Streamlined Constraint Reasoning: An Automated Approach from High Level Constraint Specifications

Patrick Spracklen

University of
St Andrews

This thesis is submitted in partial fulfilment for the degree of

Doctor of Philosophy (PhD)

at the University of St Andrews

November 2021

Abstract

Constraint Programming (CP) is a powerful technique for solving large-scale combinatorial (optimisation) 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 thesis 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.

First, we propose a method for the generation and evaluation of streamliner conjectures automatically from the type structure present in an Essence specification. Second, the possible streamliner combinations are structured into a lattice and a multi-objective search method for searching the lattice of combinations and building a portfolio of streamliner combinations is defined. Third, the problem of *"Streamliner Selection"* is introduced which deals with selecting from the portfolio an effective streamliner for an unseen instance. The work is evaluated by presenting two sets of experiments on a variety of problem classes. Lastly, we explore the effect of model selection in the

context of streamlined specifications and discuss the process of streamlining for Constrained Optimization Problems.

Underpinning Research Data or Digital Outputs

Candidate's declaration

I, Patrick Spracklen, understand that by declaring that I have original research data or digital outputs, I should make every effort in meeting the University's and research funders' requirements on the deposit and sharing of research data or research digital outputs.

Date	12/8/22	Signature of candidate

Permission for publication of underpinning research data or digital outputs

We understand that for any original research data or digital outputs which are deposited, we are giving permission for them to be made available for use in accordance with the requirements of the University and research funders, for the time being in force.

We also understand that the title and the description will be published, and that the underpinning research data or digital outputs will be electronically accessible for use in accordance with the license specified at the point of deposit, unless exempt by award of an embargo as requested below.

The following is an agreed request by candidate and supervisor regarding the publication of underpinning research data or digital outputs:

No embargo on underpinning research data or digital outputs.

Date	12/8/22	Signature of candidate
Date	12/8/22	Signature of supervisor

Funding

This work was supported by: EPSRC funding award EP/N509759/1

Research Data/Digital Outputs access statement

Research data underpinning this thesis is available at: `https://doi.org/10.17630/9805a859-d08d-4390-b986-7fce98f7fe70`

Contents

Chapter One

Introduction

Challenging combinatorial problems, from domains such as planning, scheduling, packing or configuration, often form *problem classes*: families of problem instances related by a common set of parameters. Constraint Programming (CP) [170] provides a declarative framework where a problem is defined as a set of decision variables, each having an associated finite domain, and constraint expressions are added to limit the assignments that are allowed in a feasible solution. In general it can be split up into two stages: *modeling* is the first and incorporates the process of turning an abstract problem definition (CSP) into a concrete model suitable for input to a solver. The second stage *solving* uses search and inference to find a solution, an assignment of values to the problem variables that satisfies all constraints, to the CSP.

CP offers a powerful means to solve these types of problem classes and over the last few decades there has been a large amount of work and progress in improving the performance of CP in these two areas. It is widely agreed that the *Modeling* stage is the most important in determining the solving difficulty of the underlying CSP. As Smith once noted *" there is abundant evidence that how the problem to be solved is modelled as a Constraint Satisfaction Problem (CSP) can have a dramatic effect on how easy it is to find a solution, or indeed whether it can realistically be solved at all."* [170] Freuder also stated *"It is one thing to say 'all one has to do is express the problem constraints.' It is another to express them in a manner which permits efficient solution"* [60] When modeling, decisions have to be made on how to represent the problem, in particular the types of the variables and how the constraints are encoded. These decisions can have considerable implications for the performance of the resultant model and overall difficulty of the problem. [164]

For the well known N-Queens problem, Figure 1.1, where the goal is to place n queens on a n x n chess board such that no queen can attack each other, two different representations

	Q		
			Q
Q			
		Q	

Figure 1.1: A solution to the 4 Queens problem [105]

can have drastically different search spaces. The first representation formulates the problem as placing each queen on one of the available squares of the chessboard. For the 4-Queens problem seen in Figure 1.1 the overall size space size would be 16^4. However, a different representation might take advantage of the knowledge that there is exactly one queen per row and instead formulate the problem as: placing a queen in each of the 4 different rows which reduces the search space to only size 4^4 [61].

With the global constraint catalog [16] containing over 350 constraints across 2800 pages it is very difficult for a novice user to formulate an efficient representation of the problem at hand. This problem known as the *modeling bottleneck* is one of the the key challenges facing the constraints field. There has been a large amount of progress in recent years in automatically improving the modeling process. These include case-based reasoning [136], machine learning to generate constraint models [21], tailoring of intermediate level model representations [168] and refinement of abstract constraint model specifications [67, 58, 142]

Even with these advancements in modelling in certain problems the search spaces involved are so vast that the model initially formulated is insufficient for the solver to find a solution to the more difficult instances of a problem class in a timely manner. In response, a natural step is to try and constrain the model further to reduce the size of the problem search space. There are various techniques for doing this, which will be discussed in more detail in Chapter 2. One approach is the identification of *symmetry* in the model. A *symmetry* is a permutation of the {variable, value} pairs that doesn't affect the solutions and if present means there are equivalent states within the search space. Once a *symmetry* has been identified *symmetry breaking* [82] can then be performed to remove these redundant parts and reduce the size of the overall space that has to be searched. Another approach is to add *implied* constraints, which can be inferred from the initial model and are therefore guaranteed to be sound. Manual [70, 72] and automated [36, 48, 71] approaches to generating implied constraints have been successful. Both *symmetry-breaking* and *implied* *constraints* are solution preserving techniques.

If none of these techniques is able to improve performance sufficiently, for satisfiable problems a more aggressive step is to add streamliner constraints [84], 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 offered by the previous approaches 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 [84, 124, 128, 130]. For example Gomes and Sellmann [84] added a streamliner requiring a latin square structure when searching for diagonally ordered magic squares [84].

The overarching goal of the work in this thesis is to: *completely automate the process of generating effective streamlined models to produce a substantial reduction in search effort across a diverse range of problems.* We begin with an overview of the architecture of our system, before explaining each of its components in detail in the subsequent chapters. Given a problem class of interest our streamlining approach, which is completely automated, proceeds in three main phases as seen by Figure 1.3. First candidate streamliners are generated from an ESSENCE specification (Section 1.2). Candidate streamliners are then evaluated and selected to construct a portfolio of streamliners with complementary strengths (Section 1.3). 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 1.4).

1.1 Brief introduction of the toolchain

To provide a better understanding for the following sections, which discuss the architecture and phases of our system, it is necessary to briefly introduce the components of the toolchain that we leverage:

Essence ESSENCE is a language for writing problem specifications which operates at a level of abstraction above which modelling decisions are made [65]. Figure 1.4 provides an example of an ESSENCE specification. ESSENCE does this by providing the ability to have decisions variables whose domain values are combinatorial objects, such as *set*, *function*, *partition*, *etc* which can be nested to an arbitrary depth. This

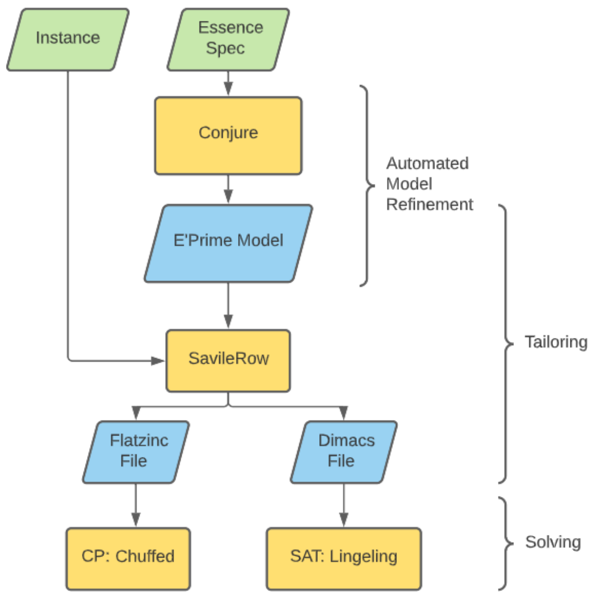


Figure 1.2: A top down view of how Essence, Conjure and Savile Row are integrated into a toolchain tasked with solving combinatorial problems

allows the problem specification to be stated more directly and naturally and avoids the need for the user to make any modelling decisions.

Conjure Conjure [1, 4] is an automated refinement system which operates on an Essence specification to produce a set of constraint models in the solver independent modelling language Essence Prime [155, 154]. Conjure operates at the problem class level and is only invoked once to create the Essence Prime model.

Savile Row Savile Row [153, 154] is a modelling assistant for constraint programming that translates an Essence Prime model into the input language of a target solver. One of the benefits of Essence Prime is that it is a solver independent modelling language and Savile Row provides the ability to target multiple different solving paradigms and solvers.

Figure 1.2 provides an overview of how these components are combined into a tool chain used for solving combinatorial problems. A more thorough description is provided in Chapter 2.

1.2 Phase 1: Streamliner Generation

Our approach is situated in the automated constraint modelling system Conjure [6]. This system takes as input a specification in the abstract constraint specification language Essence [62, 66]. Figure 1.4 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.

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 Figure 1.4. For instance, a streamliner that is generated from this specification enforces that *approximatelyHalf* of the *range* of the *car* function takes *odd* values. We cannot expect a streamliner of this nature to be universally satisfiable, however for certain instances it may retain at least one solution. In these cases as the search space has been restricted by a high degree a solver may be able to uncover this solution faster. The potential to remove solutions is what differentiates the process of *streamlining* from that of other techniques such as *symmetry* and *implied constraints*.

The modified specification is refined automatically into a streamlined constraint model by Conjure. These streamliners are generated through a set of prebuilt rules in Conjure that pattern match against the types of the decision variables in the Essence specification. Generally these rules will generate a number of candidate streamliners, 36 are generated for the specification listed in Figure 1.4. The derivation and application of these rules will be discussed in more detail in Chapter 3. Identifying and adding the streamlining constraints at this level of abstraction is considerably easier than working directly with the lower level 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.

1.3 Phase 2: Portfolio Construction

Gomes & Selman [129] first identified the benefit of combining different streamliner constraints together to further constrain a model and improve the deductions that can be made during search. In order to utilize streamliner combinations the generated candidate streamliners can be composed into a lattice structure with the root being the original Essence specification. Search is then performed on this lattice structure to identify effective streamlined combinations and build a portfolio of streamlined models (Chapter 5).

The rule based generation system in Conjure works purely upon the *types* present in the specification and is oblivious to whether or not the generated constraint is effective or satisfiable for the given problem. To reason about the effectiveness, each streamlined model is evaluated on a diverse (to ensure an unbiased estimate) instance set drawn from the problem class. To evaluate the performance of a streamlined specification, Conjure is used to refine the specification into a constraint model (Figure 3.9). Via the constraint modelling assistant tool Savile Row [153, 154], the streamlined model is translated into the input for a CP or SAT solver (Figure 3.9)

In order to facilitate the evaluations, a set of benchmark instances from the given problem class is required. We perform automated instance generation from the Essence specification of the problem class using the system proposed by Akgun et al [2]. After obtaining the instances, Feature Generation and Clustering are used to create a compressed training set suitable for use when searching the streamliner lattice. See Chapter 4 for further details.

1.4 Phase 3: Streamliner Selection and Application

Once the streamliner portfolio has been constructed it may be used in the solution of unseen instances from the problem class under consideration. Given such an instance, one or more streamlined models are selected from the portfolio and scheduled in an attempt to reduce the search effort to find a solution. We have implemented several automated selection procedures, which are explained in detail in Chapter 6. One approach is to use the features of the instance to decide which streamlined model to employ. This may result in a better selection, but does incur the additional cost of training a selector for the portfolio. Other approaches are less informed, but have a lower overhead. These include simply selecting the streamlined model from the portfolio that was observed to perform best on average during portfolio construction, and scheduling streamlined models to promote either aggressive search reduction or retaining instance satisfiability.

Overall these three phases may seem computationally expensive and time consuming. However it should be noted that this process is operating at the problem class level (families of problems related by a common set of parameters) and so the cost is amortised over the entire problem class for which the streamliner portfolios are applicable.

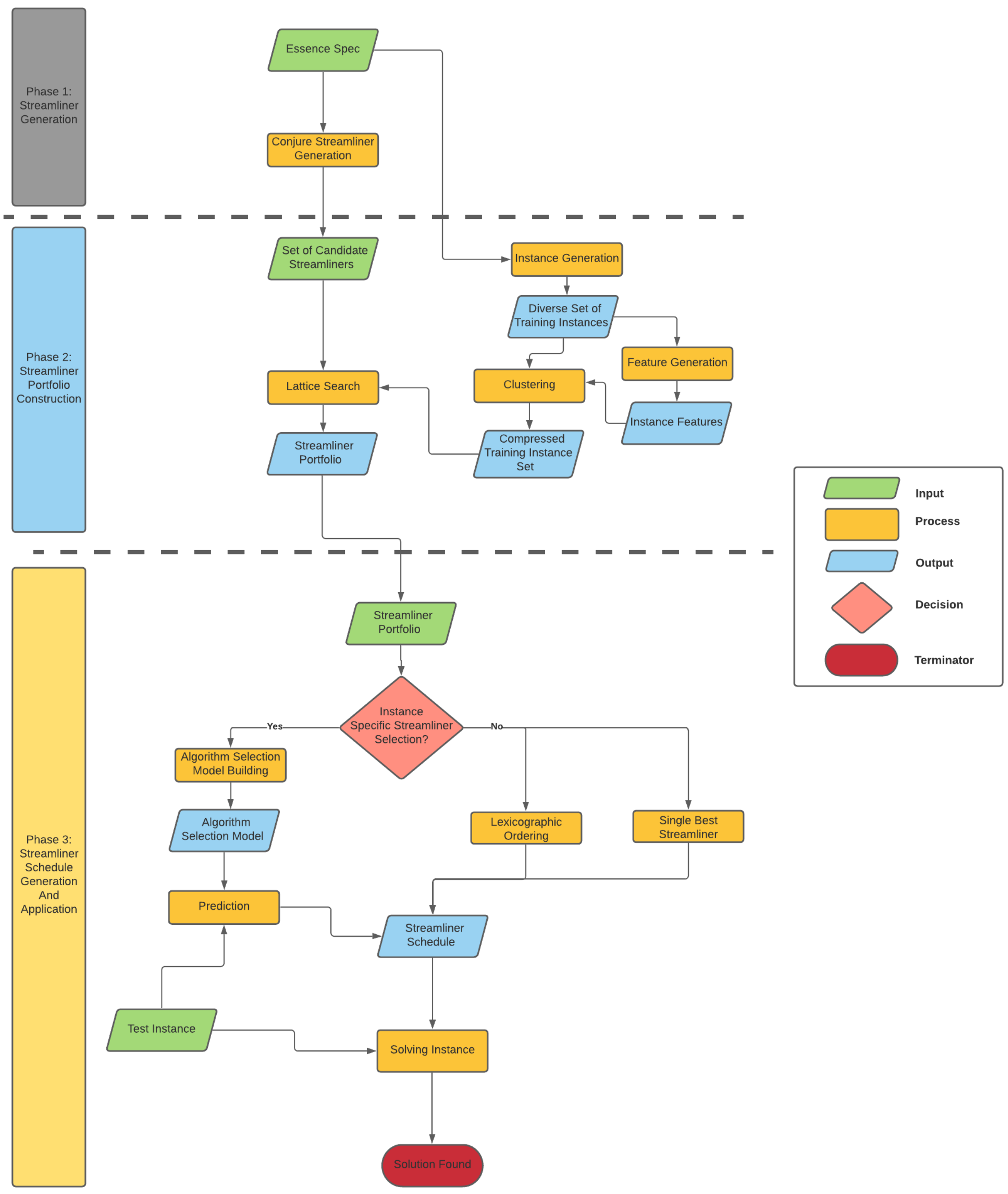

Phase 1: Streamliner Generation
Essence Spec
Conjure Streamliner Generation
Phase 2: Streamliner Portfolio Construction
Set of Candidate Streamliners
Instance Generation
Diverse Set of Training Instances
Feature Generation
Clustering
Instance Features
Lattice Search
Streamliner Portfolio
Compressed Training Instance Set
Input
Process
Output
Decision
Terminator
Phase 3: Streamliner Schedule Generation And Application
Streamliner Portfolio
Instance Specific Streamliner Selection?
Yes
No
Algorithm Selection Model Building
Algorithm Selection Model
Lexicographic Ordering
Single Best Streamliner
Prediction
Streamliner Schedule
Test Instance
Solving Instance
Solution Found

Figure 1.3: A summary of the main components and flow of our procedure for automatic generation of Streamliner Portfolios. Our approach begins with the abstract specification of a problem in ESSENCE. CONJURE is used to automatically generate a set of candidate streamliners (Chapter 3). Instance generation is performed on the ESSENCE specification via generator instances [2] to produce a diverse training set (Chapter 4). Feature Generation (Section 4.2) and Clustering (Section 4.3) are then used to create a compressed Training Set suitable for input into our Lattice Search (Chapter 5). The output of the system is a portfolio of streamliners with complementary strengths that will be used for solving any unseen instance of the same problem (Chapter 6).

```
given n_cars, n_classes, n_options : int(1..)
given quantity : function (total) Class  --> int(1..),
      maxcars : function (total) Option --> int(1..),
      blksize_delta : function (total) Option --> int(1..),
      usage : relation (minSize 1) of ( Class * Option )
letting Slots  be domain int(1..n_cars),
        Class  be domain int(1..n_classes),
        Option be domain int(1..n_options),
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-(maxcars(opt)+blksize_delta(opt))) .
    (sum i : int(s..s+(maxcars(opt)+blksize_delta(opt))-1) .
      toInt(usage(car(i),opt))) <= maxcars(opt)
```

Figure 1.4: Essence specification of the Car Sequencing Problem [178], shown to be NP-complete [80]. 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`). The cars requiring a certain option must not be bunched together, otherwise the station will not be able to cope (`maxcars`). Furthermore, the stations have been designed to handle at most a certain total number of the cars passing along the assembly line (`maxcars + blksize_delta`) at a time. Consequently, the cars must be arranged in a sequence so that the capacity of each station is never exceeded.

1.5 Thesis Statement

The thesis defended in this dissertation is as follows: Through the use of structure present in a high level abstract Essence specification a range of effective streamliner constraints can be automatically generated. A best first search method can then be used to search the single candidate streamliners and the lattice of combinations to find effective conjectures and produce high quality streamliner portfolios. Large reductions in search effort across unseen instance distributions can then be attained through the scheduling of these respective portfolios.

1.6 Publications

In chronologic order of publication date:

- **Automatic generation and selection of streamlined constraint models via monte carlo search on a model lattice** [184]

  This work showcased the automatic generation of streamliner constraints from an abstract ESSENCE specification as well as the implementation of a best first methodology to search for effective streamliners and its comparison to simpler Breadth First Search and Depth First Search methods.

  Disclaimer: There were two additional co-authors of this paper that contributed to the development of these ideas. However, I had significant contributions : I added the streamliner generation rules to the CONJURE framework and built the system for searching the lattice of combinations and evaluating each streamlined model. I designed and implemented the best-first search algorithm that was used. I also designed, implemented and ran all of the experiments.

- **Automatic streamlining for constrained optimisation** [185]

  This work presented the first automated approach to generating streamliners automatically for constrained optimisation problems. Prior limitations of generating only a single *'best'* streamliner were removed by providing a method to produce a portfolio of streamliners each representing 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.

  Disclaimer: There were three additional co-authors of this paper that contributed to the development of these ideas. However, I had significant contributions : I composed the multi-objective reward function that was used to balance the competing properties of the streamliner conjectures. I designed and built the UCB selector that experimentally performed the best. I also designed, implemented and ran all of the experiments.

- **Towards Portfolios of Streamlined Constraint Models: A Case Study with the Balanced Academic Curriculum Problem** [186]

  The refinement of streamlined Essence specifications into constraint models suitable for input to constraint solvers gives rise to a large number of modelling choices in addition to those required for the base Essence specification. Prior automated streamlining approaches were limited in evaluating only a single default encoding for each streamlined specification. In this work we explore the effect of model selection in the context of streamlined specifications with reference to the Balanced Academic Curriculum Problem.

Disclaimer: There were three additional co-authors of this paper that contributed to the development of these ideas. However, I had significant contributions : I was in charge of modifying the streamliner generation system to allow it to handle multiple models. I implemented the three different forms of racing that we utilized as well as designed, implemented and ran all of the experiments.

1.7 Contributions

The principal contribution of this thesis is: *the complete automation of the previously laborious manual task of streamliner constraint generation, evaluation and application.*

The following provides a breakdown of individual contributions:

Streamliner Generation Rules The additional of first-order and higher-order rules to the Conjure tool-chain so that streamliner constraints can be automatically generated based upon the type constructors of *decision* variables within an Essence specification.

Instance Generation Empirical identification that streamliner constraints perform differently across the instance space of a problem class. In order to ensure non-biased streamliner evaluation a procedure for the automatic generation of a diverse training instance set was created.

Lattice Search Implementation of a best-first search algorithm on the lattice of streamliner combinations. Empirical identification that for most problem classes there does not exist a dominating streamliner and the use of multi-objective optimization to build a portfolio of non-dominating streamliner constraints.

Streamliner Scheduling and Selection The creation of an architecture to schedule streamliners from a portfolio onto an unseen instance. The use of Algorithm Selection methods combined with instance features to predict streamliners on an instance specific basis.

Model Portfolios The exploration during lattice search of the impact that different refinement choices made by Conjure during modeling have on the performance of streamliner constraints. The construction of streamliner model portfolios.

Streamliners for Constrained Optimization and Boolean Satisfiability Adapting the original use of streamliners for constrained optimization problems and the application of streamliners in the Boolean Satisfiabiltiy (SAT) paradigm

1.8 Thesis Structure

The thesis is structured as follows:

Chapter 2 initially outlines related work in automated modeling. It then discusses in more detail alternative methods to reformulate and tighten constraint models using some of the methods previously discussed such as *symmetry* or *implied constraints*. An explanation of the theory underpinning *streamlining* and a history of prior work is presented.

Chapter 3 focuses on the method used to generate streamlined models automatically from an Essence specification. A brief overview of the Essence modeling language is given and the advantages of situating this system in a high level modeling language are discussed. An overview is given of the rules embedded within Conjure and an example is used to walkthrough how these work to automatically produce streamliner conjectures directly from the types present in Essence. The modeling pipeline and the process for how an abstract streamliner constraint at the Essence level is formulated down into a lower level solver dependent representation is explained. Lastly, the seven constraint satisfaction problems used within this thesis are formally defined and more generally the number of conjectures generated by our rules across the entire CSPlib [110] is shown.

Chapter 4 discusses the importance of problem instances in the automatic evaluation of model candidates and the construction of high quality streamliner portfolios. The automatic process by which training instances are generated from the Essence specification of a problem class and certain requirements these instances must possess is shown. An analysis of the "footprint" that different streamliners have in the feature space is presented and how performance similarity between instances can be used to construct a compressed representative training distribution.

Chapter 5 introduces the concept of combining streamliner conjectures and discusses the benefits in search reduction this can provide as well as the adverse effect on the complexity of streamliner search. The structuring of the possible combinations into a lattice structure and the use of pruning methods to remove ineffective combinations are defined. Various approaches for searching through the lattice structure and producing a portfolio of streamliner combinations are discussed and the search algorithm utilized in this thesis is defined.

Chapter 6 begins by showing that the performance characteristics of a streamliner conjecture can vary drastically between solving paradigms and demonstrates why it is necessary to perform paradigm specific streamliner search and generate independent streamliner portfolios. The constructed portfolios generated through the search method seen in the previous chapter are then summarized; in particular visualizing the distribution and size of the generated pareto fronts. For a subset of the problems, a more in-depth analysis is then performed to discuss some of the interesting properties of the constructed portfolios and analyze the performance of the constituent conjectures. Lastly, the problem of *"Streamliner Selection"* is introduced which deals with selecting from the portfolio an effective streamliner for an unseen instance. Various uninformed and informed methods for solving this problem are discussed.

The prior chapters have presented a completely automated approach to the generation and selection of streamliner constraints, hitherto a laborious manual task. Chapter 7 contains two sets of experiments to test the efficacy of this approach. The first is designed only to measure the frequency with which streamlining results in a reduction in search, and the magnitude of that reduction. The second experiment aims to provide a more practical setting in which the overall impact of streamlining across an entire instance distribution is analyzed. Results across two unseen instance distributions with varying solving difficulty are shown.

Chapter 8 presents the method and results on the application of streamlining for Constrained Optimization Problems.

Chapter 9 explores the effect of model selection in the context of streamlined specifications. We explore augmenting our best-first search method to generate a portfolio of Pareto Optimal streamliner-model combinations by evaluating for each streamliner a portfolio of models to search and explore the variability in performance and find the optimal model. Various forms of racing are utilised to constrain the computational cost of training. Empirical results are shown that demonstrate drastic improvements in solving time for some problem classes in comparison to a single-model approach.

Chapter Two

Background And Related Work

In this chapter we are initially going to look at the background work related to this thesis. Firstly, an overview of modelling, in particular automated constraint modelling, and a survey of different approaches that have been used to model CSPs. Related techniques used to reformulate and constrain a model such as *symmetry breaking*, *implied* and *dominance constraints* will then be discussed. Lastly an in-depth explanation behind the concept of *streamlining* and a review of the prior literature will be presented.

2.1 Constraint Programming

Constraint Programming (CP) is a powerful tool for solving large scale combinatorial problems such as vehicle routing [117], planning and scheduling [15]. To better illustrate the fundamentals of CP let us use Sudoku, a very well known puzzle played throughout the world. In Sudoku the objective is to complete a partially filled 9x9 grid by filling in the missing entries with numbers from 1 to 9 such that for each row, column and individual 3x3 squares there are no duplicate numbers. An example is shown in Figure 2.1. The aim is to solve the problem through deduction and reasoning rather than guessing. For a particular empty square possible values can be removed via deduction based upon the existing values in the row, column and square and the rule preventing duplicate numbers. Most who do this however do not realize that this process of using propagation and inference to remove values and arrive at a solution is at the heart of constraint programming [174].

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

(a) An unsolved Sudoku instance[174]

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

(b) The unique solution to the Sudoku instance [174]

Figure 2.1

A Constraint Satisfaction Problem (CSP) is a general way in which we can represent and solve these sorts of decision-making problems, such as the Sudoku problem. A CSP is formally defined as a triple {X, D, C}, where:

X is a set of variables: $\{X_1, \ldots, X_N\}$

D is a set of domains : $\{D_1, \ldots, D_N\}$

C is a set of constraints: $\{C_1, \ldots, C_N\}$

Each variable in $X_i \in X$ is defined over a domain $D_i \in D$. Each constraint $C_i \in C$ is a relation on a subset of the variables in X. This relation restricts the range of values the variables can be assigned from their respective domains. The goal is to find an assignment to all of the variables (from their respective domains) such that no constraints are violated.

Figure 2.2 provides an overview of the two main stages involved within CP, Constraint Modelling and Constraint Solving. Modelling involves mapping the features of a problem onto the features of a constraint satisfaction problem (CSP). Modelling the Sudoku problem produces a compact constraint model containing 81 variables, one for each of the squares in the grid. Each variable (ignoring squares already populated) would have a finite domain from $\{1 \ldots 9\}$ and *allDifferent* constraints would be posted on each row, column and square to prevent duplicates. Solving the CSP then typically involves the interleaving of two processes, *systematic search* and *constraint propagation*. Systematic search attempts to extend the set of partial assignments with the ability to backtrack if the chosen assignment cannot be extended to a solution. Constraint propagation attempts to use deduction based upon the current domains of the variables and the constraints to remove domain values that cannot exist in a solution.

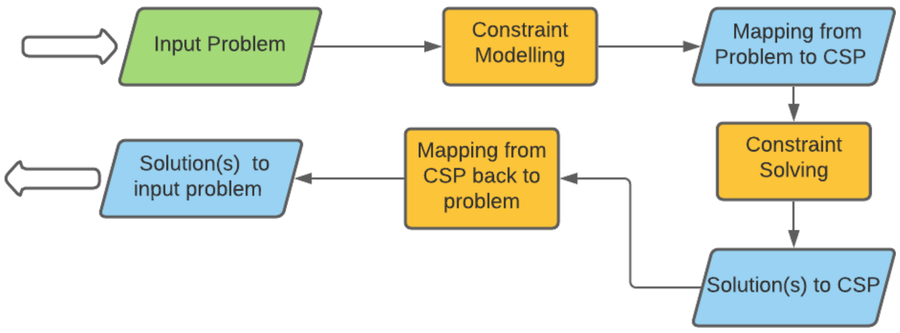

Input Problem
Constraint Modelling
Mapping from Problem to CSP
Constraint Solving
Solution(s) to CSP
Mapping from CSP back to problem
Solution(s) to input problem

Figure 2.2: Constraint Programming: Constraint Modelling & Solving

2.2 Modelling Languages

Over the years there has been a considerable amount of research into problem modelling which has resulted in a progression of different modelling languages. In modern languages the concept of solver independence, which allows for the same model to target different solvers and paradigms, is considered to be very important [65, 166].

2.2.1 OPL

The programming language OPL [191, 93] was introduced in an aim to unify the orthogonal strengths of the constraint and integer programming languages that came before it, such as CHIP [175], AMPL [59], GAMS [113]. It combines the rich language of constraint programming, that supports global, linear and non-linear constraints with the mathematical modelling languages that through their support for algebraic and set notations can concisely describe mathematical problems. Even though OPL only supports decision variables with integer and enumerated types and has no support for abstract domains it is one of the first high level modelling languages that abstracted away from the internals of the constraint solver.

2.2.2 Minizinc & Zinc

Minizinc [150] is a medium level solver independent modelling language and is probably the most popular and widely supported for constraint programming. It supports boolean and integer domains, and arrays for collections of these variables but does not have support for high level abstract types such as *function*, *partition*, *etc.* Minizinc is used to define parametrized problem class models which are then instantiated into an instance model using the instance data however no automated reformulations of the specified model take place. The instance model is compiled down into FlatZinc, a solver dependent language, for use in a solver.

Zinc [142, 166, 14] is a higher level constraint modelling language in comparison to MiniZinc. It supports more abstract domains than Minizinc, providing decision variables with set domains as well as user defined record-like domains. The main novelty of Zinc was that it was designed to support a modelling methodology where the same conceptual model could be be mapped to different solving techniques and solvers. As Zinc is a very high-level, expressive modeling language there is a considerable gap between a conceptual Zinc model and a model designed to target a particular solver or search technique. Zinc can be translated into Solver Independent Flattened Zinc Model (SI-FZM) which is at a much closer level to existing constraint solvers [166]

2.2.3 Essence

ESSENCE is a language for writing problem specifications which operates at a level of abstraction above which modelling decisions are made [65]. ESSENCE does this by providing the ability to have decisions variables whose domain values are combinatorial objects, such as *set*, *function*, *partition*, *etc* which can be nested to an arbitrary depth. A full list of domain elements is given in Figure 2.3. This allows the problem specification to be stated more directly and naturally and avoids the need for the user to make any modelling decisions.

To better understand ESSENCE let us walk through the example specification seen in Chapter 1, Figure 1.4. Line 9 declares the *find* statement which defines the decisions variables that the problem is searching for and their associated types. A specification can have one or more *find* statements. To constrain the specification, *such that* statements are used to post constraints onto the decision variables (Lines 10-16). *given* statements allow parametrization of a problem class and the values of which define a given instance (Lines 1-5). *letting* statements are optional and allow aliases to be defined.

Type	Domains
Concrete	Int, Bool
Abstract	tuple, record, variant, matrix, set, mset, function, sequence, relation, partition

Figure 2.3: In ESSENCE there are two types of domains, *concrete* and *abstract*. A *concrete* domain is one which is natively supported by the target solver. An *abstract* domain is one which is not natively supported and will be represented as a collection of more primitive variables.

2.3 Automated Modeling Tools

As discussed initially in Chapter 1 modelling is a complex task and if done manually requires great expertise to be done correctly and efficiently. *Automated Constraint Modelling* aims to solve this by automating the process of defining a concrete model and there have been a number of successful and different approaches to tackling this problem which we will now discuss.

2.3.1 O'CASEY

O'Casey was a new approach that leveraged Case-Based Reasoning (CBR) to store, retrieve and reuse prior experience to help write efficient constraint programs automatically [136]. In CBR a problem is not solved by reasoning from scratch instead stored solutions that are similar to the problem at hand are retrieved and adapted to the current problem [122]. Here a *case* was a problem description combined with instances of that problem. The experience that the system generally provided was the selection of propagators and search heuristics. It did not reformulate constraint expressions or variable representations.

2.3.2 CONACQ

Conacq [25] is a SAT-based version space algorithm that builds a constraint network consistent with an example set of solutions and non solutions of the given problem. In the first version of Conacq [19, 22, 21] they utilized passive learning where the user provides examples of solutions and non-solutions. The system is also provided with a set of variables X, domains D as well as a learning bias β that contains constraints from the target solver. Machine learning is then used to automatically generate constraints consistent with the examples. One weakness of passive learning is that in order for the set of target constraints to be learnt a diverse set of examples needs to be provided. In the second version of Conacq [23] active learning was used where the system proposed examples that the user had to classify as a solution or non solution. The use of active learning has several advantages. Firstly it can decrease the number of examples required to converge upon the set of target constraints. Secondly, it removes the need for a human user. QUACQ [20], also an active learner, was then introduced. From negative examples, QUACQ by asking the user to classify partial queries, was able to learn a constraint from the target network in a number of queries logarithmic to the number of variables.

2.3.3 Constraint Seeker and Model Seeker

Constraint Seeker [17] was a tool for finding and ranking global constraints based upon examples. It didn't create models but given a set of positive and negative examples, the tool aimed to find candidate global constraints that satisfied these ground truths and ranked them based upon perceived importance. Model Seeker [18] built upon this work to build complete finite domain constraint models from positive example solutions.

2.3.4 Conjure & SavileRow

Conjure [1, 4] is an automated refinement system which operates on an Essence specification to produce a set of constraint models in the solver independent modelling language Essence Prime [155, 154]. Conjure operates at the problem class level and is only invoked once to create the Essence Prime model. As Essence Prime is a solver independent modeling language with only concrete domains and no support for abstract types one of the main jobs of Conjure is to choose the representations for variable domains. There is not just one way this decomposition can occur and variables can be represented in different ways at a lower level. If we use the Social Golfers Problem (shown later in Chapter 3, Figure 3.1b) [92] as an example the problem definition requires partitioning a set of golfers across a schedule such that no golfer plays in the same group as any other golfer on more than one occasion. Subject to additional constraints this can be thought of as searching for a *set* of *regular partitions*. In order to solve this problem this would require representing the *set* of *regular partitions* in terms of the lower level primitive types. There are at least 72 different ways that this can be done [65]. Conjure can explore these choices to create a portfolio of all models but defaults to producing just a single model.

Savile Row [153, 154] is a modelling assistant for constraint programming that translates an Essence Prime model into the input language of a target solver. One of the benefits of Essence Prime is that it is a solver independent modelling language and Savile Row provides the ability to target multiple different solving paradigms and solvers. At this moment Savile Row can target CP: with output to Minion [81], Minizinc [150], Gecode [189], Chuffed [39], as well as output to SAT [156], MaxSAT and SMT [51]. During the translation process Savile Row applies various reformulations, such as common subexpression elimination and domain reduction, to improve the performance of the model. These reformulations, especially when combined together can result in significant gains in performance [154].

2.4 Reformulation of the model

Even with all of these advancements there are still a number of problems from domains such as Combinatorial Design [46] where even small instances of the problem class can be very difficult to solve [85]. In these cases if the model has been efficiently encoded with no obvious deficiencies a natural next step to take is to try and further constrain the model in order to strengthen the inferences the solver can make, therefore detecting dead ends in the search earlier and reducing overall search effort.

2.4.0.1 Symmetry breaking

Even though symmetry occurs in many constraint satisfaction problems [195, 144] it is difficult to find one concrete definition and in the past it has been defined in fundamentally different ways [82, 44]. Cohen et al [44] summarized the existing definitions into two distinct viewpoints of symmetry, *solution* and *constraint symmetry. solution symmetry* defines symmetry as a property of the set of solutions, any mapping that preserves the solutions is a symmetry. *constraint symmetry* on the other hand preserves the set of constraints, and therefore as a consequence also preserves the solutions. For all definitions, a symmetry maps solutions to solutions and non-solutions to non-solutions.

To get a better understanding let us use the $4x4$ chessboard shown in Figure 1.1 to explore the different forms of symmetry that exist in the NQueens problem. For the NQueens problem there are 8 inherent symmetries, as shown by Figure 2.4, obtained through rotations and reflections of the chessboard. There are four rotations of: 0 (known as the *identity symmetry*), 90, 180 and 270. There are then reflections on the *x,y* axes and on the main diagonals. The concept of *symmetry* is important as the complexity of a problem can often be reduced by detecting and exploiting these intrinsic symmetries [167].

The most common way of doing this is through symmetry breaking in which the goal is to prevent search from ever visiting symmetric search states. If a particular set of assignments is proven to be satisfiable or unsatisfiable then this inherent symmetry can be used to infer that the 8 other states are equivalent without having to visit them specifically during search. This effectively reduces the size of the search space by a factor of 8 and shows how powerful the application of *symmetry* can be in a problem. It is useful to note that *symmetry* is important both when searching for all solutions and when searching for just one valid solution. The reason being that symmetry not only helps with finding equivalent solutions but also in removing unsatisfiable assignment states.

One form of symmetry breaking involves reformulation of the model to reduce or remove symmetry. Again referring back to the CarSequencing problem, Dincbas, Simonis and van Hentenryck [52] discussed in their work the removal of symmetries where identical cars could be swapped in the production line by switching to a viewpoint which used *classes* of cars. This viewpoint is the one that we utilize in this thesis. This reduced the complexity of the problem from N^N to M^N where N is the number of cars and M is the number of classes.

Another popular method to break symmetry is the alteration of the base model with the addition of *symmetry breaking constraints* [50, 165, 57]. The idea here is compose a constraint that is satisfied by exactly one member of each of the symmetric points in the search space. An example of this is if a model contained a matrix of integers (X) the order of the matrix could be permuted which would give rise to symmetric search states. To remove this a symmetry breaking constraint could be added to the model to impose an ordering upon the values, $X[0] \leq X[1] \ldots \leq X[100]$. This constraint could be further constrained if we know that the values of the matrix must be unique, $X[0] < X[1] \ldots < X[100]$. Up to now this has generally been done manually by the modeler who may discover symmetry within the model and then add constraints in an attempt to remove it. There is no known polynomial time algorithm for detection of symmetries however it can be done automatically through reduction of the CSP to an instance of the graph isomorphism problem [140].

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

(a) identity

13	9	5	1
14	10	6	2
15	11	7	3
16	12	8	4

(b) r90

16	15	14	13
12	11	10	9
8	7	6	5
4	3	2	1

(c) r180

4	8	12	16
3	7	11	15
2	6	10	14
1	5	9	13

(d) r270

4	3	2	1
8	7	6	5
12	11	10	9
16	15	14	13

(e) x

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

(f) y

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

(g) d1

16	12	8	4
15	11	7	3
14	10	6	2
13	9	55	1

(h) d2

Figure 2.4: The 8 inherent symmetries obtained through rotations and reflections of the chessboard

2.4.0.2 Implied Constraints

An implied constraint is a redundant constraint that captures an implicit property that exists for all solutions [167]. Implied constraints retain all solutions to the original problem but attempt to further constrain the search space through supplementary domain reductions. They have been shown in the past to provide significant reductions in search on a variety of domains such as the Golomb ruler [75, 181] and quasigroup existence problems [181]. A good example of an implied constraint can be taken from the work of Regin et al [167]. Suppose there is a constraint network with 100 variables each taking the domain of $\{0,1,2,3,4,5\}$. The constraints on the network specify that every value except for 0 must be taken at least 5 times. Also suppose that the variable ordering is min-domain (selecting the variable first with the smallest domain) and an ascending value ordering is used. With this setup the solver can assign 95 of the variables to 0 before it detects an inconsistency at which point it will start to try and backtrack. Fundamentally we know that *at most* 80 of the variables can be assigned 0 $(100-4\times5=80)$ under the initial constraints. Enforcing this as an implied constraint would allow the inconsistency to be detected once 81 variables have been assigned 0, saving $5^{14}=6,103,515,625$ instantiations.

Implied constraints can be very powerful but care has to be taken in making sure that the pruning offered by the constraint must offset its overhead during propagation. They are only useful if they allow for inconsistencies to be detected earlier and for partial assignments to be failed where search would have continued otherwise. It is hard to reason about their effectiveness without experimentation. For a long time choosing useful implied constraints remained an art [181]. Colton and Miguel initially worked on an approach to generating implied, symmetry breaking and specialization constraints directly from the specification of a problem class and their application to group theory, quasigroup construction and BIBD [47]. This initial implementation was semi-automatic and started with the logical specification of a problem class. From there small instances of the problem were generated and solved and the HR automated theory formation system was used to invent concepts and theorems which were then proven by the Otter automated theorem prover. Manual interpretation of the theorems led to the construction and addition of implied constraints often exhibiting large reductions in solving time. In Charnley et al [37] they removed the manual component to produce a fully automated system for generating implied constraints. In some cases however they did find that the system failed to create an improved solver model. The effectiveness of implied constraints are limited by the constraint that all solutions to the original problem must be retained.

