Automatic Core-Guided Reformulation via Constraint Explanation and Condition Learning

Kevin Leo[1], **Graeme Gange**[1], **Maria Garcia de la Banda**[1,2], **Mark Wallace**[1]

[1] Department of Data Science & AI (DSAI), Monash University, Australia
[2] ARC Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Australia
kevin.leo@monash.edu, grame.gange@monash.edu, maria.garciadelabanda@monash.edu, mark.wallace@monash.edu

Abstract

SAT and propagation solvers often underperform for optimisation models whose objective sums many single-variable terms. MaxSAT solvers avoid this by detecting and exploiting *cores*: subsets of these terms that cannot jointly take their lower bounds. Previous work demonstrated that manual analysis of cores can help define model reformulations likely to speed up solving for many model instances. This paper presents a method to automate this process. For each selected core the method identifies the instance constraints that caused it; infers the model constraints and parameters that explain how these instance constraints were formed; and learns the conditions that made those model constraints generate cores, while others did not. It then uses this information to reformulate the objective. The empirical evaluation shows this method can produce useful reformulations. Importantly, the method can be useful in other situations that require explaining a set of constraints.

1 Introduction

Combinatorial problems are often tackled using a modelling+solving approach, whose first step is to *model* the problem's parameters, variables, constraints and objective function (if any) using a modelling language such as AMPL (Fourer, Gay, and Kernighan 1987), OPL (Van Hentenryck 1999), Essence (Frisch et al. 2007) or MINIZINC (Nethercote et al. 2007). Each instantiation of the model parameters with input data yields a model *instance*, which is then compiled to the format required by the selected *solver* to find its solutions. This compilation step uses sophisticated methods to generate a *flattened* instance (often written in a leaner formalism such as Essence' (Rendl 2010) and FLATZINC (Nethercote et al. 2007)) that is no longer intuitive for humans but is efficient for the selected solver. This approach gives users expressive and intuitive languages to model their problems, and frees them from knowing how to best map models onto solving algorithms. Further, model-to-model transformation methods exist to improve a model for many/all its instances, rather than just the one being flattened (e.g., (Hentenryck et al. 2005; Charnley, Colton, and Miguel 2006; Mears et al. 2015; Leo et al. 2013)).

A promising model-to-model transformation method is that of (Leo et al. 2020), which takes advantage of advances made by Lazy Clause Generation (Ohrimenko, Stuckey, and Codish 2007) and MaxSAT *core-guided solvers* (Andres et al. 2012; Morgado, Dodaro, and Marques-Silva 2014) to improve a common class of models: those with an *additive* (or separable) objective function, i.e. a sum of *terms*, each with a single variable. Constraint Programming (CP) and SAT solvers often underperform for models of this class because sums do not yield much propagation. MaxSAT solvers can avoid this by detecting and exploiting *cores*: subsets of terms that cannot collectively take their lower bounds. (Leo et al. 2020)'s method uses information from these detected cores to identify terms that can yield better bounds when *grouped*. It then adds *new variables* that group those terms, and reformulates the objective to use those variables.

While the above method automatically identifies sets of objective terms that cannot collectively take their lower bounds, all remaining (and very challenging) steps were manual. These include identifying *which instance constraints blocked* the lower bounds from being assigned to those terms, and what *properties of the instance data caused* those constraints to be posted. Further, this knowledge needs to be lifted from the instance level to the *model level*. All these manual steps require a significant degree of knowledge of both the model and the underlying domain.

This paper closes the gap by showing how to automate each of these challenging stages, building up to a fully-automatic method that constructs reformulated models similar to those manually developed in (Leo et al. 2020).

2 Background

Constraints: A constraint optimisation model $M[\Delta]$ is a tuple $(X[\Delta], C[\Delta], D[\Delta], f[\Delta])$, where for every element δ of the model's *parameter space* Δ mapping each parameter to its value, $X[\delta]$ is a set of variables, $C[\delta]$ a set of constraints over $X[\delta]$, $D[\delta]$ a *domain* mapping each variable $x \in X[\delta]$ to set of values $D[\delta](x)$, and $f[\delta]$ an objective function over $X[\delta]$. $C[\delta]$ is logically interpreted as the conjunction of its elements, and $D[\delta](x)$ as the conjunction of unary constraints on $x \in X[\delta]$. Thus, $M[\delta]$ denotes instance $(X[\delta], C[\delta], D[\delta], f[\delta])$. A *literal* of $M[\delta]$ is a unary constraint whose variable is in $X[\delta]$. To solve instance $M[\delta]$, CP solvers first apply constraint propagation to reduce domain

$D[\delta]$ to $D'[\delta]$ by executing the *propagators* associated with the constraints in $C[\delta]$ until fixpoint. If $D'[\delta]$ is equivalent to *false* ($D'[\delta](x)$ is empty for some $x \in X[\delta]$), we say $M[\delta]$ fails. If $D'[\delta]$ is not equivalent to *false* and fixes all variables, we found a solution to $M[\delta]$. Otherwise, $M[\delta]$ is split into n sub-instances $M[\delta]_i \equiv (X[\delta], C[\delta] \wedge c_i, D'[\delta], f[\delta]), 1 \leq i \leq n$, where $C[\delta] \wedge D'[\delta] \Rightarrow (c_1 \vee c_2 \vee \ldots \vee c_n)$ and c_i are literals (the *decisions*). These sub-instances are then iteratively searched using traditional *branch-and-bound*.

