Core-Guided Model Reformulation[⋆]

Kevin Leo$^{[0000-0003-4720-4265]}$, Graeme Gange$^{[0000-0002-1354-431X]}$, Maria Garcia de la Banda$^{[0000-0002-6666-514X]}$, and Mark Wallace$^{[0000-0001-7326-8110]}$
{kevin.leo,graeme.gange,maria.garciadelabanda,mark.wallace}@monash.edu

Faculty of Information Technology, Monash University, Australia

Abstract. Constraint propagation and SAT solvers often underperform when dealing with optimisation problems that have an *additive* (or separable) objective function. The core-guided search introduced by MaxSAT solvers can overcome this weakness by detecting and exploiting *cores*: subsets of the objective components that cannot collectively take their lower bounds. This paper shows how to use the information collected during core-guided search, to reformulate the objective function for an entire *class of problems* (those captured by the problem *model*). The resulting (currently manual) method is examined on several case studies, with very promising results.

1 Introduction

Modern approaches for solving combinatorial optimisation problems first specify a *model* that formally describes the problem's parameters, variables, constraints and objective function. All parameters are later instantiated with input data, describing an *instance* of the problem. Each instance is then compiled to the format required by the selected *solver*, which explores the model's search space to find high quality solutions.

The economic impact of combinatorial optimisation problems has fuelled the design of powerful modelling languages, such as AMPL [9], OPL [23], Essence [10] and MINIZINC [18], and powerful solvers within the Mixed Integer Programming (MIP), Constraint Programming (CP), and MaxSAT solving paradigms. However, while there have been many advances in the variety and quality of solvers available, advances in technology that helps users improve their models have been scarce. This is unfortunate since, while the way in which a problem is modelled can significantly affect its solving time, designing good models is still very challenging, even for expert users. As a result, users must follow a time consuming, iterative, modify-and-test approach that can still yield poor results.

This paper aims at helping users identify model improvements, by taking advantage of some of the great advances achieved by Lazy Clause Generation (LCG) [19,8] and MaxSAT solvers. LCG solvers, such as Chuffed [6], GEAS [11] and ORTools [20], combine the strengths of the CP and SAT solving paradigms. This allows LCG solvers to infer *nogoods* (i.e., reasons for failure), and use them to avoid repeatedly exploring infeasible subproblems. Previous work [24] showed that the nogoods inferred by LCG solvers for a model's instances can be used to identify (a) existing constraints that may

⋆ Partly funded by Australian Research Council grant DP180100151.

be strengthened, and (b) new redundant constraints on the existing model variables that are likely to increase performance.

Our work complements this line of research by providing a (currently manual) method that uses the information inferred by the core-guided search of MaxSAT solvers, to improve a common class of optimisation models: those with an *additive* (or separable) objective function, i.e. a sum of components, each containing a single variable. In particular, we show how the cores found by a core-guided solver can help identify components in the model (rather than in the instance) that can yield better bounds when grouped. We then show how to use these components to reformulate the model itself, by adding *new variables* to the model. Note that most previous works on model reformulation, have only used variables appearing in the original model. One of the few exceptions is [5], which introduced new variables to achieve a lower computational complexity in handling the *SEQUENCE* constraint. While adding new variables to a model is very unusual and challenging, our experimental results indicate that, if done appropriately, it can significantly speed up LCG (and sometimes CP) solvers. In addition, we show how the first two steps of the method can be automated. Automating the last step is a challenging and important future goal.

2 Background

Constraint Optimisation Problems: A constraint optimisation problem P is a tuple (C, D, f), where C is a set of constraints, D a *domain* mapping each variable x appearing in C to set of values $D(x)$, and f an objective function. C is logically interpreted as the conjunction of its elements, and $D(x)$ as the conjunction of unary constraints on x. A *literal* of P is a unary constraint whose variable appears in C. To solve problem $P \equiv (C, D, f)$, a CP solver first applies constraint propagation to reduce domain D to D', by executing the propagator associated with the constraints in C until reaching a fixpoint. If D' is equivalent to *false* ($D'(x)$ is empty for some variable x), we say P fails. If D' is not equivalent to *false* and fixes all variables, we have found a solution to P. Otherwise, the solver splits P into n subproblems $P_i \equiv (C \wedge c_i, D', f), 1 \leq i \leq n$, where $C \wedge D' \Rightarrow (c_1 \vee c_2 \vee \ldots \vee c_n)$ and c_i are literals (the *decisions*), and then iteratively searches these subproblems.

The search proceeds propagating and making decisions until either (1) a solution is found, or (2) a failure is detected. In case (1) the search computes the value of f, constraints the next value of f to be better (greater or smaller, depending on f) and continues the search for this better value (the traditional *branch-and-bound*). In case (2) the search usually backtracks to a previous point to make a different decision.