2.4.0.3 Dominance Breaking Constraints

Dominance relations, which are exhibited in many constraint problems, are a generalization of *symmetry relations* and as such can offer a similar or larger reduction in search space if exploited [145, 41]. Assuming a standard tree search, we can define S as the set of all search states and $f(s)$ an objective function on the best possible extension of s. Dominance relations are easily to conceptualize in optimization problems, but they are still valid in satisfaction problems where a solution can be assigned a minimum value of 0 and a non-solution a value of ∞.

A dominance relation is a binary relation on search states that follows the following properties [162]:

- $s_i \preceq s_j \implies f(s_i) <= f(s_j)$
- $\preceq$ is a partial ordering
- If $s_i \preceq s_j$ and $s_i \neq s_j$ then there exists some extension of $s'_i \in s_i$ such that for every extension of $s'_j \in s_j$, $s'_i \preceq s'_j$

Let us consider a simple CSP where we have a set of variables X: $\{x_0, x_1, \ldots, x_{10}\}$ with domain $D \in \{1 \ldots 10\}$ and an *all_different* constraint posted on the 10 variables: *all_diff*$(\{x_0, x_1, \ldots, x_{10}\})$. The objective is to minimize the sum $\sum_1^{10} x_i * i$. Now if we consider two partial assignments to this problem, A_1: $\{x_0 = 2, x_1 = 1, \ldots\}$ and A_2: $\{x_0 = 1, x_1 = 2, \ldots\}$. The best extension of A_1 is no better than the best extension of A_2 and so we can stipulate that A_2 dominates A_1 ($A_2 \preceq A_1$). The search tree located under the partial assignment A_1 could then be pruned without affecting the optimality of the objective function. The concept of dominance relations however remains far less exploited than symmetry in CP [162]. Chu et al. developed a generic method for the identification and exploitation of dominance relations via dominance breaking constraints [41]. The enforcement of dominance breaking constraints, unlike *symmetry* and *implied* constraints, can result in losing the full set of solutions.

2.4.0.4 Streamlining Constraints

Symmetry breaking and *Implied constraints* are solution preserving techniques up to isomorphism. In Constraint Satisfaction Problems with all solutions being of equal value it is not always necessary to preserve all solutions, one solution will do. Streamlined constraint reasoning, introduced first by Gomes and Sellmann [84], differs drastically from these previous approaches in that the conjectured constraints are not typically proven to follow from a given constraint model and are often not proven to be sound or redundant. The generated constraints attempt to focus effort on a highly restricted but promising portion of the search space but the lack of proof means they often do drastically alter the set of solutions for a given problem class and instance. Streamliners trade the completeness offered by implied, symmetry-breaking and dominance-breaking constraints for potentially much greater search reduction.

Gomes and Selmann in their introductory work on *Streamlining* worked on problems from the field of Combinatorial Design [84]. Combinatorial design is a generic term for a field that deals with the existence and construction of families of finite sets whose arrangement satisfy certain properties and is one of the fields for which even very small instances of a problems can be incredibly difficult for modern solvers. The difficulty arises from a combination of the large assignment space and the presence of only a few solutions which creates a low solution density meaning that without any special variable or value ordering that leads directly to a solution the solver has to churn through a large search space. The idea Gomes and Selman proposed behind *streamliner constraints* is that by *"imposing additional structural properties in advance we are steering the search first towards a small and highly structured area of the search space."* [84]. Restricting the search space can also be thought of as a partitioning of the space into two halves: one much smaller streamlined half for which the added constraint must hold and the other complement half containing the assignment space for which the constraint does not hold as can be seen from Figure 2.6. Search is then conducted in the streamlined subspace *P1*. If a solution is not found then the complement half can then be streamlined again and the process repeated.

One of the problems that they investigated was the construction of *Diagonally Ordered Magic Squares* (DOMS). Informally Magic Squares are $n \times n$ grids which contain numbers in the range from $1 \dots n^2$ such that each cell in the grid contains a different number and each row, column and diagonal must sum to the same number known as the magic constant. This magic constant can be defined by $M = n(n^2 + 1)/2$. Figure 2.5a shows a solution for an order 4 DOMS. There are known polynomial time construction methods for normal Magic Squares however for DOMS which enforce that additional to all the other constraints the diagonals must be strictly ordered there are no known polynomial time construction methods. Adding in the diagonal constraint reduces the number of solutions and thus reduces solution density making it a much more difficult problem to solve. Gomes and Sellman noted that even for small order magic squares finding solutions was considerably difficult and on the base models only solutions up to order 9 were found.

Through analysis of the solutions of smaller order DOMS they noticed that certain structural regularities often existed. One of these regularities was that numbers within the magic square are quite evenly distributed, meaning that the large or small numbers are not clumped together, Figure 2.5b. Intuitively this makes sense as if all of the rows and columns and diagonals have to sum to the same number its not going to be feasible if all of the larger order numbers are grouped in the same row. To formalize this regularity they define L (Figure 2.5c) which associates with each entry in the order N Magic Square a number drawn from $\lfloor (m_{i,j} - 1)/n + 1 \rfloor$, essentially splitting the numbers into evenly sized intervals. If no two numbers from the same interval appear in the same row or column, this will enforce even distribution of the numbers throughout the square and also means that L represents a Latin Square structure. A Latin Square of order N is an N *by* N matrix where each cell has one of N symbols and each symbol must be all different across the row and column.

To exploit this realization they added *Latin Squareness* as an extra constraint to help focus search onto magic squares that are hiding a latin square structure. Empirical results showed they were able to substantially reduce the amount of time required to find a solution. For instance, on an order 8 DOMS they reduced the time from over 5000s using the base model to just under a second by adding in this latin square streamliner. There is no guarantee that all orders of magic square will contain this structure but this enabled them to extend the construction of DOMS up to order 18 which was clearly infeasible before. This identification of a structural property that exists in the solutions of certain instances and the formalization and use of that property to try and solve more difficult instances from the same problem class is the general idea behind *Streamlining.*

6	9	13	6
12	7	13	2
15	4	10	5
1	14	8	11

(a) A solution for a Diagonally Ordered Magic Square of order 4.

			*
		*	
*			
	*		

(b) The largest numbers are evenly distributed throughout the square

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

(c) The solution can be represented as a Latin Square

Figure 2.5

When constructing *streamliner constraints* there are a few factors that should be taken into account. Firstly it is important that the constraint can effectively propagate within the solver. All constraints add overhead and a poorly designed streamliner may be able to reduce the search space but any pruning achieved will be outweighed by the expense of propagation. Secondly, it may seem intuitive that streamliners should be constructed with the goal of capturing common solution properties so that the number of solutions in the streamlined subspace is maximized. However, this conflicts with the core aim of restricting the space to reduce search. In fact very restrictive constraints that only retain one or two solutions can be very effective as they reduce the search space so considerably [84]. There is a tradeoff however with the use of very restrictive streamliners. As they maintain only a few solutions they struggle to generalize across the instance space as the property they capture may only exist for a few instances.

Since the introduction *Streamlining* has been used in a number of different contexts. Kouril et al. refer to streamlining as “tunneling" [124]. In their work they were leveraging SAT solvers to try and improve upon the bounds of the VanDerWaerden problem, where still to this day only a few numbers are proven and most just have a known lower bound. Their motivation was based upon an analysis of the search behavior of an off the shelf SAT solver on a smaller VanDerWaerden formulae. They noticed that there was, as termed by them, a *"performance mountain"* in the early stages where the breadth of search would explode early in the search tree and the solver would struggle to make progress. Similar to Gomes and Sellman [84] they devised three different *tunnels* through analysis of patterns in the variable assignments for smaller Van der Waerden formulas. These additional clauses were termed as *tunnels* as they were designed to tunnel under the *"performance mountain"* at which point the breadth of search becomes moderately small and so search can continue quickly. Empirically this led to a dramatic improvement in run time of the solver, allowing much tighter bounds to be computed.

Gomes et al. used streamlining in the context of model counting, the classical problem of

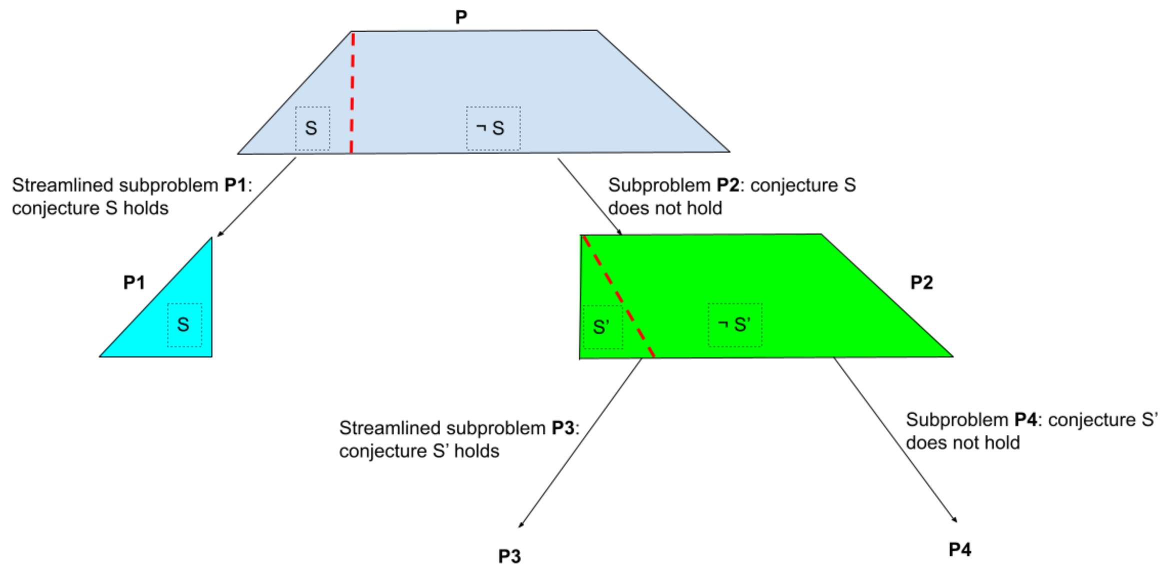

P
S
¬ S
Streamlined subproblem **P1**: conjecture S holds
Subproblem **P2**: conjecture S does not hold
P1
S
P2
S'
¬ S'
Streamlined subproblem **P3**: conjecture S' holds
Subproblem **P4**: conjecture S' does not hold
P3
P4

Figure 2.6: Streamlining aims to focus search onto a small and highly structured search space containing solutions. Here **P** represents the original problem to be solved. To streamline conjecture **S** is enforced to divide **P** into two complementary subproblems **P1** and **P2**. If the streamlined subproblem **P1** does not contain a solution, the complementary space **P2** can again be streamlined.

computing the number of solutions of a given propositional formula [87]. In this work they continuously added randomly generated *XOR* or parity streamliner constraints to constrain the problem space in a controlled manner. They proved that with high probability the number of additional *XOR* constraints added to the model before rendering the problem unsatisfiable can be used to compute a bound on the model count of an input formula. Using streamliners they were able to obtain the first non-trivial lower bounds on the number of solutions of several complex combinatorial problems.

Fujita et al. [74] used streamlining in a different approach in their soft constraint guided SAT solver on instances from the Ramsey graphs problem. In this work they prepared a set of additional streamliner constraints (S) which are added to the problem to restrict search in a preferable direction. In the original approach by Gomes et al. the streamliner splits the space into two halfs, the streamlined subspace (S) and its complement ($\overline{S}$). If S, is unsatisfiable, search would continue on its complement $\overline{S}$. Here instead if search became unsatisfiable they do not work on $\overline{S}$ but instead form a relaxed view of the set of constraints in S. They devised an iterative procedure to perform a series of relaxations and restarts until a single model is found. Using this they were able to raise the known best lower bound for the Ramsey number R(4,8) from 56 to 58.

Smith et al. built upon the introductory work of Gomes and Sellman and used streamlining [182] in the local search setting to construct larger Spatially Balanced Latin Squares (SBLS) up to order 35. The enormity of the search space for this problem and the low density of solutions made the odds of local search finding a solution considerably low. In prior work it was shown that a standard local search approach performed "remarkably poorly at finding SBLS". In this work they streamlined the space substantially by starting the local search solver at a randomly generated Latin Square and then performed swaps and permutations of the columns until a SBLS was found.

Le Bras et al. used streamlining to help construct graceful double wheel graphs [128]. Constraints forcing certain parts of the colouring to form arithmetic sequences allowed for the construction of colourings for much larger graphs. These constraints led to the discovery of a polynomial time construction for such colourings, proving 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 [130]. Here constraints enforcing periodicity in the solution, the occurrence of the improved Walters sequence, and a partially multiplicative property improved solver performance, allowing the discovery of new bounds.

In all of these prior works the concept of *streamlining* has proven to be incredibly valuable allowing computational results to be found that were not feasible before. Given the results that can be achieved it may be expected that the literature on streamliners would be comprehensive, however it has had a rather lackluster adoption and application; mainly being applied just to mathematical and combinatorial design problems. The main drawback is that it has always involved a laborious manual component, the analysis of solution patterns and derivation of streamliner constraints which has to be performed individually for each problem class and takes considerable time and expertise. In this work, we aim to increase the allure of streamlining by eradicating the need for this manual component and showcasing how it can be applied to a wide number of realistic domains with impressive results. We propose a fully automated system that supports the generation of streamliners directly from the abstract specification of a problem, the automatic search for effective candidate streamliners and combinations and the automatic scheduling and selection onto unseen instances.

2.5 Summary

In this chapter we have looked at the background work related to this thesis. Firstly, an overview of modelling, in particular automated constraint modelling, and a survey of different approaches that have been used to model CSPs was provided. Related techniques used to reformulate and constrain a model such as *symmetry breaking*, *implied* and *dominance constraints* were then discussed. Lastly an in-depth explanation behind the concept of *streamlining* and a review of the prior literature was presented.

Chapter Three

From Conjectures to Streamlined Specifications

This chapter focuses on the method used to generate streamlined models automatically from an Essence specification. The advantages of situating this system in Essence, a high level specification language, such as the ability to exploit the structure present within the specification, the ability to target multiple model formulations from the same base specification and automated symmetry breaking are discussed. An overview is given of the rules embedded within Conjure and an example is used to illustrate the process by which streamliner conjectures are generated directly from the types present in Essence. The modeling pipeline and the process for how an abstract streamliner constraint at the Essence level is formulated down into a lower level solver dependent representation is explained. Lastly, the seven constraint satisfaction problems used within this thesis are formally defined and more generally the number of conjectures generated by our rules across the entire CSPlib [110] is shown.

3.1 Utilizing Essence

In the following sections we will give an overview of why we have chosen to build our streamliner generation system around Essence specifications and an overview of some of the advantages provided by using an abstract constraint specification and a refinement based approach to streamlined model generation.

3.1.1 Utilizing Structure

The highly structured description of a problem an Essence specification provides is better suited to streamliner generation than a lower level representation, such as Essence Prime. This is because, in Essence Prime, nested types like multiset of sets, functions, relations must be represented as a constrained collection of more primitive variables, obscuring the structure that is useful to drive streamliner generation. Figure 3.1 shows a comparison between the representation of the Social Golfers Problem [92] in the constraint modeling language Essence Prime (Figure 3.1a) and Essence (Figure 3.1b). As Essence Prime does not support higher level types, this problem description has to be decomposed into more primitive types; in this case matrices of *integer* values. Without prior knowledge it would be difficult to extract the structure present and to understand the objective of partitioning a set of golfers across a schedule such that no golfer plays in the same group as any other golfer on more than one occasion.

It is possible to formulate streamliners without any knowledge of the actual types however they would be limited in nature and lack sophistication. Also as the complexity of the types grow, with the addition of nested domains such as multisets or sets of functions, the complexity of type identification in Essence Prime and the generation of streamliners would grow. Type identification in Essence is always possible regardless of the complexity of the specification as arbitrarily nested types are supported natively. In fact having more structure present within the Essence specification is beneficial to streamliner generation as it allows more complex far reaching streamliners to be generated that can have large effects on the search within the solver.

3.1.2 Multiple Formulations

With Essence the act of specifying problems does not require CP modeling decisions to be made. This means that streamlined specifications generated in Essence are abstract and are not bound to any particular formulation of a constraint model. With reference to Figure 3.1b there are multiple ways in which the *set of partitions* at the Essence level could be formulated, for instance using an *occurrence* or *explicit* representation [109]. Conjure by default uses a *default* heuristic to generate just one output constraint model, however it can generate a portfolio of models representing the different modeling choices made. As streamliners are generated at the Essence level, Conjure will automatically encode them for the particular formulation chosen. Contrast this to Essence Prime in which for the specification shown in Figure 3.1 modelling decisions have already been made. Any streamliners generated from this specification will be compatible only with this particular formulation. If a different viewpoint is taken, the streamliners would have to be regenerated to target that new viewpoint.

As discussed earlier an important property of a streamliner is its ability to effectively propagate within the solver. Different formulations can affect the ability of a constraint to propagate and as such the chosen formulation for a streamliner is important. With Essence and Conjure it is easy to produce a portfolio of different formulations to investigate which provides the best performance for the streamlined model. In Essence Prime however in order to accomplish this you would need to maintain multiple different models. Additionally, Conjure can produce multiple alternative (redundant) representations of the same variable that may appear simultaneously in the same model and additionally automatically generate *channelling* constraints between these representations to maintain consistency [143]. This would have to be done by hand in Essence Prime. When streamliners are combined, each individual constraint can target a different representation of the decision variable to maximize its propagation. The ability to target multiple different representations is discussed further in Chapter 9.

3.1.3 Automated Symmetry Breaking

Most *symmetry* enters constraint models through the process of constraint modelling [68]. An advantage of using Essence is that Conjure automates the process of generating *symmetry breaking constraints* and can break symmetry in a model as it is introduced by the modeling process [3]. With respect to streamliner generation this gives the use of Essence a distinct advantage as when streamliner constraints are added to the original model there is the possibility that they could introduce additional symmetries. With Essence Prime these symmetries would have to be broken on a per instance basis.

```
given w: int(1..)
given g: int(1..)
given s: int(1..)
letting let1 be g * s
find golfers:
        matrix indexed by [int(1..w), int(1..g), int(1..s)] of int(1..let1)
branching on [golfers]
such that
    and([sum([toInt(or([or([golfers[q23, q28, q31] = g1
                                | q31 : int(1..s)])
                        /\
                        or([golfers[q23, q28, q33] = g2
                                | q33 : int(1..s)])
                            | q28 : int(1..g)])
                    /\ (or([g1 = q26 | q26 : int(1..let1)]) /\ or([g2 = q26
                        | q26 : int(1..let1)])))
                  | q23 : int(1..w)])
         <= 1
             | g1 : int(1..g * s), g2 : int(1..g * s), g1 != g2]),
    and([flatten([[golfers[q1, q11, q12] | q12 : int(1..s)]
                      | q11 : int(1..g)])
         <lex
         flatten([[golfers[q1 + 1, q13, q14] | q14 : int(1..s)]
                      | q13 : int(1..g)])
             | q1 : int(1..w - 1)]),
    and([allDiff([golfers[q2, q15, q16]
                      | q15 : int(1..g), q16 : int(1..s)])
             | q2 : int(1..w)]),
    and([and([s >= 1 | q17 : int(1..g)]) | q2 : int(1..w)]),
    and([and([[golfers[q2, q6, q18] | q18 : int(1..s)] <lex
              [golfers[q2, q6 + 1, q19] | q19 : int(1..s)]
                  | q6 : int(1..g - 1)])
             | q2 : int(1..w)]),
    and([and([and([golfers[q2, q7, q8] <
                   golfers[q2, q7, q8 + 1]
                       | q8 : int(1..s - 1)])
                  | q7 : int(1..g)])
             | q2 : int(1..w)]),
    and([let1 = sum([s | q21 : int(1..g)]) | q2 : int(1..w)])
```

(a) ESSENCE PRIME specification

```
given w, g, s : int(1..)
letting Golfers be new type of size g * s
find sched : set (size w) of
                $ regular is implied by numParts g and partSize s
                partition (regular, numParts g, partSize s) from Golfers
such that
forAll g1, g2 : Golfers, g1 != g2 .
    (sum week in sched . toInt(together({g1, g2}, week))) <= 1
```

(b) ESSENCE specification

Figure 3.1: Comparison between the ESSENCE specification and ESSENCE PRIME model of the Social Golfers Problem [92]

3.2 Streamliner Generation

Our process is driven by the decision variables in an Essence specification, such as the function in Figure 3.1b. Candidate conjectures are generated by applying a system of conjecture forming rules. A conjecture forming 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 conjecture forming 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. For each variable, the system forms conjectures of 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.

3.2.1 Conjecture-forming Rules

In order to generate a large variety of useful conjectures we employ a small set of rules, categorised into two classes, *First-Order* and *Higher-Order* rules. *First-Order* rules add constraints to reduce the domain of a decision variable directly. Let us look at the example below where the decision variable X has just a simple *integer* domain. Here a *First-Order* rule that may apply could be the *odd* rule that operates only on an *integer* domain and restricts the acceptable assignments to only *odd* values. In this primitive example the application of this streamliner would reduce the size of the search space by a factor of 2.

```
find X: int(1..10)
such that
$ odd(X)
x % 2 = 1
```

High-Order rules on the other hand allow for streamliners to be generated on the nested structure that is often present within an Essence specification. We have modified the example slightly so that now X is a collection (a *set*) containing *integer* values. Here if we want to enforce a similar constraint enforcing that all of the values of X must also be odd we cannot just use the *odd* rule directly as we did before because this doesn't apply to a *set* domain. *High-Order* rules take another rule as an argument and lift its operation onto a decision variable with a nested domain. Here we can use one of the *Higher-Order* rules such as *all* combined with the *First-Order* rule *odd* to enforce the constraint on all elements of the set X. Imposing extra structure in this manner can reduce search very considerably.

```
find X: set of int(1..10)
such that
$ all(odd(X))
and([q1 % 2 = 1 | q1 <- x])
```

Streamliner generator rules given in Figure 3.2 and Figure 3.3 cover all domain constructors in Essence and they can be applied recursively to nested domains.

3.2.2 Streamlining by Example

To make clear the process of streamlining directly from an Essence specification we are going to walk through an example of how a real streamliner constraint is generated for the Car Sequencing problem outlined in Figure 1.4. The streamliner rules outlined thus far operate on the domains of the decision variables within the Essence specification. For Car Sequencing this is a *total function* mapping between an available slot on the assembly line and the class of car to be produced, both *integer* domains.

```
letting Slots  be domain int(1..n_cars),
letting Class  be domain int(1..n_classes)
find car: function (total) Slots --> Class
```

When this specification is streamlined the domain of the decision variable is extracted and pattern matched against the conjecture rules within Conjure. In this case as a *function* is a higher-level type both *Higher-Order* and *First-Order* rules can be applied. Only a subset of the rules will fire however as they do not all accept or operate on a *function* domain.

Some *First-Order* rules can apply directly to a *function* domain to generate streamliner constraints. These are *monotonicIncreasing*, *monotonicDecreasing*, *smallestFirst* and *largestFirst*. On this specification four streamliner constraints are generated by the application of these *First-Order* rules.

Only two *Higher-Order* rules, *range* and *defined* are directly applicable to a *function* type. Let us restrict ourselves for this example to looking at just the *range* rule. When

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

Name lowerHalf (Similarly upperHalf)
Input `X: int(l..u)`
Output `X < l + (u - l) / 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(l..u) --> int`
Output `forAll i in defined(X) . X(min(defined(X)) <= X(i)`
Tag FunctionSmallLarge

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

Figure 3.2: The *First-Order* streamlining rules. For each rule we present the rule name, rule's input, 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.

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))`

Figure 3.3: 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.

applied within CONJURE this will extract the *set* domain from the *function* and also take another rule as an argument. By default all rules will be matched and only those which operate on a *set* domain will be applicable. As the domain that range produces is a *{set: Int}* none of the first order rules can apply but a few of the higher-order rules do such as *approximatelyHalf*, *all*, *half* and *atMostOne*. Being *Higher-Order* these again will also take another rule as an argument. This recursive process continues until a first-order rule is selected. For instance the *odd* rule may be selected to create the complete rule *range(approximatelyHalf(odd(car)))*. Figure 3.4 shows a sample of *First-Order* and *Higher-Order* rules that fire on this specification alongside the generated streamliner constraints. CONJURE combines these constraints with the original specification to produce a streamlined ESSENCE specification.

These generated constraints can be quite effective in reducing search effort by a large degree. Figure 3.5 shows that the aforementioned constraint *approximatelyHalf(range(car, odd))* is applicable on almost all of the 100 randomly selected training instances. In most cases it is also able to achieve a drastic reduction in solving time for instances widely dispersed across the x-axis showing that the streamliner's performance is not just applicable to one form of instance.

3.2.3 Domain Attributes & Softened Rules

Domain attributes can be added to ESSENCE domains to restrict the set of acceptable values. For example, a function variable domain may be restricted to be *total* or *injective*, or a partition variable domain may be restricted to regular partitions. Hence, the simplest source of streamliners is the systematic annotation of the decision variables in an input specification. 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.

If we take the BIBD ESSENCE specification, Section 3.4, as an example it contains a decision variable of the form *find bibd : relation of (Obj * Block)*. A generated streamliner from the rules encompassed in Figure 3.2 and Figure 3.3 might be *symmetric(bibd)* which enforces that the relation must be *symmetric*. Enforcing this does restrict the size of the search space and help propagation as when a pair (q1, q2) is added to the relation it can be immediately propagated that (q2, q1) must also be in the relation to satisfy the *symmetric* property. The problem however is that for a lot of BIBD instances this additional constraint may be too strict and remove all solutions. It is thus important to

```
First-Order Rules
----------
        $ Enforces that the car function must be monotonically increasing
           such that if a value is less than another in the defined values
           it must also be less than in the function
        monotonicallyIncreasing(car)
        --------
                forAll i,j in defined(car).
                        i < j -> image(car, i) <= image(car, j)

        $ Enforces that the smallest defined value of the function must
           have the lowest value in the function
        smallestFirst(car)
        --------
                forAll i in defined(car).
                        car(min(defined(car)) <= car(i)

Higher-Order Rules
----------
        $ Enforces that the range of the function must be even
        range(even(car))
        --------
                forAll i in range(car).
                        i % 2 = 0
        $ Enforces that approximately half (with softness parameter) of the
            range of the function must be odd
        range(approximatelyHalf(odd(car)))
        -------
                |range(car)| / 2 + k >= sum([toInt(q11 % 2 = 1) | q11 <-
                    range(car)]) /\\ |range(car)| / 2 - k <=
sum([toInt(q11 % 2 = 1) | q11 <- range(car)])
```

Figure 3.4: The application of *First* and *Higher-Order* rules on the *function* decision variable domain in Car Sequencing

consider that there may be structural regularities that exist within problems that only partially apply to a variable and not the variable as a whole. This premise has been seen before in the literature where in their work on Double-Wheel Graphs, Bras et al showed that one of the effective streamliners they generated enforced most of the inner wheel to be odd [129].

We want to be able to generate streamliners that are flexible and can capture these smaller regularities. Some of the rules that comprise our system are designed such that they can take a softness argument which can control how strict the generated streamliner constraint is going to be. Within CONJURE there are hard coded softness parameters that are drawn from the set $\{1, 2, 4, 8, 16, 32\}$. Softness works by relaxing the strictness of the streamliner on a given variable by the degree of the softness parameter. As a convention,

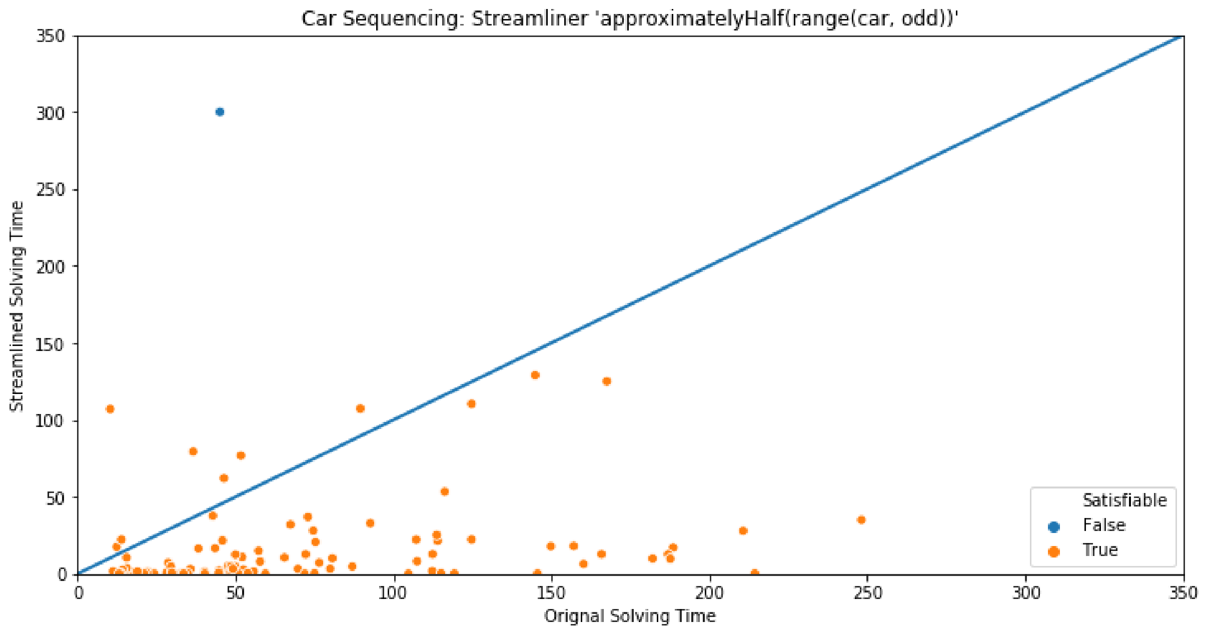


Figure 3.5: For the Car Sequencing problem the performance comparison between the unstreamlined model and streamliner approximatelyHalf(range(car, odd)) on 100 random training instances with unstreamlined solving times between [10,300]s

```
2 : and([|‘int(1..v)‘|*|‘int(1..v)‘| / 2 <=  sum([toInt(bibd(q1, q2) ->
    bibd(q2, q1)| q1 : int(1..v)])
4 : and([|‘int(1..v)‘|*|‘int(1..v)‘| / 4 <=  sum([toInt(bibd(q1, q2) ->
    bibd(q2, q1)) | q1 : int(1..v)])
8 : and([|‘int(1..v)‘|*|‘int(1..v)‘| / 8 <=  sum([toInt(bibd(q1, q2) ->
    bibd(q2, q1)) | q1,q2 : int(1..v))
```

Figure 3.6: A sample of the softened rules for the *symmetric* streamliner constraint on the BIBD problem

smaller values of the softness parameters produce comparatively strict streamliners (hence potentially causing greater reductions in the amount of search) and larger values produce more applicable streamliners. For instance the *symmetric* rule noted above is one of these softened rules and for each softness parameter (N) a softened constraint will be generated. Figure 3.6 shows three of the softened constraints that would be generated for the softness values of {2,4,8}.

As can be seen from Figure 3.7 the degree of softness can have a large impact on both the applicability and reduction performance of a streamliner across an instance set. Here *applicability* refers to the proportion of training instances for which the streamlined model admits a solution and *reduction* the mean search reduction in solving time achieved by the streamliner on those satisfiable instances. For the *symmetric* constraint described

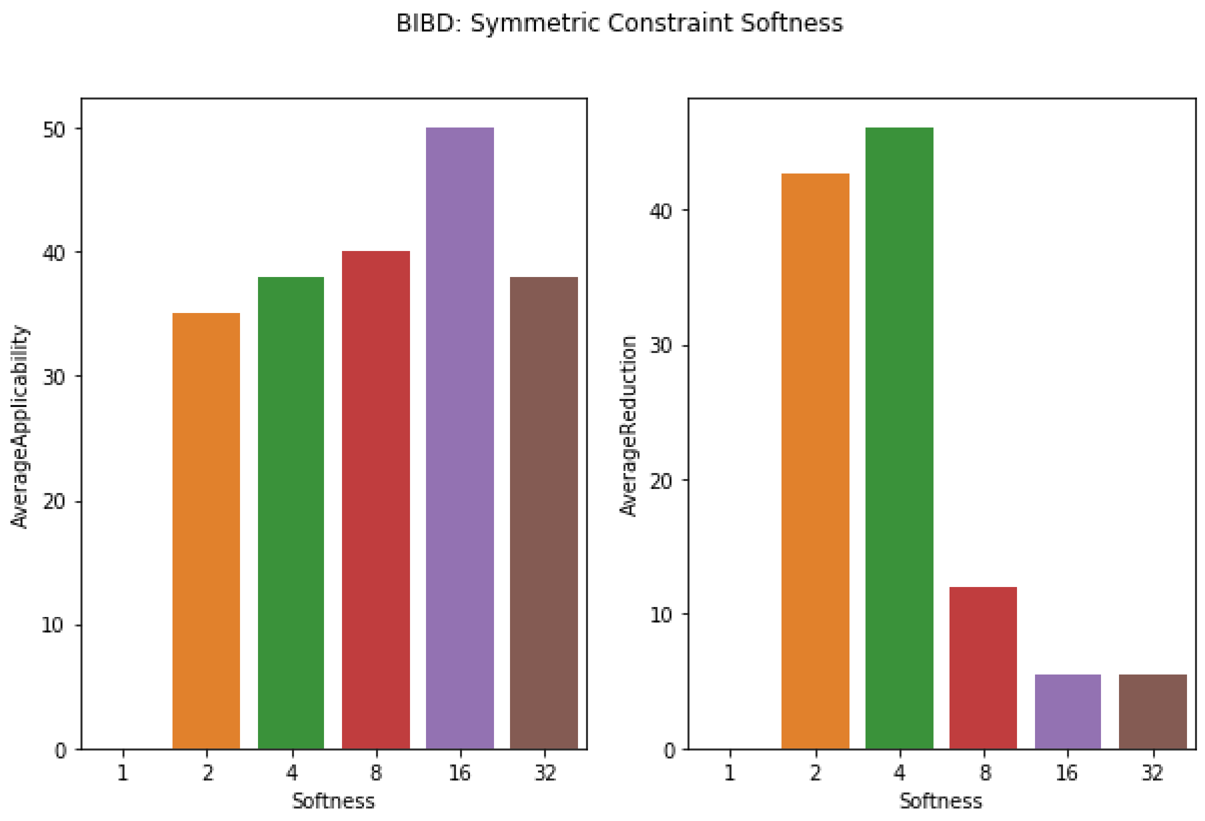


Figure 3.7: For the BIBD problem the Average Applicability and Reduction for various softness parameter on the symmetric streamliner constraint

above it can be seen that enforcing the relation *find bibd : relation of (Obj * Block)* to be symmetric results in all training instances being unsatisfiable (a softness value of 1 in this context enforces no softness). If *Obj* and *Block* are not of the same domain size then a fully symmetric relation is not achievable. If that is relaxed slightly such that at least half of the relation elements are symmetric then solutions are found on over 35% of the instances. As the softness parameter increases the percentage of instances for which a solution is admitted also increases up to almost 50% for value 16. As the constraint is relaxed it is able to capture smaller solution regularities that exist and so achieves greater coverage of the instance set. This trend does not extend to softness value 32 where the average applicability actually drops. Even though the search space of softness value 32 is a strict superset of all prior values, as streamliners get encoded as additional constraints they add overhead either in the form of extra clauses or extra constraints depending on the paradigm. For a streamliner to be effective the benefit from search restriction must outweigh the additional overhead and for value *32* because it is relaxed to such a high degree this does not occur for all instances.

For each softened constraint, and for each value of softness parameter an independent constraint is generated by Conjure. Because of this there is a practical limit in the range of the softness values that can be used. A larger range allows for the generation of more fine grained streamliners that can detect the presence of smaller regularities in the problem structure. As the range grows however so does the number of generated conjectures which increases the complexity of identifying which conjectures are effective. Also as the conjectures are continually softened their ability to restrict search and achieve reductions is reduced. This can be seen from Figure 3.7 where for any softness value greater than 4 the average reduction consistently drops as the streamliner is further relaxed.

3.2.4 Application on CSPlib

Table 3.1 shows the application of these rules across the set of satisfaction problems in the CSPlib [110], a problem library for constraints. There are a diverse range of types represented in this table, from partitions to functions to sequences to multi-sets to relations, and the rules within Conjure are able to generate streamliners for each problem listed.

Problem	# Streamliners	Essence features	Refs
Transshipment	68	2 partial functions	[103]
VanDerWaerden	65	partition of numbers	[94, 53]
CarSequencing	36	function	[52]
Template Design	40	1 1D function, 1 2D function	[163]
QuasiGroupExistence	35	2D function	[202]
AllIntervalSeries	72	2 bijective functions	[78]
VesselLoading	144	4 functions	[30]
Perfect Square Placement	72	2 functions	[176]
Social Golfers Problem	260	set of partition	[173]
Nonogram	64	2D matrix	[54]
Schurs Lemma	65	partition	[90]
Ramsey Numbers	32	function	[83]
Magic Squares	68	2D matrix, int	[193]
Magic Hexagon	68	2D matrix, int	[194]
Langford	52	sequence and injective function	[5, 179]
Sports Tournament Scheduling	12	relation of weeks, periods and sets of teams	[171]
BIBD	168	relation of unnamed types	[146]
Balanced Academic Curriculum Problem	36	binary relation, function variable	[97]
Fixed Length Error Correcting Codes	64	set of functions	[69]
NFractionsPuzzle	36	total, surjective function	[64]
Steiner	64	matrix of set	[45]
Covering Array	64	mset of function	[98]
Number Partitioning	32	2 set variables	[121]
DiamondFree	236	symmetric & irreflexive relation total function	[147]
Graceful Double Wheel Graphs	72	injective functions	[180]
Graceful Gears	72	injective functions	[180]
Graceful Helms	72	injective functions	[180]
Graceful Wheel Graphs	72	injective functions	[180]
Nqueens	36	total, bijective function	[105]
EFPA	144	set	[104]
Killer Sudoku	64	matrix	[152]
Tail Assignment	160	relations, sets of nested functions	[89]
A Layout Problem	32	function	[7]
Peg Solitaire	224	2 functions	[108]
Production Line Sequencing	48	2 total functions, 1 total injective function	[9]

Table 3.1: For the set of decision problems listed in CSPLib, the number of streamliners automatically generated from their ESSENCE specifications as well as the types of decisions variables used.