Lazy Clause Generation (LCG): LCG solvers extend CP solvers by modifying their propagators to explain domain changes via literals of the form $x = d$, $x \neq d$, $x \geq d$, and $x \leq d$ for $d \in D[\delta](x)$. An inferred literal ℓ is explained as $S \rightarrow \ell$, where S is a set of literals (interpreted as a conjunction). For example, literal $y \neq 5$ inferred by the propagator of constraint $x \neq y$ given literal $x = 5$, is explained by $\{x = 5\} \rightarrow y \neq 5$. Each literal inferred when solving instance $M[\delta]$ is stored with its explanation, forming an *implication graph*. If failure is detected for sub-instance $M[\delta]_i$, LCG solvers use this graph to compute a clause L (or *nogood*): a disjunction of literals that holds under any solution of $M[\delta]$ but is inconsistent under $M[\delta]_i$. L is then added to $C[\delta]$ to avoid failing for the same reasons.

Core-Guided Optimisation: CP solvers underperform for additive objectives because the lower bound of any objective term, say ot_i of variable x_i for minimising function $f[\delta] \equiv ot_1 + \cdots + ot_n$, can often be achieved by increasing the value of others, and $f[\delta]$'s lower bound is inferred from those of its terms. Core-guided solvers avoid this by first fixing all terms to their lower bounds and then searching for a solution. If this succeeds, an optimum has been found. Otherwise, they return a *core*: a (hopefully small) subset of terms that cannot collectively take their lower bounds. They then update $f[\delta]$'s bound and adjust the lower bounds of the core terms. Finally they re-solve, iterating until a solution is found. Core-guided solvers differ in their handling of cores, term bounds, and $f[\delta]$. We assume they all return a set S that is empty if the current sub-instance $M[\delta]_i$ is satisfiable; and otherwise contains literals of the form $x \geq k$ where variable x appears in $f[\delta]$ and at least one literal holds. Extending LCG solvers to support this interface is straightforward.

The LCG core-guided solver GEAS (Gange et al. 2020) is used herein. It is based on OLL (Andres et al. 2012), which progressively reformulates $f[\delta]$ to use the cores: upon finding core S, OLL adds a new variable $p = \sum_{((x \geq k) \in S)} x$ to $M[\delta]$ (with lower bound up by at least 1) and rewrites $f[\delta]$ in terms of p. GEAS improves the basic OLL with stratification (Marques-Silva et al. 2011; Ansótegui et al. 2012), extracting cores on high-coefficient terms first; weight-aware core extraction (Berg and Järvisalo 2017), delaying adding new variables until no cores are found; and hardening (Ansótegui et al. 2012), upper-bound propagation on new variables. Since the value of k in $\forall (x \geq k) \in S$ is irrelevant to our method, we will refer to S as the *raw* core and instead use the set $\{x | (x \geq k) \in S\}$ as our core.

Paths: *Variable* and *constraint paths* (Leo and Tack 2017) assign a unique identifier to each variable and constraint in a flattened instance that connects them to the model's source code. They describe the path the compiler took when flattening those variable and instance constraints.

Minimum Unsatisfiable Subset (MUS): Given instance $M[\delta] \equiv (X[\delta], C[\delta], D[\delta], f[\delta])$ where $C[\delta]$ is unsatisfiable, the subset $C'[\delta] \subseteq C[\delta]$ is a MUS of $C[\delta]$ iff $C'[\delta]$ is unsatisfiable and removing any constraint from $C'[\delta]$ makes it satisfiable. Our method uses FINDMUS (Leo and Tack 2017) – a MUS enumeration tool available for MINIZINC– to find the MUSes associated to a core (or rather, to the nogood obtained by negating its raw core).

Running example: We use the running example of (Leo et al. 2020): the Resource-Constrained Project Scheduling Problem with Weighted Earliness and Tardiness cost, which schedules tasks of a given duration and desired start time, subject to precedence and cumulative resource constraints. Its aim is to find a schedule that minimises the weighted earliness and tardiness costs of tasks not completed by their desired times. The model $M[\Delta]$ used (`rcpsp-wet` in the MINIZINC benchmarks) has the following objective $f[\Delta]$:

```
objective = sum (i in Tasks)                    |Original
  % earliness cost
  deadline[i,2]*max(0,deadline[i,1]-s[i]) +
  % tardiness cost
  deadline[i,3]*max(0,s[i]-deadline[i,1]));
```

i.e., the sum of earliness and tardiness costs for every task `i` in input set `Tasks` where parameter `deadline[i,1]` is the desired start time for `i`, parameter `deadline[i,2]` (`deadline[i,3]`) is the cost per time unit for `i` to start before (after) its desired time, and variable `s[i]` represents `i`'s start time. For reasons of space, we will denote the terms `deadline[i,2]*max(0,deadline[i,1]-s[i])` and `deadline[i,3]*max(0,s[i]-deadline[i,1])` by `e(i)` and `t(i)`, respectively. Note that `|Original` is used to mark the code as part of the original model.

3 Method Overview

Our automated method follows the three main steps of (Leo et al. 2020) shown in Figure 1. While Step 1 was mostly automated in (Leo et al. 2020), the other two were manual. This section illustrates the main issues found when automating these two steps and how we tackled them, using as example the `rcpsp-wet` model $M[\Delta]$ instantiated with input data $\delta_1 \in \Delta$ from file `j30_1_3-wet.dzn` and $\delta_2 \in \Delta$ from `j30_43_10-wet.dzn`. Consider the four cores found by Step 1 shown in Figure 2: **`{e(16),t(25)}`**, **`{e(8),t(14)}`**, **`{e(17),t(27)}`**, and **`{e(21),t(8)}`**; the first three from instance $M[\delta_1]$, the last from $M[\delta_2]$.

