Problems/prob001.
[15] I.P. Gent, Two results on car-sequencing problems, Report University of Strathclyde, APES-02-98 7, 1998.
[16] M. Kouril, J. Franco, Resolution tunnels for improved sat solver performance, in: Theory and Applications of Satisfiability Testing, Springer, 2005, pp. 143–157.
[17] R. Le Bras, C.P. Gomes, B. Selman, Double-wheel graphs are graceful, in: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, AAAI Press, 2013, pp. 587–593.
[18] R. Le Bras, C.P. Gomes, B. Selman, On the Erdos discrepancy problem, in: Principles and Practice of Constraint Programming: 20th International Conference, Proceedings, CP 2014, Lyon, France, September 8-12, 2014, vol. 8656, Springer, 2014, p. 440.
[19] Ö. Akgün, Extensible automated constraint modelling via refinement of abstract problem specifications, Ph.D. thesis, University of St Andrews, 2014.
[20] Ö. Akgün, I.P. Gent, C. Jefferson, I. Miguel, P. Nightingale, Breaking conditional symmetry in automated constraint modelling with Conjure, in: ECAI, 2014, pp. 3–8.
[21] Özgür Akgün, A.M. Frisch, I.P. Gent, C. Jefferson, I. Miguel, P. Nightingale, Conjure: automatic generation of constraint models from problem specifications, Artif. Intell. 310 (2022) 103751, https://doi.org/10.1016/j.artint.2022.103751, https://www.sciencedirect.com/science/article/pii/S0004370222000911.
[22] A.M. Frisch, C. Jefferson, B.M. Hernandez, I. Miguel, The rules of constraint modelling, in: Proc. of the IJCAI 2005, 2005, pp. 109–116.
[23] A.M. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, I. Miguel, Essence: a constraint language for specifying combinatorial problems, Constraints 13 (3) (2008) 268–306, https://doi.org/10.1007/s10601-008-9047-y.
[24] P. Nightingale, Özgür Akgün, I.P. Gent, C. Jefferson, I. Miguel, Automatically improving constraint models in Savile row through associative-commutative common subexpression elimination, in: Principles and Practice of Constraint Programming - CP 2014, Springer, 2014.
[25] P. Nightingale, Ö. Akgün, I.P. Gent, C. Jefferson, I. Miguel, P. Spracklen, Automatically improving constraint models in Savile row, Artif. Intell. 251 (2017) 35–61.
[26] P. Van Hentenryck, The OPL Optimization Programming Language, MIT Press, 1999.
[27] N. Nethercote, P.J. Stuckey, R. Becket, S. Brand, G.J. Duck, G. Tack, Minizinc: towards a standard CP modelling language, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2007, pp. 529–543.
[28] P. Nightingale, A. Rendl, Essence' description, arXiv preprint, arXiv:1601.02865, 2016.
[29] Ö. Akgün, A.M. Frisch, I.P. Gent, B.S. Hussain, C. Jefferson, L. Kotthoff, I. Miguel, P. Nightingale, Automated symmetry breaking and model selection in Conjure, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2013, pp. 107–116.
[30] P. Kerschke, H.H. Hoos, F. Neumann, H. Trautmann, Automated algorithm selection: survey and perspectives, Evol. Comput. 27 (1) (2019) 3–45.
[31] J. Wetter, Ö. Akgün, I. Miguel, Automatically generating streamlined constraint models with Essence and Conjure, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2015, pp. 480–496.
[32] P. Spracklen, Ö. Akgün, I. Miguel, 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, Springer, 2018, pp. 362–372.
[33] G. Chu, M. de la Banda, P. Stuckey, Exploiting subproblem dominance in constraint programming, Constraints 17 (1) (2012) 1–38, https://doi.org/10.1007/s10601-011-9112-9, code available from https://github.com/chuffed/chuffed.
[34] A. Biere, Lingeling essentials, a tutorial on design and implementation aspects of the sat solver lingeling, in: POS@SAT, vol. 27, 2014, p. 88.
[35] P. Spracklen, N. Dang, Özgür Akgün, I. Miguel, Stacs-cp/automated-streamliner-portfolios, https://doi.org/10.5281/zenodo.7747141, Mar. 2023.
[36] R. Le Bras, C.P. Gomes, B. Selman, Double-wheel graphs are graceful, in: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence, IJCAI '13, AAAI Press, 2013, pp. 587–593, http://dl.acm.org/citation.cfm?id=2540128.2540214.