3.3 Modeling Pipeline

From an abstract specification CONJURE generates JSON output detailing the streamliner constraints that were automatically generated. A sample of this output is shown in Figure 3.8. This JSON output defines for each streamliner the following:

- **Numerical Id:** Each streamliner constraint is allocated a numerically ascending unique id which can be used to reference it when generating streamlined models.
- **onVariable:** The decision variable for which the streamliner constraint is imposed upon
- **groups:** A list denoting the groups that this streamliner is part of. This is used later to identify constraints that may be trivially incompatiable.
- **constraint:** The ESSENCE description of the streamliner constraint

The streamlined conjectures are generated at the ESSENCE specification level. ESSENCE specifications however cannot be solved directly by systematic constraint solvers, which lack support for abstract types such as functions and partitions, and nested types. Hence, an ESSENCE specification must be *refined* into a constraint model where these types,

```
{"1": {"onVariable": "car",
      "groups": ["FuncIncreaseDecrease"],
      "constraint": "and([and([q1 < q2 -> image(car, q1) <= image(car, q2)
                       | q2 <- defined(car)])          | q1 <- defined(
           car)])"},
 "2": {"onVariable": "car",
      "groups": ["FuncIncreaseDecrease"],
      "constraint": "and([and([q3 < q4 -> image(car, q3) >= image(car, q4)
                       | q4 <- defined(car)])          | q3 <- defined(
           car)])"},
 "3": {"onVariable": "car",
      "groups": ["FuncIncreaseDecrease"],
      "constraint": "and([image(car, min(defined(car))) <= image(car, q5)
                  | q5 <- defined(car)])"},
 "4": {"onVariable": "car",
      "groups": ["FuncIncreaseDecrease"],
      "constraint": "and([image(car, max(defined(car))) >= image(car, q6)
                  | q6 <- defined(car)])"},
```

Figure 3.8: Example JSON generated by CONJURE detailing the streamliners automatically generated from the specification

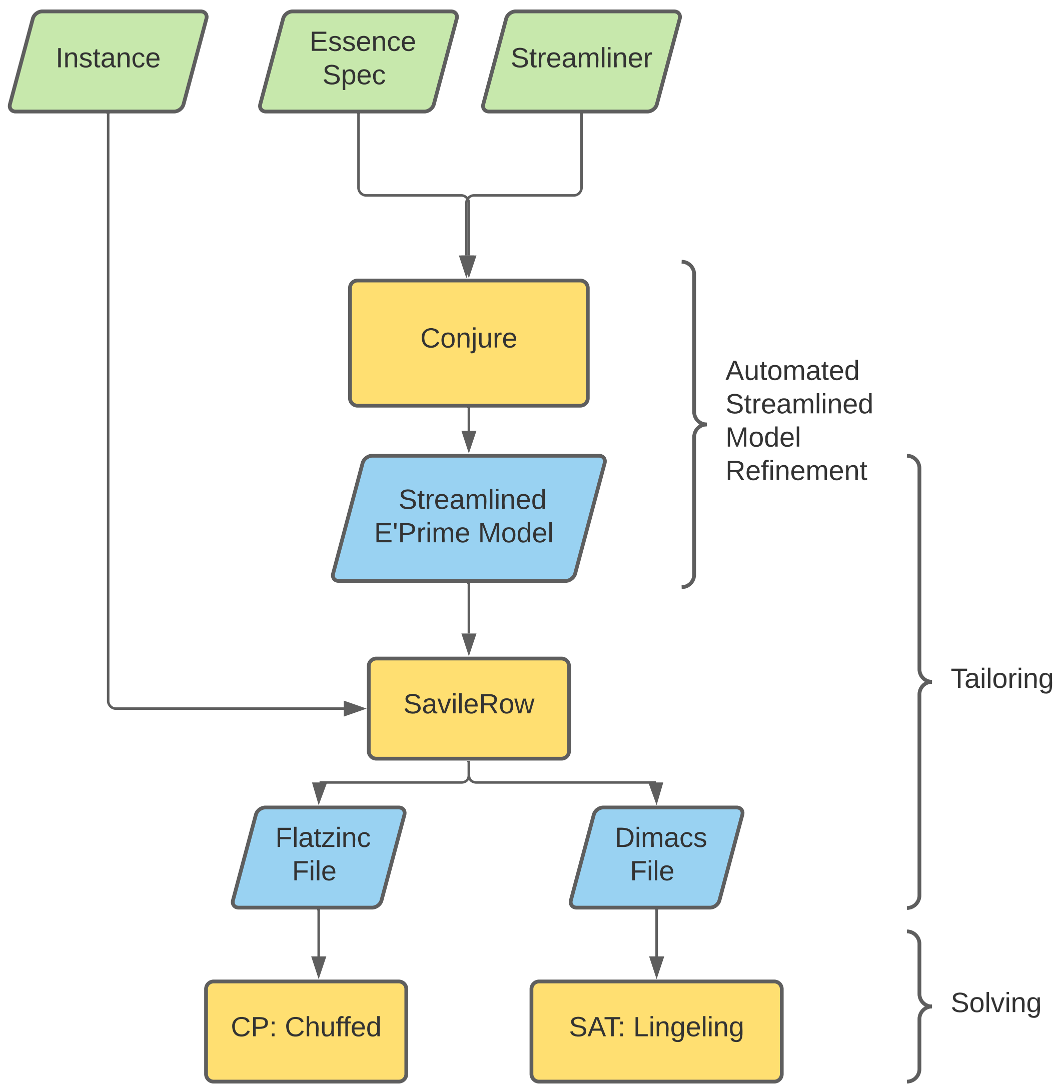


Figure 3.9: Process of evaluating a streamliner

and the constraints on them, are modeled in terms of the target constraint language. The process of evaluating a streamliner can then be split into three main components: Conjure Modelling, Savile Row Reformulation and Solver evaluation as can be seen from Figure 3.9.

Conjure modeling involves the refinement of the streamlined Essence model into the solver-independent constraint language Essence Prime [155, 154]. Streamliner constraints add additional complexity to the model and as such can have a negative impact on the modelling time taken by Conjure. However for a given streamliner combination Conjure only has to be invoked once to generate the Essence Prime model which can then be used for any instance of that problem class. The additional time added by the streamliner is then amortized over the entire instance space.

The constraint modelling tool Savile Row is then invoked to translate and reformulate the Essence Prime model together with the target instance into the input language of the solver. During the translation process Savile Row applies various reformulations, such as common subexpression elimination and domain reduction, to improve the performance of the model.

One of the main benefits of utilizing this pipeline is that it is trivial to target multiple different solver paradigms; for instance CP, SAT and SMT, without having to change the generated streamlined specifications. In this stage the generated solver-model produced by Savile Row is evaluated by the solver of choice.

3.4 Problems

We now introduce the problem classes studied in this thesis, in addition to the Car Sequencing Problem presented in Figure 1.4 and the Social Golfer's Problem listed in Figure 3.1a. These problems will be used both to illustrate the remainder of our method and for our empirical evaluation. We selected these problems, presented in Figure 3.10 to give good coverage of the abstract domains available in Essence, including matrices, sets, partitions, relations and functions.

The Covering Array problem [172] (Figure 3.10a) 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 [57].

The Fixed Length Error Correction Codes problem (FLECC) [63] (Figure 3.10b) 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 Balanced Incomplete Block Design problem (BIBD) [161] (Figure 3.10c) 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.

The Transshipment problem [8] (Figure 3.10d) 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 Balanced Academic Curriculum Problem (BACP) [96] (Figure 3.10e) (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.

```
given t : int(1..) $ strength (size of subset of rows)
given k : int(1..) $ rows
given g : int(2..) $ number of values
given delta_b : int(1..) $ columns
where k >= t
letting b be g**t  + delta_b

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

such that
  forAll rows : sequence (size t) of int(1..k) .
    (forAll i : int(2..t) . rows(i-1) < rows(i)) ->
    forAll values : sequence (size t) of int(1..g) .
      exists column : int(1..b) .
        forAll i : int(1..t) .
          CA[rows(i), column] = values(i)

$ row & col symmetry breaking
such that forAll i : int(2..k) . CA[i-1,..] <=lex CA[i,..]
such that forAll i : int(2..b) . CA[..,i-1] <=lex CA[..,i]
```

(a) Essence specification for the Covering Array Problem (decision version) [172]

```
      given nbCharacter : int(1..)
      letting Character be domain int(1..nbCharacter)
      maxDist       : int(1..) ,
      codeWordLength : int(1..),
      numOfCodeWords : int(1..)
letting Index be domain int(1..codeWordLength),
        String be domain function (total) Index --> Character
find c : set (size numOfCodeWords) of String
such that
  forAll s1, s2 in c, s1 != s2 .
    (sum i : Index . dist((s1(i),s2(i)))) >= minDist
```

(b) Essence specification for Fixed Length Error Correcting Codes problem [63]

```
given v, k, lambda, b, r : int(1..)
letting Obj   be domain int(1..v)
letting Block be domain int(1..b)
find bibd : relation of (Obj * Block)
such that
  forAll o  : Obj   . |toSet(bibd(o,_ ))| = r,
  forAll bl : Block . |toSet(bibd(_,bl))| = k,
  forAll o1, o2 : Obj , o1 != o2 .
    |toSet(bibd(o1,_)) intersect toSet(bibd(o2,_))| = lambda
```

(c) ESSENCE specification for Balanced Incomplete Block Design problem [161]

```
given n_warehouses, n_transshipment, n_customer, maxCost : int(1..)
letting W  be domain int(1..n_warehouses),
        T be domain int(1..n_transshipment),
        C be domain int(1..n_customer)
given   costWT : function (total) (W, T) --> int(1..),
        costTC : function (total) (T, C) --> int(1..),
        stock : function (total) W --> int(1..),
        demand : function (total) C --> int(1..),
find amountWT : function (W, T) --> int(1..max(range(stock)))
find amountTC : function (T, C) --> int(1..max(range(demand))
such that
  forAll w : W .
    sum([amount | ((w', t), amount) <- amountWT, w = w']) <= stock(w),
  forAll t : T .
    sum([amount | ((w, t'), amount) <- amountWT, t = t']) =
    sum([amount | ((t', c), amount) <- amountTC, t = t']),
  forAll c : C .
    sum([amount | ((t, c'), amount) <- amountTC, c = c']) = demand(c),
  sum([costWT(key) | (key, amount) <- amountWT]) +
  sum([costTC(key) | (key, amount) <- amountTC]) <= maxCost
```

(d) ESSENCE specification for Transshipment [8]

```
given n_courses, n_periods,load_per_period_lb, load_per_period_ub,
   courses_per_period_lb, courses_per_period_ub : int(1..)
letting Course be domain int(1..n_courses),
        Period be domain int(1..n_periods)
given prerequisite : relation of (Course * Course),
      course_load : function (total) Course --> int(1..)
find curr : function (total) Course --> Period
such that
  forAll c1,c2 : Course . prerequisite(c1,c2) -> curr(c1) < curr(c2),
  forAll p : Period . (sum c in preImage(curr,p) . course_load(c)) <=
      load_per_period_ub /\ (sum c in preImage(curr,p) . course_load(c)) >=
      load_per_period_lb,
  forAll p : Period . |preImage(curr,p)| <= courses_per_period_ub /\ |
      preImage(curr,p)| >= courses_per_period_lb
```

(e) ESSENCE specifications for Balanced Academic Curriculum Problem [96]

Figure 3.10: ESSENCE specifications for six problem classes used for evaluation, in addition to the Car Sequencing problem given in Figure 1.4.

3.5 Summary

This chapter has focused on providing an overview of the method used to generate streamlined models automatically from an Essence specification. An introduction to the Essence specification language was given and the advantage of situating the system in a high level specification language were discussed. An overview was given of the rules embedded within Conjure and a walkthrough of how an example streamliner conjecture is generated directly from the types present in Essence was given. The modeling pipeline and the process for how an abstract streamliner constraint at the Essence level is formulated down into a lower level solver dependent representation is explained. Lastly, the seven constraint satisfaction problems used within this thesis were formally defined and the application of the rules across the entire CSPlib is shown.

Chapter Four

Generating and Selecting Training Instances

The core idea underpinning this thesis is that an effective portfolio of streamliners can be built on a training set and then employed to solve unseen instances from the same problem class with substantially less effort than the *original* model. Because the streamliner conjectures are generated purely based upon the types present in Essence, the automatic evaluation of conjectures is essential to construct a high quality portfolio of streamlined models. For this purpose, training instances from the problem class under consideration are required. In this chapter we describe how training instances are automatically generated and selected directly from the Essence specification of a problem class. Lastly an analysis of the "footprint" that different streamliners have in the feature space is presented and how performance similarity between similar instances can be used to construct a compressed representative training distribution.

4.1 Automated Instance Generation

A potential for confusion in this chapter results from the use of the word *instance* and its specific meaning. An *instance* can normally be thought of as the result of combining a model and the instance data (particular values given for its parameters). Referring back to the Social Golfers Problem, two models for the problem are detailed in Figure 3.1 and a valid instance data example is provided below:

```
language Essence 1.3

letting g be 17
letting s be 10
letting w be 2
```

To evaluate the performance of a particular streamliner we want to compare the same instance under the original model and a streamlined model to gauge its reduction in search. However, as streamlining typically alters the model through the addition of conjectures, the instance produced given the same instance data is different between the original and streamlined models. For this reason: in the rest of this chapter we often refer to an *original* and *streamlined* representation of an instance and what is meant by that is using the same *instance data* but under the *original* and *streamlined* ESSENCE models.

We employ the automated instance generation system proposed in by Akgun et al [2]. Starting from the unstreamlined problem specification of a problem class in the ESSENCE language, several satisfiable instances with desirable properties are generated in a completely automated fashion. The system includes two main steps, which we illustrate in what follows with the Car Sequencing problem.

First, the original specification is automatically rewritten to produce an *instance generation specification*. Rewrite rules embedded in CONJURE transform each input parameter of the problem specification into decision variables and constraints on the valid assignments for these variables. Hence, solutions to this new specification correspond to instances of the original specification. The instance generation specification is itself parameterised so that it can be tuned to admit solutions from different parts of the instance space. An example of the generator specification created by the system for the Car Sequencing problem is presented in Figure 4.1.

Second, instances are generated by searching in the parameter space of the generator using the automated algorithm configuration tool IRACE [137]. IRACE uses an iterated racing approach to automatic algorithm configuration which consists of three steps. 1) The generation of a set of candidate configurations by sampling from the parameter space, which in our case is the space of the *given* parameters given to the generator specification. 2) Racing [141] to evaluate the performance and selecting the best configurations. Within each race the CP solver MINION [81] is firstly invoked to search the *generator specification* (Figure 4.1) for a valid instance to the Car Sequencing problem. If MINION is able to find a solution, that generated instance is then evaluated by the solver of choice to determine

```
language ESSENCE' 1.0

given n_cars_middle: int(1..200)
given n_cars_delta: int(0..99)
find n_cars: int(1..200)
given quantity_range_middle: int(1..200)
given quantity_range_delta: int(0..99)
find quantity_Function1DPartial_Flags: matrix indexed by [int(1..200)] of
   bool
find quantity_Function1DPartial_Values: matrix indexed by [int(1..200)] of
   int(1..200)

such that
    n_cars >= n_cars_middle - n_cars_delta,
    n_cars <= n_cars_middle + n_cars_delta,
    and([q1 >= 1 /\ q1 <= n_classes <-> quantity_Function1DPartial_Flags[q1
       ] | q1 : int(1..200)]),
    and([quantity_Function1DPartial_Flags[q16] ->
         quantity_Function1DPartial_Values[q16] >= quantity_range_middle -
             quantity_range_delta
              | q16 : int(1..200)]),
    and([quantity_Function1DPartial_Flags[q17] ->
         quantity_Function1DPartial_Values[q17] <= quantity_range_middle +
             quantity_range_delta
              | q17 : int(1..200)]),
    and([quantity_Function1DPartial_Flags[q5] = false ->
       quantity_Function1DPartial_Values[q5] = 1 | q5 : int(1..200)])
```

Figure 4.1: Parts of Car Sequencing generator's ESSENCE specification automatically created by the system in [2]. Given a generator's parameter file created by IRACE during the tuning, a random instance is created by solving this specification using the constraint solver MINION [81]. For brevity, we omit the specification for `nclasses`, `noptions`, `maxCars`, `blksize` and `usage`, as they are rewritten in exactly the same way as other parameters of the same types.

whether it is *satisfiable* and the difficultly required to solve it. 3) Updating the sampling distribution in order to bias the sampling towards the best configurations. These three steps are repeated until a termination criterion is met.

4.1.1 Instance Requirements

IRACE searches for instances under one particular representation, the *original* model. Not all instances found during search are selected to compose the training set however, only those that are *satisfiable*. This requirement is necessary as an *unsatisfiable* instance provides little utility in analyzing the performance of a streamliner as there is no way to reason about whether it retains solutions or aids in search reduction. Streamliners cannot be used to prove unsatisfiability, because they are not inferred from the model, and so an *unsatisfiable* instance is of little utility in our training set.

Secondly the instance must be *graded*, which we define as the solving difficulty falling within a required range; the instance cannot be too easy or too difficult for the chosen solver. In this work instances are selected that have a solving time in the range $[10, 300]s$.

An *upper bound* on instance difficult is enforced for two main reasons. Firstly, as IRACE initially randomly samples it generally needs to perform multiple iterations before it can focus onto good configurations. If no bound on instance difficultly is enforced it is possible that one of the initial candidate configurations lands in a difficult part of the instance space and the entire budget given to *instance generation* could be consumed without completing one iteration. By enforcing an upper bound it can be assured that IRACE will complete at least a certain number of iterations in the given time budget. Secondly, streamlining in the past has generally been used to solve difficult instances of a problem, i.e taking a few hours or days to solve, and so ideally our training set would be composed of instances of a similar character. The problem however is that during *streamliner evaluation* the worst case time occurs on *timeout* where the solving time is then lower bounded by the time taken by the *original* model. Based upon the number of conjectures that are generated from a specification and need to be evaluated, to keep the cost of training manageable an upper limit needs to be placed on the instance difficulty. We will show later in our results that streamliner portfolios learned on instances of this character do extrapolate onto more difficult instance distributions.

A *lower bound* is also defined to prevent instances from being trivially satisfiable as they are not an effective medium for evaluating a streamliner. On evaluation they could be used to tell if a streamliner was satisfiable, however as the solving time is negligible it would be difficult to gauge whether the streamliner actually helped in reducing search. Also as discussed in the previous chapter streamliners can add complexity to the solver dependent model. As trivial instances require little or no search to find a solution it is difficult to evaluate whether the complexity added will impact the performance of the solver as this would only become apparent during search.

4.1.2 Irace Search

The search of IRACE is guided via a scoring that rewards generator configurations covering unsatisfiable, satisfiable and non-trivial instances. Each race is allocated a score based upon its outcome:

- No instance produced by MINION: *score = 0*
- The generated instance is *unsatisfiable*: *score = 0*
- The generated instance times out: *score = 0*
- *instance is satisfiable:*
  - $instance\ trivial : score = solvingTime$
  - $instance\ graded : score = minSolvingTime$

Runs that generate no *instance* or an *instance* that is *unsatisfiable* or times out are penalized to the highest degree with the lowest score of zero as they represent the worst possible outcome during instance generation. If a *satisfiable* instance is found, the score is based upon the solving time taken by the solver. This means that trivial instances will be allocated a lower score than graded ones. As the objective of IRACE is to maximize the score obtained on each run this will progressively push search towards the instance space containing graded instances. During the *iterated racing* approach the sampling distributions are updated in order to bias the sampling towards the best configurations (those with the highest score) found during search. The space of all possible generator configurations is large and this bias in the sampling aims to focus the search on a part of the space that has already produced promising configurations. The hope is by focusing the search other successful generator configurations can be found and as a result generate more instances than would have been found under just a random exploration. This was shown to be the case in the work done by Akgun et al [2]. The downside of this however is that the IRACE search is going to focus on a particular region of the generator configuration space and instances generated from this region may be similar and possess little diversity.

Each run of IRACE tends to focus in on a particular region of the generator configuration space meaning that instances generated from the search tend to clump together in the feature space. To overcome this IRACE can be run multiple times with different starting *seed* values so that each search operates in a different region of the feature space as can

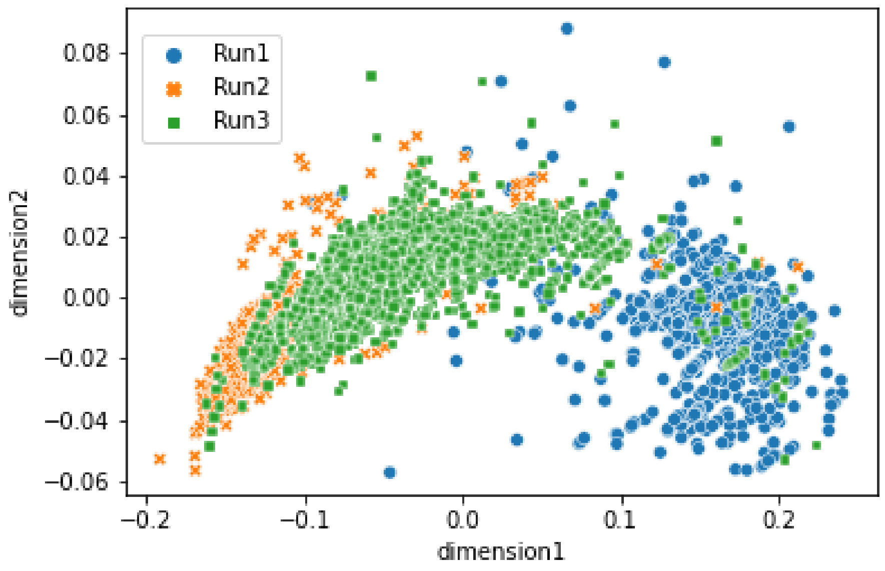


Figure 4.2: The instances generated for the Car Sequencing problem by the three independent runs of IRace. The instances are plotted on the two dimensional projection of the original multi-dimensional instance feature space via Principle Component Analysis [40]. About 99.4% of the variance in the feature space is captured in these two dimensions.

be seen by Figure 4.2. In this work for each problem class instances are collected from three different IRACE runs to ensure diversity. Table 4.1 provides a breakdown on the distribution of the different outcomes during the IRACE searches. *sat-graded* is the only type that results in an instance being collected for streamliner evaluation. Even though each IRACE search helps to increase coverage of the feature space as can be seen from the listed computation times *instance generation* is an expensive process and this is why we limit it to only three runs per problem class.

Irace Iteration	Iteration Result						Time Spent (cpu hours)
	no instance	sat: too-easy	sat : graded	unsat: too-easy	unsat: graded	solver time-out	
1	1542	1542	1429	3003	422	1405	609.61
2	1427	1927	1185	3893	362	1173	443.77
3	1576	841	3036	2854	348	1207	546.93
Total	4545	4310	5650	9750	1132	3785	1600.31

Table 4.1: Details of the three irace instance generation executions for the Car Sequencing problem. The *Iteration Result* category details the number of iterations that resulted in each category. The breakdown of the categories is defined as follows: *no instance* indicates the parameters chosen for the generator configuration did not generate an instance in the allotted time. The *sat* and *unsat* types indicate satisfiability and unsatisfiability respectively. The subtypes of *too-easy* and *graded* indicate how long the solver took to arrive at a solution. An instance is marked *too-easy* if it took $< 10s$ and is marked *graded* if the solver finished in the defined time range of [10, 300]s. *solver-timeout* indicates the solver went over $300s$ with no answer

4.2 Streamliner Footprint Analysis

We have seen in the previous chapters that different instances can react quite differently to the application of a streamliner constraint, both in terms of *satisfiability* and the degree of search reduction. Similar to the concept of an *algorithm footprint* [183], we can consider the footprint of a streamliner to be its search reduction performance across a set of instances. To visualize the streamliner performance FlatZinc features were extracted for all instances using the `fzn2feat` tool (part of `mzn2feat`[10]). There are 95 features grouped into 6 categories (variables, constraints, domains, global constraints, objective, and solving features) [10]. Figure 4.3 shows in a transformed 2-D version of the feature space, using Principle Component Analysis [40], the feature space for instances from the Fixed Length Error Correcting Codes problem. In the feature space the instances arrange themselves into two independent clusters. It is not possible however, because the dimensions produced by PCA are a combination of features, to tell what is distinguishing attribute between these two clusters.

These streamliner "footprints" vary across the feature space as can be seen from Figure 4.3. If we analyse the performance on the bottom grouping of instances the performance achieved by different streamliners varies drastically. Streamliner '2', which enforces that each *function* representing a code word is *monotonicallyDecreasing*, is mostly satisfiable across that group but generally it only achieves a reduction of less than 50%. Streamliner '121', which enforces that for approximately half of the code words the characters have an odd domain value, 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'. Streamliner '8' enforces that each code word is only composed of characters in the upper half of the character domain. 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.

Because of this variation in how different instances respond to *streamliners* ideally the set of instances we use for *training* should be diverse otherwise the generated streamliner portfolio may be skewed towards instances of one or a limited number of types and so not generalise across the problem class. As cautioned by Hooker [99, 100], there is a need to be careful about the conclusions that can be drawn beyond the selected instances. For optimization problems there have been documented cases where the benchmark library instances are not very diverse [95] and as such there is a danger of overfitting where the algorithms are developed and tuned to perform well on these instances without understanding the performance that can be expected on instances with diverse properties.

In analyzing Figure 4.3 it can be seen that the variation of the footprints across the feature space does not appear to be random. For each streamliner there are regions of the space that exhibit satisfiability and a degree of reduction and then there are other parts that tend to produce infeasibility. It appears that there is a *relationship between the instance features and the streamliner performance*, such that instances located close together in the high dimensional feature space will possess similar performance characteristics [112] in relation to streamliner application. We can exploit this fact and use *instance features* as a cheap proxy for performance data. For instance, if a streamliner has been proven *unsatisfiable* on a particular instance we can draw the conclusion that other instances located nearby in the feature space may have a similar result.

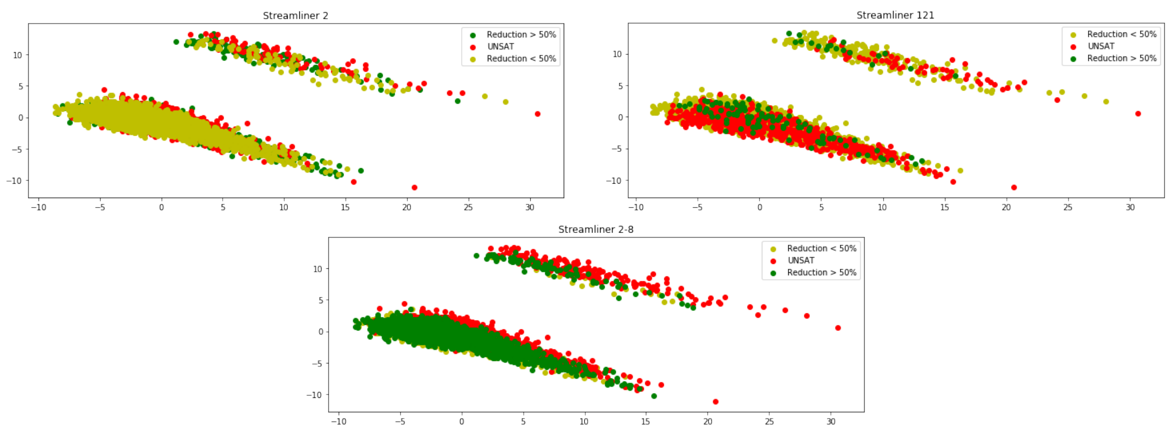


Figure 4.3: Performance of three example streamliners on the training set of 4647 instances generated for the Fixed Length Error Correcting Code problem with CHUFFED. Each two dimensional plot is a projection of the original multi-dimensional instance feature space via Principle Component Analysis [40].

4.3 Training Set Construction

For some of the examined problem classes the instance sets generated through the automated instance generation procedure are too large to use in their entirety when searching for effective streamliners. For example, on the Fixed Length Error Correcting Codes problem 4647 and 6930 instances were generated for Chuffed and Lingeling respectively as seen in Table 4.2. When using all instances, one candidate streamliner evaluation could take around 15 CPU days, assuming a worst case of 5 minutes per instance. It is thus necessary to form a compressed representation of the instance space that we can use directly in search. During compression we want to retain as best as possible the performance characteristics of the instance space. If our training set includes sampling bias this will affect the conclusions that can be drawn and directly limit the ability of our streamliner portfolios to generalize across the problem class.

As noted in Section 4.2 it appears that instances located close together in the high dimensional feature space possess similar performance characteristics [112]. Based upon this we can look for clusters of instances present in the feature space and select one or more per cluster to compress the dataset but still retain the performance characteristics. The accuracy of this method may not be perfect, however it should allow us to retain some of the diversity of the dataset while being relatively cheap to use. GMeans [91] clustering is used on the feature space to detect the number of instance clusters (column 4 of Table 4.2). An example of clustering results for the Fixed Length Error Correcting Code with

CHUFFED is shown in Figure 4.4. We can then build a compressed representation of the training set by selecting a subset of instances per cluster.

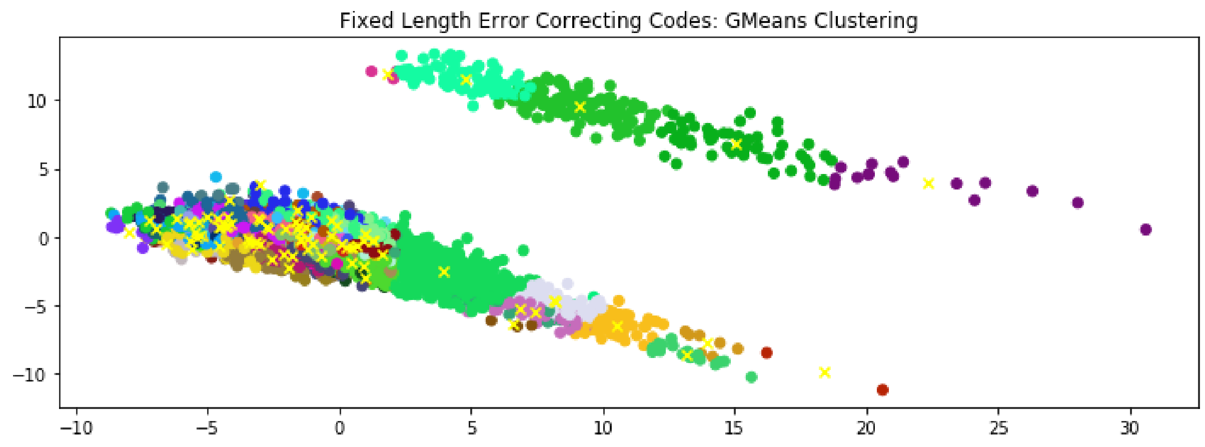


Figure 4.4: GMeans clustering results, with 128 clusters detected, on the instance feature space (projected to 2-dimensional space by PCA) for the Fixed Length Error Correcting Codes problem with CHUFFED. Each color represents a cluster.

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 then, in order to keep the computational cost manageable while still retaining the diversity of the instance space, only one representative instance per cluster is selected. In the scenarios where the number of detected clusters is less than 50, a subset of instances per cluster are chosen until the minimum number (50) is met. 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.

Problem	#Candidate Streamliners	#Instances		#Clusters	
		Chuffed	Lingeling	Chuffed	Lingeling
BACP	108	235	133	-	-
BIBD	200	427	272	-	-
CoveringArray	64	4301	1641	153	54
Car Sequencing	36	4376	5651	171	149
FLECC	144	4647	6930	128	162
Transshipment	68	1534	3889	-	96
SocialGolfersProblem	260	709	340	-	-

Table 4.2: 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. A - denotes that the number of identified clusters was < 50.

4.4 Summary

This chapter discussed the importance of automatic conjecture evaluation in the construction of high quality streamliner portfolios. The process by which a diverse set of benchmark instances is generated from the ESSENCE specification of the problem class was introduced. Finally, the use of Feature Generation and Clustering to create a compressed training set suitable for exploration is discussed.

Chapter Five

Identifying Effective Combinations of Streamliners

In this Chapter we are initially going to look at the concept of combining streamliner conjectures, introduced first by Lebras et al in their work on double-wheel graphs [128]. The benefits that this can bring in solving difficult problems is discussed as well as the adverse effect that it can have on the complexity of search. The structuring of the possible combinations into a lattice structure and several pruning methods utilized to remove ineffective combinations are defined. Various approaches for searching and producing a portfolio of streamliner combinations are discussed and the search algorithm utilized in this thesis is defined.

5.1 Combining Streamliners

A major research challenge in discrete mathematics is to characterize what families of graphs are graceful. A graceful graph is one in which all of the vertices can be labeled with distinct integer values from $\{0, \ldots, e\}$ such that each edge has a unique value corresponding to the absolute difference between its end points. A number of different families of graphs have previously been proven to be graceful such as Wheels [73], Gears [138], Helms [13], Webs [114]. Lebras et al used streamlined reasoning to help them construct the first polynomial time construction method for double-wheel graphs thus proving them to be graceful [129]. They used the same procedure as in their previous work where solutions would be collected and would be manually analyzed for patterns and the onus was on the human expert to construct a streamliner constraint that captured the regularity. What is interesting in this work however is that in order to find larger double-wheel graphs Lebras et al combined different streamliners together to a model simultaneously and showed that it can result in larger performance gains than any of the conjectures in isolation. Figure 5.1 shows the combinations of streamliners that were used to solve varying sizes of double-wheel graphs.

With the original model, only graceful double wheel graphs up to size 9 could be constructed within 60 seconds using standard constraint reasoning. Figure 5.1 shows that even relatively simple streamliner conjectures can have a profound effect on performance. For instance the streamliner r_2 only enforces that the middle of the double-wheel graph is labelled 0 however it is able to allow solutions of size 18 to be discovered, twice the size of the original model. The application of single streamliners can be very effective but the impact is limited by the degree to which they can restrict the search space. To improve upon this conjectures can be combined together to further restrict the search space and focus the solver allowing solutions to be found faster. For instance, Lebras et al enforced that the centre of the double wheel is labelled 0 combined with the fact that the inner circle only contains odd numbers. The combination of these two regularities allowed larger graphs of up to size 19 to be solved. This procedure can be recursively repeated to create larger and larger combinations of conjectures.

Whilst combining streamliners can be very beneficial in providing larger reductions in search and allowing more difficult problems to be solved it does have an adverse effect on the complexity of search. To find effective streamliners not only does the set of candidate streamliners have to be evaluated but now potentially the power set of all combinations has to be explored.

Set of Streamliners	Size
$r_0 : \emptyset$	9
r_1 : { C_1 is odd }	11
r_2 : {0 at center }	18
$r_3 : r_1 \cup r_2$	19
$r_4 : r_3 \cup$ {Inc. steps of 2 in C_1}	-
$r_5 : r_2 \cup$ {Inc. steps of 2 in C_1} $\cup$ { C_1 mostly odd}	21
$r_6 : r_5 \cup$ {Dec. steps of 2 in C_1}	22
$r_7 : r_6 \cup$ {Inc. steps of 2 in C_2}	23
$r_8 : r_7 \cup$ {Dec. steps of of 2 in C_2 }	38

Figure 5.1: In their work on Double Wheel Graphs, Lebras et al [84] combined conjectures to further restrict search and solve graphs of a previously unattainable size

5.2 Providing Structure: Definition of the Streamliner Lattice

The possible combinations of streamliners can be formulated as a lattice structure: the root is the original Essence specification and an edge represents the addition of a streamliner to the combination associated with the parent node as seen in Figure 5.2.

5.2.1 Exploiting Structure: Pruning the streamliner lattice

It is possible to apply some basic rules to the lattice to remove streamliner combinations from evaluation that we know are sure to render an instance unsatisfiable. In this work we employ three forms of pruning which are detailed in the following sections.

5.2.1.1 Pruning based upon tags

As discussed earlier in Chapter 3 when an Essence specification is streamlined each generated conjecture is assigned one or more *group* tags. Conjectures that apply to the same decision variable and share groups are not combined as they will be conflicting and unsatisfiable. For instance these *groups* would prevent combining the conjectures that would force a set simultaneously to contain only odd numbers and only contain even numbers. This also removes the possibility of combining two different streamliners that differ only in the values of their softness parameters.

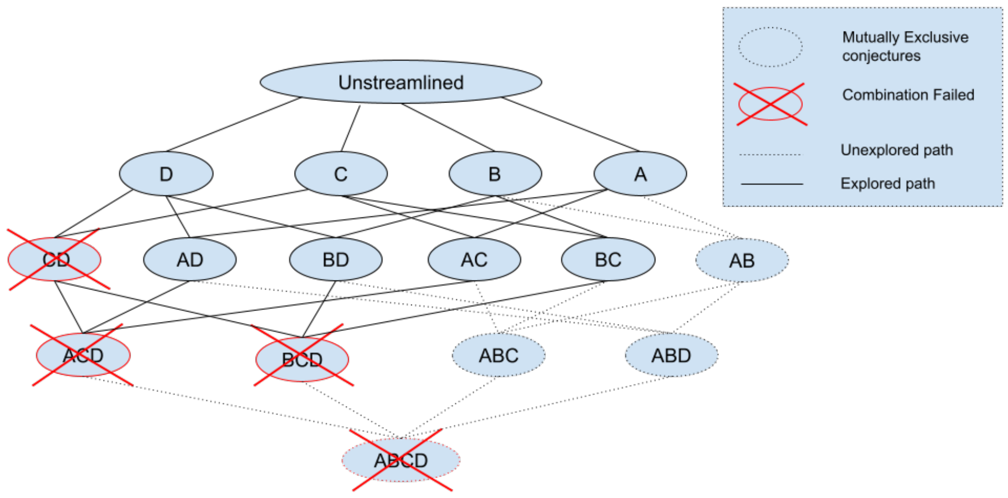


Figure 5.2: The power set of singleton candidate streamliners is explored to identify combinations that result in powerful streamlined specifications. If small sets of conjectures that fail to retain solutions are identified, such as CD, all supersets can be pruned from the search, vastly reducing the number of vertices to be explored. Here streamliners A and B are tagged (Section 3) mutually exclusive, and so no streamliner combinations containing both will be evaluated however the lattice states are shown for informational purposes.

5.2.1.2 Pruning based upon failure

Up to now streamliners have been treated independently from one another; each conjecture is evaluated independently and it is assumed that the performance of one conjecture has no connection with another. This is valid for the single candidate streamliners as because each conjecture imposes a different ESSENCE restriction the performance characteristics may differ vastly from one to another. The vast majority of states in the lattice however are not single streamliners but instead combinations of conjectures that build upon other combinations. These combinations will share performance characteristics with the conjectures that comprise it. If upon evaluation the model with the addition of the streamliner combination is proven to be *unsatisfiable* on a given instance, we know that any superset of this combination will also be *unsatisfiable*. This knowledge can be used to reduce the number of instances being unnecessarily evaluated at each lattice node. At a given node in the lattice the intersection of the sets of satisfiable instances from all available parents 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 that AB is only satisfiable on instances 1,2 and 3, while AC is only satisfiable on instances 2 and 3. For ABC we only need to evaluate the streamliner combination on instances 2 and 3. This reduces the computation on the current node by 60%. If a conjecture has zero applicability across the instance space, i.e it removes all solutions for all instances all supersets of this combination are removed from the lattice (see Figure 5.2 for an example).

In order for this pruning to be enacted the conjecture must have been proven to be unsatisfiable and this is an important stipulation. There can often be cases, as finding a proof of unsatisfiability in a streamlined subspace can take far longer than finding a solution to the original problem, where a streamliner is evaluated and *timesout* in the given time limit. Pruning all supersets from evaluation could then be premature and potentially remove good configurations from consideration. This need for a proof can often limit the effectiveness of this pruning strategy.

Additionally 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 Figure 5.2 it can be seen that the streamliner *CD* is unsatisfiable which allows for certain 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 have a large impact on the amount of pruning that can be performed.

5.2.1.3 Pruning Softened Streamliners

Softened Streamliners generate an independent constraint for each possible defined softness value and as such have a large impact on the combinatorial size of the lattice to be searched. Using the BIBD problem as an example if we take one of its softened streamliners, such as (symmetric {1,2,4,8,16,32,64}), 7 independent constraints will be generated. These constraints can be strictly ordered in terms of their strictness. This ordering can be used for pruning during search as we know that these streamliners are semantically the same, i.e they enforce the same underlying constraint but only differ in the required coverage. Based upon this ranking if one softened constraint (*symmetric(bibd)/16*) is proven to be unsatisfiable on an instance it then allows us to conclude that all stricter versions (symmetric {1,2,4,8}) must also be unsatisfiable and can be pruned.

This pruning can have profound effects but its impact is governed by a large extent on the ordering of traversal of the softened constraints. Two ways in which this could be traversed are using a descending or ascending ordering. An ascending order, $(1 \rightarrow 2 \rightarrow 4 \ldots)$ has limited pruning ability as you can't extrapolate the result onto any other softened constraints. Descending order $(64 \rightarrow 32 \rightarrow 16 \rightarrow 8 \ldots)$ does profit from the ability to prune however the downside is that the most relaxed version of the streamliner is being applied first and as we have seen earlier this may not be that effective because of its limited restrictive power. A more nuanced approach is to use a binary search method where initially the softness parameter is started in the middle at *8*. If the chosen value is unsatisfiable we can then prune all stricter versions and then move to evaluate *32*.

5.3 Searching for a Streamliner Portfolio

The three pruning rules described above only remove combinations that are sure to fail, or are equivalent to a smaller set of conjectures. 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.

5.3.1 Focused Search

For each problem the size of the lattice and number of valid streamliner combinations is defined by $2^{NumberCandidateStreamliners}$. For even a relatively small number of candidate conjectures the size of the space will quickly become infeasible to search. Whilst many of the states within the lattice are theoretically valid they are likely to be ineffective. As more conjectures are combined the resultant model becomes more and more restrictive until at a certain point it will become so strict that for any given instance all solutions will be removed. These ineffective states can encompass large swaths of the total lattice. For instance, if we look at the number of combinations of size *300* for the *BIBD* problem there will be $\approx 2.2 \times 10^{96}$ valid states in the lattice. It is unlikely that for any of these states the resultant model produced by combining 300 different conjectures will be effective.

Given our limited computational resources evaluating these *"large"* states, which we believe a priori are going to be ineffective, would be wasteful. If it is believed that the combination of too many streamliners is detrimental one could cap the maximum combination size which would significantly limit the size of the search space to be explored. The problem is that it would be difficult to define what is this bound for a given problem. For instance in the work done by Lebras et al on Double Wheel Graphs they combined up to ten different conjectures to allow them to construct graphs of up to size 38. If they had enforced an arbitrary limit they may not have found this combination and been able to achieve this result. In this work we focus search such that it always starts at the root, which represents the original ESSENCE specification. From there the lattice is explored, with the single candidate streamliners and smaller combinations being explored first.