Step 2 selects candidate cores by determining their cause. (Leo et al. 2020) do this by first grouping cores that follow identical-up-to-renaming patterns. It is easy to see (and automatically infer; see Step 2.1) that our four cores follow pattern **`{e(A),t(B)}`**, shown in the "Patterns" column in Figure 2. Automatically identifying the *cause* for the cores (and thus for the pattern) is more complex. We do this by using FINDMUS to find the instance constraints that generate the nogood associated to each core (see Step 2.2). As shown by the "Explanations" column labeled ① in Figure 2, this yields one instance constraint for each of the first two cores and two instance constraints for each of the last two cores. We then use constraint paths to automatically trace back all

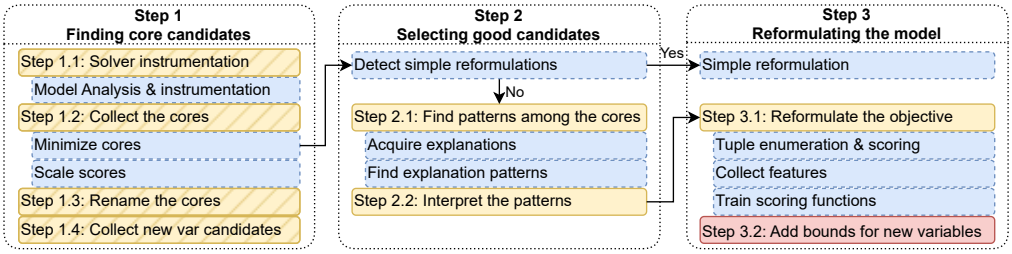

Figure 1: Method overview showing changes w.r.t. (Leo et al. 2020). Blue marks new, automated sub-steps; hatched yellow, old ones that were already automated but executed separately by hand; yellow (red), old ones that are now (still not) automated.

six instance constraints (arrow labeled ②) to the following model constraint (shown at the top of Figure 2), which ensures each task `i` finishes before its successor task `j` starts:

```
forall (i in Tasks, j in suc[i])    |Original
       (s[i]+d[i]<=s[j]);
```

where variable `s[i]` is as before, and parameters `d[i]` and `suc[i]` give `i`'s duration and successor set, resp. With this information, modellers should know that the first two cores are caused by a task and its successor (**16** and **25** in the first core; **8** and **14** in the second), while the last two are caused by a chain of three tasks that starts and ends with the tasks in the cores (**17** and **27** in the third core; **21** and **8** in the fourth) and has a middle task (24 and 12, resp). To automatically achieve this, our method first automatically annotates the model (see extra sub-step "Model analysis and instrumentation"of Step 1) to connect the data in the instance constraints to the parameters in the model constraints. This is used later (Step 2.2, shown here as an arrow labelled ③) to automatically generate *explanations* for each core (the column labelled "Generators"), and the associated *explanation patterns* (the two boxes outlined in orange coming from the arrows labeled ④). Importantly, if the cores of a core pattern are caused by different explanations (as is the case with **{e(A),t(B)}**), they should be treated differently.

Once explanation patterns for all cores are found, Step 3 reformulates the model. Automating this is also complex; it requires finding the conditions that make those explanations *constraining* enough to generate cores for those tasks but not for other tasks. Otherwise, we might group tasks in the

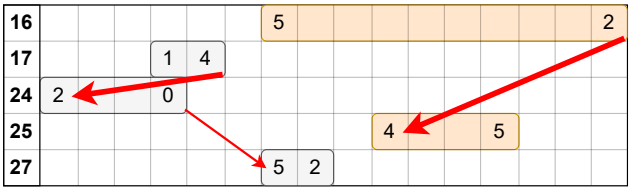

Figure 3: Part of a Gantt chart for instance $M[\delta_1]$

objective that do not generate cores. Consider the Gantt chart in Figure 3, which shows some of the tasks in $M[\delta_1]$, where the y axis shows the task number `i`, the x axis represents time, the lengths of the rectangles show the task durations `d[i]`, each task `i` appears at its desired start time `deadline[i,1]`, and arrows represent `suc[i]` dependencies.

The three grey tasks correspond to core **{e(17),t(27)}** while the two orange ones correspond to **{e(16),t(25)}**. However, many other tasks satisfy the same explanation patterns and generate no cores; mostly chains of two or three tasks. This is because the constraints posted by the instantiation of those explanations are not "tight" enough to fail and, thus, do not change the objective's bound (in contrast to those associated to cores). Modellers should be able to see that the explanation pattern of a chain of three tasks is constraining only if:

`deadline[A,1] + d[A] + d[C] > deadline[B,1],`

i.e., if the period of time between the ideal start times of tasks **A** and **B** is shorter than the sum of durations of tasks

```
constraint forall (i in Tasks, j in suc[i]) (s[i] + d[i] <= s[j])
```

Nogoods	Patterns	① Explanations	Generators	Explanation Patterns
e(16), t(25)	e(A), t(B)	s[16] + 8 <= s[25]	16 in Tasks, 25 in suc[16]	A in Tasks, B in suc[A]
e(8), t(14)	e(A), t(B)	s[8] + 7 <= s[14]	8 in Tasks, 14 in suc[8]	A in Tasks, B in suc[A]
e(17), t(27)	e(A), t(B)	s[17] + 2 <= s[24] s[24] + 4 <= s[27]	17 in Tasks, 24 in suc[17] 24 in Tasks, 27 in suc[24]	A in Tasks, C in suc[A] C in Tasks, B in suc[C]
e(21), t(8)	e(A), t(B)	s[21] + 8 <= s[12] s[12] + 8 <= s[8]	21 in Tasks, 12 in suc[21] 12 in Tasks, 8 in suc[12]	A in Tasks, C in suc[A] C in Tasks, B in suc[C]

Figure 2: Step 2 with cores **{e(16),t(25)}**, **{e(8),t(14)}**, **{e(17),t(27)}**, and **{e(21),t(8)}**.