[37] K. Smith-Miles, T.T. Tan, Measuring algorithm footprints in instance space, in: 2012 IEEE Congress on Evolutionary Computation, IEEE, 2012, pp. 1–8.
[38] M. Lindauer, H.H. Hoos, F. Hutter, T. Schaub, Autofolio: an automatically configured algorithm selector, J. Artif. Intell. Res. 53 (2015) 745–778.
[39] F. Dunlop, J. Enright, C. Jefferson, C. McCreesh, P. Prosser, J. Trimble, Expression of graph problems in a high level modelling language, in: Proceedings of the International Workshop on Graphs and Constraints, 2018.
[40] B. Hnich, Z. Kiziltan, T. Walsh, CSPLib problem 030: balanced academic curriculum problem (BACP), http://www.csplib.org/Problems/prob030.
[41] S. Prestwich, CSPLib problem 028: balanced incomplete block designs, http://www.csplib.org/Problems/prob028.
[42] E. Selensky, CSPLib problem 045: the covering array problem, http://www.csplib.org/Problems/prob045.
[43] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh, Breaking row and column symmetries in matrix models, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2002, pp. 462–477.
[44] P. Nightingale, CSPLib problem 055: equidistant frequency permutation arrays, http://www.csplib.org/Problems/prob055.
[45] S. Huczynska, P. McKay, I. Miguel, P. Nightingale, Modelling equidistant frequency permutation arrays: an application of constraints to mathematics, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2009, pp. 50–64.
[46] A. Frisch, C. Jefferson, I. Miguel, CSPLib problem 036: fixed length error correcting codes, http://www.csplib.org/Problems/prob036.
[47] A.M. Frisch, C. Jefferson, I. Miguel, Constraints for breaking more row and column symmetries, in: Principles and Practice of Constraint Programming–CP 2003: 9th International Conference, Proceedings, CP 2003, Kinsale, Ireland, September 29–October 3, 2003, vol. 9, Springer, 2003, pp. 318–332.
[48] Özgür Akgün, CSPLib problem 83: transshipment problem, http://www.csplib.org/Problems/prob083.
[49] Özgür Akgün, CSPLib problem 115: tail assignment, http://www.csplib.org/Problems/prob115.
[50] W. Harvey, CSPLib problem 010: social golfers problem, http://www.csplib.org/Problems/prob010.
[51] K. Brown, CSPLib problem 008: Vessel loading, http://www.csplib.org/Problems/prob008.
[52] Ö. Akgün, N. Dang, I. Miguel, A.Z. Salamon, C. Stone, Instance generation via generator instances, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2019, pp. 3–19.
[53] I.P. Gent, C. Jefferson, I. Miguel Minion, A fast scalable constraint solver, in: ECAI, vol. 141, 2006, pp. 98–102.
[54] Ö. Akgün, N. Dang, I. Miguel, A.Z. Salamon, P. Spracklen, C. Stone, Discriminating instance generation from abstract specifications: a case study with CP and MIP, in: International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Springer, 2020, pp. 41–51.

[55] N. Dang, O. Akgün, J. Espasa, I. Miguel, P. Nightingale, A framework for generating informative benchmark instances, in: C. Solnon (Ed.), 28th International Conference on Principles and Practice of Constraint Programming (CP 2022), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 235, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2022, 18, https://drops.dagstuhl.de/opus/volltexte/2022/16647.
[56] M. López-Ibáñez, J. Dubois-Lacoste, L.P. Cáceres, M. Birattari, T. Stützle, The irace package: iterated racing for automatic algorithm configuration, Oper. Res. Perspect. 3 (2016) 43–58.
[57] K. Smith-Miles, D. Baatar, B. Wreford, R. Lewis, Towards objective measures of algorithm performance across instance space, Comput. Oper. Res. 45 (2014) 12–24.
[58] R. Amadini, M. Gabbrielli, J. Mauro, An enhanced features extractor for a portfolio of constraint solvers, in: SAC 2014: Proceedings of the 29th Annual ACM Symposium on Applied Computing, ACM, 2014, pp. 1357–1359, code available from https://github.com/CP-Unibo/mzn2feat.
[59] K. Pearson, Principal components analysis, Lond. Edinb. Dubl. Philos. Mag. J. Sci. 6 (2) (1901) 559.
[60] G. Hamerly, C. Elkan, Learning the k in k-means, in: Advances in Neural Information Processing Systems, 2004, pp. 281–288.