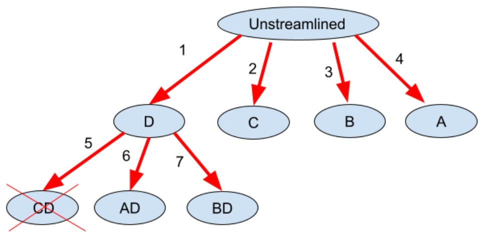
5.3.2 Uninformed Search

The simplest approach for searching this lattice would be to utilize systematic search methods such as breadth-first search (BFS) and depth-first search (DFS) to explore the streamliner lattice in an uninformed manner. Both of these methods however have intrinsic deficiencies in their ability to effectively search the lattice structure. Using the full lattice structure defined in Figure 5.2 as reference let us analyze these deficiencies.

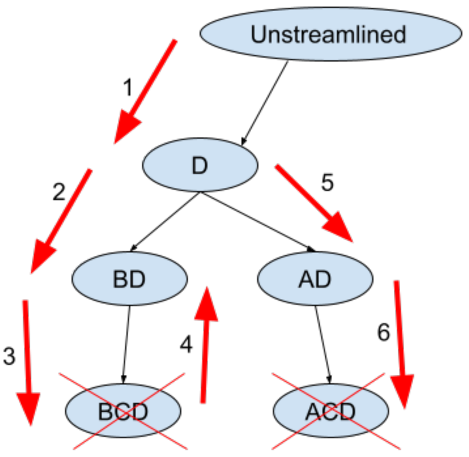
Depth-First search will dive down into the lattice structure, evaluating larger and larger combinations until the current set of conjectures has been proven to remove all solutions at which point it can backtrack. For instance, Figure 5.3b shows an example of how DFS may explore. One of the main issues with DFS is that its traversal ordering often affects the amount of dynamic pruning that can be performed. In this example, as DFS dives before evaluating all smaller combinations it means that combination {CD} is not found to have failed which means that supersets {ACD} and {BCD} are unnecessarily evaluated. Whilst the impact of this may seem small in this example as the number of candidate streamliners increases and the size of the lattice grows the ability to prune whole supersets from evaluation can have a significant impact on the complexity of exploration.

Another issue is that DFS will only backtrack if a proof of unsatisfiability can be obtained, showing that the current combination removes all solutions on the given instance{s}. If a streamliner is evaluated and times out before finding a solution it can't be removed as this is too strict; we can't be sure that when combined with another conjecture it won't perform well. In the work done by Gomes et al there were cases where a single conjecture by itself did not produce a reduction in search but when combined it became effective. This proof of unsatisfiability however can be hard to obtain and often means DFS traverses deep into an unsatisfiable subtree before eventually backtracking.

As seen by Figure 5.3a Breadth-First search performs sequential exploration of the levels in the lattice. This sequential exploration solves both of the issues faced by DFS. Firstly, the failure of combination {CD} would be discovered first and any supersets could be removed from the lattice without evaluation. Secondly, this search will not get stuck exploring large unsatisfiable subtrees in the lattice where no proof of unsatisfiability can be obtained. The detriment with BFS however is in its lack of exploitation. Larger combinations can be an effective component of a streamliner portfolio as theoretically because of their more restrictive nature they can often achieve a higher reduction in search. Because every level has to be fully explored before moving onto the next, in the time allocated to search the lattice it is often infeasible to evaluate these larger combination sizes. For instance in this thesis the smallest problem in terms of lattice size is CarSequencing with only 36 candidate streamliners generated. As discussed more in Chapter 6 the largest combination in the streamliner portfolio for the CarSequencing problem is composed of 3 conjectures. In the worst case for BFS to find this combination it would have had to traverse and evaluate through 46,656 conjectures. Due to the graded nature of the instances that we use, as discussed in Chapter 4, the evaluation of one streamliner can take considerable time and it is unlikely that in the allocated search budget this many conjectures could be evaluated.



(a) Breadth-First exploration of the streamliner lattice



(b) Depth-First exploration of the streamliner lattice

Figure 5.3

5.3.3 Defining a reward

The lattice is composed of conjectures automatically generated from the Essence specification and so it is likely that there is going to be a distribution in how the conjectures perform; some may be ineffective (providing little or no search reduction) or may even remove all solutions and others may achieve substantial reductions. During traversal of the lattice, when search arrives at a particular node, the combination of conjectures that it represents is evaluated across the training set. After evaluation (Figure 3.9) information is available on how the combination performed on each training instance. With the previous uninformed search methods the lattice is systematically searched and the same amount of effort is spent trying to extend suboptimal combinations rather than focusing on combinations that have already proven to be effective. If our search is *complete* and we can fully explore the lattice this is not a problem, however because of the cost of streamliner evaluation and the inordinate size of the lattice a *complete* search is rarely if ever possible.

Instead if we can structure the information available from evaluation into a reward we can use that to the search the lattice in a more informed manner. 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. As the search space is further restricted however solutions will be continually removed up until the point that no solutions remain. If a streamliner is too restrictive its ability to apply onto unseen instances may be effected.

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 achieved by the streamliner on the satisfiable instances.

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

$$r_x \text{ dominates } r_{x\prime} \iff \forall i \in [1..d], r_i \geq r_{i\prime}, \exists j \in [1..d], r_j > r_{j\prime} \tag{5.1}$$

Given a set of streamliners, the non-dominated solution set is a set of all the streamliners that are not dominated by any other streamliner member of the solution set. The aim of search is to find the pareto optimal set of streamliners which is defined as the non-dominated set of the entire feasible decision space. We want our concept of reward to help guide the search towards finding these non-dominated streamliner combinations. Initially search begins with an empty pareto front and as it progresses streamliner combinations in the lattice are evaluated and their vectorial reward for *AverageApplicability* and *AverageReduction* across the instance space are computed. In order to generate a reward for that particular state and streamliner combination the current portfolio of non-dominated streamliner combinations is used to compute the Pareto dominance test. 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 reward of 1 is given, otherwise 0. Formally defined as:

$$r_{u;dom} = \begin{cases} 1 & 1 \text{ if } \nexists r \in P, r \succ r_u \\ 0 & \text{otherwise} \end{cases}$$

When the pareto dominance test is calculated, any dominated streamliner combinations presently in the portfolio are removed. One consideration that has to be taken into account is that as the pareto front evolves over time then the reward values of the Pareto dominance test are non stationary since they depend on the portfolio. This means that earlier in search when less combinations have been evaluated it is easier for a combination to get a positive reward value as it is only being compared against a small subset of the configuration space. As search progresses the portfolio will begin to approximate the pareto optimal set and so it becomes more and more difficult to achieve a positive reward. The effect of this is that in the lattice there will be actions that appear to have a high reward but this may not be representative of the true value of the action but just related to the fact that is was evaluated early on in search. To combat this we use the cumulative discounted dominance (CDD) [196] reward mechanism during reward update.

5.3.4 Exploration vs Exploitation

We want to use the reward of the conjectures to help guide the search and indicate which parts of the lattice to focus our resources on. Doing this however raises the issue of the exploration/exploitation problem: if we can identify a combination of streamliners that performs well, should we try and exploit that combination further by evaluating the addition of further streamliners, or should we explore other combinations that may at present seem less promising?

When traversing the lattice structure we can formalize the exploration/exploitation problem faced at each node as a Multi-Armed Bandit. Bandit Problems [177, 115, 12, 139] are sequential decision problems where at each stage there are k possible actions, each yielding an unknown payoff, and one of these k actions must be chosen in order to maximise the total reward. The reward distributions amongst the actions are initially unknown however information on rewards can be gained through playing each action. Formally a set of K probability distributions $(D_1, \ldots, D_k)$ with associated expected values $(\mu_1, \ldots, \mu_k)$. The probability distributions $(D_1, \ldots, D_k)$ generally correspond to different arms on a slot-machine. For each turn the player selects one of the arms, with index j(t) and receives a reward r(t) $D_j(t)$. The idea behind the Bandit-Problem, is to try and find out which of the arms yields the highest payoff whilst still maximising our possible reward [125]. Whilst the player balances their exploration and exploitation of the different arms they will inevitably not pick the arm with the highest reward on every turn. The difference between the actual given reward for a player and the theoretical maximum possible reward for any turn T is know as the total expected regret and can be defined as:

$$R_T = T_{\mu*} - \sum_{t=1}^{T} \mu_j(t)$$

where $\mu*$ represents the reward from the best arm. The total regret for a player can then be defined as:

$$R_N = \mu * T - \mu_j \sum_{j=1}^{K} \mathbb{E}(T_K(T))$$

where $\mu*$ represents the reward from the best arm and $\sum \mathbb{E}(Tk(T))$ represents the predicted number of plays for arm k after turn T.

From Lay and Robbins (1985) [127] it can be seen that for any algorithm designed to solve the bandit-problem the growth of regret is lower bounded by $\sigma(logt)$. If an algorithm is able to get within constant factors of this growth then it is said to solve the bandit-problem.

5.3.5 MOMCTS

Monte Carlo Tree Search (MCTS) [32, 49] attempts to solve the exploration/exploitation dilemma during tree search by treating the choice of where to explore at each node as a multi-armed bandit problem. It has been proven to be very effective in a number of competitive settings [56, 77]. However its application is not limited just to games: it can also be effective for single-agent sequential decision problems [187].

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 [196]. The algorithm has four phases, as summarised below. The full Algorithm is defined in Algorithm 1 and an illustration example of the four phases are shown in Figure 5.4.

1. **Selection**: Starting at the root node, the Upper Confidence Bound applied to Trees (UCT) [31] policy is applied to traverse the explored part of the lattice until an unexpanded node is reached.

   A child node is selected to maximise:

   $$UCT = X_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

   where n is the number of times the current (parent) node in the lattice has been visited, n_j the number of times child j has been visited, X_j is the cumulative reward associated with child j and $C_p > 0$ is a constant

   One deviation from standard MCTS and rollout algorithms in general is that X_j represents a cumulative reward here instead of an average. The reason for this is that only a very small number of the simulations performed will be able to discover a non-dominated streamliner and so if an average is used the rewards of even good trajectories through the lattice will tend towards 0 over time which makes the exploration factor the dominating term in the UCT calculation thus making MCTS effectively degenerate to random search.

2. **Expansion**: Uniformly select and expand an admissible child.

3. **Simulation**: The collection of streamliners associated with the expanded node are evaluated.

4. **BackPropagation**: The result of the evaluation is propagated back up through all paths in the lattice to update CDD reward values, as shown in Figure 5.4. At each node updated during back-propagation a discount mechanism is used to discount the effects of non stationarity, introduced by the evolving portfolio, by attempting

to forget old rewards and reflect up to date information. A reward is updated as follows:

$$r\prime_{s,a;dom} \leftarrow r\prime_{s,a;dom} * \sigma^{\delta t} + r_{u;dom}$$

$$t_{s,a} \leftarrow t;$$

where $t_{s,a}$ denotes the index of the last update that affected (s, a) and $\sigma \in [0, 1]$ and $r_{u;dom}$ represents the reward of the most recent simulation.

Since our search is operating over a lattice, a node may have multiple parents. This requires an alteration to the back propagation employed in MCTS: when we perform back propagation that reward value is back propagated up all paths from that node to the root. To illustrate consider a problem with two streamliners {A,B} and we are back propagating from a node in the lattice representing the combination {AB}. There are two paths by which this node could have been reached, {root → A → B} and {root → B → A}. Even though the algorithm will have only descended one of these paths, because the reward value of a node in the lattice is representative of the ability of the streamliner combination represented by that node to combine and produce effective reductions in search the node in the lattice which represents streamliner combination {B} should also receive this reward. For this reason both paths are rewarded accordingly and the reward generated is back propagated up all paths from that node to the root. We also ensure that a node that lies on more than one such path is rewarded only once.

We must also consider the situation where a node in a path back to the root has not yet been expanded. If we simply ignore such nodes, their true reward will not be reflected in their reward values because all reward values back propagated from child nodes prior to their creation will be lost. Our approach is that when a node is expanded, it absorbs the reward value and visit count of any of its immediate children that already exist in the lattice. This avoids caching a potentially large set of values but still means that reward values are maintained for nodes around the focus of our tree search.

Algorithm 1 Multi Objective Monte Carlo Tree Search

```
 1: procedure MOMCTS-DOM(originalModel, numIterations, VBS, paretoFront,
    cache)
 2:     node ← originalModel,
 3:     P ← paretoFront                                   ▷ Streamliner Portfolio
 4:     n ← 1
 5:     while n < numIterations do
 6:         node ← UCTSelection(node)
 7:         child ← randomly expand child of node
 8:         streamliners ← pathFromRootToNode(child)
 9:         if streamliners in cache then
10:             (Applic,MeanReduc) ← cache{streamliners}
11:         else
12:             (Applic,MeanReduc) ← eval(streamliners, VBS, cache)
13:             n ← $n+1$                                  ▷ Increment iteration count
14:         end if
15:         BackProp((Applic, MeanReduc), child)
16:     end while
17: end procedure
18: procedure BackProp($r_u$, currNode)
19:     if $r_u$ is not dominated by any point in P then
20:         Prune all points dominated by $r_u$ in P
21:         $P \leftarrow P \cup \{r_u\}$
22:         r ← 1                                           ▷ Reward Value
23:     else
24:         r ← 0                                           ▷ Reward Value
25:     end if
26:     NodesToVisit ← getAllNodesInPathToRoot()
27:     for TreeNode ← NodesToVisit do
28:         discount ← CalcDiscount()
29:         reward ← (TreeNode.reward * discount) + r
30:         updateRewardValue(TreeNode, reward)
31:         updateVisitCount()
32:     end for
33: end procedure
34: procedure Eval(streamliners, VBS, cache)
35:     ResultsDict ← run(streamliners)          ▷ Run streamliners on training set
36:     cache[streamliners] = ResultsDict                   ▷ Cache results
37:     for (instance, streamlinerResult) ← ResultsDict do
38:         if VBS[instance] *better* streamlinerResult then
39:             ResultsDict[instance] ← VBS[instance]
40:         end if
41:     end for
42:     return {Applic(ResultsDict), MeanReduc(ResultsDict)}
43: end procedure
```

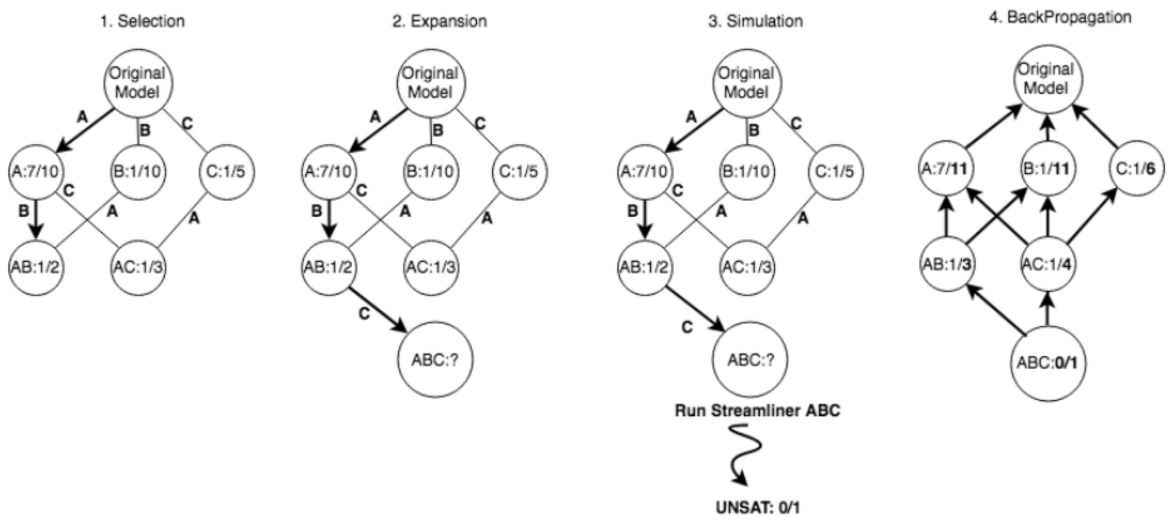


Figure 5.4: MOMCTS-DOM operating on the streamliner lattice. A, B and C refer to single candidate streamliners generated from the original ESSENCE specification. As MOMCTS-DOM descends down through the lattice the streamliners are combined through the conjunction of the individual streamliners (AB, ABC). The nodes are labelled with CDD reward value divided by the number of times visited.

After the streamliner has been evaluated the vectorial reward ⟨Applicablity, Search Reduction⟩ across the set of training instances is calculated and returned.

5.3.6 Improving Portfolio Strength Using Hydra

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 5.3. 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} unsatifiable, 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 resultant 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.

In order to solve this issue, we adapted our search to incorporate elements of Hydra [198], 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 complement the strengths of the combined portfolios built in the prior rounds.

Following the ideas of Hydra, 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 is 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 at. 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. Hydra is very effective in this domain and is able to from round to round find new streamliner combinations that complement the solving strengths of previous portfolios. This can be seen from Figure 5.5

where for all 4 of the problems present Hydra produces a portfolio in each round that produces a drastic decrease in logarithmic solving time showing that it is finding new streamliners that complement the previous solving strengths.

5.3.7 The risk of overfitting

Even though Hydra has great strengths one important thing that has to be taken into account is the tendency to produce portfolios that over-fit the training data. In Machine Learning there is a tradeoff between the bias and variance of a model. The bias represents the accuracy of the model and the bias error is calculated as the difference between the expected model prediction and the correct value. The variance represents the consistency of the model in its predictions and the variance error is due to the variability in the model prediction for a given data point. In general simpler models tend to have higher bias as they fail to fit regularities in the training data and as the model complexity increases the bias decreases but as a result the variance begins to increase [149]. Ideally a learnt model should have low bias and low variance however the bias-variance trade-off is a well documented problem in machine learning [120, 76].

The aim of Streamliner Search is to try and identify structural regularities that exist in the instance space and use these to solve unseen instances. However if Hydra is left to run it has a tendency to produce a highly complex portfolio which over-fits the training set. From Figure 5.5 it can be seen that the idea of complementary portfolios works very well and a large decrease in solving time can be noticed in the first few rounds. However, for all 3 problems they reach a point in which the decrease in solving time plateaus. In the given time budget this plateau can extend for a large number of rounds and during this time as seen from Figure 5.5 the size and complexity of the portfolio steadily increases. This can create two problems. Firstly, a streamliner can be added to the portfolio if it provides a greater performance on just one or more instances from the training set. This means that in later stages there is a risk that streamliners will be added that don't represent any structural property in the instance space but just by chance happen to work on a particular instance and produce a good reduction. It is not expected that the performance will translate onto unseen instances and as such it complicates the scheduling process. Secondly, as can be seen from Figure 5.5 for the 3 problems shown there is a large increase in portfolio complexity which is not accompanied by a substantial decrease in Solving Time. Because of this we may want to cap the size of the portfolio so we can benefit from the positive benefits of Hydra but not have a resultant complex portfolio.

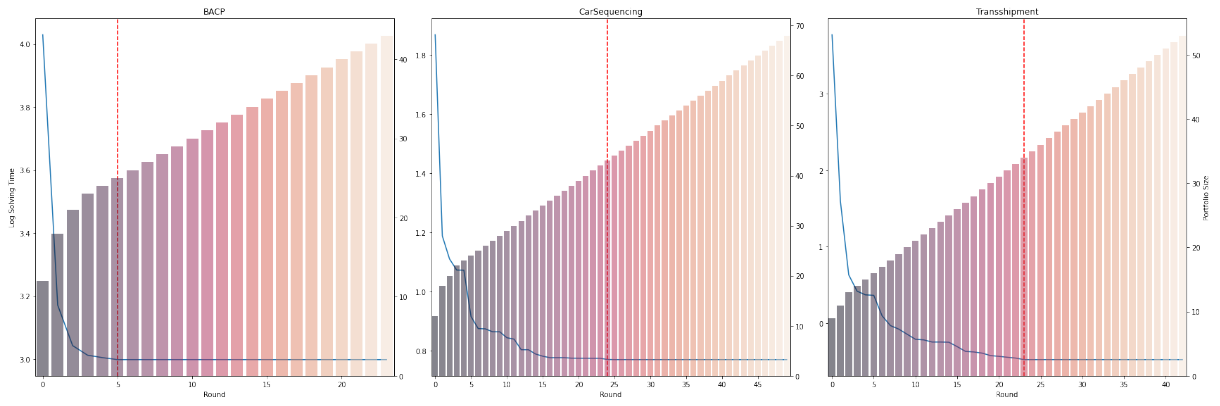


Figure 5.5: For three problems, BACP, CarSequencing and Transshipment a comparison between the log solving time (blue line) of the validation distribution and the size of the portfolio (histogram) across multiple rounds of our search algorithm. The red dotted line displays the round at which there is no further reduction in solving time.

A validation set can be used to try and alleviate some of these issues. The validation set is constructed in a similar means to the Training set from Section 4.3. When the set of clusters are identified using GMeans for each cluster another instance can be chosen to form a validation set of the same size and distribution of the training set. The idea behind this validation set is to validate that the streamliner does capture some structural regularity and that the performance across the feature space is maintained. There are a variety of ways in which the validation set could be leveraged. The simplest method is to take each streamliner evaluated on the training set and additionally evaluate its performance on the validation set but this would constitute a large increase in the training overhead.

Another method, which we use in this work, is to only utilize the validation set when the current streamliner is non pareto dominated and would be added to the portfolio. Addition to the portfolio occurs rarely and means that the increase in cost of utilizing the validation set is only marginal. The validation set here is used by evaluating the streamliner on the same satisfiable clusters from the training set. More precisely for every instance (which constitutes a cluster) upon which the streamliner on the training set is satisfiable the streamliner will be evaluated on the same satisfiable clusters within the validation set. Through evaluation it will become apparent, based upon if the conjecture maintains its performance, whether or not it captures a structural regularity. The validation set helps to control the complexity of the generated portfolio. Firstly, it helps to reduce overfitting of the training set where streamliners can be added to the portfolio purely by improving on one or more instances by chance. In this case we can prevent increasing the complexity of the portfolio unnecessarily.

5.4 Summary

In this Chapter the idea of combining streamliner conjectures to provide larger overall reductions in search was introduced. The "streamliner lattice" was defined and its utility in providing structure for search and pruning was discussed. Alternative approaches for searching the lattice and building a streamliner portfolio were explored and the algorithm utilized in this thesis was defined.

Chapter Six

The Streamliner Selection Problem

This chapter initially shows how the performance characteristics of a streamliner conjecture can vary drastically between solving paradigms and why it is necessary to perform paradigm specific search and generate independent streamliner portfolios. The constructed portfolios generated through the search method seen in the previous chapter are then summarized; in particular visualizing the distribution and size of the generated pareto fronts. For a subset of the problems, a more in-depth analysis is then performed to discuss some of the interesting properties of the constructed portfolios and analyze the performance of the constituent conjectures. Lastly, the problem of *"Streamliner Selection"* is introduced which deals with selecting from the portfolio an effective streamliner for an unseen instance. Various uninformed and informed methods for solving this problem are discussed.

6.1 Independent Portfolio Construction

For each problem an independent streamliner search is performed for both of the solvers and paradigms used in this paper, Chuffed (CP) [43] and Lingeling (SAT) [26]. As streamliners are generated from the Essence specification of a problem class which is solver independent it may seem unnecessary to perform two independent searches. An effective streamliner captures a structural regularity within the problem and this should exist regardless of the solver used; a streamliner that is effective for Chuffed will also be effective in Lingeling and vice-versa. 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 incredibly verbose in the SAT encoding, which may result in substantial overhead during search and as such affect performance. In Savile Row the default encodings for *SAT* are order encoding [188] for sums and support encoding [79] for binary constraints.

Figure 6.1 shows that 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. This could be for a variety of reasons. The same streamliner might not achieve the same propagation in SAT as it does in CP or the encoding of the streamliner may become too verbose and the overhead of the additional clauses outweighs any reduction the streamliner might achieve.

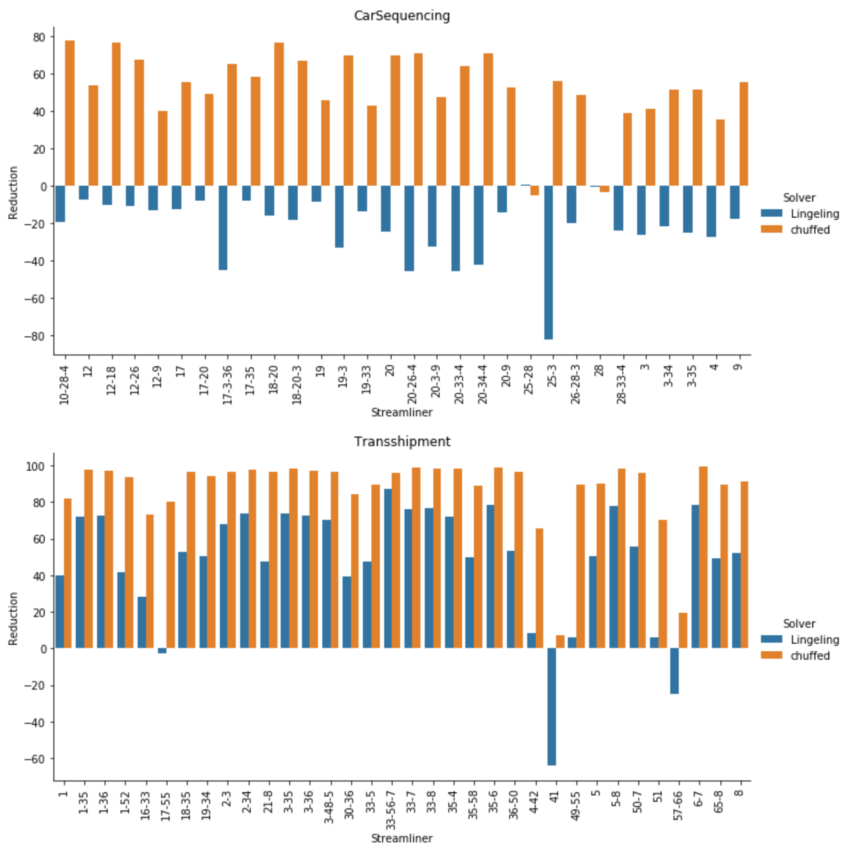


Figure 6.1: For Transshipment and CarSequencing the portfolios generated during the Chuffed streamliner search are tested on Lingeling on the same set of test instances. The Average Reduction across the two portfolios are represented for both paradigms. The same set of test instances are used so that any variation in the reductions of the streamliners is purely due to the different setting.

6.2 Constructed Portfolios

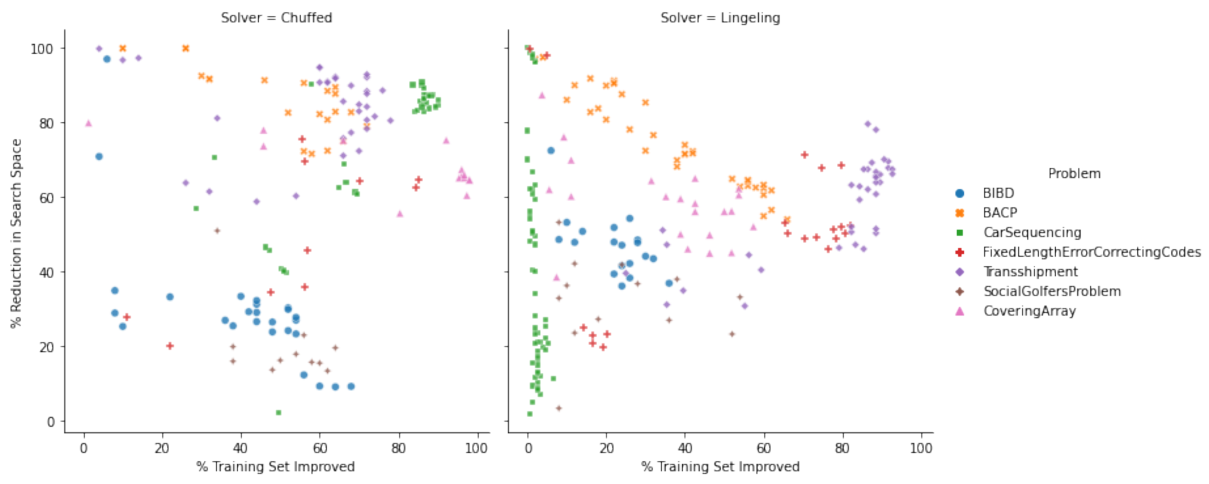
The output of *Streamliner Search* (Chapter 5) is a portfolio of streamliners possessing complementary strengths on the training distribution.

Figure 6.2a shows for each portfolio the performance of the constituent streamliners in terms of the two objectives used, *AverageReduction* and *AverageApplicability*. Due to the portfolio builder technique that is used during search the figures will not be a true representation of a pareto front as any conjectures added in later rounds may be pareto dominated but will remain. Comparing the performance of the portfolios across problems it does seem that some problems are more conducive to streamlining than others. For instance, across both solvers the performance of the portfolios for Transshipment and BACP dominate those of BIBD and SocialGolfers. It is an open research question however whether this is due to an inherent property of those problems that makes them difficult to streamline; for instance the structural regularities that do exist in solutions cannot be captured concisely; or due to nature of the streamliners generated from ESSENCE. Transshipment and BACP both have *function* domains whereas BIBD and SocialGolfers have *relation* and *set* domains.

It may initially be expected that as streamlining is such an aggressive technique that generated conjectures may work well for some instances but in general will have limited success across the instance space. Figure 6.2a shows the opposite however in that many of the conjectures generated by our system attain a high % applicability and reduction across the training set. Excluding one or two problems in general there are consistently conjectures in the portfolio that work on more than 50% of the training set and achieve more than a 50% reduction in search. These conjectures generated automatically from the ESSENCE specification must be capturing concisely some structural regularity that exists in many of the instances within the training set.

Figure 6.2a also shows the benefit of constructing a portfolio of streamliners. For all of these problems there is no one dominating conjecture but instead the conjectures form a distribution in terms of the two competing objectives. The use of a portfolio approach during search allows this diversity to be captured. From Figure 6.2b, which details the overall size as well as the max combination size of the generated portfolio for each problem and solver, it can be seen that the constructed portfolios are not trivial and generally contain a number of different combinations ranging from the smallest for FLECC at 11 conjectures to the largest with CarSequencing at 64.

Across all portfolios the largest combination found is composed of three different conjectures. Initially this may seem relatively small and it may be expected with the use of MOMCTS that search could dive deep into the lattice and the portfolio would contain larger combinations. One of the reasons for this is that the large numbers of conjectures generated from the ESSENCE spec means with the limited resources available it can be difficult to descend far into the lattice before the resources are exhausted. MCTS is a best first search variant. However, because it explicitly aims to balance *exploration* and *exploitation*, it has the problem that when search reaches a new node, all potential children have to be explored before that combination can be further exploited. For some problems this means that even getting down to a combination of size 3 could involve the generation of ≈ 900 conjectures which could take a considerable amount of time given that streamliner evaluation is not trivial. Also, for many of the problems there are multiple effective candidate streamliners and this means that MCTS has to balance its exploitation



(a)

Problem	Chuffed		Lingeling	
	Portfolio Size	Max Comb. Size	Portfolio Size	Max Comb. Size
BIBD	27	2	20	2
BACP	26	3	41	3
CarSequencing	45	3	64	3
FLECC	11	2	21	2
Transshipment	35	3	37	3
SocialGolfersProblem	12	1	14	2
CoveringArray	15	2	23	2

(b)

Figure 6.2: Portfolio sizes

amongst multiple viable paths in the lattice which ultimately affects the overall depth it can achieve.

It should not be seen as a failure of search however to have absent large combinations in the generated portfolio. A large combination is not necessarily more effective, it depends on the strictness of the composing conjectures. For instance, if the problem contains a decision variable of the type *Set (int 1. . .)*, a single conjecture enforcing that the set only contains *odd* numbers would be more restrictive than two combined conjectures enforcing that approximatelyHalf of the set contains *odd* numbers and approximatelyHalf of the set is from the upperHalf of the int domain. However the lack of combinations in some problems such as SocialGolfersProblem may be indicative that we need to inspect the type present within the problem to generate different types of conjectures or more fine grained conjectures that can be better combined.

6.3 Portfolio Analysis

In what follows we are going to analyze and discuss some of the interesting properties of the constructed portfolios and analyze the performance of the constituent conjectures.

6.3.1 BACP-Chuffed

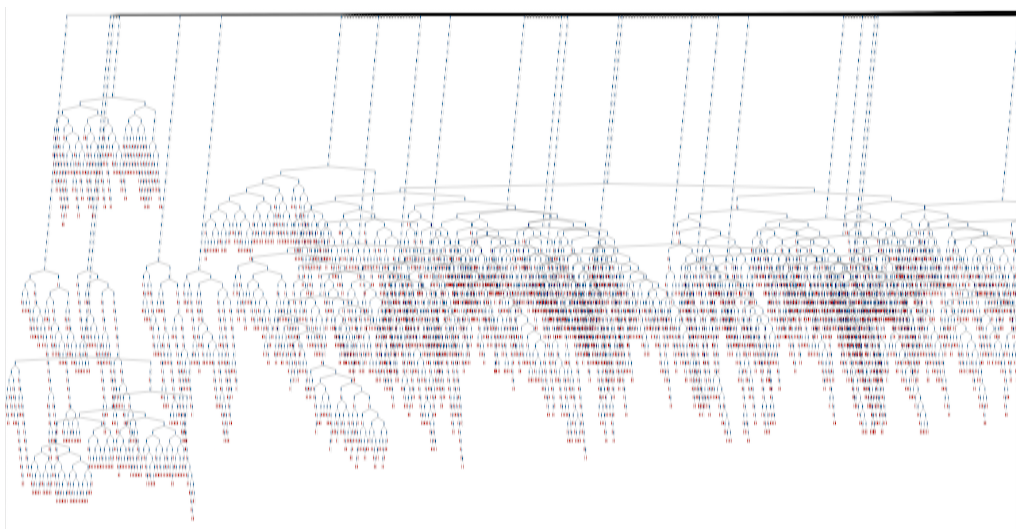
For *BACP-Chuffed* what is interesting about the constructed portfolio is the substantial reduction in search that some constituent conjectures can achieve. There are seven streamliners in the portfolio that can achieve greater than a 99.9% reduction in search on the training distribution. Streamliner *1* is an example that works on 26% of the training distribution with a mean reduction in search time of *99.89%* and a mean reduction in search nodes of *99.97%*. This streamliner enforces that the *curr* function which maps *courses* to *periods* is monotonicallyIncreasing, such that for the defined values of *curr*:

$$q_1 < q_2 \implies curr(q1) \leq curr(q2)$$
$$q1, q2 \leftarrow defined(curr)$$

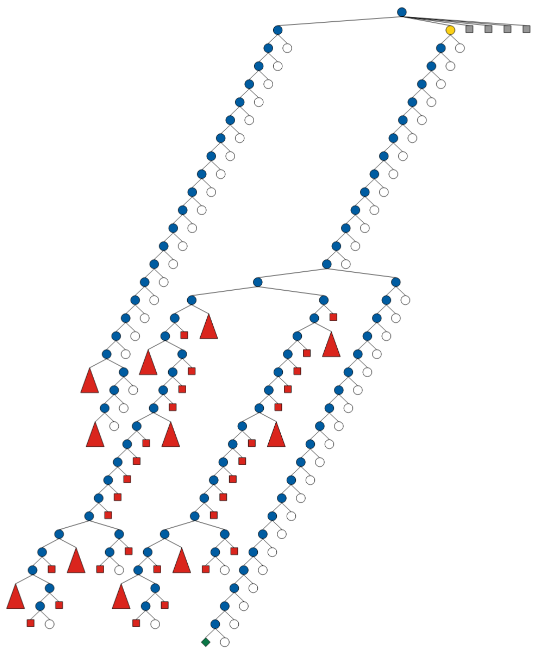
This streamliner often doesn't work however, being proven to be unsatisfiable on 74% of the training distribution but in some cases it can be an aggressive technique that reduces search substantially. Figure 6.3 provides a comparison between the search tree of an *unstreamlined* model (Figure 6.3a) vs a *streamlined* model (Figure 6.3b) on a representative training instance. The streamlined model on this instance is able to elicit a 99.97% reduction in search time and a 99.8% reduction in search nodes to solution. There is a stark difference between the search trees of the two models which really depicts the substantial impact that streamliners can have on the search procedure of the solver and the effort required to arrive at a solution. Without the focus provided by the streamliner the solver has to descend, backtrack and explore large swaths of the assignment space before a solution is found. In both models, the function decision variable is represented as a 1-D matrix of size int(1 . . . n_courses) indexed by int(1 . . . n_periods). Figure 6.3c shows the assignment to each search node on the path to solution for the streamlined model. An ascending variable ordering is being used by CHUFFED here. On each assignment to the matrix variable the presence of the streamliner constraint removes any values from subsequent variables that violate the *monotonicallyIncreasing* condition through the propagation mechanisms of the solver. In this case it helps prevent search from getting into unsatisfiable subtrees and allows the solver to almost walk to a solution with minimal effort.

6.3.2 CoveringArray-Chuffed

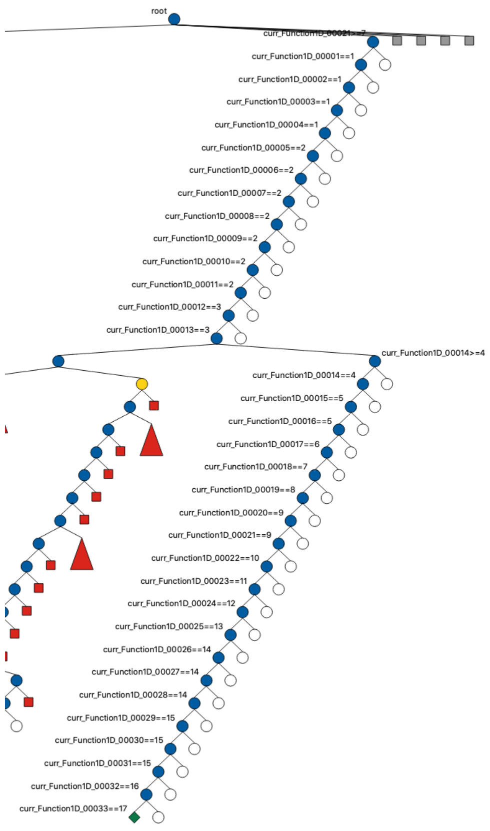
For *CoveringArray-Chuffed* what is interesting about the generated portfolio is that there are a number of conjectures that are able to achieve over 99% applicability across the Training Set, which equates to ≈ 151 instances out of the 153 evaluated. These same conjectures also achieve reductions in search time/nodes of $\approx 60\%$ and $\approx 80\%$ respectfully. One of the conjectures that performs well is Streamliner 7 which enforces that for all of the matrix rows, the values of exactly half of the columns must be in the lower half of the matrix domain. To understand why this streamliner is effective let us use an example instance with k = 20, v = 90 and t = 1. In the Covering Array problem the decision variable is represented as a 2-D matrix. This should be a trivial instance as the *strength* of the covering is only set to 1 meaning that the coverage of configurations only applies individually to each row. Any valid covering array must satisfy the t-covering property which states that when any t of the k rows are chosen, all v^t of the possible t-tuples must appear among the columns. For $t = 1$, a valid covering array would be one in which for each row all of the 90 possible configurations are listed. The problem is that the original model from CSPLib does not enforce that each configuration must appear at least once or limit the number of times a value can appear within a row in the matrix and because of this the Chuffed solver thrashes a lot during search. One of the common problems it faces is that it will assign a large chunk of the variables in one row the same values $v_0 = 1, v_1 = 1, v_2 = 1, v_3 = 1\dots$ and it takes a while before the constraints detect that this prevents all configurations from being assigned. This streamliner is effective as it helps mitigate this issue to some degree by forcing the solver to spread out the values equally amongst the lower and upper half of the domain.



(a) A slice of the search tree under the *unstreamlined* model. Overall solver statistics for CHUFFED are detailed: {solveTime: 30.854s, nodes: 130216, nogoods: 119015, backjumps: 3954, propagations: 3252465, restarts: 271 }



(b) The complete search tree under streamliner *1* which enforces the *curr* function to be *monotonicallyIncreasing*. {solveTime: 0.008, nodes: 232, nogoods: 161, backjumps: 17, propagations: 28181, restarts: 1}



(c) The assignment to each search node on the path to solution. During assignment the streamliner constraint through solver propagation removes domain values from subsequent nodes that would violate the *monotonicallyIncreasing* condition.

Figure 6.3: A comparison of the search trees in the CHUFFED solver between a streamlined and unstreamlined model representation.