`A` and `C`, implying that `A` and/or `B` must be scheduled earlier or later than their respective target times. To infer this automatically we must identify the properties (e.g., duration and desired start time) that might affect the tightness of the posted constraints. We can then use standard machine learning techniques (such as regression) to train predictors of nogood importance. Section 6 explains how we tackled this and Section 7 shows the associated results.

4 Step 1: Finding Core Candidates

Step 1 in (Leo et al. 2020) starts by executing the already instrumented GEAS solver (Step 1.1 in Figure 1) on each instance $M[\delta]$ to find the cores (Step 1.2). It then renames them to map the variables introduced by either the flattening process or the solver, back to those in the model $M[\Delta]$ (Step 1.3), and collects any core with more than one variable (Step 1.4). We extend these four sub-steps to perform three extra ones needed to automate later steps: *core minimisation*, *scoring*, and *model analysis & instrumentation*.

Core minimisation: The order in which core variables are introduced by solvers can yield cores with redundant terms, i.e., terms that if removed, do not affect the core's reduction in objective bound. This sub-step aims to find a minimal subset of the core terms that yields the same bound, as this can uncover patterns between cores with redundant terms and those without them. We do this by first computing the bound achieved by minimising the sum of the core's terms, and then using an adapted MUS enumeration strategy to find a minimal cardinality subset that yields the same bound. As this is expensive, our method can be asked to only analyse cores with high scores and/or a small number of terms, and can be terminated with a time-limit yielding the smallest core found so far. Core minimisation was not in (Leo et al. 2020) because that work focused on small, easy to understand cores that happened to be already minimal.

Scoring: Each core of $M[\delta]$ is assigned a *score* representing its effectiveness in solving $M[\delta]$. It is computed as its objective bound improvement divided by the sum of the improvements for all cores of $M[\delta]$. Importantly, the division allows us to compare the effectiveness of cores across instances, which is needed in Step 2.1 for scoring core patterns. Scoring was not part of (Leo et al. 2020) because that work assumed the most effective cores are those found early in the search, which is often the case for a single instance.

Model Analysis & Instrumentation: This sub-step modifies model $M[\Delta]$ to generate information used by Step 2 to find each core's explanation, e.g., to find (`17 in Tasks`,`24 in suc[17])`,(`27 in Tasks`,`27 in suc[24]`) in the upper right box of Figure 2 for core `{e(17),t(27)}`. This requires linking the instance constraints that caused the core (i.e., `s[17]+2<=s[24]`, `s[24]+4<=s[27]`) with the *generator conditions* of the model constraint `(i in Tasks, j in suc[i])` that, when instantiated with `(i=17,j=24)` and `(i=24,j=17)`, produced the instance constraints.

Information about generator conditions is lost during flattening due to, e.g., loop unrolling. To keep it, we automatically add to each constraint in $M[\Delta]$ annotations that describe the data dependencies used (and lost) when flattening its expressions. This is achieved by a depth-first traversal of the MINIZINC Abstract Syntax Tree (AST) that traverses conjunctions (`forall`, `/\`) and `if` constraints to collect their generator variables (e.g., the `i` and `j` above), data dependencies (`Tasks`, `suc[i]`), and any conditions in `if` or `where` expressions of an AST node. Boolean expressions are annotated with the collected data dependencies and either recursed upon (if one of the above) or backtracked over.

The generated code below shows the result of this process for the model constraint that yields core `{e(17),t(27)}`:

```
forall ( i in Tasks ) (                                    |Generated
 forall ( j in suc[i] ) (
    (s[i]+d[i]<=s[j])
      ::data(3, 5, "in", "j", "suc[i]")
      ::data(3, 4, "assign", "j", show(j))
  )::data(1, 2, "in", "i", "Tasks")
   ::data(1, 1, "assign", "i", show(i)));
```

where the `forall` expression has been split into two nested `forall`s (for `i` and for `j`) to illustrate the depth-first traversal. The first argument of each `data` annotation is the AST depth, used to alias generator variables correctly. The second is a counter used for ordering the annotations to simplify pattern matching. The third is the annotation *type*, which currently only includes: `in`, the index set of a generator variable (e.g., `i in Tasks`); `assign`, the value assigned to a generator variable (`i=1`); and `if`, the condition on which this constraint depends (none in this case). Note that `|Generated` is used to mark the code as generated MINIZINC code. During flattening, annotations are propagated down yielding FLATZINC constraints with instantiated data annotations that act as *path conditions*: the conjunction of constraints that allowed us to reach this program point. For example, annotation `data(1, 1, "assign", "i", show(i))` indicates that generator variable `i` took the value given by function `show(i)` when the instance constraint was flattened.

5 Step 2: Selecting Good Candidates

The first new sub-step of Step 2 (see Figure 1) bypasses the expensive parts of Steps 2 and 3 if it can detect cores that lead to simple, pre-determined reformulations. This sub-step is discussed in Appendix A; the remaining sub-steps here.