[61] W. Wang, M. Sebag, Hypervolume indicator and dominance reward based multi-objective Monte-Carlo tree search, Mach. Learn. 92 (2–3) (2013) 403–429.
[62] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P.I. Cowling, S. Tavener, D. Perez, S. Samothrakis, S. Colton, et al., A survey of Monte Carlo tree search methods, IEEE Trans. Comput. Intell. AI Games (2012).
[63] L. Xu, H. Hoos, K. Leyton-Brown, Hydra: automatically configuring algorithms for portfolio-based selection, in: Twenty-Fourth AAAI Conference on Artificial Intelligence, 2010.
[64] L. Kotthoff, Algorithm selection for combinatorial search problems: a survey, in: Data Mining and Constraint Programming, Springer, 2016, pp. 149–190.
[65] J.R. Rice, et al., The algorithm selection problem, Adv. Comput. 15 (1976) 65–118.
[66] L. Xu, F. Hutter, J. Shen, H.H. Hoos, K. Leyton-Brown, Satzilla2012: improved algorithm selection based on cost-sensitive classification models, in: Proceedings of SAT Challenge, 2012, pp. 57–58.
[67] F. Hutter, H.H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: International Conference on Learning and Intelligent Optimization, Springer, 2011, pp. 507–523.
[68] M. Kouril, J. Franco, Resolution tunnels for improved SAT solver performance, in: International Conference on Theory and Applications of Satisfiability Testing, Springer, 2005, pp. 143–157.
[69] C.P. Gomes, a. Sabharwal, B. Selman, Model counting: a new strategy for obtaining good bounds, 2006.
[70] I.P. Gent, B. Smith, Symmetry Breaking During Search in Constraint Programming, Citeseer, 1999.
[71] I.P. Gent, K.E. Petrie, J.-F. Puget, Symmetry in Constraint Programming, Foundations of Artificial Intelligence, vol. 2, Elsevier, 2006, pp. 329–376.
[72] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh, Matrix modelling, in: Proc. of the CP-01 Workshop on Modelling and Problem Formulation, 2001, p. 223.
[73] J.-F. Puget, Automatic detection of variable and value symmetries, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2005, pp. 475–489.
[74] Ö. Akgün, I. Miguel, C. Jefferson, A.M. Frisch, B. Hnich, Extensible automated constraint modelling, in: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI Press, 2011, pp. 4–11.
[75] O. Akgün, A.M. Frisch, I.P. Gent, B.S. Hussain, C. Jefferson, L. Kotthoff, I. Miguel, P. Nightingale, Automated Symmetry Breaking and Model Selection in Conjure, Lect. Notes Comput. Sci., vol. 8124, 2013, pp. 107–116.
[76] C. Mears, M.G. De La Banda, Towards automatic dominance breaking for constraint optimization problems, in: Twenty-Fourth International Joint Conference on Artificial Intelligence, 2015.
[77] E. Nudelman, K. Leyton-Brown, A. Devkar, Y. Shoham, H. Hoos, Satzilla: an algorithm portfolio for sat, solver description, SAT competition 2004, 2004.
[78] L. Xu, F. Hutter, H.H. Hoos, K. Leyton-Brown, Satzilla: portfolio-based algorithm selection for sat, J. Artif. Intell. Res. 32 (2008) 565–606.
[79] E. O'Mahony, E. Hebrard, A. Holland, C. Nugent, B. O'Sullivan, Using case-based reasoning in an algorithm portfolio for constraint solving, in: Irish Conference on Artificial Intelligence and Cognitive Science, 2008, pp. 210–216.
[80] R. Amadini, M. Gabbrielli, J. Mauro, Sunny-cp: a sequential cp portfolio solver, in: Proceedings of the 30th Annual ACM Symposium on Applied Computing, 2015, pp. 1861–1867.
[81] R. Amadini, M. Gabbrielli, J. Mauro, A multicore tool for constraint solving, arXiv preprint, arXiv:1502.03986, 2015.
[82] M.-R. Van Dongen, C. Lecoutre, O. Roussel, Proceedings of the 2nd international constraint solver competition, 2008.
[83] P. Spracklen, N. Dang, Ö. Akgün, I. Miguel, Automatic streamlining for constrained optimisation, in: International Conference on Principles and Practice of Constraint Programming, Springer, 2019, pp. 366–383.
[84] P. Spracklen, N. Dang, Ö. Akgün, I. Miguel, Towards portfolios of streamlined constraint models: a case study with the balanced academic curriculum problem, arXiv preprint, arXiv:2009.10152, 2020.