6.3.3 Transshipment-Lingeling

In the last two sections we have looked at the portfolios generated for the CP domain, however it is very reassuring to see that our approach can also generate effective portfolios for the SAT domain. LINGELING, a SAT solver, takes a considerably different approach to solving problems than CHUFFED however streamliners can still be found that produce large reductions in search time and decisions. Figure 6.4 shows the performance of the constructed portfolio across the 96 instances that compose the training distribution. On the top of the x-axis the values detail the maximum reduction in solving time achieved by any of the constituent streamliners for each training instance. On the whole reductions are overwhelmingly positive with large reductions being found across the training distribution with the *min,max,mean* values at *64.2%, 98.2%, 81.5%* respectfully. Every instance from the distribution is also solved by at least one streamliner from the portfolio, there are no gaps in applicability.

It is interesting to see that some instances from the training distribution seem to be much more receptive to streamlining than others. For example, on instance 44 all streamliners from the constructed portfolio are *satisfiable* and almost all are incredibly effective with most attaining a reduction in solving time in the $\approx 90\%$ range. In contrast there are other instances for which only a few constituent streamliners work. Instance 94 provides such an example. Here, only 3 streamliners from the portfolio find a solution 53, 36 and 26. This shows the strength of the portfolio builder technique that the final portfolio contains these Pareto dominated streamliners because they maintain applicability on these difficult instances.

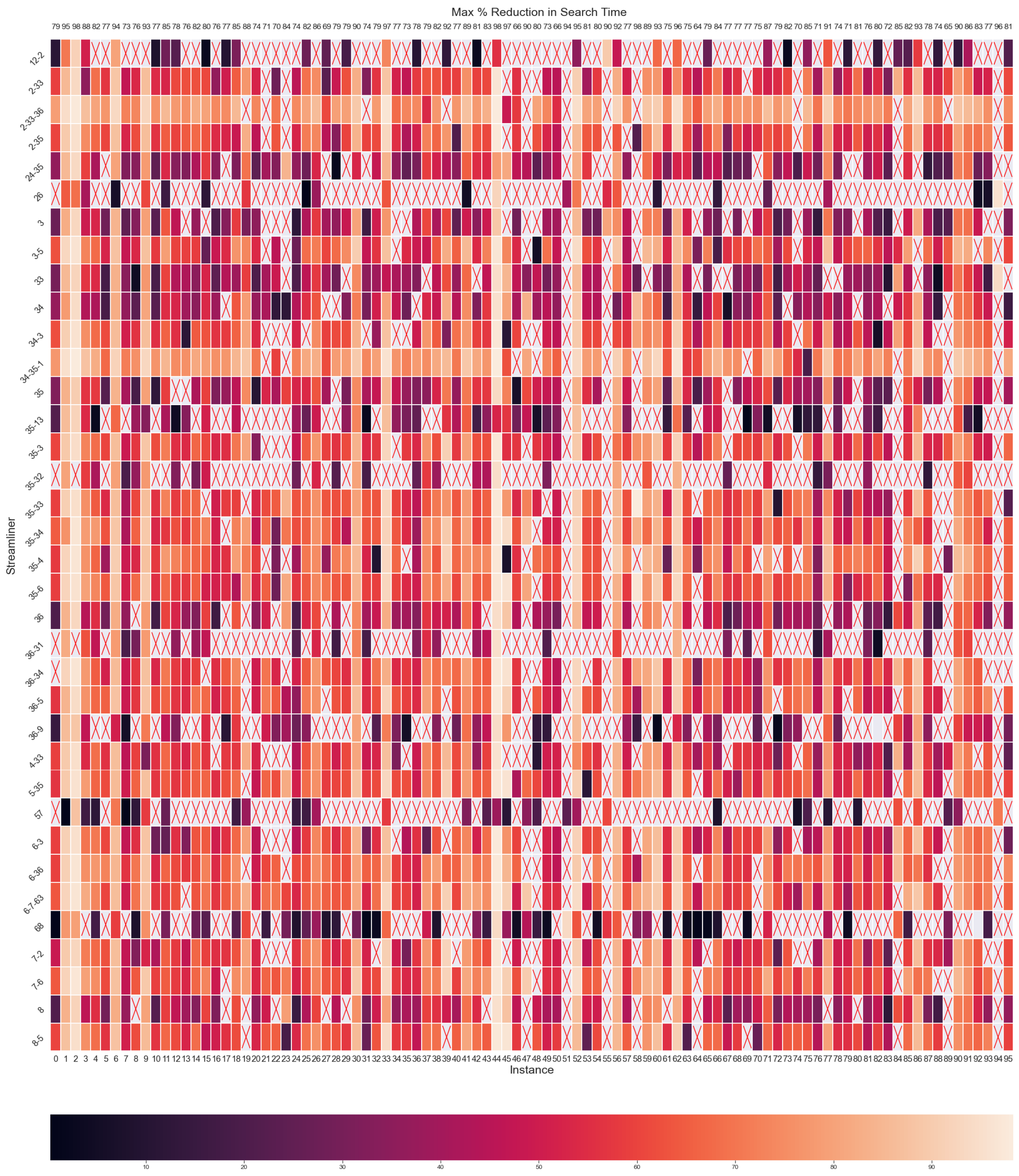


Figure 6.4: Performance of the constructed portfolio across the 96 instances that composed the Training distribution. The red cross indicates that the evaluated streamliner timed out or was proven to be unsatisfiable on the instance. The labels across the top of the x-axis detail the maximum reduction in solving time achieved by any of the streamliners in the portfolio.

6.4 Streamliner Selection

Having constructed a streamliner portfolio for a particular problem class using MOMCTS and the set of training instances, for a given test instance the question arises as to which streamlined models from the portfolio should be used, in what order, and according to what schedule. The reason a portfolio is constructed, which complicates this selection process, is that in general there is no one dominating streamliner for a problem. Larger more aggressive streamliner combinations will generally yield a higher reduction but as a result of their strictness they are very specific to a certain subset of the instance space and cannot generalize well. This creates the problem of for a particular test instance which algorithm to choose. It has long been observed in many domains such as SAT, CP, ASP where there are multiple high-performance algorithms that in general there is no one dominating algorithm and different algorithms will exhibit complementary solving strengths and dominate on different types of problem instances known as performance complementarity [116, 169, 88, 126, 131, 199, 157]. In general this gives rise to the problem of for a particular instance what is the best algorithm or algorithms to use to solve the instance. There are many closely related methods that aim to solve this same general problem such as *algorithm schedules* [111, 101, 11], *parallel algorithm portfolios* [88, 133] and *Algorithm Selection* [169, 123, 116]. In this work both *algorithm schedules* and *Algorithm Selection* methods are utilized and compared.

6.4.1 Single Best Solver

The most basic streamliner application approach, namely the SingleBestStreamliner (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 may not perform well on average across the instance space are neglected even if they may exhibit good performance on a subset of the instances.

6.4.2 Streamliner Scheduling

No Algorithm Selection system is perfect and will always make some degree of miss-classifications. This can happen for a variety of reasons such as uninformative instance features used for training or using a classification model that has over-fit the training data

Algorithm 2 Lexicographic Streamliner Selection

```
procedure SELECTION(Portfolio P, Ordering, $Time_{total}$, Instance)
    $P \leftarrow$ sort(P, by = Ordering)
    $Time_{Taken} \leftarrow 0$
    while $Time_{Taken} \leq Time_{Total}$ do
        Streamliner ← P.next()
        Stats ← Apply(Streamliner, Instance)
        if Stats→sat() then
            Return                                    ▷ Instance Solved
        end if
        $Time_{Taken} += $ Stats.time
    end while
end procedure
```

and that cannot generalize to unseen instances. An extension of algorithm selection is to select a schedule of multiple algorithms at least one of which performs well [135, 132]. The general idea is to have an ordered sequence of Algorithms which are run one after another each for a given budget in the hope that one of the Algorithms will be successful. The budgets can differ between Algorithms and are generally calculated based upon the performance on the training set. Utilizing a schedule of well-performing algorithms provides another way of exploiting performance complementarity as the idea is that the schedule will contain the dominating algorithm for the instance. In this work we focus on the use of static Algorithm Schedules which are instance agnostic and are applied uniformly across the instance space.

6.4.2.1 Lexicographic Selection Methods

It is possible to order the streamlined models in a portfolio lexicographically by, for example, prioritizing 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 as shown in Algorithm 2. The schedule is static in that it only moves to the next streamlined model when the search space of the current one is exhausted.

6.4.2.2 Dynamic Portfolio Filtering

When traversing a schedule it is possible to dynamically filter streamliners based upon prior results. If for a given instance we are evaluating the static 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.

6.4.3 Automated Algorithm Selection Methods

Automated Algorithm Selection (AS) techniques [169, 123, 116] utilize 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 [134, 201].

This is a very similar setting to our portfolio of streamliners, with complementary solving strengths and no single dominating streamliner exists. In this work we employ the algorithm selection system Autofolio [134]. 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* [134] 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 tools SMAC [106]. 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.

6.5 Summary

This chapter initially demonstrated the drastic differences in performance that can occur between identical streamliner conjectures in the SAT and CP paradigms and explained why it is necessary to perform paradigm specific search to generate independent streamliner portfolios. Analysis of the constructed portfolios was then performed, in particular the visualization of the distribution and size of the generated pareto fronts was presented. For a subset of the problems, a more in-depth analysis was then performed to highlight interesting properties of the constructed portfolios and analyze the performance of a sample of constituent conjectures. Lastly, the problem of *"Streamliner Selection"* was introduced and the application of various uninformed and informed methods for solving this problem were discussed.

Chapter Seven

Experimental Results

In the preceding chapters we have presented a completely automated approach to the generation and selection of streamliner constraints, hitherto a laborious manual task. In this chapter we present two sets of experiments to test the efficacy of this approach. The first is designed only to measure the frequency with which streamlining results in a reduction in search, and the magnitude of that reduction. Imperfect streamliner selection however means that streamlining is not always guaranteed to be effective and on certain instances negative reductions can occur. The second experiment aims to provide a more practical setting in which the overall impact of streamlining across an entire instance distribution is analyzed. Results across two unseen instance distributions with varying solving difficulty are shown.

7.1 Experimental Setup

All experiments were run on a cluster of 280 nodes, each 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 solving paradigm (CP/SAT).

The generated streamliner portfolios were evaluated on two different instance distributions, distinguished by the comprising instance difficulty. The first one, denoted *Distribution A*, consists of instances with similar difficulty to those used during the portfolio construction phase (Chapter 5), i.e. satisfiable instances with a solving time within $[10, 300]$ seconds by the original model. We use this to analyze the generalization 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 4.1), but drawn from a different distribution with a solving time limit of $(300, 3600]$ seconds (by the original model). *Distribution B* allows us to study the ability of the portfolio and streamliner scheduling/selection methods to generalise to instances of higher difficulty.

Each IRACE run of the instance generation process (for either instance distribution) was given a wall-time limit of 48 CPU hours on a cluster node with 36 parallel processes. As the instance distribution space is problem dependent and given a fixed budget the number of generated instances for *Distribution A* and *Distribution B* (listed in Table 7.1 & Table 7.2) varies. For the CP paradigm the learning solver CHUFFED [39] was used with its default parameter setting as the target solver. For the SAT paradigm the LINGELING [26] solver was used. In SAVILE ROW the default encodings for *SAT* are order encoding [188] for sums and support encoding [79] for binary constraints. SAVILE ROW was also used to produce the FlatZinc inputs for the `fzn2feat` feature extraction tool. These features are used by the Algorithm Selection method Autofolio for prediction as described in Section 6.4.3.

We report performance of the two simple streamliner scheduling approaches ApplicFirst and ReducFirst (Section 6.4.2.1), the Single Best Streamliner (SBS, Section 6.4.1), and the automated algorithm selection approach Autofolio (Section 6.4.3). We also report as a reference point the theoretically best performance, namely the *Oracle*, where we assume that the best solving model (either original or streamlined) for each instance is used.

7.1.1 Algorithm Selection Setup

In this work we are evaluating performance on two instance distributions with disjoint solving difficulties. For Algorithm Selection the simplest approach would be to generate a single model that predicts streamliners for instances from both distributions. A question then arises as to whether or not the performance of the constituent streamliners will vary across the distributions which could affect the validity of the predictions. Figure 7.1 compares on two different instance distributions with different solving difficulty the average speedup performance of the seven portfolios in both the SAT and CP paradigms. As can be seen there is a large amount of variation in performance with some streamliners attaining differing reductions in the two distributions. This variances complicates the problem of learning a single effective prediction model. As such we build two different prediction models for each distribution evaluated. The prediction model for *Distribution A* is learnt on the set of instances used for streamliner portfolio construction, detailed in Section 4.1, as these are of a similar solving difficulty. For *Distribution B* the prediction model is again built on a similar set of instances generated with a solving time in $[300, 3600]s$.

In order to evaluate Autofolio for each Solver/Problem variant the ASlib format [28], which defines a standard format for representing algorithm selection scenarios, was generated for each Training set. For each problem Autofolio was run ten times in tuning mode with a 1 day CPU budget to configure claspfolio2 for that particular domain. 10-Fold cross validation was used and the model that provided the best average cross-validation score was selected for prediction.

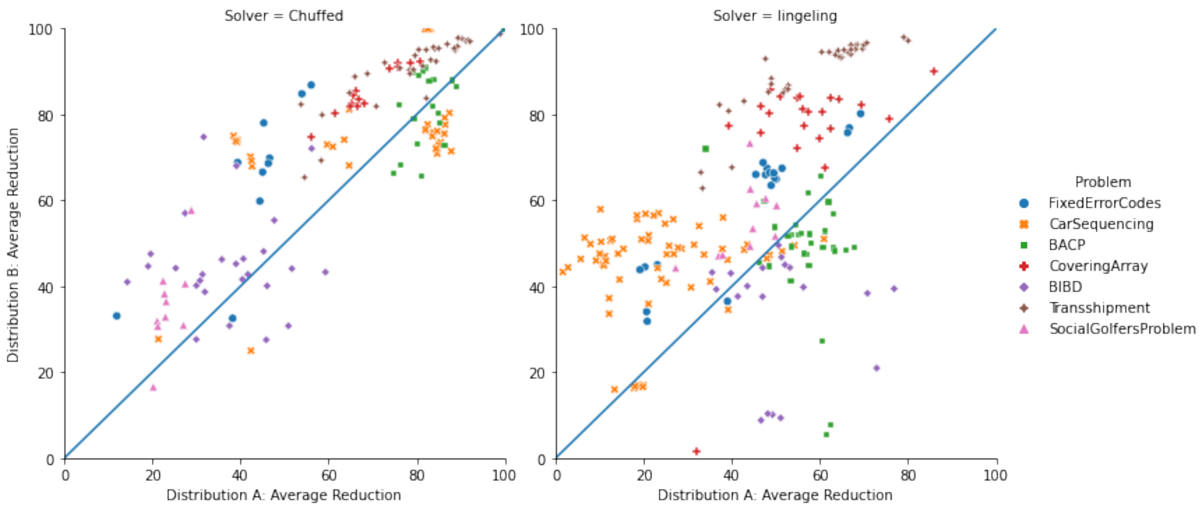


Figure 7.1: Average Percentage Reduction achieved by the streamliner portfolios on the Distributions {A & B} across the 7 problem classes

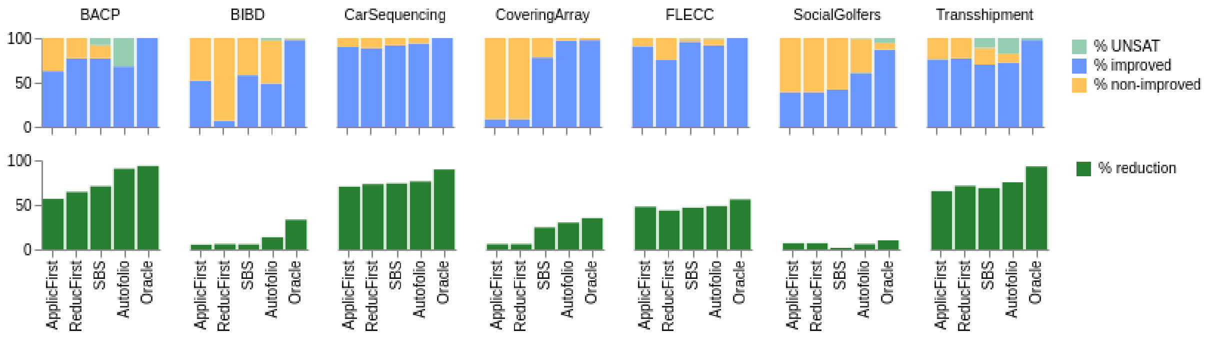
7.2 Frequency and Magnitude of Search Reduction

The results for the two distributions are presented in Figures 7.2 and 7.3. We begin by considering the setting where the instances are solved with the CP solver Chuffed. The very strong performance 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. For BACP, Car Sequencing, and Transshipment it is most pronounced, approaching one hundred percent, which would indicate a solution obtained without resorting to search. For others, like Social Golfers and BIBD, the reduction is less, and this will of course depend on the nature of the problem class and the streamliners we are able to generate for it.

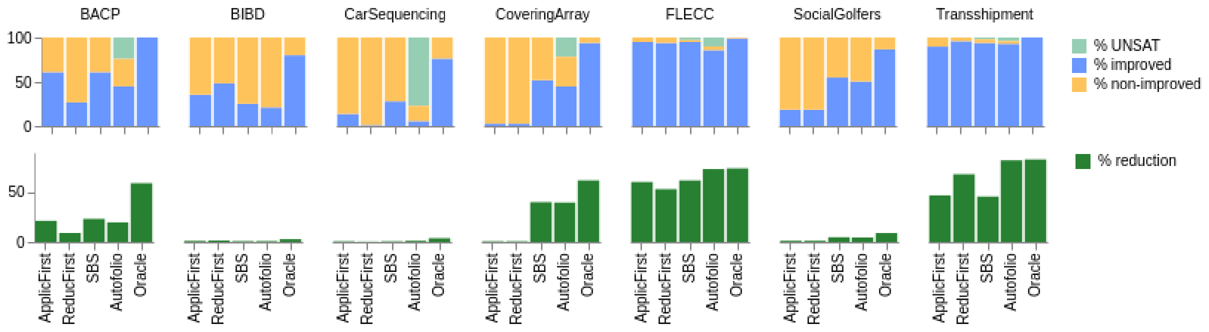
Performance across all problem classes significantly improves for Distribution B, suggesting that the impact of streamlining grows with the difficulty of the problem instance. This is as expected: the size of the search space typically increases with that of the instance, providing the opportunity for the selected streamliner to prune larger parts of the search space, and so reduce search further.

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 most robust performance in this respect comes from Autofolio. The simple lexicographic approaches sometimes perform well, but for some problem classes, such as Covering Array, their performance is relatively weak. The Single Best Streamliner approach, which requires no further training following the streamliner search, offers a good compromise between performance and cost before solving unseen instances. Where automated selection is selecting a streamliner that does not lead to an improvement, in most cases this is due to the search exceeding the time taken by the unstreamlined model, rather than the instance being rendered unsatisfiable. This suggests a bad interaction with the search strategy, which we will study in future.

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 three of the seven problem classes the magnitude of this reduction is small. On the remaining four problem classes, performance is stronger and in some cases exceeds the improvement delivered in the corresponding CP setting. Once again performance clearly improves on Distribution B relative to Distribution A, suggesting that for SAT also the impact of streamlining grows with difficulty.

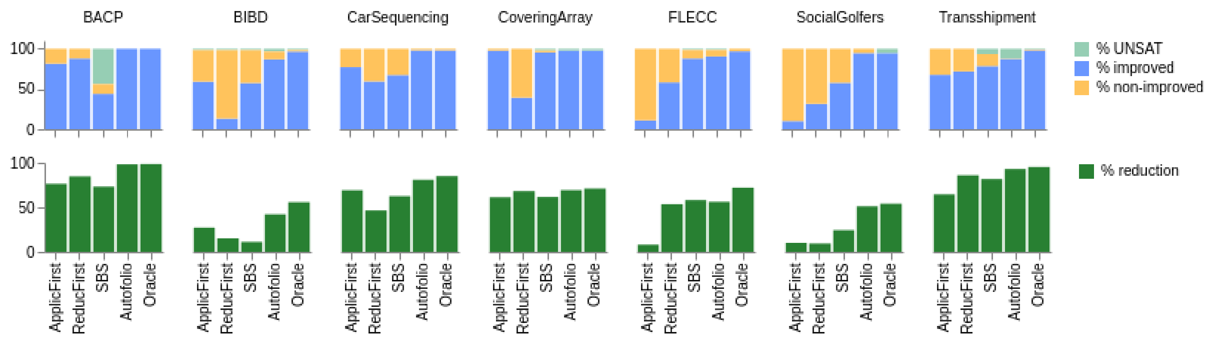


(a) Chuffed on *Distribution A*.

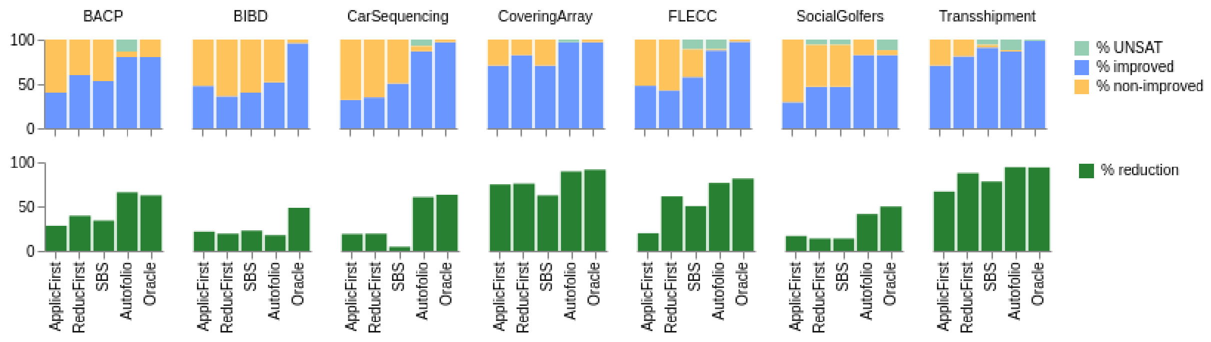


(b) Lingeling on *Distribution A*.

Figure 7.2: 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 *original* 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. The best approaches are both frequently applicable and result in a large search reduction.



(a) CHUFFED on *Distribution B*



(b) LINGELING on *Distribution B*

Figure 7.3: Results with CHUFFED and LINGELING on *Distribution B*. Detailed meaning of the plots are described in Figure 7.2.

7.3 A Practical Setting

By employing the algorithm selection techniques described in Section 6.4 our aim is to maximise the occasions on which streamlining produces a reduction in search effort. In the previous section we restricted ourselves to only looking at such occasions. However to a practitioner who is leveraging this method the alternative cases where the streamliner failed to be effective are also important. 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, or the streamliner may be ineffective on the given instance. Even though streamlined models operate on a subspace of the *original* model it is not guaranteed that they will finish faster. Figure 7.4 combines the conjectures from all constructed portfolios and displays their performance across three different instance distributions. Performance is represented on the y-axis as the speedup in solving time relative to the original model defined as:

$$Speedup = (Original\_SolvingTime)/(Streamlined\_TotalSolvingTime)$$

Note because some evaluations time out or are proven unsatisfiable in these cases the diagram is comparing the time taken for the original model to find a solution vs the total time taken by the streamlined model. The red dotted line separates the cases: any entry above represents an improvement (a positive speedup) and any value below represents a negative improvement (a negative speedup) in solving time relative to the *original model*. The vast majority of these evaluations are ideal from the perspective of the practitioner; a solution is found in the streamlined subspace and a positive speedup in solving time is achieved on the instance.

What is interesting however is that not all evaluations produce a positive speedup, there are many cases where the application of a streamliner actually increases the time taken. This raises the practical issue of trying to define the amount of time that should be allocated to the streamliner{s} in the schedule when solving an unknown instance. The most basic method would be to apply the streamliners composing the schedule until either a solution is found or each streamliner has completed. As no schedule or AS prediction is going to be perfect and will always make some degree of mis-classifications this basic method however has a very real danger for some instances of actually increasing the time to find a solution substantially relative to the *original* model. In this section we discuss different scheduling procedures that attempt to balance achieving substantial performance

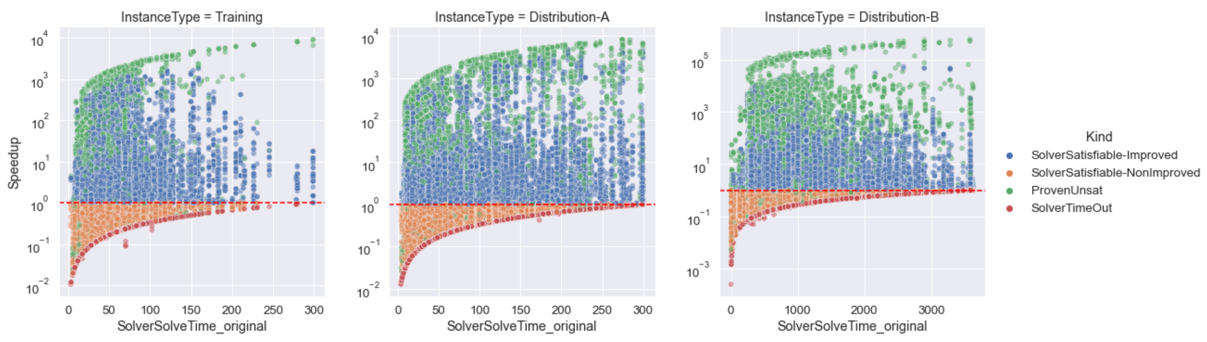


Figure 7.4: The diagram groups the evaluations into four different categories. *SolverTimeOut* represents the cases where the solver timed out during evaluation. The *timeout* value for the Training Set and Distribution A is set at 5 minutes and for Distribution B at 1 hour. *ProvenUnsat* represents the cases where the streamlined subspace was proven to contain no solutions in the given time budget. The last two categories, {*SolverSatisfiable-Improved*, *SolverSatisfiable-NonImproved*} represent the cases where a solution was found in the streamlined subspace. What differentiates them is whether or not the time taken to find this solution was an improvement over the *original model*

whilst restricting the worst case possible reduction.

7.3.1 Streamliner Scheduling with fixed cutoff time

In order to restrict the worst possible reduction one approach is to give a fixed cutoff to the evaluation of the streamliner{s} before reverting back to running the *original* model. The benefit of this fixed cutoff is that it will still allow streamliners that achieve large reductions to be effective whilst capping those that perform poorly. There is however no easy way to decide what value should be used for that cutoff as for an unseen instance we do not know its difficulty. If the instance is difficult and the allotted cutoff time is too short then large potential reductions could be missed. For example, if the instance takes an hour to solve under the *original* model and the streamliner achieves a large reduction of 50% with a cutoff less than 30 minutes this reduction will be missed. On the other hand if the cutoff time is too high then the runtime of ineffective or unsatisfiable streamliners can exceed that of the *original* model producing large negative reductions.

There are two ideal solutions to this problem. Firstly, if there existed an oracle that could predict precisely the amount of time the original model would take on an instance this would allow us to cap the amount of time the streamliner portfolio would run for and as a result cap the maximum negative reduction. Unfortunately, computationally cheap, perfect oracles of this nature are not available for any NP-complete problem and we cannot precisely determine an arbitrary algorithm's runtime on an arbitrary instance without actually running it [200]. A second approach would be to try and predict the satisfiability of a streamlined model before running it. If the prediction was accurate we would be able to remove any streamliners in the portfolio that render the current instance unsatisfiable and as such avoid the potential for negative reductions. However again there is currently no way to accurately predict the satisfiability for any NP-complete problem. Xu et al [197] using deep learning were able to achieve high prediction accuracy of satisifiability but only on boolean binary constraint satisfaction problems.

Figure 7.5 displays the distribution of both positive and negative reductions as a factor of the cut-off time for instances from Distribution B. It can be seen that as the cutoff time is increased the spread of the distribution also largely increases. For a small cutoff time of 30s there is a dense cluster of reductions centered close to 0% reduction and another small cluster centered close to the 100% reduction mark. This occurs as in most cases the cutoff time is too short to allow a selected streamliner to solve the instance and as such the time spent on streamliner evaluation results in a small negative reduction. In some cases however the streamliner can attain a very high reduction in search near 100%. As the cutoff time is increased the distribution starts to spread out and the probability of receiving larger negative reductions is increased. For 3600s for instance, negative reductions of up to -1000% are found. These large negative reductions can occur because as we have seen in Figure 7.4 search can take considerably longer in a streamlined subspace than the *original* model. The probability of these large negative reductions is often very low with respect to the entire instance space however when analyzing the overall impact of streamlining their presence can mask the positive reductions attained.

7.3.2 Portfolio Approach

Another approach is one 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 *original* model. This helps in two ways. Firstly if the streamliner selected renders an instance unsatisfiable the *original* model will produce a solution. Secondly if the selected streamliner is ineffective it cannot exceed the wall time of the *original* model and in terms of CPU time the worst case negative speedup is capped at 0.5.

We evaluated this parallel configuration on both Distribution A and B. Our results are summarised in Tables 7.1 and 7.2, which present the *Overall Speedup* of each approach

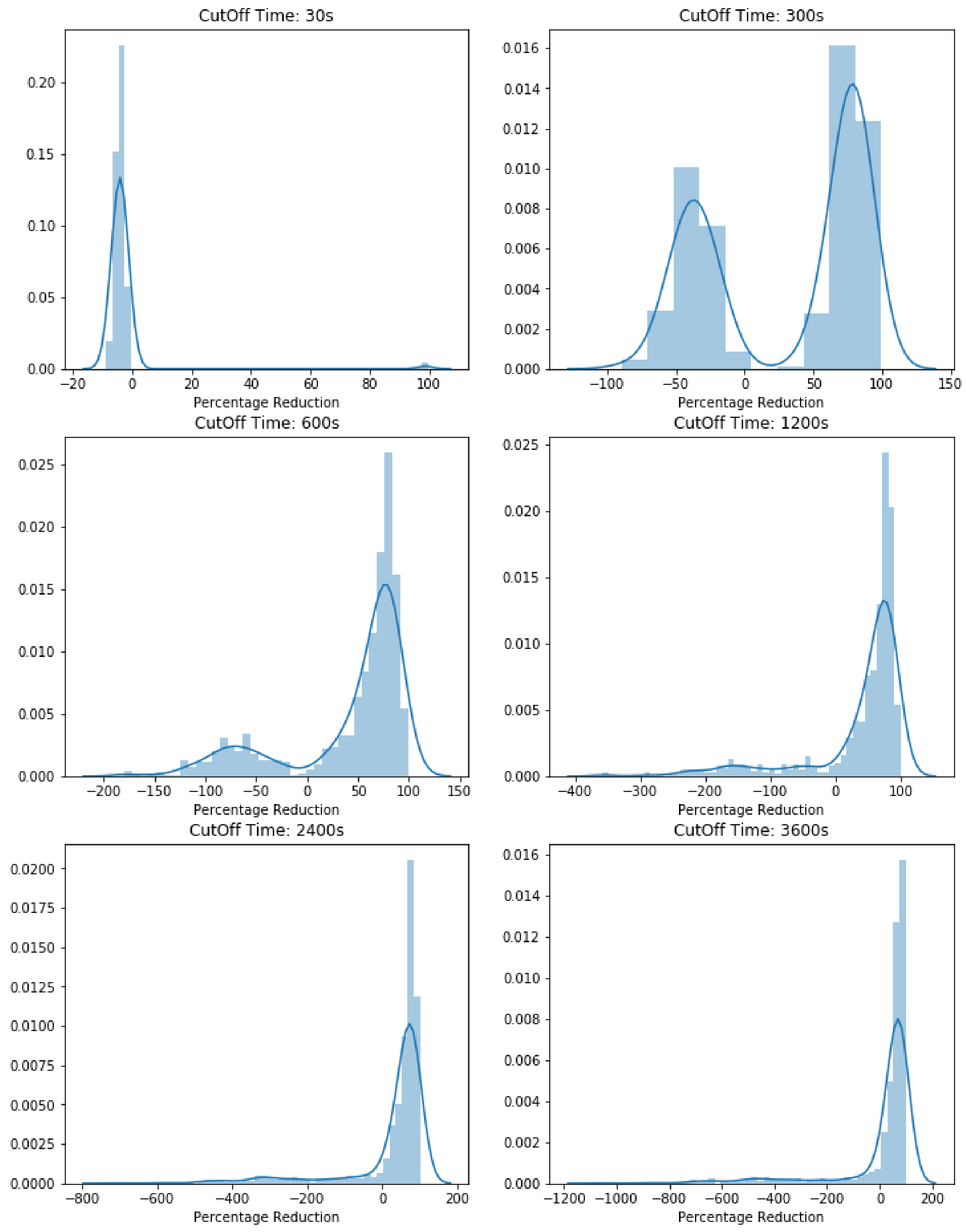


Figure 7.5: For the Fixed Length Error Correcting codes problem under the Chuffed solver: The distribution of reductions along with the Kernel Density Estimation [118, 29, 190] of the probability density function is shown as the fixed cutoff time is varied.

across the seven 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 1.9× and 1.57× for Distribution A and 4.72× and 3.53× 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 7.2, there is a pronounced increase in the speedups achieved for the more difficult *Distribution B* instances.

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**
	FLECC	100	2.42	**1.90**	1.88	1.71	**1.90**
	SocialGolfersProblem	100	1.12	1.02	1.07	1.07	**1.09**
	Transshipment	100	9.19	2.44	2.58	**2.63**	2.17
Geometric-Mean Speedup			4.12	1.86	1.68	1.76	**1.90**
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
	FLECC	100	4.06	2.72	2.79	2.40	**3.08**
	SocialGolfersProblem	91	1.10	**1.05**	1.01	1.01	1.04
	Transshipment	99	6.13	1.84	1.95	3.20	**4.15**
Geometric-Mean Speedup			2.31	1.41	1.33	1.38	**1.57**

Table 7.1: 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	16	53.47	1.47	3.98	5.99	**46.56**
	BIBD	59	2.25	1.13	1.32	1.09	**1.71**
	CarSequencing	52	8.77	1.91	2.75	1.88	**6.77**
	CoveringArray	46	3.36	2.20	2.42	1.41	**3.20**
	FLECC	192	3.95	2.18	1.04	1.63	**2.24**
	SocialGolfersProblem	19	2.53	1.28	1.04	1.05	**2.53**
	Transshipment	216	16.21	2.77	1.89	2.79	**5.39**
Geometric-Mean Speedup			6.65	1.77	1.84	1.88	**4.72**
Lingeling	BACP	15	5.92	2.20	1.52	1.66	**4.91**
	BIBD	25	2.26	1.25	1.19	1.21	**1.30**
	CarSequencing	69	3.32	1.06	1.16	1.16	**2.95**
	CoveringArray	34	16.65	2.19	2.10	3.14	**10.81**
	FLECC	166	5.89	1.62	1.18	1.51	**3.39**
	SocialGolfersProblem	17	2.23	1.14	1.20	1.14	**1.89**
	Transshipment	68	12.42	3.59	2.13	4.32	**5.25**
Geometric-Mean Speedup			5.32	1.71	1.45	1.77	**3.53**

Table 7.2: 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.

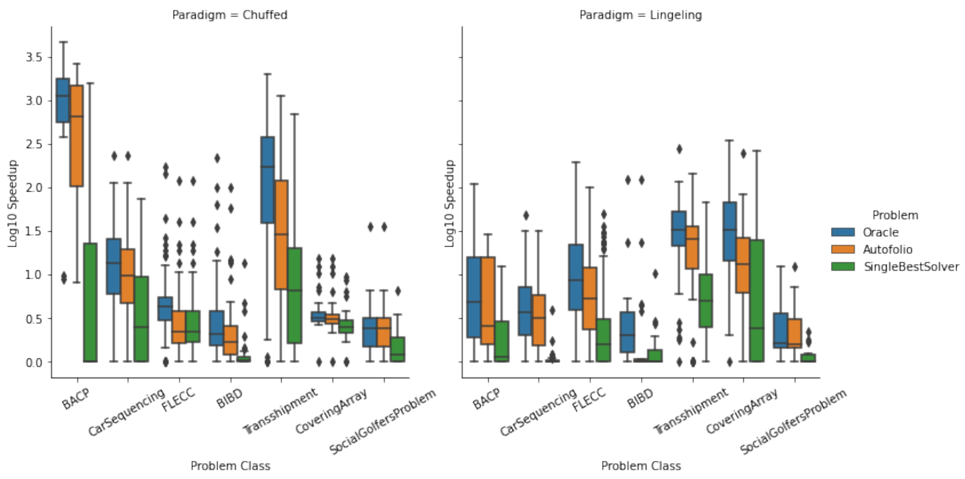
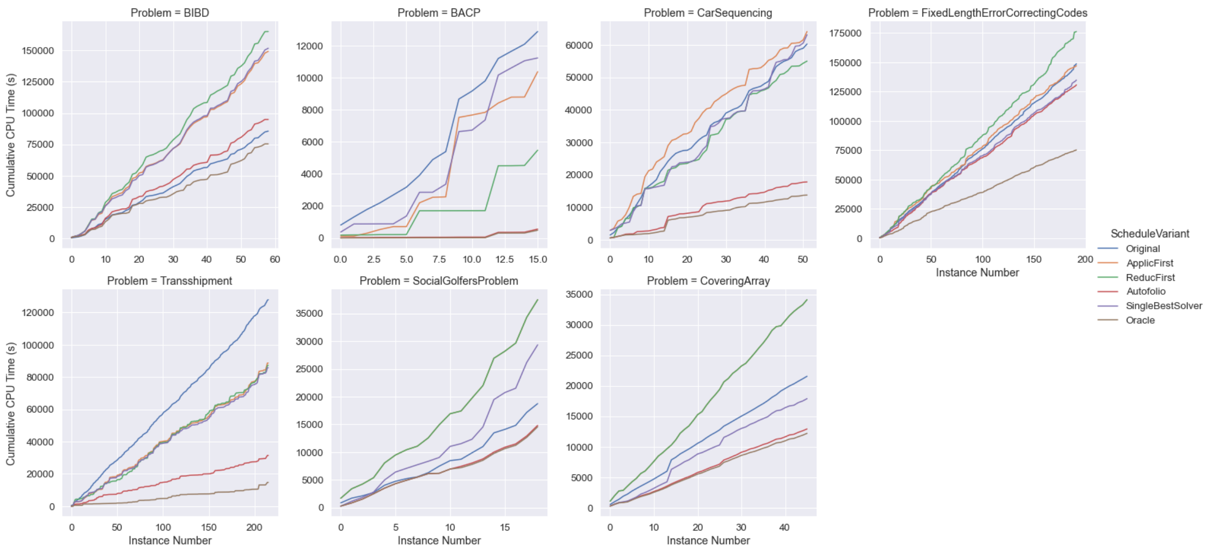


Figure 7.6: Distribution of speedup values (in base 10 logarithmic scale) for the Oracle, Autofolio and Single Best Solver on *Distribution B*

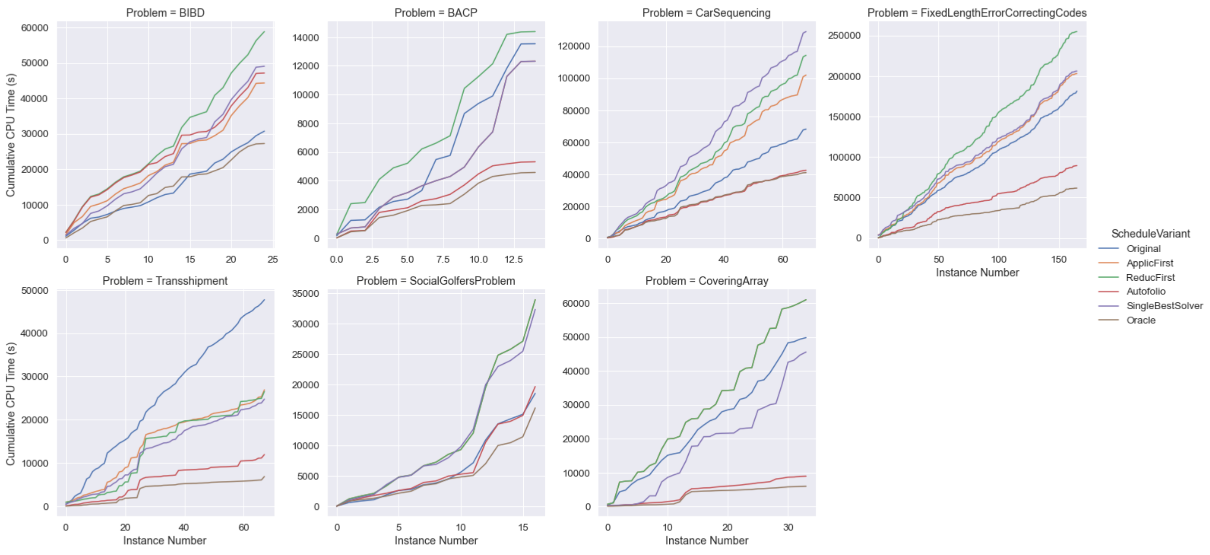
Overall Speedup is quite a coarse metric, which can obscure the individual instance speedups being obtained. For example, if a streamliner is evaluated on 10 instances and for half its application makes them trivial reducing the solving time to near zero but for the other half it is unsatisfiable the *Overall Speedup* will be $\approx 2\times$. This does not provide a good indication that on half of the instance distribution the streamliner is restricting the search space to such a high degree. In order to better visualize this Figure 7.6 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 decimate the search space and 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 $\{q_1 \approx 6\times, q_3 \approx 25\times\}$.