Step 2.1: Find Patterns Among the Cores

We automatically generate core patterns using the approach of (Zeighami et al. 2018) to compute a most specific generalisation (MSG) per subset of cores with similar terms. In doing this, we keep track of the mapping between each pattern and its cores, and of each pattern's score computed as the sum of those of its cores. For example, the MSG of cores `{e(21),t(8)}` and `{e(17),t(27)}` yields core pattern `{e(A),t(B)}` and mapping `{{A/21,B/8},{A/17,B/27}}`, stating they both contain terms of the form `e(A)` and `t(B)`, where pattern variables `A` and `B` take respectively values `21` and `8` in the first core, and `17` and `27` in the second. If each of these cores improves the objective by 3%, their pattern will have a score of 0.06. Currently, the MSG is approximated by first sorting the terms in every core and then assuming an ordered match between terms. Even then, this

step can be expensive if there are many cores. Importantly, when computing an MSG our system always checks whether simple pre-defined relationships occur between the instantiated pattern variables. In particular, it looks for $=, \neq, <, >$, and $= \pm 1$ relationships either between the variables themselves or when used as index into parameter arrays, such as `A=suc[B]+1`. We refer to these relationships as *facts* if they occur in all cores of a pattern, and use them in Step 3.

Step 2.2: Interpret the Patterns

As illustrated in Section 3, each identified core pattern needs to be interpreted by first acquiring explanations for its cores and then finding patterns among these explanations.

Acquire explanations: To acquire a core explanation, our method must find the instance constraints that caused the core, match them to the associated model constraints and use the instrumented version of these constraints to generate the explanation. Let c be the (raw) core we want to explain from flattened instance $M[\delta]$. Our method starts by negating c to obtain nogood $N = \neg c$ and construct the unsatisfiable instance $\varphi = M[\delta] \wedge N$. Then, it uses FINDMUS to identify a MUS in φ, i.e., a minimal subset of instance constraints that are incompatible with N and, thus, caused c. Currently, our method stops after finding the first MUS, as finding and later using more can be very time consuming (see next subsection). Consider, for example, the following output produced by GEAS for the flattened instance of model $M[\Delta]$ =`rcspc-wet` with data $\delta \in \Delta$ from `j30_1_3-wet.dzn`:

```
NEW BOUND: 63                                    |Output
 CORE: max(0, deadline[17,1]-s[17]) >= 3,
       max(0, s[27]-deadline[27,1]) >= 1
 VAR: x242 = max(0, deadline[17,1]-s[17])
           + max(0, s[27]-deadline[27,1])
NEW BOUND: 83
 CORE:   x242 >= 3
```

where the last line shows raw core $c \equiv$ `x242>=3` whose associated nogood is $N \equiv$ `x242<3`. Note that `|Output` is used to mark plain output. Applying the definition of variable `x242` yields raw core **`e(17)`**+**`t(27)`**`>=3` and associated nogood **`e(17)`**+**`t(27)`**`<3`. Applying FINDMUS to the result of appending N and its variables' definitions to the end of $M[\delta]$ yields a MUS combining N with 2 additional constraints: `s[17]+2<=s[24]` and `s[24]+4<=s[27]`. These are the instance constraints in Figure 2 (arrows labeled ①) that caused this core.

Once a MUS is found, our method uses constraint and variable paths to identify the model constraint from which each instance constraint in this MUS comes, and match its model variables with the instance ones. Then, thanks to the analysis and instrumentation of Step 1, we can substitute any occurrences of constants that match the pattern variables in the constraint and data annotations to generate the core explanation. For example, consider again the MUS with constraints `s[17]+2<=s[24]` and `s[24]+4<=s[27]`, in addition to nogood N. These two instance constraints are mapped to the model constraint shown in Section 3, and the instance variables (e.g., `s[27]`) to the model ones (`s[i]`). With the annotated constraint shown in Section 4, they are used to generate FLATZINC code similar to the following:

```
(s[24] - s[27] <= -4)                            |FlatZinc
  :: data("in", "j", "suc[i]")
  :: data("assign", "j", 27);
  :: data("in", "i", "Tasks")
  :: data("assign", "i", 24);
(s[17] - s[24] <= -2)
  :: data("in", "j", "suc[i]")
  :: data("assign", "j", 24)
  :: data("in", "i", "Tasks")
  :: data("assign", "i", 17);
```

which tells us **`17`** is the successor of `24`, and `24` of **`27`**. From this our method generates the explanations shown in the upper right blue box of Figure 2 for core **`{e(17),t(27)}`**.

Find Explanation Patterns: As shown in Figure 2, cores with the same pattern can have different explanations, which may require different model reformulations. Thus, we should only group cores with the same "kind" of explanation, i.e., the same explanation pattern. To do this, for each core pattern our method collects the explanations of all its cores. It then substitutes the core pattern variable mappings into the explanations. Finally, it computes the MSG of the resulting explanations. For example, consider again the cores **`{e(17),t(27)}`**, **`{e(21),t(8)}`**; pattern **`{e(A),t(B)}`** with mapping `{{`**`A/17,B/27`**`}, {`**`A/21,B/8`**`}}`; and their explanations shown in the upper and lower right blue boxes in Figure 2. Applying the mapping for core **`{e(17),t(27)}`** to its explanation generates:

```
( A in Tasks, 24 in suc[A])
(24 in Tasks, B in suc[24])
```

The same is generated for core **`{e(21),t(8)}`**with `12` rather than `24`. Thus, their MSG yields explanation pattern:

```
(A in Tasks, C in suc[A])
(C in Tasks, B in suc[C])
```

and mapping `{{`**`A/17,B/27`**`,C/24}, {`**`A/21,B/8`**`,C/12}}`. This automatically infers that these cores share an explanation pattern different (as shown in Figure 2) from that of cores **`{e(8),t(14)}`** and **`{e(16),t(25)}`**. Thus, the two kinds are later used separately. Note that with only one MUS per core, our method may miss some explanations. Exploring the accuracy/speed trade-off is interesting future work.

6 Step 3: Reformulating The Model

Step 3.1 Reformulate the Objective

As explained in Section 3, the method must now determine the conditions that make some instantiations of an explanation pattern yield constraints tight enough to fail creating a core that changes the objective's bound, while other instantiations do not. This section discusses how we achieve this.