We must be mindful that all results presented thus far in this section are in terms of the reduction/speedup in *wall-time* of the portfolio approach vs the *original* model. Since the portfolio method utilizes two cores, one for the *original* model and one for the portfolio a possible critique may be this is an unfair comparison as we are consuming more resources to arrive at the solution. To combat this Figure 7.7 displays the cumulative *cpu time* of the scheduling methods across the instances comprising Distribution B. Here the total CPU 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 6 of the problems, BIBD being the exception, Autofolio is still able to reduce the overall time with in some cases a substantial reduction. BACP, CarSequencing and Transshipment experience speedups of $\approx 23x$, $\approx 3.38x$ and $\approx 4.07x$ approximately. For LINGELING, on every problem except BIBD and SocialGolfers positive speedups in overall time are achieved again with substantial results for some problems. For Transshipment, CoveringArray and BACP speedups of $\approx 5.56x$, $\approx 3.99x$ and $\approx 2.54x$ are achieved.

It is useful to note that the *cpu time* speedup of the portfolio approach is not always half that of the *wall time* speedup. The reason for this is that the portfolio approach does not always use twice the cpu resource of the *original* model. Let us use an example to illustrate this point. We have an instance *I_0* which takes 100s under the *original* model. If during evaluation the selected streamliner (*S_0*) is proven to be *unsatisfiable* at $T_0 = 50s$ then from *50s* $\rightarrow$ *100s* only one core will utilized running the *original* model. The *cpu time* speedup of the portfolio approach is then $\frac{2}{3}$ compared to a speedup of 1.0 for the *wall time* approach.



(a) Chuffed



(b) Lingeling

Figure 7.7: Across the 7 problems the cumulative CPU time of the different scheduling variants is shown for instance Distribution B for the Chuffed a) and Lingeling b) solvers

7.4 Results: Further Analysis

In this section we discuss and analyse in further detail the results presented thus far. Firstly, we look more closely into various factors that impact performance when streamliners are added into a model specification. Those include: (i) the reduction in number of search nodes required to reach a satisfiable solution and the change in solver performance in terms of number of search nodes (CP) or number of decisions (SAT) made per second

(Section 7.4.1); and (ii) the various optimisation strategies provided by Savile Row during its automated reformulation process (Section 7.4.2) and its cost. Those factors can lead to either substantial speedups or decreases in performance depending on certain situations. Secondly, we provide more insights into the behaviours of all streamliner portfolio application methods under study (Section 7.4.4 and Section 7.4.5).

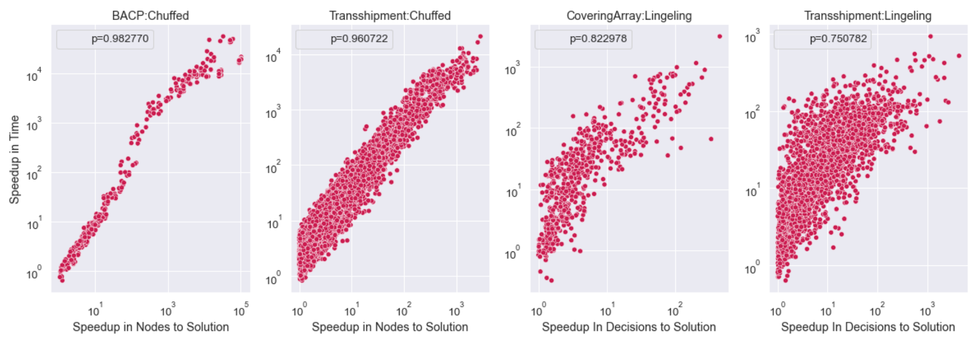
7.4.1 Node Reduction and Solver Performance

The substantial speedup in solving time enabled by some streamliners can be explained by the large reduction in the number of search nodes (CP) or decisions (SAT) the solver has to traverse before reaching a solution. Streamliners help to partition the search space into two disjoint sub-problems where the streamlined subspace is generally much smaller than its complement [84], and if a solution for a decision problem exists in the streamlined subspace, the considered instance can be solved efficiently by the solver. Figure 7.8a shows the number of search nodes/decisions to a solution (for instances from *Distribution B*) on a sample set of streamliners where large speedups in solving time are obtained. In all four problems, there is a strong correlation between the reduction in search nodes/decisions and the corresponding speedup in solving time. For some cases such as BACP (where the largest speedups are found), streamliners can turn a difficult instance taking millions of search nodes to a trivial instance taking less than 1000 nodes to solve.

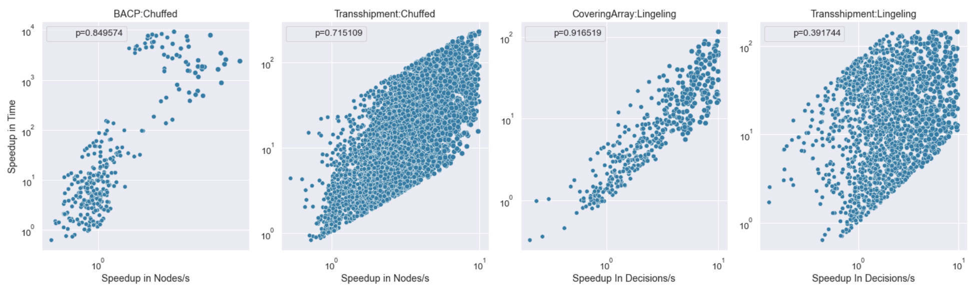
Reducing the number of nodes/decision required to reach a solution is not the only way that streamliners achieve their speedups. In various cases, the addition of streamliner constraints into a problem specification may actually increases a solver's performance: the number of nodes (for CP) or decisions (for SAT) the solver can traverse per second. As shown in Figure 7.8b, there were a large number of cases where the solver's performance on the streamlined model was faster than on the original model. In particular, for Transshipment-Chuffed and CoveringArray-Lingeling, increases in solver performance up to 10× were often observed. Such increases were possible thanks to a reduction in model complexity introduced during the formulation process by Savile Row, which will be discussed in further detail in the next section.

7.4.2 SavileRow Formulation

The presence of streamlined conjectures in the Essence Prime can impact the formulation of the solver model. The most profound impact is on the size of the generated solver



(a) For four different problem/solver settings a comparison of the speedup in time to arrive at a solution (y-axis) vs the speedup in nodes to solution (x-axis) under a streamlined representation.



(b) For the same four settings as fig. 7.8a the speedup in time to arrive at a solution (y-axis) vs the speedup in nodes/decisions per second completed by the solver (x-axis) under a streamlined representation.

Figure 7.8

dependent model produced by Savile Row. Figure 7.9 presents the effect of a portfolio of different streamliners in the *SAT* paradigm on the size of the resulting dimacs representation across three problems classses. The impact the streamliners impart is very problem dependent. For two of the problems listed, BIBD and CarSequencing, the streamliners drastically increase the number of clauses and variables and in turn the complexity of the model. For BIBD the delta is $\approx 8\times$ increase in the number of clauses and a $\approx 1.5\times$ increase in the number of variables. In Savile Row the default encodings for *SAT* are order encoding [188] for sums and support encoding [79] for binary constraints. A majority of the generated streamliners for Car Sequencing and BIBD contain sum expressions which explode when formulated in the dimacs format.

The opposite effect however was seen on Transshipment which is initially surprising. It might be expected that as streamliner constraints are additional constraints added onto an Essence Prime model that they would always increase the complexity of the generated solver-dependent model. Having a closer look into the Savile Row formulation process, we find out that there are two optimisation strategies in Savile Row, namely *domain*

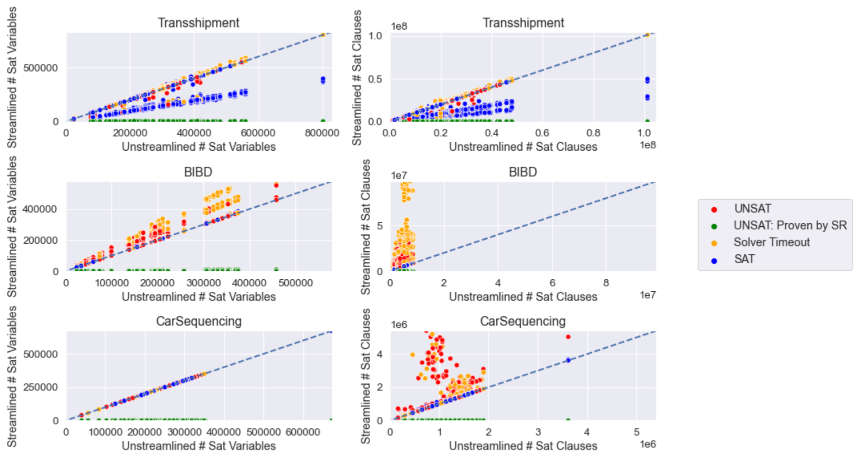


Figure 7.9: Effect of Streamliners on #Variables/#Clauses in a SAT model

filtering and *identical common sub expressions (cse)*, that contribute to the simplification of the generated output.

The first strategy, domain filtering, removes values that cannot take part in any solution. It is implemented in Savile Row by using the standard propagation of the CP solver Minion in combination with single consistency applied to the upper and lower bound of variables' domains. In some cases, domain filtering can help Savile Row to quickly prove that a streamlined model is unsatisfiable on an instance without even fully formulating the model or invoking the solver. Those cases are represented by the green dots in Figure 7.9. A trivial example of how this could occur is presented in Figure 7.10

```
find x : int(1..100)
such that
        x % 2 == 0,
        $ Added streamliner
        x % 2 == 1
```

Figure 7.10: By forcing x to both be simultaneously *odd* and *even* this model becomes trivially unsatisfiable and through the application of domain filtering Savile Row will be able to prove that the plausible domain for x is empty.

The second strategy, identical-cse, assigns the same auxiliary variable for all syntactically identical sub-expressions that need to be flattened. For Transshipment, the addition of streamliners helps to increase the amount of domain filtering and identical-cse that occurs, resulting in simpler and smaller solver input sizes.

7.4.3 Savile Row Formulation Time

The Savile Row formulation process is not trivial and can take considerable time and resources depending on both the complexity of the Essence Prime specification and the given instance. Unlike Conjure, Savile Row has to be invoked for every application of a streamliner on a given problem instance. This means that any overhead a streamliner imparts during formulation needs to be taken into account when evaluating performance on an instance. Figure 7.11 shows that streamliners can also have a profound impact on the time taken by Savile Row to formulate the solver output with respect to the *original* model. There is a high correlation between the delta in Savile Row formulation time and the delta in complexity of the streamlined and original solver-dependent models. In the cases where the complexity of the output model drastically increases this is generally associated with an increase in formulation time and vice versa.

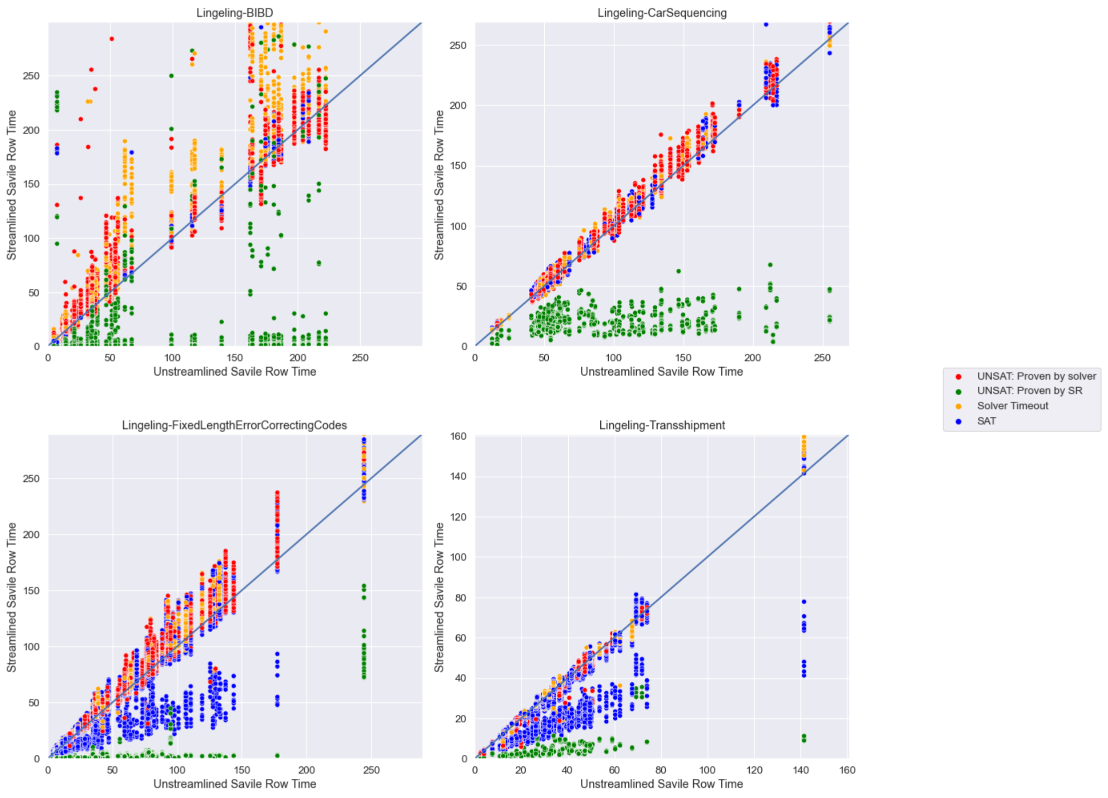


Figure 7.11: Effect of Streamliners on Savile Row formulation time

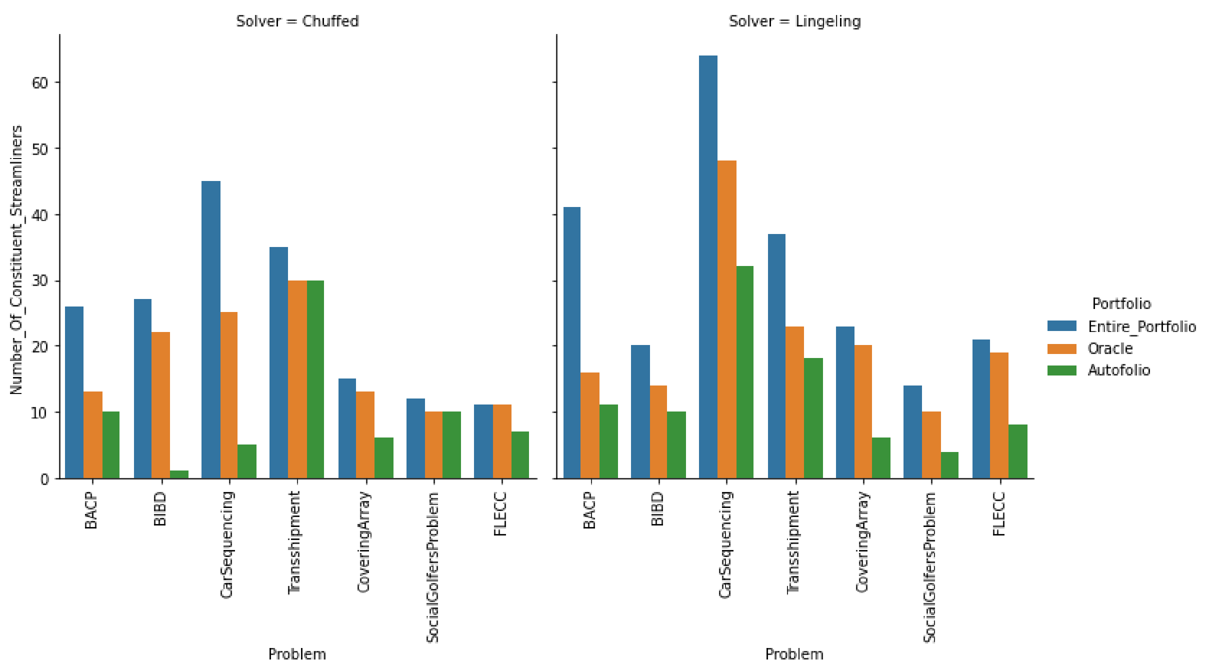


Figure 7.12: The size of the constructed portfolios and the number of streamliners used by the Oracle and Autofolio for instances from Distrbution B.

7.4.4 Instance-oblivious Streamliner Application

The simplest streamliner application method on an unseen problem instance is to use the Single Best Streamliner (SBS), i.e., the streamliner with the lowest average solving time across all training instances. The SBS shows competitive performance on the easy instances (Distribution A, Table 7.1). However, on the more difficult instances (Distribution B, Table 7.2), SBS is no longer performing well compared to the instance-specific approach based on Autofolio. It indicates that for the difficult instances, performance of the streamliners in the constructed portfolio varies among instances and there is no single best streamliner that works well on all instances. This is illustrated in Figure 7.12 where we show the number of streamliners used by the Oracle for all instances from Distribution B. In all cases the Oracle is composed of at least 10 streamliners, showing that to achieve the performance of the Oracle a substantial part of the portfolio needs to be used.

Other instance-oblivious streamliner application methods used in this work include ApplicFirst and ReducFirst, the two lexicographic schedules built based on prioritising streamliners with higher applicability (ApplicFirst) and solving time reduction (ReducFirst) across all training instances. ApplicFirst is a conservative approach since it always starts with the "safest" streamliners from the portfolio, i.e., the ones that are less restrictive to the solution space and therefore more likely to provide a solution given enough time.

Those streamliners are mostly composed of only one or a few candidate streamliners. ReducFirst, on the other hand, starts with the most restrictive streamliners from the portfolio, which can offer great reduction but are also more likely unsatisfiable. The key to whether or not ReducFirst manages to work more effectively compared to ApplicFirst is the speed at which it can dismiss unsatisfiable streamliners from the schedule. As an example, for BACP-Chuffed on Distribution B where it manages to achieve a speedup range with the 1^{st} and 3^{rd} quartiles at $\approx 68\times$ and $\approx 1564\times$, respectively, on average 5 streamliners are applied before either the time budget is exhausted or a solution is found. This is in contrast to BIBD-Lingeling where only marginal speedups are attained, on average only 2 streamliners from the schedule are evaluated.

7.4.5 Instance-specific Streamliner Application

Autofolio is considered an instance-specific approach since it takes into account instance features to decide which streamliner(s) to apply on an unseen instance. Compared with instance-oblivious approaches, Autofolio is very competitive on Distribution A and dominates on Distribution B. However, for some problems such as Transshipment, the fairly large gap with the Oracle indicates that there is still quite some room for improvement. This is expected, as algorithm selection has long-known to be a difficult problem and has been a research focus for several years [116]. Figure 7.12 shows the number of streamliners used by AutoFolio compared to the Oracle. Interestingly, for some cases such as CoveringArray-Chuffed, SocialGolfersProblem-Chuffed and CarSequencing-Lingeling, Autofolio only needs a small subset of the streamliner portfolio to almost reach the ideal performance of the Oracle.

7.5 Summary

This chapter demonstrated two sets of experiments that tested the efficacy of our automated approach to the generation and selection of streamliner constraints, across seven problem classes in both SAT and CP. The first examined the frequency for which streamlining resulted in a reduction in search, and the magnitude of that reduction. One deficiency of this experiment was that it only examined cases where the chosen streamliner was effective. The second experiment displayed a more practical setting and analyzed the overall impact of streamlining across an entire instance distribution. Overall positive results in both experiments for a majority of the problems evaluated were shown.

Chapter Eight

Streamlining for Constrained Optimization

Up to this point the use of Streamlined Constraint Reasoning has solely been reserved for Constraint Satisfaction problems where all solutions are of equal value. A great many problems however can be defined as Constrained Optimization Problems (COP) where aside from finding a solution to the problem there is a secondary goal of maximizing or minimizing a specific objective. In this chapter we discuss the process of streamlining Constrained Optimization Problems and the required adaptations to our base search and scheduling methods. Empirical results shown demonstrate drastic improvements in both time to optimality and time to proof for all problems evaluated.

8.1 Problems

We now introduce the three problem classes studied in this chapter, which will be used both to illustrate the remainder of our method and for our empirical evaluation. We selected these problems, presented in Figure 8.1 to give good coverage of the abstract domains available in ESSENCE, such as set, multi-set and function. Furthermore, SONET and Progressive Party have nested domains: multi-set of set and set of function respectively.

The Progressive Party Problem defines the problem of trying to timetable a party at a yacht club. Certain boats are designated hosts, and the crews of the remaining boats in turn visit the host boats for defined periods. The crew of a host boat remains on board to act as hosts while the crew of a guest boat together visits several hosts. Every boat can only hold a limited number of people at a time (its capacity) and crew sizes are different. The total number of people aboard a boat, including the host crew and guest crews, must not exceed the capacity. A table with boat capacities and crew sizes can be found below; there were six time periods. A guest boat cannot not revisit a host and guest crews cannot meet more than once. The problem facing the rally organizer is that of minimizing the number of host boats [192].

The Minimum Energy Broadcast (MEB) problem aims to configure the power level in each device of an ad hoc network such that if a specified source device broadcasts a message it will reach every other device either directly or by being retransmitted by an intermediate device (a broadcast tree is formed). The desired configuration is that which minimises the total energy required by all devices, thus increasing the lifetime of the network [33].

In the SONET problem a number of rings and nodes are defined. For each pair of nodes a demand is given (which is the number of channels required to carry network traffic between the two nodes), which may be zero. A node is installed on a ring using a piece of equipment called an add-drop multiplexer (ADM). Each node may be installed on more than one ring. Network traffic can be transmitted from one node to another only if they are both installed on the same ring. Each ring has an upper limit on the number of nodes, and a limit on the number of channels. The demand of a pair of nodes may be split between multiple rings. The objective is to minimise the total number of ADMs used while satisfying all demands [151].

```
$ SONET
given nnodes, nrings, capacity : int(1..)
letting Nodes be domain int(1..nnodes)
given demand : set of set (size 2) of Nodes

find network : mset (size nrings) of
            set (maxSize capacity) of Nodes
minimising sum ring in network . |ring|,
such that forAll pair in demand .
  exists ring in network . pair subsetEq ring

$ Minimum Energy Broadcast
letting dNodes  be domain int(1..n_nodes)
letting dDepths be domain int(1..n_nodes)
find parents: function (total)
                dNodes --> dNodes,
     depths : function (total)
                dNodes --> dDepths
find optVar : int(0..numberNodes * max([cost | (_,cost) <- linkCosts]))
minimising optVar


$ Progressive Party
letting Boat be domain int(1..n_boats)
find hosts: set (minSize 1) of Boat,
     sched: set (size n_periods) of
              function (total) Boat --> Boat
find optVar : int(0..n_boats)
     optVar = |hosts|
minimising optVar
```

Figure 8.1: ESSENCE specifications for the three problem classes considered herein. Synchronous Optical Networking (SONET) [151] is given in full. For brevity, only the parameters, decision variable declarations (from which streamliners are generated) and optimization variables are shown for the Progressive Party Problem [192] and the Minimum Energy Broadcast Problem [33]

8.2 Searching for a Streamliner Portfolio

From a cursory analysis of Figure 8.2 one might come to the conclusion that streamliner combination *44-5* is the most effective as in terms of these three instances it always finishes first and achieves the largest speedup relative to the *Original* model. The problem however is that as streamlining can modify the solution space a candidate streamlined model may find a solution quickly, but of poor quality, and may exclude the set of optimal solutions entirely. Even though *44-5* does finish the fastest it does not for any of the three instances reach the same objective value as that of the *Original* model. In this case the gaps are relatively small with the largest occurring on instance 494-1-238598565 where the *Original* reaches a best objective value of *6* vs *8* for streamliner *44-5*. In comparison Streamliner *41* takes slightly longer to complete however in contrast on all three instances it retains the optimal objective value. For COP it is no longer sufficient to just analyze the time to complete; as with satisfaction problems; but additionally the value of the objective must be taken into account. Depending on the goal of the practitioner both of these streamliners could be an effective choice. For *44-5* considering the speedup attained one could argue that not maintaining exactly the optimal solution is an acceptable tradeoff. However dependent on the domain even small disparities in the objective value could have large consequences in overall performance and so *41* might be the proficient choice. Neither of these streamliners dominate each other and so the portfolio constructed during search should contain both.

To accomplish this, we modify the search show in Chapter 5, to add a third objective to allow us explicitly to balance considerations of solution quality against how aggressively the streamlined model reduces search:

1. **Applicability**. The proportion of training instances for which the streamlined model admits a solution.
2. **Search Reduction**. The mean reduction in time to prove optimality in comparison with the *original* model.
3. **Optimality Gap**. The mean percentage difference between the optimal value found by the streamlined model and the true optimal value under the *Original* model.

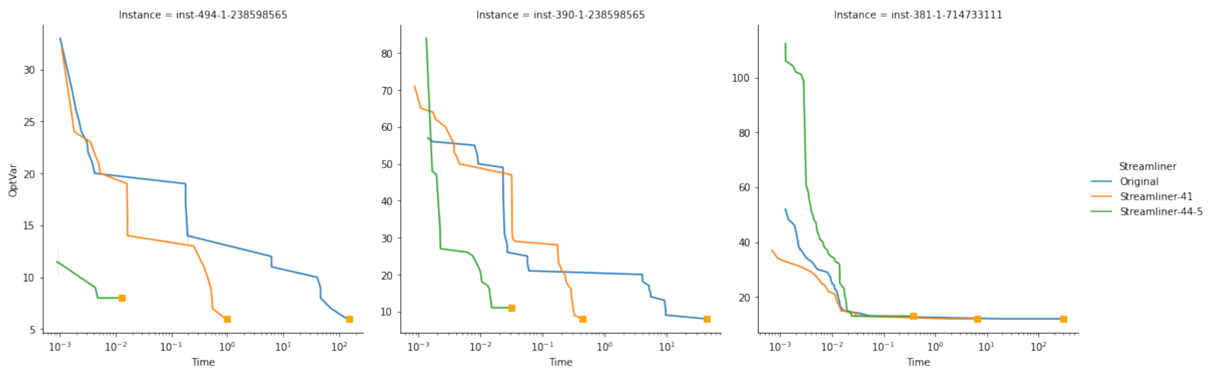


Figure 8.2: A comparison of the search progress between the Original model and two streamliner combinations (*41* and *44-5*) for the MEB problem across three instances. The solver in this setting is Minion and the objective is to be minimized

8.3 Pruning the Streamliner Portfolio

As the number of objectives increases so, typically, does the size of the Pareto front, and hence the size of the generated streamliner portfolio. This is demonstrated in Table 8.1, which, in column 2, records the size of the streamliner portfolios generated through MOMCTS for our three problem classes. A large portfolio is cumbersome when considering streamliner selection and scheduling. We observed, however, that the streamlined models were not distributed evenly across the Pareto front. Therefore, GMeans clustering is used to identify the number of clusters present in the portfolio and a point from each cluster is then selected to form a representative subset of the full portfolio (see column 3 of Table 8.1).

Problem	Initial Portfolio Size	Pruned Portfolio Size
SONET	57	6
MEB	56	3
PPP	64	9

Table 8.1: We prune an initially generated streamliner portfolio through GMeans clustering and select a representative point from each cluster.

Table 8.2 presents some of the candidate streamliners that compose the final portfolio for each problem. Detailed alongside is the % Applicability, % Reduction and % Δ OptVar achieved on the training distribution during the search process. The last three columns gives a glimpse of how the search balances the three considerations of applicability vs solution quality vs how aggressively the streamlined model reduces search. For instance, for the PPP problem streamliner combination *7 - 27* achieves 100% applicability with an average $\approx 17.9\%$ reduction to a solution that is $\approx 3.6\%$ worse than than the optimal. *164-103* also achieves 100% applicability however it attains an average $\approx 99.9\%$ reduction to a solution that is $\approx 71.2\%$ worse than than the optimal. In comparing these two conjectures there is a tradeoff between the speed of completion vs the value of the objective retained. Having both conjectures present in the portfolio is important such that a practitioner can weigh which objective is most important for their particular setting.

Interestingly there is one streamliner combination found *41-18* for the MEB problem that is able to maintain 100% applicability and in all cases maintain the optimal solution. Upon further analysis however it became apparent that it is not an implied constraint as on a number of instances the solution space was modified and not all optimal solutions retained.

Problem	Streamliner Id	Description	% Applic	% Reduction	% Δ OptVar
Sonet	13	Approx. half the nodes installed on each ring are odd	100	63.7	0.01
	13 -15	*13* & Approx. half the nodes on each ring are from the lower half of the Nodes domain	100	83.2	1.5
	13-67	*13* & The objective variable is constrained to the lower half of its domain	42	56.9	0
	13-59	*13* & For Approx. half of the rings at most one node can be from the lower half of the Nodes domain	100	73.3	0.81
	6-67	Exactly half the nodes installed on each ring are odd & The objective variable is constrained to the lower half of its domain	32	97.9	3.8
	6-52-67	Exactly half the nodes installed on each ring are odd & The objective variable is constrained to the lower half of its domain & For Approx. half of the rings at most one node can be from the upper half of the Nodes domain	32	98.9	4.3
MEB	41-18	The range of the depths function contains all odd entries & Approx. half of the entries in the range of the parents function must be even	100	99.7	0
	42-44	The range of the depths function contains: all even entries & all values must be from the upper half of the Nodes domain	84	99.9	98.7
	41-25	The range of the depths function contains all odd entries & exactly half of the defined values of the *parents* function are *odd*	34	99.8	0
PPP	7 - 14	For half of the hosts the boats must be in the lower half of the Boats domain & For approx. half of the hosts the Boats must be odd	100	30.7	3.8
	164-103	The objective must lie in the upper half of the Boats domain & there can be at most one period where more than one entry from the range of the period is in the lower half of the boats domain	100	99.9	71.2
	7 - 27	For all periods, for half the range of the function the values must lie in the lower half of the Boat domain & For half of the hosts the boats must be in the lower half of the Boats domain	100	17.9	3.6

Table 8.2: Sample streamliner combinations from the three generated portfolios for the problem classes we consider (see Figure 8.1 for their ESSENCE specifications). References to odd/even are with respect to the integer identifiers associated with entities such as nodes or boats. Streamliner Id is a unique reference given to a streamliner when generated through CONJURE; we shall refer to these examples in Section 8.6.1

8.4 Selecting from the Streamliner Portfolio

Having constructed a streamliner portfolio for a particular problem class using MOMCTS and the set of training instances, for a given test instance the question arises as to which streamlined models from the portfolio should be used, in what order, and according to what schedule. We consider both static lexicographic selection methods, which establish a priority order over our three objectives of Applicability, Search Reduction and Optimality Gap, and a dynamic method, which adjusts the selection based on the performance on the instance thus far.

Algorithm 3 Lexicographic Streamliner Selection

```
procedure SELECTION(Portfolio P, Ordering, $Time_{total}$, Instance)
    $P \leftarrow sort(P, by = Ordering)$
    $Time_{Taken} \leftarrow 0$
    while $Time_{Taken} \leq Time_{Total}$ do
        Streamliner ← P.next()
        Stats ← Apply(Streamliner, Instance)
        if Stats→sat() then
            setBound(Instance, Stats.bound)            ▷ Set new bound on the instance
        end if
        $Time_{Taken} += $ Stats.time
    end while
end procedure
```

8.4.1 Lexicographic Selection Methods

As discussed earlier in Section 6.4.2.1 it is possible to order the streamlined models in a portfolio lexicographically. In the optimization setting there are three objectives and six such orderings to consider. Through preliminary testing it became apparent that only two of these orderings are effective, where the Applicability objective is prioritised. The other orderings trade Applicability for either Search Reduction or a better Optimality Gap. On more difficult test instances, significant search effort can be required to prove that an aggressive streamliner has rendered an instance unsatisfiable, which can lead to poor overall performance. Thus two lexicographic selection methods are used herein: {*Applicability First, Optimality Second, Reduction Third*} and {*Applicability First, Reduction Second, Optimality Third*}.

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 as shown in Algorithm 3. The schedule is static in that it only moves to the next streamlined model when the search space of the current one is exhausted. A key parameter is $Time_{total}$, which specifies the total budget in seconds for traversing the streamliner portfolio. In Section 9.4 for each selection method four different settings for this parameter are experimented with to explore its effect on overall performance.

Algorithm 4 UCBSelection

```
procedure SELECTION(Portfolio, Ordering, $Time_{total}$, Instance)
    $Time_{taken} \leftarrow 0$
    UCBTimeLimit ← 1
    NumberOfIterations ← 0
    Map                                  ▷ Mapping from Streamliner to Process
    while $Time_{taken} \leq Time_{total}$ do
        Streamliner ← UCTSelection(Portfolio)
        if  Map[Streamliner].restart then
            Process ← remodel(instance, streamliner)    ▷ Remodel with the new bound
            Map[Streamliner].process ← Process
            Stats ← run(Process, UCBTimeLimit)
        else
            Process ← Map[Streamliner].process
            Stats ← run(Process, UCBTimeLimit)  ▷ Continue running existing process
        end if
        Map[Streamliner].visits += 1
        NumberOfIterations += 1
        if Stats→sat() then
            Map[Streamliner].reward += 1
            setBound(Instance, Stats.bound)         ▷ Set new bound on the instance
            for $S \leftarrow Map$ do
                if S != Streamliner then
                    Map[S].restart = True       ▷ New Bound was found; restart all other
processes
                end if
            end for
        end if
        $Time_{taken} += $ Stats.time
    end while
end procedure
```

8.4.2 UCB Streamliner Selection

During optimisation, typically a number of feasible solutions are discovered before the optimal objective value is found. This intermediate information can be used as an indicator of the performance of the streamlined model. For a given instance we have no prior knowledge of the suitability of a particular streamlined model and as such it is important to balance the time taken exploring the portfolio to identify the performance of each model while exploiting those that have already found solutions. Representing this as a multi-armed bandit problem allows us to employ well known regret-minimising algorithms to deal with the exploration/exploitation dilemma.

For each streamliner k we record the average reward x^k and the number of times k has been tried in the selection n_j out of a total of n iterations. On each iteration a streamliner is chosen that maximizes $x_j + \sqrt{2\log(n)/n_j}$. The reward distributions for an individual streamliner are not fixed, so this is not a Stationary Multi-Armed Bandit problem. However, if a streamliner performs well, we expect it will continue performing well during search even if there is a slight variation in the mean reward. We have found that using UCB1 gives good results. Future work could investigate the use of Upper Confidence Bound policies for non-stationary bandit problems, such as the family of Exp3 algorithms [119, 148].

When traversing the portfolio UCB performs incremental evaluation, it runs a streamliner for a set time, observes the results, and potentially moves on before the corresponding search space has been exhausted. When the streamliner is pre-empted it is necessary to pause the search in order to avoid repeating work if it is rescheduled at a later point. The only exception to this is whenever a new bound on the objective is discovered all of the streamliners from the portfolio, aside from the current streamliner, are restarted and remodeled with the new bound. There are two main benefits to doing this. Firstly, by restarting the streamliner has the newly constrained bound at the top of the search tree which allows it to make more informed decisions higher up without descending into unsatisfactory subtrees. Secondly, by remodeling it takes advantage of the toolchain (Conjure and Savile Row) which may be able to reformulate the model based upon this new information and produce reductions at the solver level. Algorithm 4 shows the UCBSelection process in detail.

8.5 Experimental Setting

We evaluate our automated streamlining approach on the three problem classes in Figure 8.1. We selected these problems to give good coverage of the abstract domains available in Essence, such as set, multi-set and function. Furthermore, SONET and Progressive Party have nested domains: multi-set of set and set of function respectively.

Our hypothesis is that a streamliner portfolio, generated automatically on a set of automatically generated training instances from a given problem class, can be employed to solve more difficult test instances to deliver substantial performance improvements relative to the *original* model. Training instances were generated as per Chapter 4, with a time limit of $[10, 300]$ seconds. Test instances are generated using the same instance generator and the tuning tool irace but with a time limit of $(300, 3600]$ seconds. 50 instances are selected randomly to form the test set.

Care must be taken when considering the proof of optimality of our test instances. Although in solving a streamlined model the constraint solver may exhaust the search space this is not a proof that the current objective value is optimal. This is because streamliners are not necessarily sound, hence a streamlined model may exclude the set of optimal solutions. For this reason, after the streamliner portfolio has been run for its allotted time, we use the remainder of the time budget to run the *original* model, starting from the best objective value found by the streamliner portfolio, to provide the optimality proof. The benefit of streamlining in this context is in finding high quality solutions much more quickly than the *original* model.

All experiments were run on a cluster of 280 nodes, each with two 2.1 GHz, 18-core Intel Xeon E5-2695 processors. MOMCTS was run on a single core with a budget of 4 CPU days for each problem class. The number of instances solved within the limit and the time reduction ratio of the streamliner selection strategies are reported. The percentage of time reduction (for both reduction to optimality and reduction to proof) is calculated with respect to the *original* model under the same random seed. Every test instance was run with three random seeds.

8.6 Results

Table 8.3 summarises results on 50 test instances for each of our three problem classes. We evaluate four different approaches: the *original* model, and streamliner portfolios with UCB selection, lexicographic ordering {*Applicability First, Optimality Second, Reduction Third*} (denoted *opt-second*), and lexicographic ordering {*Applicability First, Reduction Second, Optimality Third*} (denoted *red-second*). For each streamliner selection method, a parameter is the amount of time allocated to the streamliner portfolio before handing over to the *original* model to prove optimality. Four different values for this time budget were tested: 30, 60, 120 and 300 seconds.