Tuple Enumeration & Scoring: For each explanation pattern EP we perform three actions. First, we generate a new MINIZINC model $M_{EP}[\Delta]$ that enumerates all possible instantiations of EP for the parameter values of any $\delta \in \Delta$. For the EP in the previous section, this means adding:

```
forall(pA in Tasks, pC in suc[pA],          |Generated
       pB in suc[pC])
  (trace("Tuple: \(pA) \(pB) \(pC)\n"));
```

to an $M_{EP}[\Delta]$ containing the parameter definitions from $M[\Delta]$ (here, the `suc` array of successors). The `trace` function, which outputs information during compilation, will

output the values for `pA`, `pB`, and `pC`, i.e., the possible assignments to the pattern variables of EP. Thus, compiling each $M_{EP}[\delta]$ enumerates all possible tuples of **`A`**, **`B`** and `C`, and thus all instantiations of EP, in $M[\delta]$.

Second, we identify the types of terms in $M[\Delta]$'s objective $f[\Delta]$, and connect them to the cores in which they appear. Identifying term types involves traversing the AST to collect any top level definitions, including those under top level conjunctions. We start from the objective variable definition and expand any sum expressions, collecting generators (`i in Tasks`), coefficients (`deadline[i,3]`), and the term variables (`max(0,s[i]-deadline[i,1])` at the leaves of the tree, with their location. We identify as leaves any non-linear expression or call to a function outside $M[\Delta]$. We also collect conditions from generators, and enter branches of `if` statements. For `rcpsp-wet`, the two term types (i.e., earliness and tardiness) in the objective yield:

```
TERM TYPE 0:                                          |Output
  Loc : "rcpsp-wet_orig.mzn|122|26|122|54|"
  Gens: ["i in Tasks"]
  Coef: ["deadline[i,3]"]
  Var : "max(0,s[i]-deadline[i,1])"
TERM TYPE 1:
  Loc : "rcpsp-wet_orig.mzn|120|26|120|54|"
  Gens: ["i in Tasks"]
  Coef: ["deadline[i,2]"]
  Var : "max(0,deadline[i,1]-s[i])"
```

where each line of a term type shows its path: a string locating the term in $M[\Delta]$ as model-name|start-line|start-column|end-line|end-column|; the generator for the term variable (if any); the coefficient for the term (if any); and the actual variable. Connecting the term types to the core variables is done by matching the paths of the FLATZINC variables in the core with those of the term type to be used.

And third, we score each tuple generated by $M_{EP}[\delta]$ by solving $M[\delta]$, minimizing the penalty the terms of the core pattern can yield when instantiated to the tuple values. For example, for the above EP and tuple (**`A`**=**`17`**, **`B`**=**`27`**, `C=24`) the objective is to minimize:

```
objective =                                           |Generated
  deadline[17,2]*max(0,deadline[17,1]-s[17])
+ deadline[27,3]*max(0,s[27]-deadline[27,1])
```

This gives us an idea of how useful this grouping will be. Flattening for each possible group can be slow, so we modified GEAS to optimize arrays of objectives (thus flattening only once), and create arrays of at most $1,000$ objectives.

Collect Features: The training data must include model parameters likely to be useful features. Our method adds any parameters in the objective terms and in the constraints used to build the explanations. For our example EP, it adds `deadline[A,1]` and `deadline[B,1]` from the objective, and `d[A]` from the constraints used to build its explanations. It then generates for each EP an expression that outputs the values of these parameters when instantiated using the tuples found for that EP. For our example EP, it yields:

```
forall (                                              |Generated
  pA in Tasks, pB in suc[pA] where pA < pB)
  (trace("\(pA) \(pB) \(deadline[pA,1])
          \(deadline[pB,1]) \(d[pA])\n"))
```

where `pA < pB`, which was discovered when computing the MSG for EP, filters out spurious candidates.

Train scoring functions: We can now construct a training dataset for each EP. Consider, for brevity, the EP for a chain of two tasks (**`A`** `in Tasks,` **`B`** `in suc[`**`A`**`]`). The method generates features by instantiating the expression:
`A B` `deadline[`**`A`**`,1] deadline[`**`B`**`,1] d[`**`A`**`] isCore`
with the values of **`A`** and **`B`** from each tuple, and `isCore`, a 0/1 value indicating if this tuple incurred a penalty. The aim is to build a scoring function that can predict how useful grouping the terms of the cores explained by EP may be. Our implementation uses the `LinearSVC` regressor from Scikit-learn (Pedregosa et al. 2011) to learn the coefficients of a function that predicts `isCore`. A trained scoring function approximating the overlap between the target start time of tasks **`A`** and **`B`** above is:

```
function float: score_0(int: A,                       |Generated
                        int: B) =
  max([deadline[A,2], deadline[B,3]]) *
 (  0.4644 * deadline[A,1] +  0.4657* d[A]
 + -0.4621 * deadline[B,1] + -0.2326);
```

where the numbers in bold are the learned coefficients. The resulting sum is multiplied by the coefficients of the terms involved (`[deadline[A,2]` and `deadline[B,3]]`) to infer a score that approximates the penalty of moving these tasks. Code is then added to the model that uses these scoring functions to compute scores for all possible groupings. These scores are then used to decide which groups should be introduced in the final objective.

Grouping terms automatically: The last step adds the following code to the original MINIZINC model:

```
objective =                                           |Generated
  decompose_bottomup(get_order_array(...));
```

where function `decompose_bottomup` takes an array of elements $[x_1, \cdots, x_n]$, creates a new variable $z_i = x_{2i-1} + x_{2i}$ for each pair of adjacent elements, and recursively calls itself with the new array; and function `get_order_array` adds new variables by ranking candidate groupings and grouping those with the highest score. It outputs an array containing these new variables and any remaining un-grouped terms.