The second column (labelled *#proved*) gives a broad view of overall performance. It reports the average number of instances (3 runs per instance with 3 different random seeds) where an optimal solution is reached and optimality proved within one CPU hour. [1] All streamliner portfolio variants significantly outperform the *original* model by this

[1]Tuning and generation of test instances is performed on the basis of one seed. On the two other seeds

	Strategy	mean #proved (1-hour)	Finding an optimal solution						Finding and prove optimality					
			time(s)			speed-up ratio			time(s)			speed-up ratio		
			p10	p50	p90	p10	p50	p90	p10	p50	p90	p10	p50	p90
MEB	unstreamlined	35	157.9	1185.2	13893.9				311.1	1976.2	16781.3			
	UCB-30s	**50**	6.1	8	**11.0**	14.2	158.2	1583	15.2	22.2	**176.7**	6.6	43.6	492.2
	UCB-60s	**50**	4.4	7.2	12	15	150.3	1552.2	16.1	24.6	188.7	6.9	35.6	521.9
	UCB-120s	**50**	4.5	7.8	12.1	14.9	158	1604	15.1	24.8	220.9	6.2	36.1	518.4
	UCB-300s	**50**	4.5	7.1	12.1	15	157.5	1605.4	14.9	24.9	345.1	5.2	32.1	416.6
	opt-second-30s	49.7	**4.1**	6.3	13.4	14.1	171.1	1701.5	11.6	22.9	221.3	7.3	44.6	605.9
	opt-second-60s	49.7	**4.1**	6.6	14.9	15.7	174.3	1833.4	11.7	22.5	199.6	7	45.3	625.4
	opt-second-120s	**50**	4.2	6.2	13.6	**19.9**	178.3	1776.7	11.7	21.8	181.6	7.3	46.5	594.9
	opt-second-300s	**50**	**4.1**	**6.1**	12.8	**19.9**	170.9	1865.8	**11.5**	21.8	176.9	7.5	**47.6**	**647.0**
	red-second-30s	49.7	**4.1**	6.7	13.6	14.1	156	1845.1	11.8	22.8	249.1	7.3	43	532.2
	red-second-60s	49.7	4.2	**6.1**	12.8	15.3	**187.0**	1878.3	11.8	**21.7**	198.7	7.3	45	646.3
	red-second-120s	**50**	**4.1**	6.2	12.6	16.9	177.4	**1903.5**	11.6	22.1	178.1	7.2	46.3	605.4
	red-second-300s	**50**	**4.1**	**6.1**	13.5	16.8	167.5	1891	11.7	22.3	178.8	**7.6**	47.5	625.1
PPP	unstreamlined	41.3	73.4	564.3	3123				313	1339.7	6908.1			
	UCB-30s	47.7	13	**73.7**	**1007.9**	**1.2**	**4.1**	52	49.2	**350.8**	1946.6	**1.0**	**3.0**	**29.3**
	UCB-60s	**48.3**	19.2	105.9	1078.7	0.9	2.9	28.7	86.1	428.8	2141.5	0.9	2.5	24.4
	UCB-120s	**48.3**	18.9	163.3	1129.7	0.7	2.5	31.8	135.5	449.6	**1936.2**	0.9	2.1	16.8
	UCB-300s	**48.3**	19	344.6	1311.3	0.4	1.6	30.1	323.9	646.3	2273.2	0.6	1.4	10.5
	opt-second-30s	46.7	8.3	105.1	1340.5	0.9	3.5	75.1	**44.1**	419.4	2592.5	0.9	2.4	26.2
	opt-second-60s	47	**8.1**	105.8	1444.2	0.8	3.4	75.2	73.7	453.5	2640.3	0.8	2.3	18.9
	opt-second-120s	47.3	8.9	142.9	1765.1	0.7	3.6	**76.5**	113.1	486	2716.7	0.8	1.9	17.6
	opt-second-300s	47.7	8.9	211	1349.3	0.5	3.1	72.4	110.8	599.1	2703.2	0.7	1.8	15.5
	red-second-30s	45	14.7	177.7	2344.7	0.7	2	18.9	73.3	626.2	3537.7	0.8	1.7	14.8
	red-second-60s	45.3	21.2	195.2	2341.6	0.6	2.1	15.6	96.1	643.2	3174.7	0.7	1.8	13.8
	red-second-120s	45.7	13.6	175.7	2384	0.6	2.1	17.5	136.5	591.5	3095	0.6	1.8	11.1
	red-second-300s	45.3	13.6	228	2731.5	0.6	1.9	16.8	157	657.6	3339.1	0.6	1.4	8.4
SONET	unstreamlined	43	539.5	1263.2	3820.3				574.4	1417.8	3954			
	UCB-30s	**50**	5	21.8	**121.9**	**10.3**	49.7	341.5	34	**42.3**	**174.0**	**6.6**	**23.4**	60.5
	UCB-60s	**50**	6.1	28	131.9	8.5	38.1	300.3	63.3	75.3	198.7	4.9	14.4	42.1
	UCB-120s	46	6	31.1	246.8	3.4	31.5	321.9	121.2	132.2	581.2	2.3	7.6	32.1
	UCB-300s	**50**	7	30.7	344.5	3.8	33.4	287.3	111.8	310.8	437.8	1.7	4.2	22.9
	opt-second-30s	49.3	3.5	9	1023.8	1.4	112.7	553.9	27.7	72.7	1023.2	1.4	19.2	**70.5**
	opt-second-60s	49.7	3.5	9	443.2	1.5	113.1	611.4	27.6	93.6	644	1.5	15.9	66.9
	opt-second-120s	49.3	3.3	**8.3**	455.6	1.3	117.3	**677.9**	**26.9**	120.7	701	1.3	14.6	68.6
	opt-second-300s	49.3	3.7	8.4	549	3.6	**121.0**	549.9	28	123.1	770.6	1.1	10.3	69.4
	red-second-30s	47.7	**3.0**	115.4	1749.6	0.8	10.6	483.5	27.7	227.3	2167.4	0.8	5.2	61.7
	red-second-60s	47.7	**3.0**	105.3	1760.9	0.8	14.2	530.9	28.1	185	2137.2	0.8	7.2	64.1
	red-second-120s	47.3	**3.0**	96.7	1532.5	0.8	16	506.3	28.3	157.8	2295.6	0.8	7.6	62.6
	red-second-300s	47.7	**3.0**	96	1451.4	0.9	18.2	533.8	27.1	221.4	1717.6	0.8	6.1	65.2

Table 8.3: Summary results on 50 test instances (3 runs/instance) on three optimisation problem classes: MEB, PPP and SONET. Reported results include: the average number of instances solved within one-hour (#proved 1-hour); the time to reach an optimal solution, the time to both reach an optimal solution and prove its optimality; and the corresponding time-reduction percentages when compared to the *original* model. A time-reduction value of 99% means the solve time was reduced by 99%. For each measurement (except #proved 1-hour), we report the 10^{th} percentile (p10), the median (p50), and the 90^{th} percentile (p90).

simple measure.

We then proceed to report more detailed results, where each run is now given a maximum amount of 96 CPU hours (4 days). The amount of time to reach an optimal solution value, as well as the time to both reach *and* prove optimality are reported. The corresponding reduction percentage of those two values compared to the *original* model are also listed. For each measurement, we report the 10th percentile, the median and the 90th percentile values. These values are reported as the mean can be skewed by outliers. In particular, if the optimal solution is not proved this results in a large time value (96 hours = 345600 seconds) for that run. The percentiles avoid this situation and show a clearer overall trend.

Results in Table 8.3 are strongly positive. They show that all the streamliner portfolio approaches can not only find an optimal solution and prove optimality on more test instances than the *original* model, but also vastly reduce the amount of time required for both tasks. In general, the UCB-30s variant has the best overall performance across the three problem classes, and provides consistently robust improvement over the *original* model.

More details on how the streamliner approaches improve on the original models on an instance basis are presented in Figure 8.3. For brevity, we only show results of the streamliner variants with the time limit of 30 seconds. Each data point in the plot corresponds to a pair of instances and random seeds. The plots show that the solving time of the test instances are well distributed across the x-axis, which is a good indication for the diversity of the test instance set. There are several cases where the *original* model cannot find or prove optimality within the time budget and the streamliner can, which are represented by the data points on the rightmost side after the vertical red lines.

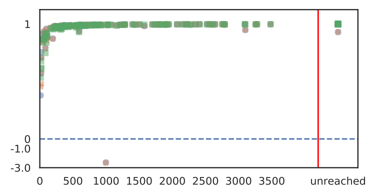
The MEB results demonstrate strong performance of all three streamliner approaches on all test instances. On SONET, UCB-30s clearly has better performance compared with the other two approaches, which aligns with the summary results in Table 8.3. While still strongly positive, on PPP the reduction provided by the streamliner approaches is not quite as strong as for the other two problem classes. There are a minority of cases where even the best streamliner approach, UCB-30s, cannot find or prove optimality within the time budget, as shown by the data points in the bottom-right corners.

As can be seen from Table 8.3 and Figure 8.3, the time to prove optimality is very significantly reduced through the application of streamliners. This stems directly from their ability to find high quality feasible solutions very quickly. Therefore, once the time

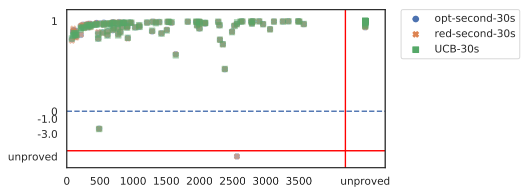
it is possible for the unstreamlined model to time out at one CPU hour allocated to the streamlined models has been spent, the *original* model is run starting from a very high quality or even optimal objective value, requiring much less effort to exhaust the search space.

8.6.1 UCB Streamliner Selection: Discussion

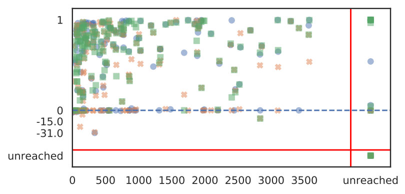
In this section, we discuss the UCB approach for streamliner selection in more detail, as UCB-30s achieves the best overall performance across the three problem classes, both in terms of reduction to finding the optimal objective value and reduction to proving



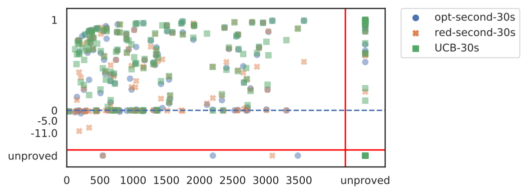
(a) MEB - time to optimal



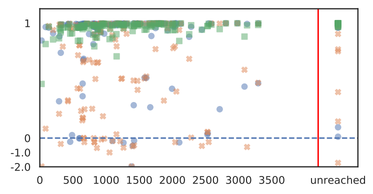
(b) MEB - time to proof



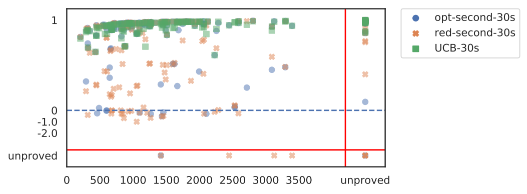
(c) PPP - time to optimal



(d) PPP - time to proof



(e) SONET - time to optimal



(f) SONET - time to proof

Figure 8.3: Reduction ratio of streamliner approaches with 30 seconds for streamliner portfolio. Two reduction ratio values are reported: reduction in time to reach an optimal solution, and reduction in time to reach an optimal solution and prove its optimality. The x-axis represents the time required by the *original* model. The y-axis shows the the reduction value. Each data point correspond to a pair of (instance, random seed). These plots focus on the region within a 1-hour time limit: all data points outside that ranges are shrunk into the same region. More specifically, runs where the (original model) streamliner approaches do not reach an optimal solution or does not prove optimality in one hour are separated by the red (vertical) horizontal lines. The reduction values, however, are still the true values calculated based on the 4-day CPU limit. As most data points lie within the range of $y \in [0, 1]$, the plot is rescaled so that this range is zoomed in for a better visualisation.

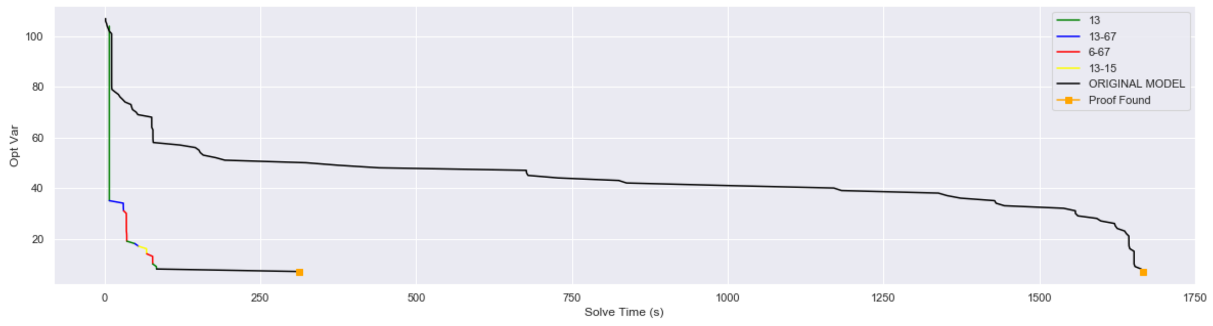


Figure 8.4: Objective value progression from the *original* model compared with its progression under the UCB selection method for a representative SONET instance.

optimality. In contrast to the lexicographic methods, which only move on to the next streamlined model when the search space of the current one is exhausted, UCB benefits from its ability to sample the entire streamliner portfolio. After the initial exploration phase, where each streamliner is given its initial application, UCB then selects streamliners based upon the observed rewards. It main advantage is the ability to balance the exploration and exploitation of the streamlined models in the portfolio.

It is not always the case that the objective is found purely through the application of one streamliner. In fact for SONET, on average three streamliners are used across the 50 test instances to arrive at the optimal objective value. Having access to the whole portfolio allows UCB to descend upon the optimal objective value more quickly and is one of the reasons for its success. The application of several different streamliners at different time points can be used to reduce the bound of the objective in an effective manner as can be seen from Figure 8.4.

The UCB algorithm exploits the streamliners that have previously been shown to produce an improvement in the objective value. This can be very clearly shown from Figure 8.4 where for an instance from SONET the streamliners 13, 13-67 and 6-67[2](explained in Table 8.2) improve the objective multiple times during the course of the selection process. This is due to the fact that UCB is continuing to exploit those streamliners as previously they had success. However, it is also crucial to continually explore the portfolio in an attempt to find streamliners that did not initially have success but may do after a certain number of iterations. Streamliner 13-15 is an example of such a case.

[2]13-67, for example, indicates a streamlined model including both streamliner 13 and 67

8.6.2 Time Allocated to the Streamliner Portfolio: Discussion

From Table 8.3 it can be seen that the $Time_{Total}$ parameter as defined in Algorithms 3 and 4 can have a large observable impact on the overall performance of the selection method. There is a general trend (excluding MEB which will be discussed separately) that as the $Time_{Total}$ increases the time both to find and prove the optimal objective value increases. This may seem puzzling initially: if using a $Time_{Total}$ of 30s reduces the time to find the optimal objective value to a certain extent, it might be expected that a $Time_{Total}$ of 300s will do equally as well. However, there are two things to consider. First, streamliners from the portfolio are not guaranteed to preserve the optimal value and so there is the potential for an optimality gap between what the streamliners can find and the true optimal of the instance. Therefore, the true optimal is only found after the switch to the *original* model occurs. Second, on average the streamliners converge upon their optimal value in a very short period of time, 17s, 7s and 12s for SONET, MEB and PPP respectively. By increasing the $Time_{Total}$ parameter it delays the point at which the switch occurs to the original model which in turn delays the point at which the true optimal is found. However, for MEB the $Time_{Total}$ does not have a large impact on performance and this is due to the fact that the streamliners in the portfolio generally exhaust their search space very quickly. This means that the whole portfolio can be traversed before $Time_{Total}$ is reached and so the time at which the switch to the *original* model occurs is generally the same across all parameter settings.

The increase in time to prove optimality occurs as if the T_{total} parameter is set too large then when the optimal value is found at time T_{opt}, the whole duration from $T_{opt} \rightarrow T_{total}$ is spent proving the optimality of that solution in the streamlined subspaces. As was discussed earlier, proving optimality with respect to the streamliners does not prove optimality on the original model and so the whole time from $T_{opt} \rightarrow T_{total}$ is wasted.

8.7 Summary

We have presented the first automated approach to generating streamliners automatically for optimisation problems, and for their selection and scheduling when employed on unseen instances. On three quite different problem classes the results are very encouraging, with vastly reduced effort both to find and to prove optimal objective values. An important question we plan to investigate further is the applicability of our method to identify in which contexts our streamliner can and cannot help. In the context of optimisation the benefit of streamlining lies in the early identification of the optimal, or at least high quality, values for the objective. Where the *original* model is able to identify the optimal value quickly, the benefit of streamlining will be limited.

Furthermore, there are several methods for devising good search strategies for constrained optimisation problems. Recent research suggest using machine learning to design a promising search ordering [42], using solution density as a heuristic indicator [159] and a number of value ordering heuristics to find good solutions early [158, 55]. Streamlining constraints can potentially be used in combination with the existing methods for devising good variable and value selection heuristics to achieve even better results.

Chapter Nine

Model Portfolios

The refinement of streamlined Essence specifications into constraint models suitable for input to constraint solvers gives rise to a large number of modelling choices in addition to those required for the base Essence specification. Up to now in this thesis the automated streamlining approaches have been limited in evaluating only a single default model representation for each streamlined specification. In this chapter we explore the effect of model selection in the context of streamlined specifications. We explore augmenting our best-first search method to generate a portfolio of Pareto Optimal streamliner-model combinations by evaluating for each streamliner a portfolio of models to search and explore the variability in performance and find the optimal model. Various forms of racing are utilised to constrain the computational cost of training. Empirical results demonstrate drastic improvements in solving time for some problem classes in comparison to a single-model approach.

9.1 From Essence Specifications to Constraint Models

In the prior chapters the Conjure automated modelling system has been used to refine a streamlined Essence specification into a constraint model for evaluation. However it is limited to accepting Conjure's default choice for each of the modelling decisions that need to be made, such as how to represent abstract variables as constrained collections of more primitive variables, resulting in a single streamlined model. The hypothesis that motivates the work in this chapter is that these default modelling choices are unlikely always to result in the most effective model, and that the best modelling choices may vary according to which streamliner constraints are added to the original specification.

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

We use the Transshipment problem (Figure 3.10d) as a running example to illustrate how Conjure produces alternative models. This specification has two *find* statements, both with function domains: *amountWT* and *amountTC*. Each of these can be modelled in three different ways in the current version of Conjure. The first is the most compact and utilises two matrices, one Boolean matrix to represent the domain of the function (as it is not known to be *total*) and another integer matrix to contain the values. Both are of the form `matrix indexed by [int(1..n_warehouses), int(1..n_transshipment)]`. An alternative interpretation models the function as a binary relation first, followed by modelling the relation in two different ways. Both representations use two integer matrices `matrix indexed by [int(1..n_warehouses * n_transshipment * maxStock)]` for each component of the relation. They differ in how they represent the cardinality of the function, one using a single integer marker and the other one using a Boolean matrix as flags to denote whether the value is in the relation.

Hence, from the two *find* statements alone there are nine possible combinations of modelling choices. However, Conjure also has to model other types of objects such as *given* parameters, *auxiliary* variables and *quantified* variables each of which could have multiple different representations. Conjure also has the ability to implement *automated channelling* [38] in which each reference to a variable in a constraint expression can be modelled differently. Each of these components increases the number of modelling choices and as a result a single abstract Essence specification can typically be refined into a large number of constraint models. Figure 9.1 shows for the Transshipment problem the number of candidate models on the unstreamlined Essence and how this changes with the introduction of streamliner constraints. Over 6000 different models can be generated for the unstreamlined specification alone and with the additional modelling choices streamliners radically increase this number. Even though a large number of these models are small modifications of other models, some models are significantly different from others and they have very different performance characteristics. Moreover, different models are also likely to benefit from the addition of streamliner constraints differently.

In Chapter 5 and Chapter 8 this choice is handled by utilising the DEFAULT heuristic of CONJURE. This is a heuristic employed during refinement to commit greedily to promising modelling choices at each point where an abstract type or a constraint expression may be refined in multiple ways [3]. The DEFAULT heuristic is a combination of two other heuristics: COMPACT for decision variables and constraint expressions and SPARSE for parameters. COMPACT favours transformations that produce smaller expressions. For an abstract type, we define an ordering as follows: concrete domains (such as bool, matrix) are smaller than abstract domains; within concrete domains, bool is smaller than int and int is smaller than matrix. Abstract type constructors have the ordering set $<$ mset $<$ sequence $<$ function $<$ relation $<$ partition. These rules are applied recursively so that COMPACT will select the smallest domain according to this order. The SPARSE heuristic is designed to choose the domain representations that will generate the smallest ESSENCE PRIME parameter files for sparse objects. Some representations represent every *potential* member of a domain explicitly whereas some representations only represent the actual members of a domain. For a constraint expression (and the objective), it chooses the refinement with the most shallow abstract syntax tree. For a parameter, consider a domain such as `set of int(1..M)` where `M` is a very large value. Representing this set using a Boolean list indexed by `int(1..M)` would create a Boolean value for every potential value, whereas an integer list indexed by `int(1..C)` (where C is the computed cardinality of the

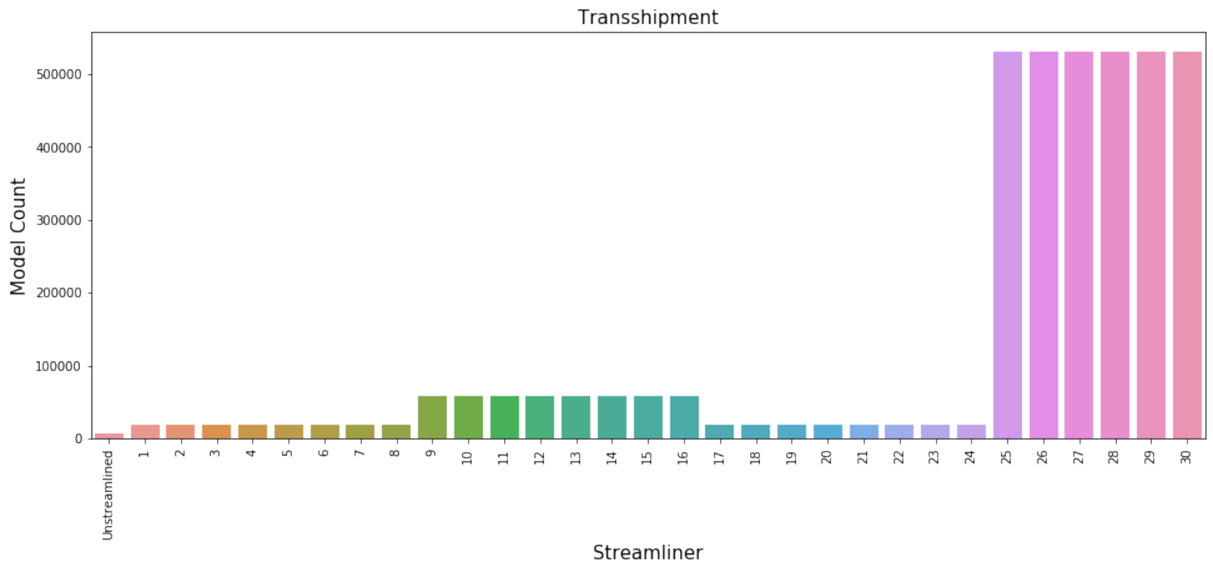


Figure 9.1: The number of models refined from the original ESSENCE specification and with a sample set of single candidate streamliners for the Transshipment problem. The streamliners are represented in CONJURE via numeric values which are presented.

set) would only create an integer value for actual members of the set.

Why this is important is that in Constraint Programming the underlying structure of the constraint model can have a drastic impact on the overall efficiency of the constraints that compose the model. A poorly performing model may inhibit the propagation of the constraints resulting in less aggressive domain value reduction leading to a larger overall search tree to be explored. Candidate Streamliners are themselves purely just additional constraints encoded on top of the base model and as such it is possible that their efficiency is also governed largely by the choices made on the underlying constraint model.

Figure 9.2 shows the performance of streamlined models for two different problem classes across 8 different models. For Transshipment the DEFAULT model does produce the best encoding for streamliner 5, which places restrictions on which Transshipment locations that can be used. It is able to achieve roughly a 61% reduction, far greater than the other models. However the picture for BACP is quite different. For Streamliner 10, which enforces that half of the courses must be placed in even numbered periods, the DEFAULT model is only able to achieve around a 20% reduction in time, far inferior to the 60% reduction achieved by the best model. This shows two things: firstly across different model representations there can be a large variability in the streamliner performance and secondly that the DEFAULT heuristic may not always provide the best performance.

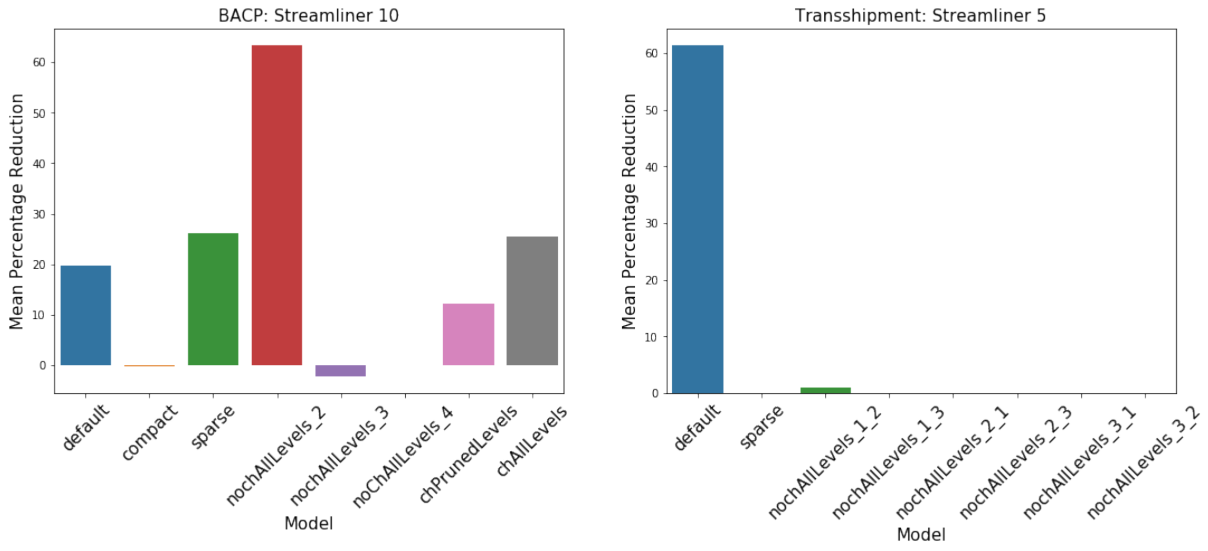


Figure 9.2: Variance in streamliner performance across 8 models for Transshipment and BACP. Each model is named with respect to the generating heuristic (table 9.1)

Heuristic	**Declaration Type**				**Additional Flags**	
	Find	**Given**	**Auxiliaries**	**Quantifieds**	**Channelling**	**Levels**
default	compact	sparse	compact	compact	False	True
compact	compact	compact	compact	compact	False	True
sparse	sparse	sparse	sparse	sparse	False	True
nochPrunedLevels	all	sparse	compact	compact	False	True
nochAllLevels	all	sparse	compact	compact	False	False
chPrunedLevels	all	sparse	compact	compact	True	True
chAllLevels	all	sparse	compact	compact	True	False
fullPrunedLevels	all	sparse	all	all	True	True
fullAllLevels	all	sparse	all	all	True	False
fullParamsPrunedLevels	all	all	all	all	True	True
fullParamsAllLevels	all	all	all	all	True	False

Table 9.1: The ranked set of heuristics we use when generating a portfolio of models.

9.2 Model Portfolios

As seen from Figure 9.1 the number of possible models that can be refined, especially from a streamlined specification, means that refining and evaluating all possible models for each streamliner is not feasible. Instead, a sample of N models can be evaluated where N is controlled to balance the computational cost against the exploratory benefit. Conjure allows for the customisation of the modelling process in several ways. First, different strategies can be employed for the modelling of different types of objects. A distinction can be made among four different forms of declarations, *{find, given, auxiliary, quantified}* depending upon their origin in the specification. *find* and *given* statements define decision variables and problem parameters, *auxiliaries* are decision variables created by Conjure during model reformulation, and *quantified variables* are defined by quantified expressions like `forAll`, `exists` and `sum`. Second, automated channelling and precedence levels can be activated and deactivated to further customise the modelling heuristic. To produce a portfolio of N models we use a set of heuristics (Table 9.1), ranked based on two criteria. First, in ascending order of the predicted number of generated models and second our perceived effectiveness of the generated models. These heuristics can be then invoked in an iterative fashion until a portfolio of N models has been generated.

9.3 Model Racing

Evaluating all candidate models for each considered streamliner combination is expensive. Poorly-performing streamliner/model combinations can timeout on several instances and consume a large amount of the training budget. Hence, we employ various *racing* techniques [27, 107, 3, 34] to terminate poorly-performing combinations early. The streamliner search can then allocate more time to evaluate promising streamliners and gain a more accurate estimation of their performance.

Racing is performed at each lattice node of the search, representing a combination of streamliners. We race among the multiple models generated by CONJURE on the given streamliner combination, and try to eliminate the poorly performing models without having to fully evaluate them. We first describe the three racing strategies we used, *ρ-Capping* [3], *racing based on statistical tests* [27, 137] and *adaptive capping* [107] and then discuss how they are combined together in our streamliner search through a multi-level model generation approach.

9.3.1 ρ-Capping

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

9.3.2 Racing using statistical tests

Racing using statistical tests for the automated configuration of parameterised algorithms [102] was first proposed in [27], and later extended and implemented in the automated algorithm configuration tool IRACE [137]. Bad algorithm configurations are discarded from a race as soon as sufficient statistical evidence is observed. We apply the same idea to remove bad models early during a race.

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

In the original implementation of IRACE, parallelisation is done on an instance-basis. If the number of models is smaller than the number of cores available, there will be idle resources not being used. This under-utilisation of cores is particularly likely to happen at the end of each race as the number of surviving models gets smaller. Therefore, in our implementation, as soon as idle cores are available, the best models in the current stage are speculatively executed on the next stage. This has the benefit of keeping our CPUs fully utilized and of calculating results that we may need in the next stage in advance. Once the current stage is completed, results of the eliminated models, if calculated, are simply discarded.

9.3.3 Adaptive Capping

Adaptive capping is another technique in automated algorithm configuration to terminate poorly-performing algorithm configurations early when optimising running time of a parameterised algorithm. It was first proposed in the local search-based automated algorithm configuration tool ParamILS [107], and was later integrated into IRACE [34]. The technique has been shown to significantly speed up the search for the best algorithm configurations in many cases [107, 106, 34]. Our streamliner search makes use of the adaptive capping mechanism of IRACE.

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

9.3.4 Multi-Level Model Generation

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

Iterative deepening of the portfolio size allows us to initially evaluate the models that we believe are most likely to perform best based upon the ranking in Table 9.1. The use of elite models derived from previous levels helps the adaptive capping mechanism to speed up the evaluations in the next levels substantially. This allows the search to explore larger portfolio sizes within a reasonable amount of time. It is especially useful where the truly best model for a particular streamliner combination is actually located in a lower ranked heuristic.

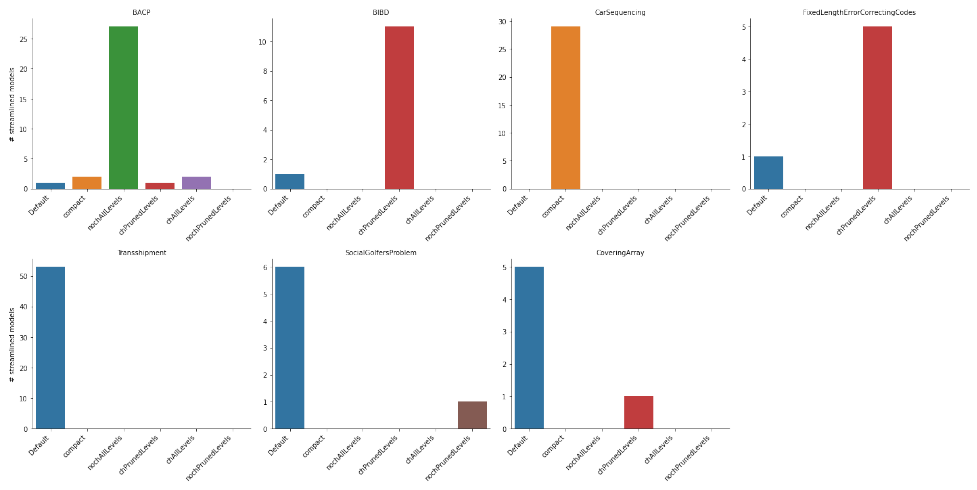
9.4 Experimental Results

Experiments were run on a cluster of 280 nodes, each with two 2.1 GHz, 18-core Intel Xeon E5-2695 processors. MOMCTS-DOM was run with leaf parallelisation [35] using up to 30 cores with a maximum budget of 4 wall-time days per problem class. Difficult test instances, with solving time in $[300s, 3600s]$, were generated automatically using the same instance generation system [2] used for training instance construction. We will show that although the portfolios of streamlined models are trained on easy instances (solved in $[10s, 300s]$), their large improvement in solving time transfers to difficult, unseen test instances.

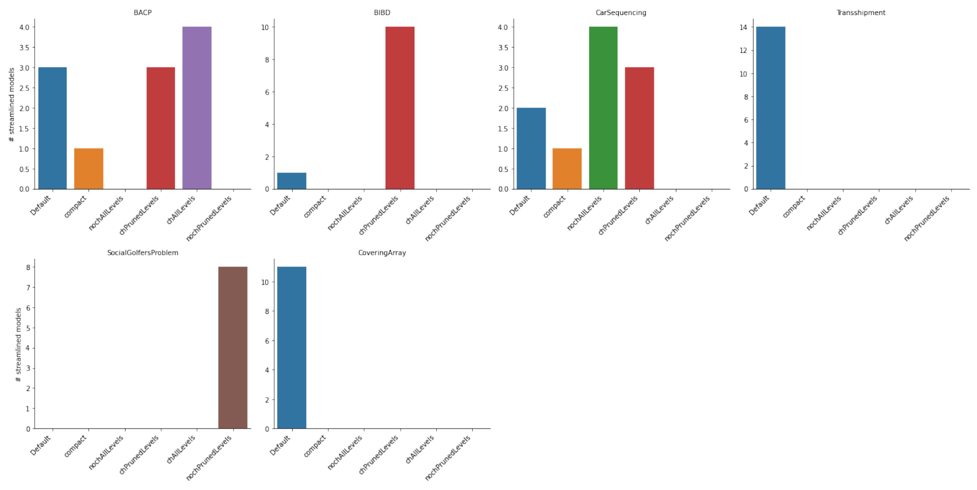
To demonstrate the positive impact of adding multiple-model exploration during streamliner search, two independent searches were conducted for each problem class: one with the single default model, and one with the multiple-model approach. Our hypothesis is that even though each round of a multiple-model search is generally more expensive than its single-model counterpart, the opportunity of exploring multiple models and our enhanced search method using model-racing will be able to produce higher-quality portfolios of streamlined models within the same time budget.

Figure 9.3 presents the composition of the portfolio of models produced for each problem class. It is immediately apparent that CONJURE's DEFAULT heuristic is not always the best choice and there are alternative representations available that provide superior performance. It is not just for one or two esoteric conjectures that this occurs: for CarSequencing-Chuffed and BACP-Chuffed there are over 25 different combinations that perform better in an alternative representation. If we focus on CHUFFED for the problems BACP, BIBD, CarSequencing and FLECC the DEFAULT heuristic actually ends up encompassing a minority share of the overall portfolio. What is interesting is that it is not always a single model that dominates the portfolio and for a number of problems there is diversity where different model representations are chosen for different streamliner combinations (BACP-Chuffed, BACP-Lingeling and CarSequencing-Lingeling). For a number of problems, no or limited diversity is found and the DEFAULT model does dominate showing it is an effective choice generally.

Table 9.2 shows our experimental results of the single-model and multiple-model settings on the seven problem classes under study. We report results of the Virtual Best Solver (VBS) for each setting to showcase the best possible improvement that a multi model approach can yield. The first observation is that streamlining is very effective on these problem classes, resulting in very substantial reduction in search effort which we would expect from our results in the previous chapters. The success of the Multi-Model approach however is quite varied. On several problems such as BACP-Chuffed, CarSequencing-Chuffed and FLECC-Chuffed large speedups relative to the DEFAULT model are observed of $\approx 2.3x$, $\approx 1.37x$ and $\approx 2.3x$ respectfully. These are significant gains over the already impressive performance of the DEFAULT model. For other problems, BIBD-Lingeling, CarSequencing-Lingeling, and SocialGolfersProblem-Lingeling speedups are observed but they are not as pronounced. For the cases where no model diversity is found during streamliner search (Chuffed-Transshipment, CoveringArray-Lingeling, etc.) the multi-model exploration does not negatively impact the speedups attained. Due to the racing and capping methods



(a) Chuffed



(b) Lingeling

Figure 9.3: Composition of the portfolio generated by our multi-model search for each problem class. The x-axis represents heuristic rules (Table 9.1) used by CONJURE. The y-axis shows the number of streamlined models in the portfolio generated by the corresponding heuristics rules. Note that models refined according to the same set of rules can still differ due to the additional modelling choices introduced by the added streamliners.

implemented during search the MM exploration is able to discover an equivalent portfolio of streamliners to that of the single model exploration within the same time limit.

There are several cases however where diversity is exhibited in the streamliner portfolio however overall it performs worse than the default single model portfolio. One of the problems with our method is that during streamliner search we identify which of the expanded models performs best for each streamliner. If we focus on BIBD-Chuffed we see that almost all streamliners perform the best under the models generated by the

Solver	Problem	# Instances	Model	VBS WallSpeedup	VBS CPUSpeedup
Chuffed	BACP	25	MM	**19.35**	**9.67**
			D	8.31	5.32
	BIBD	93	MM	1.47	0.80
			D	**1.82**	**0.96**
	CarSequencing	26	MM	**8.55**	**4.27**
			D	6.21	3.11
	CoveringArray	69	MM	2.96	1.53
			D	**3.03**	**1.56**
	FixedLengthErrorCorrectingCodes	80	MM	**2.79**	**1.76**
			D	1.20	1.14
	SocialGolfersProblem	35	MM	1.55	0.83
			D	**1.72**	**0.86**
	Transshipment	50	MM	105.02	52.51
			D	105.02	52.51
Lingeling	BACP	31	MM	3.42	1.71
			D	**3.79**	**1.90**
	BIBD	40	MM	**2.05**	**1.03**
			D	1.84	0.92
	CarSequencing	103	MM	**3.10**	**1.57**
			D	2.67	1.33
	CoveringArray	50	MM	9.82	4.91
			D	9.82	4.91
	SocialGolfersProblem	33	MM	**2.53**	**1.73**
			D	2.47	1.41
	Transshipment	101	MM	9.32	4.81
			D	9.32	4.81

Table 9.2: Results of the Virtual Best Solver (VBS) on the test instances of the seven problem classes, under the DEFAULT model setting (D) and the multi-model (MM) setting. Overall speedup in terms of wall time and CPU time are presented.

chPrunedLevels heuristic. During evaluation when a streamliner is scheduled each unseen instance will then be modelled with regards to that particular heuristic. The problem is that not every instance evaluated is going to perform well under the model generated by the *chPrunedLevels* heuristic; this representation may be a bad choice for that particular instance. By blanket modeling every unseen instance with this heuristic in some cases we could be making them much harder to solve than say using the DEFAULT model. The DEFAULT heuristic is chosen to be the *"default"* as it is a relatively safe choice by choosing the most compact option at every decision point. By choosing a bad representation, it can make the instance much more difficult to solve and even though the streamliner performs best under that representation it begins at a detriment and so the gains it achieves relative to the DEFAULT model are inhibited.

A more nuanced approach to the use of multiple models in a streamlined portfolio would be for each unseen instance to firstly try and predict using instance features which heuristic was the best representation and then attempt to match that with a streamliner from the portfolio that performed best under that representation.

9.5 Portfolio Analysis

9.5.1 Balanced Academic Curriculum Problem (BACP)

The BACP portfolio is the most diverse of the four problem classes. It is comprised of various models generated from five different heuristics. Some models have both *channelling* and *precedence levels* for representations enabled whereas others have both disabled. Those modelling choices interact with the candidate streamliners and enable efficient propagation during the search, leading to the improvement in all performance measurements as shown in Table 9.2.

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

9.5.2 Fixed Length Error Correcting Codes (FLECC)

For FLECC, switching from single-model to multi-model search results in a large increase in the percentage of improved test cases (nearly 60% for both VBS and Autofolio). There are two clearly dominating heuristic rules. When the additional modelling choices for the added streamliners are also considered, four different streamlined models are found. Three of these make use of automated channelling, an option that is disabled in the DEFAULT model. In these models, channelling allows a decision variable to be represented using multiple viewpoints, which is likely to allow better constraint propagation. Streamliner *7* which achieves high applicability of 95% across the Training Set enforces that the Characters that compose the error correcting codes are drawn from the lower half of the Character domain. This streamliner achieves better performance under a channelled representation than under the DEFAULT model. In one case, however, moving to the DEFAULT heuristic (channelling disabled) provides the best performance. This occurs for the streamliner combination *8-6* which enforces that not only are the Characters that compose the error correcting codes drawn from the upper half of the Character domain but additionally their integer index is odd.

9.5.3 Car Sequencing

This portfolio comprises models generated using a single heuristic, COMPACT, which differs from DEFAULT in the way in which it treats parameters. In the ESSENCE specification for CarSequencing Figure 1.4 3 of the given parameters are *total* functions and are represented using an *integer* matrix by both heuristics. The *usage* relation in the COMPACT model is represented using an *occurrence* representation: `matrix indexed by [int(1..n_classes), int(1..n_options)] of bool`. The DEFAULT model, however, has an *explicit* representation with two integer matrices. Even though the occurrence representation will generally have a larger footprint the size is only related to the product of the *Class* and *Option* parameters, which are generally not large enough to become prohibitive. The use of a single matrix then allows for easier representation of base constraints and the propagation of the generated streamliner constraints. When combining this heuristic set with the additional modelling choices for the streamliners, there are five models presented in the final portfolio. Although the number of improved test instances are the same as in the single-model search, the solving time reduction is increased by 4.7% for the VBS and by 18.9% for Autofolio.

9.5.4 Transshipment

For Transshipment the DEFAULT heuristic performs best across the streamliner lattice. The reason for this is that the other models used in the portfolio have quite a verbose encoding, especially with the addition of streamlining constraints, which increases the time taken to refine the ESSENCE and for SAVILE ROW to prepare the resulting model for input to a constraint solver. DEFAULT favours the least verbose option for each modelling choice and hence avoids this expense. For instance in the ESSENCE specification there are two *function* decision variables. As the functions are not known to be *total* the DEFAULT model represents each function as two independent matrices: one Boolean matrix which contains the domain of the function and another integer matrix which contains the values of the function. An alternative model in the portfolio represents these same decision variables using a representation which will encode these functions as a *relation*. This representation creates 3 independent matrices to contain the values of the relation each with a defined max size of `int(1..n_warehouses * n_transshipment * maxStock)`. Not only can the size of the these matrices become prohibitive for larger Transshipment instances but also the base model constraints are much more verbose for this representation.

9.6 Summary

We have demonstrated that searching for the right model to pair with a streamliner can improve performance significantly over a fixed, default model. Racing can be used to constrain the computational search cost by removing inferior models early. Future work is to investigate additional modelling choices such as when and how it is beneficial to encode streamliners to different representations such as SAT or SMT.

10

Chapter Ten

Conclusions

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 conjectures as to good streamliners, and then testing their efficacy in practice. In this work we have presented a 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.