Step 3.2 Add Bounds for New Variables

While the reformulations from (Leo et al. 2020) were useful for LCG solvers, traditional CP solvers required the introduction of stronger upper bounds on new variables to improve performance. Our automated approach produces semantically equivalent objective functions, as the terms are simply reordered. Adding bounds to the introduced variables requires stronger reasoning since, if performed incorrectly, could yield incorrect solutions. This is not tackled yet.

7 Experimental Evaluation

This section evaluates the results of applying our automated method to the `rcpsp-wet` model. In doing this, it uses a data set of 12 instances, split into 6 easy (30 task), 3 medium (60 task), and 3 hard (90 task). All experiments were performed on an Intel Xeon 8260 CPU with 24 cores with 268.55GB of RAM, and using the GEAS solver in free-search mode with core-guided features disabled for solving. The results of

EMH:N	*T(s)*	*Cs*	*Es*	*EPs*	*PVs*	*S*	*F*
300:1	39.9	40	20	2	3	0.8	1.3
210:1	60.2	54	26	3	4	1.0	1.6
120:1	88.0	61	35	4	5	1.0	2.5
111:1	100.7	84	38	3	4	0.9	1.7
300:A	56.4	40	26	2	3	0.8	1.3
210:A	258.4	54	40	4	5	0.9	11.6
120:A	352.3	61	51	5	5	1.0	12.2
111:A	473.9	84	60	4	5	0.9	12.5

Table 1: Using different training sets for `rcpsp-wet`

Instance	*O*	*H*	*N*	*NW*	300:1	111:A
30_27_5	0.52	0.18	2.11	1.60	**0.15**	0.71
30_43_10	6.18	**0.67**	10.99	5.07	0.70	1.17
30_44_8	1.49	**0.24**	1.66	1.24	0.30	1.07
60_19_6	158.72	4.26	180.32	84.36	**0.79**	3.63
60_28_3	TO	**2.43**	TO	TO	3.86	5.72
60_36_8	TO	4.64	TO	TO	**4.60**	10.36
90_10_10	TO	TO	TO	TO	TO	TO
90_19_7	TO	TO	TO	TO	TO	TO
90_48_4	TO	TO	TO	TO	TO	TO

Table 2: Flatten and solve times for models of `rcpsp-wet`

applying our method to the other benchmarks in (Leo et al. 2020) are discussed in Appendix B.

Table 1 shows a summary of the impact of several training sets on the training process and on the resulting models. The *EMH:N* column shows the number of easy, medium, and hard instances used for training, followed by whether the resulting reformulation is based on all (:A) patterns or the highest scoring one (:1). Column T shows the training time; *Cs* the number of cores found in the selected instances; *Es* the number of explanations acquired; *EPs* the number of resulting EPs; and *LVs* the largest number of variables in a pattern. Each instance was solved twice per reformulation, and the ratio of the average solving and flattening times to those of the hand-reformulated model were computed. The *S* and *F* columns show the geometric means of these ratios. The choice of training set can significantly impact the resulting reformulation. Using the highest scoring pattern resulted in faster training and models that flatten faster than using all patterns. Interestingly, the solve times were not very different to the hand-written model.

Table 2 compares the performance of the original `rcpsp-wet` model (column *O*) with that of five reformulations for several instances. The first three come from (Leo et al. 2020): its hand-written direct model, which groups the earliness/tardiness terms of direct successor tasks that overlap based on the cost of enforcing their precedence (*H*); a naïve reformulation that groups terms in order of occurrence (*N*); and a "weighted" naïve reformulation that groups terms sorted by their coefficients (*NW*). The last two (300:1 and 111:A) are the best and worst performing reformulations from Table 1. A timeout of 600 seconds was used and represented by TO, underscored if the best objective value was found (but not proven optimal). Results for other reformulations and instances are presented in Appendix B. The results show that while the naïve reformulations are not faster than the original model, the performance of the reformulations produced by our method (300:1 and 111:A) is comparable to that of the hand-written model.

8 Interesting Applications of the Method

At its heart, our method first groups cores by their common explanation patterns (Step 2), and finds the conditions that make those explanations generate cores rather than feasible constraints (Step 3.1). While this is done for cores, the same can be done for any set of infeasible constraints.

A particularly interesting application is explainability. Consider answering query "why didn't you perform task X after task Y?" for an instance of `rcpsp-wet` by modifying it to force X be executed before Y. If it is is infeasible, a useful system will try to find MUSes to help users understand why. This would be easier if the explanation for each MUS (first part of Step 2.2) is also displayed. Further, often, the number of MUSes found is large enough to overwhelm users. This could be avoided if they can be grouped according to their explanation patterns (Step 2) reducing their number and, explaining the failure. Consider, for example, an instance of `rcpsp-wet` where the user asked to perform a task before five of its predecessors. It would be clearer for the user to get one MUS pattern and its explanation indicating that task needs to be after all its predecessors, than five MUSes each for one predecessor. Further, MUS enumeration could be sped up if patterns among MUSes could be detected and the search modified to forbid more MUSes that match the pattern. We will explore such applications in future work.

9 Conclusions and Future Work

This paper shows how to automate the process defined in (Leo et al. 2020) to use the cores inferred by a core-guided solver to reformulate a model in such a way it is likely to speed up solving for many model instances. To achieve this, the method instruments the model to find the constraints and associated explanations that caused each core, group cores that share the same explanation pattern, and train a scoring function to predict how useful it is to group the objective terms associated to the cores in that explanation pattern. The results of applying the method to the `rcpsp-wet` model show that the performance of the reformulated model (1) depends on the instances used to detect cores, find the associated explanation patterns and train the scoring function; and (2), in the case of `rcpsp-wet`, it is comparable to that of a hand-crafted reformulation.

Future work includes exploring several accuracy/speed trade-offs such as the use of more than one MUS per core (particularly disjoint ones), more complex MSGs, and different instance selection strategies. We also plan to explore other uses of the method, with special focus on applications related to constraint explainability. Finally, we would like to explore the relationship between the new variables introduced by our system with the theory of backdoors (Williams, Gomes, and Selman 2003).

Acknowledgements

This work was partly funded by Australian Research Council grant DP180100151. This material is based on research partially sponsored by the DARPA Assured Neuro Symbolic Learning and Reasoning (ANSR) program under award number FA8750-23-2-1016.

References

Andres, B.; Kaufmann, B.; Matheis, O.; and Schaub, T. 2012. Unsatisfiability-based optimization in clasp. In *Proc. ICLP Technical Communications*, volume 17 of *LIPIcs*, 211–221. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Ansótegui, C.; Bonet, M. L.; Gabàs, J.; and Levy, J. 2012. Improving SAT-Based Weighted MaxSAT Solvers. In *Proc. CP*, volume 7514 of *Lecture Notes in Computer Science*, 86–101. Springer.

Berg, J.; and Järvisalo, M. 2017. Weight-Aware Core Extraction in SAT-Based MaxSAT Solving. In *Proc CP*, volume 10416 of *Lecture Notes in Computer Science*, 652–670. Springer.

Charnley, J.; Colton, S.; and Miguel, I. 2006. Automatic Generation of Implied Constraints. In *Proceedings of the 2006 conference on ECAI 2006: 17th European Conference on Artificial Intelligence August 29 – September 1, 2006, Riva del Garda, Italy*, 73–77. Amsterdam, The Netherlands, The Netherlands: IOS Press. ISBN 1-58603-642-4.

Fourer, R.; Gay, D. M.; and Kernighan, B. W. 1987. *AMPL: A mathematical programming language*. AT & T Bell Laboratories Murray Hill, NJ 07974.

Frisch, A. M.; Grum, M.; Jefferson, C.; Martínez, B.; and Miguel, H. I. 2007. The design of ESSENCE: a constraint language for specifying combinatorial problems. In *IJCAI-07*, 80–87.

Gange, G.; Berg, J.; Demirović, E.; and Stuckey, P. J. 2020. Core-guided and Core-boosted Search for CP. In Hebrard, E.; and Musliu, N., eds., *Proceedings of Seventeenth International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming (CPAIOR2020)*, 205 – 221. Springer.

Hentenryck, P.; Flener, P.; Pearson, J.; and Ågren, M. 2005. Compositional Derivation of Symmetries for Constraint Satisfaction. In Zucker, J.-D.; and Saitta, L., eds., *Abstraction, Reformulation and Approximation*, volume 3607, chapter LNCS, 234–247. Springer Berlin Heidelberg. ISBN 978-3-540-27872-6.

Leo, K.; Gange, G.; de la Banda, M. G.; and Wallace, M. 2020. Core-Guided Model Reformulation. In Simonis, H., ed., *Principles and Practice of Constraint Programming*, 445–461. Cham: Springer International Publishing.

Leo, K.; Mears, C.; Tack, G.; and Banda, M. G. d. l. 2013. Globalizing Constraint Models. In Schulte, C., ed., *CP*, volume 8124 of *LNCS*, 432–447. Springer. ISBN 978-3-642-40626-3.

Leo, K.; and Tack, G. 2017. Debugging Unsatisfiable Constraint Models. In Salvagnin, D.; and Lombardi, M., eds., *CPAIOR 2017*, volume 10335 of *Lecture Notes in Computer Science*. Springer. ISBN 978-3-319-59775-1.

Marques-Silva, J.; Argelich, J.; Graça, A.; and Lynce, I. 2011. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, 62(3-4): 317–343.

Mears, C.; Garcia De La Banda, M.; Wallace, M.; and Demoen, B. 2015. A method for detecting symmetries in constraint models and its generalisation. *Constraints*, 20(2): 235–273.

Morgado, A.; Dodaro, C.; and Marques-Silva, J. 2014. Core-Guided MaxSAT with Soft Cardinality Constraints. In O'Sullivan, B., ed., *Principles and Practice of Constraint Programming*, 564–573. Cham: Springer International Publishing. ISBN 978-3-319-10428-7.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and Tack, G. 2007. MiniZinc: Towards a Standard CP Modelling Language. In Bessiere, C., ed., *CP*, volume 4741 of *LNCS*, 529–543. Springer. ISBN 978-3-540-74969-1.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2007. Propagation = Lazy Clause Generation. In Bessiere, C., ed., *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, 544–558. Springer. ISBN 978-3-540-74969-1.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; and Duchesnay, E. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12: 2825–2830.

Rendl, A. 2010. *Effective Compilation of Constraint Models*. Ph.D. thesis, Univ. of St Andrews.

Van Hentenryck, P. 1999. *The OPL Optimization Programming Language*. Cambridge, MA, USA: MIT Press. ISBN 0-262-72030-2.

Williams, R.; Gomes, C. P.; and Selman, B. 2003. Backdoors to Typical Case Complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, IJCAI'03, 1173–1178. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Zeighami, K.; Leo, K.; Tack, G.; and de la Banda, M. G. 2018. Towards Semi-Automatic Learning-Based Model Transformation. In Hooker, J. N., ed., *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, 403–419. Springer.