In Chapter 3 we looked at the method used to generate streamlined models automatically from an Essence specification. We discussed the advantages of situating this system in Essence, a high level modeling language, such as the ability to exploit the structure present within the model, the ability to target multiple model formulations from the same base specification and automated symmetry breaking. An overview of the process and description of the rules embedded with Conjure with an example walkthrough of how a streamlined model is generated was shown.

In Chapter 4 we discussed the need for training instances from the problem class under evaluation in order to construct high quality streamliner portfolios. The process by which training instances are automatically generated from the Essence specification of a problem class and the requirements that these instances must meet was shown. An analysis of the *"footprint"* that different streamliners have in the feature space was presented and how performance similarity between similar instances can be used to construct a compressed representative training distribution.

In Chapter 5 we initially discussed the concept introduced first by Lebras et al in their work on double-wheel graphs on the process of combining streamliner conjectures. The benefits that this can bring in solving difficult problems was discussed as well as the adverse effect that it can have on the complexity of search. The structuring of the possible combinations into a lattice structure and the use of several pruning methods to reduce the complexity and remove ineffective combinations were defined. Various approaches for searching through the lattice structure and producing a portfolio of streamliner combinations were discussed and the search algorithm utilized in this thesis defined.

Chapter 6 initially shows how the performance characteristics of a streamliner conjecture can vary drastically between solving paradigms and why it is necessary to perform paradigm specific streamliner search and generate independent streamliner portfolios. The constructed portfolios generated through the search method are then summarized; in particular the visualization of the distribution and size of the generated pareto fronts. For a subset of the problems, a more in-depth analysis was performed to discuss interesting properties of the constructed portfolios and analyze the performance of the constituent conjectures. Lastly, the problem of *Streamliner Selection* was introduced which deals with selecting from the portfolio an effective streamliner for an unseen instance and various uninformed and informed methods for solving this problem were discussed.

In Chapter 7 two sets of experiments were presented designed to test the efficacy of our automated streamlining approach. The first analyzed the frequency with which streamlining results in a reduction in search, and the magnitude of that reduction. Due to the nature of imperfect streamliner selection the second experiment aimed to provide a more practical setting in which the overall impact of streamlining across an entire instance distribution was analyzed. Results across two unseen instance distributions with varying solving difficulty were shown.

In Chapter 8 the process of streamlining Constrained Optimization Problems and the required adaptations to the base search and scheduling methods were discussed. Empirical results were shown demonstrating drastic improvements in both time to optimality and time to proof for all problems evaluated.

Chapter 9 explored the effect of model selection in the context of streamlined specifications. The augmentations to the best-first search method to evaluate at each lattice node a portfolio of models to search and explore the variability in performance and find the optimal model were discussed. Various forms of racing to constrain the computational cost of training were introduced. Empirical results demonstrated drastic improvements in solving time for some problem classes in comparison to a single-model approach.

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 conjectures as to good streamliners, and then testing their efficacy in practice. This work has 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. The empirical results demonstrate the success of this approach. In conclusion this thesis meets its original goal outlined in Chapter 1 to: *completely automate the process of generating effective streamlined models to produce a substantial reduction in search effort across a diverse range of problems.*

10.1 Future Work

10.1.1 Online Learning

One of the pitfalls of the Streamliner Selection system presented in this thesis Chapter 6 is that the learning procedure is static. For instance, for Autofolio a prediction model is learnt based upon the performance of the streamliners in the portfolio across the training distribution and that model is never updated. This means that if for a particular part of the instance space the generalizations learnt by Autofolio are ineffective the imperfect predictions made by the model will never be adjusted. The reason for this limitation in the thesis is that Autofolio is a supervised learning system and so to perform online learning after each instance would require complete information on how each streamliner from the portfolio performs. This is not possible as in practise we only know how the selected streamliner performed.

An avenue of future research could be to investigate leveraging an online reinforcement learning algorithm to update the predictions of the model based upon the reward of the chosen streamliner. In our portfolio setting, this reward could be defined as the speedup achieved relative to the *original* model.

10.1.2 Performance Based Selection

Currently selection is a black box system where for a given instance the streamlined model is invoked and only the output is inspected to see whether the model was *satisfiable,* *unsatisfiable* or *timed out*. We saw in Section 7.4 that streamliners can have a large impact not only on the complexity of the model but also how the solver performs. A potential avenue of research would be to investigate whether some dynamic analysis of how the streamlined conjectures are performing in the solver could be used to predict whether or not the chosen streamliner is going to be effective. A potential example could be: if for the given instance the streamlined model is so complex that it adversely affects the performance of the solver, search could be stopped prematurely to evaluate a different model.

10.1.3 Streamlining from the other end: A Constraint Acquisition Approach

In this thesis the streamliners are built from the structure and types present in the Essence specification of the problem. One of the pitfalls of this is that many of these generated conjectures may not be *satisfiable* and a great amount of resource can be spent evaluating and removing these invalid conjectures. Another approach that could be investigated would be to take an approach more similar to that done initially by Gomes et al where streamliners are generated instead from the solutions of the problem.

Here an Essence specification would be provided and instances of the problem automatically generated as done in this thesis. However, instead of generating and evaluating conjectures from the specification the instances would be unaltered and solved to provide a set of solutions. Similar solutions could be grouped together and from here, the system would use a process similar to that of constraint acquisition where the goal would be to induce a constraint network that uses combinations of constraints from the language and that is consistent with the solutions [24]. These different constraint networks generated based upon different sets of solutions could be thought of as different streamlined models. The advantage here is that these models are generated directly from the solutions and so we know that they represent a valid structural regularity within the problem space.

10.1.4 Harder instances for portfolio construction

The approach detailed within this thesis attempts to limit training time by restricting the set of training instances used to those that can be solved within five minutes of compute time. A problem however may be that the patterns identified by the streamliner conjectures may not exist or be as effective in the more difficult instance distributions.

Another approach would be to utilize a training distribution of a similar character to the instances that are being solved to investigate whether there is better generalization from training to test.

10.1.5 Generic Portfolio Prediction

The approach detailed in this thesis is a compute intensive process. For each problem the set of conjectures is generated from the ESSENCE specification and search is invoked to build the streamliner portfolio. Currently each search is independent and no knowledge is shared or reused amongst problems. Even though the cost is amortized across the entire instance space it is acknowledged that this is a large upfront cost for each problem that a practitioner wants to solve.

An interesting avenue of research would be to investigate the concept of building a generic streamliner prediction model. If a practitioner did not want to spend the upfront training cost they could leverage this generic model to predict a portfolio of streamliners for their current problem. This prediction model would operate upon features generated and extracted from the high-level ESSENCE specification. An ESSENCE specification contains a wealth of data: the types of the decision variables, the types of the *givens* as well as all of the constraints posted on those variables.

Using a variety of diverse problems, with diverse type information, one could build (using Machine or Deep Learning methodologies) a generic model that could with limited cost predict potentially powerful portfolios on unseen problem classes.

10.1.6 Solution Counting in streamlined subspaces

Currently during evaluation of a streamliner on a given instance the solver will return once a solution in the streamlined subspace has been found. Instead of stopping on first solution, solution counting heuristics [86, 160] could be employed to try and estimate the number of solutions in the streamlined subspace. These counts could be used to develop more robust search heuristics in the streamliner lattice.

Bibliography

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

[2] Özgür Akgün, Nguyen Dang, Ian Miguel, András Z Salamon, and Christopher Stone. “Instance generation via generator instances”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2019, pp. 3–19.

[3] Özgür Akgün, Alan M Frisch, Ian P Gent, Bilal Syed Hussain, Christopher Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale. “Automated symmetry breaking and model selection in Conjure”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2013, pp. 107–116.

[4] Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. “Breaking Conditional Symmetry in Automated Constraint Modelling with Conjure.” In: *ECAI*. 2014, pp. 3–8.

[5] Özgür Akgün and Ian Miguel. “Modelling Langford’s Problem: A Viewpoint for Search”. In: *Proceedings of the 17th International Workshop on Reformulating Constraint Satisfaction Problems*. 2018.

[6] Özgür Akgün, Ian Miguel, Christopher Jefferson, Alan M. Frisch, and Brahim Hnich. “Extensible Automated Constraint Modelling”. In: *AAAI-11: Twenty-Fifth Conference on Artificial Intelligence*. 2011.

[7] Özgür Akgün. *CSPLib Problem 132: A Layout Problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob132`.

[8] Özgür Akgün. *CSPLib Problem 83: Transshipment Problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob083`.

[9] Özgür Akgün and Zeynep Kiziltan. *CSPLib Problem 131: Production Line Sequencing*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob131`.

[10] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. “An Enhanced Features Extractor for a Portfolio of Constraint Solvers”. In: *SAC 2014: Proceedings of the 29th Annual ACM Symposium on Applied Computing*. Code available from `https://github.com/CP-Unibo/mzn2feat`. Gyeongju, Republic of Korea: ACM, 2014, pp. 1357–1359. ISBN: 978-1-4503-2469-4. DOI: `10.1145/2554850.2555114`.

[11] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. “SUNNY-CP: a sequential CP portfolio solver”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 2015, pp. 1861–1867.

[12] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. “Finite-time analysis of the multiarmed bandit problem”. In: *Machine learning* 47.2-3 (2002), pp. 235–256.

[13] Jacqueline Ayel and Odile Favaron. *The helms are graceful.* Université Paris-Sud, Centre d’Orsay, Laboratoire de recherche en Informatique, 1981.

[14] Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. “The modelling language Zinc”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2006, pp. 700–705.

[15] Philippe Baptiste, Philippe Laborie, Claude Le Pape, and Wim Nuijten. “Constraint-based scheduling and planning”. In: *Foundations of artificial intelligence*. Vol. 2. Elsevier, 2006, pp. 761–799.

[16] Nicolas Beldiceanu, Mats Carlsson, and Jean-Xavier Rampon. *Global constraint catalog.* 2010.

[17] Nicolas Beldiceanu and Helmut Simonis. “A constraint seeker: Finding and ranking global constraints from examples”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2011, pp. 12–26.

[18] Nicolas Beldiceanu and Helmut Simonis. “A model seeker: Extracting global constraint models from positive examples”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2012, pp. 141–157.

[19] Christian Bessiere, Remi Coletta, Eugene C Freuder, and Barry O’Sullivan. “Leveraging the learning power of examples in automated constraint acquisition”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2004, pp. 123–137.

[20] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. “Constraint acquisition via partial queries”. In: *Twenty-Third International Joint Conference on Artificial Intelligence*. 2013.

[21] Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. "Acquiring constraint networks using a SAT-based version space algorithm". In: *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*. Vol. 21. 2. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. 2006, p. 1565.

[22] Christian Bessiere, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. "A SAT-based version space algorithm for acquiring constraint satisfaction problems". In: *European Conference on Machine Learning*. Springer. 2005, pp. 23–34.

[23] Christian Bessiere, Remi Coletta, Barry O'Sullivan, Mathias Paulin, et al. "Query-Driven Constraint Acquisition." In: *IJCAI*. Vol. 7. 2007, pp. 50–55.

[24] Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. "Constraint acquisition". In: *Artificial Intelligence* 244 (2017), pp. 315–342.

[25] Christian Bessiere, Frédéric Koriche, Nadjib Lazaar, and Barry O'Sullivan. "Constraint acquisition". In: *Artificial Intelligence* 244 (2017). Combining Constraint Solving with Mining and Learning, pp. 315–342. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2015.08.001. URL: https://www.sciencedirect.com/science/article/pii/S0004370215001162.

[26] Armin Biere. "Lingeling Essentials, A Tutorial on Design and Implementation Aspects of the the SAT Solver Lingeling." In: *POS@ SAT* 27 (2014), p. 88.

[27] Mauro Birattari, Thomas Stützle, Luis Paquete, Klaus Varrentrapp, et al. "A Racing Algorithm for Configuring Metaheuristics." In: *Gecco*. Vol. 2. 2002. 2002.

[28] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, et al. "Aslib: A benchmark library for algorithm selection". In: *Artificial Intelligence* 237 (2016), pp. 41–58.

[29] Zdravko I Botev, Joseph F Grotowski, and Dirk P Kroese. "Kernel density estimation via diffusion". In: *The annals of Statistics* 38.5 (2010), pp. 2916–2957.

[30] Kenneth N Brown. "Loading supply vessels by forward checking and unenforced guillotine cuts". In: *17th Workshop of the UK Planning and Scheduling SIG*. 1998.

[31] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter I. Cowling, Stephen Tavener, Diego Perez, Spyridon Samothrakis, Simon Colton, and et al. "A survey of Monte Carlo tree search methods". In: *IEEE Transactions on Computational Intelligence and AI* (2012).

[32] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. “A survey of monte carlo tree search methods”. In: *IEEE Transactions on Computational Intelligence and AI in games* 4.1 (2012), pp. 1–43.

[33] David A. Burke and Kenneth N. Brown. *CSPLib Problem 048: Minimum Energy Broadcast (MEB)*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob048`.

[34] Leslie Pérez Cáceres, Manuel López-Ibáñez, Holger Hoos, and Thomas Stützle. “An experimental study of adaptive capping in irace”. In: *International Conference on Learning and Intelligent Optimization*. Springer. 2017, pp. 235–250.

[35] Tristan Cazenave and Nicolas Jouandeau. “On the parallelization of UCT”. In: 2007.

[36] John Charnley, Simon Colton, and Ian Miguel. “Automatic generation of implied constraints”. In: *ECAI*. Vol. 141. 2006, pp. 73–77.

[37] John Charnley, Simon Colton, and Ian Miguel. “Automatic generation of implied constraints”. In: *ECAI*. Vol. 141. 2006, pp. 73–77.

[38] BMW Cheng, Kenneth M. F. Choi, Jimmy Ho-Man Lee, and JCK Wu. “Increasing constraint propagation by redundant modeling: an experience report”. In: *Constraints* 4.2 (1999), pp. 167–192.

[39] Geoffrey Chu, Maria de la Banda, and Peter Stuckey. “Exploiting subproblem dominance in constraint programming”. In: *Constraints* 17.1 (2012). Code available from `https://github.com/chuffed/chuffed`, pp. 1–38. ISSN: 1383-7133. DOI: `10.1007/s10601-011-9112-9`.

[40] Geoffrey Chu and Peter J Stuckey. “A generic method for identifying and exploiting dominance relations”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2012, pp. 6–22.

[41] Geoffrey Chu and Peter J Stuckey. “Dominance breaking constraints”. In: *Constraints* 20.2 (2015), pp. 155–182.

[42] Geoffrey Chu and Peter J Stuckey. “Learning value heuristics for constraint programming”. In: *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*. Springer. 2015, pp. 108–123.

[43] Geoffrey Chu, PJ Stuckey, A Schutt, T Ehlers, G Gange, and K Francis. *Chuffed, a lazy clause generation solver, 2016.*

[44] David Cohen, Peter Jeavons, Christopher Jefferson, Karen E Petrie, and Barbara M Smith. “Symmetry definitions for constraint satisfaction problems”. In: *Constraints* 11.2-3 (2006), pp. 115–137.

[45] Charles J Colbourn. “Embedding partial Steiner triple systems is NP-complete”. In: *Journal of Combinatorial Theory, Series A* 35.1 (1983), pp. 100–105.

[46] Charles J Colbourn and Jeffrey H Dinitz. *The CRC handbook of combinatorial designs*. Vol. 5005. CRC Press, Boca Raton, FL, 2007.

[47] S. Colton and I. Miguel. “Constraint Generation via Automated Theory Formation”. In: *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming*. Ed. by T. Walsh. 2001, pp. 575–579.

[48] Simon Colton and Ian Miguel. “Constraint generation via automated theory formation”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2001, pp. 575–579.

[49] Rémi Coulom. “Efficient selectivity and backup operators in Monte-Carlo tree search”. In: *International conference on computers and games*. Springer. 2006, pp. 72–83.

[50] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. “Symmetry-breaking predicates for search problems”. In: *KR* 96.1996 (1996), pp. 148–159.

[51] Ewan Davidson, Özgür Akgün, Joan Espasa, and Peter Nightingale. “Effective Encodings of Constraint Programming Models to SMT”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2020, pp. 143–159.

[52] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. “Solving the Car-Sequencing Problem in Constraint Logic Programming.” In: *ECAI*. Vol. 88. 1988, pp. 290–295.

[53] Michael R. Dransfield, Victor W. Marek, and Mirosław Truszczyński. “Satisfiability and Computing van der Waerden Numbers”. In: *Theory and Applications of Satisfiability Testing*. Ed. by Enrico Giunchiglia and Armando Tacchella. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 1–13. ISBN: 978-3-540-24605-3.

[54] Gary Duncan and Ian Gent. *CSPLib Problem 012: Nonogram*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. http://www.csplib.org/Problems/prob012.

[55] Jean-Guillaume Fages and Charles Prud’Homme. “Making the first solution good!” In: *2017 IEEE 29th International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE. 2017, pp. 1073–1077.

[56] Hilmar Finnsson and Yngvi Björnsson. "Simulation-Based Approach to General Game Playing." In: *Aaai*. Vol. 8. 2008, pp. 259–264.

[57] Pierre Flener, Alan M Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. "Breaking row and column symmetries in matrix models". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2002, pp. 462–477.

[58] Pierre Flener, Justin Pearson, and Magnus Ågren. "Introducing ESRA, a relational language for modelling combinatorial problems". In: *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer. 2003, pp. 214–232.

[59] Robert Fourer, David M Gay, and Brian W Kernighan. "A modeling language for mathematical programming". In: *Management Science* 36.5 (1990), pp. 519–554.

[60] Eugene C Freuder. "In pursuit of the holy grail". In: *Constraints* 2.1 (1997), pp. 57–61.

[61] Eugene C Freuder. "Progress towards the holy grail". In: *Constraints* 23.2 (2018), pp. 158–171.

[62] A. M. Frisch, C. Jefferson, B. Martinez Hernandez, and I. Miguel. "The Rules of Constraint Modelling". In: *Proc. of the IJCAI 2005*. 2005, pp. 109–116.

[63] Alan Frisch, Chris Jefferson, and Ian Miguel. *CSPLib Problem 036: Fixed Length Error Correcting Codes*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob036`.

[64] Alan Frisch, Christopher Jefferson, Ian Miguel, and Toby Walsh. *CSPLib Problem 041: The n-Fractions Puzzle*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob041`.

[65] Alan M Frisch, Matthew Grum, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. "The Design of ESSENCE: A Constraint Language for Specifying Combinatorial Problems." In: *IJCAI*. Vol. 7. 2007, pp. 80–87.

[66] Alan M. Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. "Essence: A constraint language for specifying combinatorial problems". En. In: *Constraints 13(3)* (2008), pp. 268–306. ISSN: 1572-9354. URL: `http://dx.doi.org/10.1007/s10601-008-9047-y`.

[67] Alan M Frisch, Warwick Harvey, Chris Jefferson, Bernadette Martínez-Hernández, and Ian Miguel. "Essence: A constraint language for specifying combinatorial problems". In: *Constraints* 13.3 (2008), pp. 268–306.

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

[69] Alan M Frisch, Christopher Jefferson, and Ian Miguel. "Constraints for Breaking More Row and Column Symmetries". In: *Proceedings of 9th International Conference on Principles and Practice of Constraint Programming {CP-2003}*. Ed. by Francesca Rossi. Vol. 2833. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 318–332. ISBN: 978-3-540-20202-8. DOI: `10.1007/978-3-540-45193-8\_22`.

[70] Alan M Frisch, Christopher Jefferson, and Ian Miguel. "Symmetry breaking as a prelude to implied constraints: A constraint modelling pattern". In: *ECAI*. Vol. 16. 2004, p. 171.

[71] Alan M Frisch, Ian Miguel, and Toby Walsh. "CGRASS: A system for transforming constraint satisfaction problems". In: *Recent Advances in Constraints*. Springer, 2003, pp. 15–30.

[72] Alan M Frisch, Ian Miguel, and Toby Walsh. "Symmetry and implied constraints in the steel mill slab design problem". In: *Proc. CP01 Workshop on Modelling and Problem Formulation*. 2001.

[73] Roberto Frucht. "Graceful numbering of wheels and related graphs". In: *Annals of the New York Academy of Sciences* 319.1 (1979), pp. 219–229.

[74] Hiroshi Fujita, Miyuki Koshimura, and Ryuzo Hasegawa. "SCSat: a soft constraint guided SAT solver". In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2013, pp. 415–421.

[75] Philippe Galinier and Centre for Research on Transportation. *A constraint-based approach to the Golomb ruler problem*. Universite de Montreal, Centre de recherche sur les transports, 2003.

[76] Stuart Geman, Elie Bienenstock, and René Doursat. "Neural networks and the bias/variance dilemma". In: *Neural computation* 4.1 (1992), pp. 1–58.

[77] Michael Genesereth and Michael Thielscher. "General game playing". In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 8.2 (2014), pp. 1–229.

[78] I P Gent, I McDonald, and B M Smith. "Conditional Symmetry in the All-Interval Seris Problem". 2003.

[79] Ian P Gent. "Arc consistency in SAT". In: *ECAI*. Vol. 2. 2002, pp. 121–125.

[80] Ian P Gent. “Two results on car-sequencing problems”. In: *Report University of Strathclyde, APES-02-98* 7 (1998).

[81] Ian P Gent, Christopher Jefferson, and Ian Miguel. “Minion: A fast scalable constraint solver”. In: *ECAI*. Vol. 141. 2006, pp. 98–102.

[82] Ian P Gent, Karen E Petrie, and Jean-François Puget. “Symmetry in constraint programming”. In: *Foundations of Artificial Intelligence* 2 (2006), pp. 329–376.

[83] Ian P Gent and Barbara Smith. *Symmetry breaking during search in constraint programming*. Citeseer, 1999.

[84] Carla Gomes and Meinolf Sellmann. “Streamlined constraint reasoning”. In: *Principles and Practice of Constraint Programming - CP 2004*. Springer, 2004, pp. 274–289.

[85] Carla Gomes and Meinolf Sellmann. “Streamlined constraint reasoning”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2004, pp. 274–289.

[86] Carla P Gomes, Willem Jan van Hoeve, Ashish Sabharwal, and Bart Selman. “Counting CSP solutions using generalized XOR constraints”. In: *AAAI*. 2007, pp. 204–209.

[87] Carla P Gomes, Ashish Sabharwal, and Bart Selman. “Model counting: A new strategy for obtaining good bounds”. In: *AAAI*. 2006, pp. 54–61.

[88] Carla P Gomes and Bart Selman. “Algorithm portfolios”. In: *Artificial Intelligence* 126.1-2 (2001), pp. 43–62.

[89] Mattias Grönkvist. “A constraint programming model for tail assignment”. In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer. 2004, pp. 142–156.

[90] R. Guy. *Unsolved Problems in Number Theory*. Problem Books in Mathematics / Unsolved Problems in Intuitive Mathematics. Springer, 2004. ISBN: 9780387208602. URL: `http://books.google.co.uk/books?id=1AP2CEGxTkgC`.

[91] Greg Hamerly and Charles Elkan. “Learning the k in k-means”. In: *Advances in neural information processing systems*. 2004, pp. 281–288.

[92] Warwick Harvey. *CSPLib Problem 010: Social Golfers Problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob010`.

[93] Pascal Van Hentenryck. “Constraint and integer programming in OPL”. In: *INFORMS Journal on Computing* 14.4 (2002), pp. 345–372.

[94] Marijn Heule and Toby Walsh. “Symmetry within solutions”. In: *Proceedings of AAAI*. Vol. 10. 2010, pp. 77–82.

[95] Raymond R Hill and Charles H Reilly. “The effects of coefficient correlation structure in two-dimensional knapsack problems on solution procedure performance”. In: *Management Science* 46.2 (2000), pp. 302–317.

[96] Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. *CSPLib Problem 030: Balanced Academic Curriculum Problem (BACP)*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob030`.

[97] Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. “Modelling a Balanced Academic Curriculum Problem”. In: *CP-AI-OR-2002*. 2002, pp. 121–131.

[98] Brahim Hnich, Steven D Prestwich, Evgeny Selensky, and Barbara M Smith. “Constraint models for the covering test problem”. In: *Constraints* 11.2-3 (2006), pp. 199–219.

[99] John N Hooker. “Needed: An empirical science of algorithms”. In: *Operations Research* 42.2 (1994), pp. 201–212.

[100] John N Hooker. “Testing heuristics: We have it all wrong”. In: *Journal of heuristics* 1.1 (1995), pp. 33–42.

[101] Holger Hoos, Roland Kaminski, Marius Lindauer, and Torsten Schaub. “aspeed: Solver scheduling via answer set programming 1”. In: *Theory and Practice of Logic Programming* 15.1 (2015), pp. 117–142.

[102] Holger H Hoos. “Automated algorithm configuration and parameter tuning”. In: *Autonomous search*. Springer, 2011, pp. 37–71.

[103] Bruce Hoppe and Éva Tardos. “The quickest transshipment problem”. In: *Mathematics of Operations Research* 25.1 (2000), pp. 36–62.

[104] Sophie Huczynska, Paul McKay, Ian Miguel, and Peter 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.

[105] Bilal Syed Hussain. *CSPLib Problem 054: N-Queens*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob054`.

[106] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "Sequential model-based optimization for general algorithm configuration". In: *International conference on learning and intelligent optimization*. Springer. 2011, pp. 507–523.

[107] Frank Hutter, Holger H Hoos, Kevin Leyton-Brown, and Thomas Stützle. "ParamILS: an automatic algorithm configuration framework". In: *Journal of Artificial Intelligence Research* 36 (2009), pp. 267–306.

[108] Chris Jefferson, Angela Miguel, Ian Miguel, and Armagan Tarim. *CSPLib Problem 037: Peg Solitaire*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob037`.

[109] Christopher Jefferson. "Representations in constraint programming". PhD thesis. Citeseer, 2007.

[110] Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent, eds. *CSPLib: A problem library for constraints*. `http://www.csplib.org`. 1999.

[111] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. "Algorithm selection and scheduling". In: *International conference on principles and practice of constraint programming*. Springer. 2011, pp. 454–469.

[112] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. "ISAC: Instance-Specific Algorithm Configuration". In: *ECAI*. Vol. 215. 2010, pp. 751–756.

[113] Josef Kallrath. *Modeling languages in mathematical optimization*. Vol. 88. Springer Science & Business Media, 2013.

[114] Q-D Kang, Z-H Liang, Y-Z Gao, and G-H Yang. "On labeling of some graphs". In: *Journal of Combinatorial Mathematics and Combinatorial Computing* 22 (1996), pp. 193–210.

[115] Michael N Katehakis and Arthur F Veinott Jr. "The multi-armed bandit problem: decomposition and computation". In: *Mathematics of Operations Research* 12.2 (1987), pp. 262–268.

[116] Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. "Automated algorithm selection: Survey and perspectives". In: *Evolutionary computation* 27.1 (2019), pp. 3–45.

[117] Philip Kilby and Paul Shaw. "Vehicle routing". In: *Foundations of Artificial Intelligence*. Vol. 2. Elsevier, 2006, pp. 801–836.

[118] JooSeuk Kim and Clayton D Scott. "Robust kernel density estimation". In: *The Journal of Machine Learning Research* 13.1 (2012), pp. 2529–2565.

[119] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". In: *ECML*. LNCS 4212. Berlin, Germany: Springer, 2006, pp. 282–293. ISBN: 978-3-540-46056-5. DOI: 10.1007\/11871842_29.

[120] Ron Kohavi, David H Wolpert, et al. "Bias plus variance decomposition for zero-one loss functions". In: *ICML*. Vol. 96. 1996, pp. 275–83.

[121] Jelena Kojić. "Integer linear programming model for multidimensional two-way number partitioning problem". In: *Computers & Mathematics with Applications* 60.8 (2010), pp. 2302–2308.

[122] Janet Kolodner and Case-Based Reasoning. "Morgan Kaufmann Publishers". In: *San Mateo, CA* (1993).

[123] Lars Kotthoff. "Algorithm selection for combinatorial search problems: A survey". In: *Data Mining and Constraint Programming*. Springer, 2016, pp. 149–190.

[124] Michal Kouril and John Franco. "Resolution tunnels for improved SAT solver performance". In: *Theory and Applications of Satisfiability Testing*. Springer. 2005, pp. 143–157.

[125] Volodymyr Kuleshov and Doina Precup. "Algorithms for multi-armed bandit problems". In: *arXiv preprint arXiv:1402.6028* (2014).

[126] Michail G Lagoudakis and Michael L Littman. "Learning to select branching rules in the DPLL procedure for satisfiability". In: *Electronic Notes in Discrete Mathematics* 9 (2001), pp. 344–359.

[127] Tze Leung Lai and Herbert Robbins. "Asymptotically efficient adaptive allocation rules". In: *Advances in applied mathematics* 6.1 (1985), pp. 4–22.

[128] Ronan Le Bras, Carla P Gomes, and Bart Selman. "Double-wheel graphs are graceful". In: *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*. AAAI Press. 2013, pp. 587–593.

[129] Ronan Le Bras, Carla P. Gomes, and Bart Selman. "Double-wheel Graphs Are Graceful". In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*. IJCAI '13. Beijing, China: AAAI Press, 2013, pp. 587–593. ISBN: 978-1-57735-633-2. URL: http://dl.acm.org/citation.cfm?id=2540128.2540214.

[130] Ronan Le Bras, Carla P Gomes, and Bart Selman. "On the Erdos Discrepancy Problem". In: *Principles and Practice of Constraint Programming: 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*. Vol. 8656. Springer. 2014, p. 440.

[131] Kevin Leyton-Brown, Eugene Nudelman, Galen Andrew, Jim McFadden, Yoav Shoham, et al. “A portfolio approach to algorithm selection”. In: *IJCAI*. Vol. 3. 2003, pp. 1542–1543.

[132] Marius Lindauer, Rolf-David Bergdoll, and Frank Hutter. “An empirical study of per-instance algorithm scheduling”. In: *International Conference on Learning and Intelligent Optimization*. Springer. 2016, pp. 253–259.

[133] Marius Lindauer, Holger Hoos, Kevin Leyton-Brown, and Torsten Schaub. “Automatic construction of parallel portfolios via algorithm configuration”. In: *Artificial Intelligence* 244 (2017). Combining Constraint Solving with Mining and Learning, pp. 272–290. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/j.artint.2016.05.004`. URL: `https://www.sciencedirect.com/science/article/pii/S0004370216300625`.

[134] Marius Lindauer, Holger H Hoos, Frank Hutter, and Torsten Schaub. “Autofolio: An automatically configured algorithm selector”. In: *Journal of Artificial Intelligence Research* 53 (2015), pp. 745–778.

[135] Marius T Lindauer. “Algorithm selection, scheduling and configuration of Boolean constraint solvers”. PhD thesis. Universit "a t Potsdam, Institut f ü r Informatik, 2015.

[136] James Little, Cormac Gebruers, Derek G Bridge, and Eugene C Freuder. “Using case-based reasoning to write constraint programs”. In: *CP*. Citeseer. 2003, p. 983.

[137] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. “The irace package: Iterated racing for automatic algorithm configuration”. In: *Operations Research Perspectives* 3 (2016), pp. 43–58.

[138] KJ Ma and CJ Feng. “On the gracefulness of gear graphs”. In: *Math. Practice Theory* 4 (1984), pp. 72–73.

[139] Aditya Mahajan and Demosthenis Teneketzis. “Multi-armed bandit problems”. In: *Foundations and applications of sensor management*. Springer, 2008, pp. 121–151.

[140] Toni Mancini and Marco Cadoli. “Detecting and breaking symmetries by reasoning on problem specifications”. In: *International Symposium on Abstraction, Reformulation, and Approximation*. Springer. 2005, pp. 165–181.

[141] Oded Maron and Andrew W Moore. “The racing algorithm: Model selection for lazy learners”. In: *Lazy learning*. Springer, 1997, pp. 193–225.

[142] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J Stuckey, Maria Garcia De La Banda, and Mark Wallace. “The design of the Zinc modelling language”. In: *Constraints* 13.3 (2008), pp. 229–267.

[143] B Martínez-Hernández and Alan M Frisch. “Towards the systematic generation of channelling constraints”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2005, pp. 859–859.

[144] Christopher Mears, M Garcia De La Banda, and Mark Wallace. “On implementing symmetry detection”. In: *Constraints* 14.4 (2009), pp. 443–477.

[145] Christopher Mears and Maria Garcia De La Banda. “Towards automatic dominance breaking for constraint optimization problems”. In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015.

[146] P Meseguer and C Torras. “Exploiting Symmetries within Constraint Satisfaction Search”. In: *Artificial Intelligence* 129.1-2 (2001), pp. 133–163.

[147] Alice Miller and Patrick Prosser. *CSPLib Problem 050: Diamond-free Degree Sequences*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. http://www.csplib.org/Problems/prob050.

[148] Rémi Munos. “From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning”. In: *FTML* 7.1 (2014), pp. 1–129. ISSN: 1935-8237. DOI: 10.1561/2200000038.

[149] Brady Neal, Sarthak Mittal, Aristide Baratin, Vinayak Tantia, Matthew Scicluna, Simon Lacoste-Julien, and Ioannis Mitliagkas. “A modern take on the bias-variance tradeoff in neural networks”. In: *arXiv preprint arXiv:1810.08591* (2018).

[150] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. “MiniZinc: Towards a standard CP modelling language”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2007, pp. 529–543.

[151] Peter Nightingale. *CSPLib Problem 056: Synchronous Optical Networking (SONET) Problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. http://www.csplib.org/Problems/prob056.

[152] Peter Nightingale. *CSPLib Problem 057: Killer Sudoku*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. http://www.csplib.org/Problems/prob057.

[153] Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, and Ian 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.

[154] Peter Nightingale, Özgür Akgün, Ian P Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. "Automatically improving constraint models in Savile Row". In: *Artificial Intelligence* 251 (2017), pp. 35–61.

[155] Peter Nightingale and Andrea Rendl. "Essence'description". In: *arXiv preprint arXiv:1601.02865* (2016).

[156] Peter Nightingale, Patrick Spracklen, and Ian Miguel. "Automatically Improving SAT Encoding of Constraint Problems through Common Subexpression Elimination in Savile Row". In: *Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP 2015)*. Springer, 2015, pp. 330–340.

[157] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry 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.

[158] Anthony Palmieri and Guillaume Perez. "Objective as a Feature for Robust Search Strategies". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2018, pp. 328–344.

[159] Gilles Pesant. "Counting-based search for constraint optimization problems". In: *Thirtieth AAAI Conference on Artificial Intelligence*. 2016.

[160] Gilles Pesant. "Counting solutions of CSPs: A structural approach". In: *IJCAI*. Citeseer. 2005, pp. 260–265.

[161] Steven Prestwich. *CSPLib Problem 028: Balanced Incomplete Block Designs*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob028`.

[162] Steven Prestwich and J Christopher Beck. "Exploiting dominance in three symmetric problems". In: *Fourth international workshop on symmetry and constraint satisfaction problems*. Citeseer. 2004, pp. 63–70.

[163] Les Proll and Barbara Smith. "Integer linear programming and constraint programming approaches to a template design problem". In: *INFORMS Journal on Computing* 10.3 (1998), pp. 265–275.

[164] Jean-Francois Puget. "Constraint programming next challenge: Simplicity of use". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2004, pp. 5–8.

[165] Jean-Francois Puget. "On the satisfiability of symmetrical constrained satisfaction problems". In: *International Symposium on Methodologies for Intelligent Systems*. Springer. 1993, pp. 350–361.

[166] Reza Rafeh, Maria Garcia de la Banda, Kim Marriott, and Mark Wallace. "From Zinc to design model". In: *International Symposium on Practical Aspects of Declarative Languages*. Springer. 2007, pp. 215–229.

[167] Jean-Charles Régin. "Minimization of the number of breaks in sports scheduling problems using constraint programming". In: *DIMACS series in discrete mathematics and theoretical computer science* 57 (2001), pp. 115–130.

[168] Andrea Rendl. "Effective compilation of constraint models". PhD thesis. University of St Andrews, 2010.

[169] John R Rice. "The algorithm selection problem". In: *Advances in computers*. Vol. 15. Elsevier, 1976, pp. 65–118.

[170] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[171] Andrea Schaerf. "Scheduling sport tournaments using constraint logic programming". In: *Constraints* 4.1 (1999), pp. 43–65.

[172] Evgeny Selensky. *CSPLib Problem 045: The Covering Array Problem*. Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. http://www.csplib.org/Problems/prob045.

[173] Meinolf Sellmann and Warwick Harvey. "Heuristic constraint propagation–using local search for incomplete pruning and domain filtering of redundant constraints for the social golfer problem". In: *CPAIOR'02*. Citeseer. 2002.

[174] Helmut Simonis. "Sudoku as a constraint problem". In: *CP Workshop on modeling and reformulating Constraint Satisfaction Problems*. Vol. 12. Citeseer. 2005, pp. 13–27.

[175] Helmut Simonis. "The CHIP system and its applications". In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 1995, pp. 643–646.

[176] Helmut Simonis and Barry O'Sullivan. "Search strategies for rectangle packing". In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2008, pp. 52–66.

[177] Aleksandrs Slivkins. "Introduction to multi-armed bandits". In: *arXiv preprint arXiv:1904.07272* (2019).

[178] Barbara Smith. *CSPLib Problem 001: Car Sequencing.* Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob001`.

[179] Barbara M Smith. "Modelling a permutation problem". In: (2000).

[180] Barbara M Smith and Jean-François Puget. "Constraint models for graceful graphs". In: *Constraints* 15.1 (2010), pp. 64–92.

[181] Barbara M Smith, Kostas Stergiou, and Toby Walsh. "Using auxiliary variables and implied constraints to model non-binary problems". In: *AAAI/IAAI.* 2000, pp. 182–187.

[182] Casey Smith, Carla Gomes, and Cesar Fernandez. "Streamlining local search for spatially balanced latin squares". In: *IJCAI.* Vol. 5. Citeseer. 2005, pp. 1539–1541.

[183] Kate Smith-Miles, Davaatseren Baatar, Brendan Wreford, and Rhyd Lewis. "Towards objective measures of algorithm performance across instance space". In: *Computers & Operations Research* 45 (2014), pp. 12–24.

[184] Patrick Spracklen, Özgür Akgün, and Ian 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.

[185] Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. "Automatic Streamlining for Constrained Optimisation". In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2019, pp. 366–383.

[186] Patrick Spracklen, Nguyen Dang, Özgür Akgün, and Ian Miguel. "Towards Portfolios of Streamlined Constraint Models: A Case Study with the Balanced Academic Curriculum Problem". In: *arXiv preprint arXiv:2009.10152* (2020).

[187] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[188] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. "Compiling finite linear CSP into SAT". In: *Constraints* 14.2 (2009), pp. 254–272.

[189] Gecode Team. *Gecode: Generic constraint development environment, 2006.* 2008.

[190] George R Terrell and David W Scott. "Variable kernel density estimation". In: *The Annals of Statistics* (1992), pp. 1236–1265.

[191] Pascal Van Hentenryck, Laurent Michel, Laurent Perron, and J-C Régin. "Constraint programming in OPL". In: *International Conference on Principles and Practice of Declarative Programming.* Springer. 1999, pp. 98–116.

[192] Toby Walsh. *CSPLib Problem 013: Progressive Party Problem.* Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob013`.

[193] Toby Walsh. *CSPLib Problem 019: Magic Squares and Sequences.* Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob019`.

[194] Toby Walsh. *CSPLib Problem 023: Magic Hexagon* . Ed. by Christopher Jefferson, Ian Miguel, Brahim Hnich, Toby Walsh, and Ian P. Gent. `http://www.csplib.org/Problems/prob023`.

[195] Toby Walsh. "General symmetry breaking constraints". In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2006, pp. 650–664.

[196] Weijia Wang and Michèle Sebag. "Hypervolume indicator and dominance reward based multi-objective Monte-Carlo Tree Search". In: *Machine learning* 92.2-3 (2013), pp. 403–429.

[197] Hong Xu, Sven Koenig, and TK Satish Kumar. "Towards effective deep learning for constraint satisfaction problems". In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2018, pp. 588–597.

[198] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. "Hydra: Automatically configuring algorithms for portfolio-based selection". In: *Twenty-Fourth AAAI Conference on Artificial Intelligence.* 2010.

[199] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "SATzilla-07: The design and analysis of an algorithm portfolio for SAT". In: *International Conference on Principles and Practice of Constraint Programming.* Springer. 2007, pp. 712–727.

[200] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. "SATzilla: portfolio-based algorithm selection for SAT". In: *Journal of artificial intelligence research* 32 (2008), pp. 565–606.

[201] Lin Xu, Frank Hutter, Jonathan Shen, Holger H Hoos, and Kevin Leyton-Brown. “SATzilla2012: Improved algorithm selection based on cost-sensitive classification models”. In: *Proceedings of SAT Challenge* (2012), pp. 57–58.

[202] Hantao Zhang. “Specifying latin square problems in propositional logic”. In: *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos* (1997), pp. 115–146.

List of Figures