Lazy Clause Generation: LCG solvers [19,8] extend CP solvers by instrumenting their propagators to explain domain changes in terms of *equality* ($x = d$ for $d \in D(x)$), *disequality* ($x \neq d$) or *inequality* ($x \geq d$ or $x \leq d$) literals. An *explanation* for literal ℓ is $S \rightarrow \ell$, where S is a set of literals (interpreted as a conjunction). For example, the explanation for the propagator of constraint $x \neq y$, which infers literal $y \neq 5$ given literal $x = 5$, is $\{x = 5\} \rightarrow y \neq 5$. Each literal inferred when solving problem P is recorded with its explanation, forming an *implication graph*. If failure is detected for subproblem P', LCG solvers use this graph to compute a clause L (or *nogood*): a

disjunction of literals that holds under any solution of P but is inconsistent under P'. [1] L is then added to P's constraints, to avoid failing for the same reasons.
Core-Guided Optimisation: CP solvers often underperform when proving optimality for additive objectives. This is because the lower bound of any objective component, say oc_i of variable x_i for minimising function $f \equiv oc_1 + \cdots + oc_n$, can often be achieved by sacrificing others, and f's lower bound is inferred only from the bounds of its components. Core-guided solvers overcome this weakness by first fixing all components 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 components that cannot collectively take their lower bounds. They then update f's bound *without* committing to *which* core component incurs the cost, and adjust the lower bounds of the components in the core. Finally they re-solve, repeating this process until a solution is found. Different core-guided solvers differ mostly in how the interaction between cores, component bounds, and the objective is handled. We assume such solvers return either an empty set, indicating the current subproblem is satisfiable, or a set S of literals of the form $x_i \geq k$ where variable x_i appears in the objective, indicating at least one literal in S must hold. Extending LCG solvers to support this interface is straightforward.

This paper uses the LCG, core-guided solver GEAS [11]. Its core-guided approach is based on the OLL [1] method, which progressively reformulates the objective to use the discovered cores: upon finding core S, OLL introduces a new variable $p = \sum S$ (with lower bound increased by at least 1), and rewrites the objective in terms of p. GEAS improves the basic OLL with stratification [17,2] (extracting cores on high-coefficient terms first), weight-aware core extraction [4] (delaying the introduction of new variables until no cores can be found), and the 'hardening rule' [2] (upper-bound propagation on new variables).

3 Motivation for Core-guided Model Reformulation

The reformulation of the objective performed by GEAS when solving an instance, can significantly reduce the search and, thus, the solving time for both LCG and branch-and-bound solvers. This is somewhat counter-intuitive, as the reformulation introduces new instance variables. The reasons for such reduction are twofold. First, LCG solvers can use the new variables to learn nogoods that shorten their optimality proof. This shows the importance of what we call the *language of learning*. Second, branch-and-bound solvers can use the bounds on the new variables to detect failed subproblems earlier.

This paper aims to achieve similar improvements to those achieved by core-guided solvers, but at the model (rather than instance) level. Thus, performance can be improved for multiple instances (rather than only for the one being executed), and also for non-core-guided solvers (either because they are not available, or are not as fast for the instance in question). Let us demonstrate via two (extreme) examples the radical performance improvements made possible by reformulating the objective to use variables whose bounds are detected by core-guided search.

[1] Note our *nogood*s denote a positive (implied) clause. In other works they denote the (conjunctive) negation of its literals.

Example 1. Consider an optimisation problem with n pairs of variables, $x_i, y_i : i \in 1..n$, where each variable has $0..m$ domain, $\forall i \in 1..n : x_i + y_i \geq k$, and the objective is to minimise the sum of the variables ($\sum_{i=1}^{n} x_i + y_i$). With a CP or LCG solver, propagation ensures assignments to x_i and y_i are mutually consistent and, if the lower values in the domains are tried first by the search, the first solution to be found will be optimal. Given a direct model for this problem, the first row of the table below shows, for $n = 10$ and $m = 5$, the number of search steps required by CP solver Gecode [12] to find the first solution and prove its optimality, for several values of k.

Search steps with:	$k = 2$	$k = 3$	$k = 4$
original objective	21	73,955	11,163,595
reformulated objective	21	21	21

The last row shows the number of search steps required after (a) adding to the model new variables xy_i and constraints $xy_i = x_i + y_i, xy_i \geq k$ for each $i \in 1..n$, and (b) reformulating its objective as $\sum_{i=1}^{n} xy_i$. With this model reformulation, once Gecode finds the first solution and starts searching for a better (smaller) objective, the bounds on the xy_i variables allow it to immediately realise that no higher value is possible for any x_i or y_i. Thus, the search to prove optimality efficiently finishes right after the first solution is found, regardless of the value of k.□

Example 2. Consider now a problem with n decision variables $x_1 \ldots x_n$, with domain $0..1$. Each triple of variables $(x_i, x_j, x_k) : i > j > k$ has a target values (a_i, a_j, a_k) and the triple incurs a benefit of 1 if $x_i = a_i, x_j = a_j$ and $x_k = a_k$. The objective is the sum of these benefits. Given a direct model for this problem, the first row of the table below gives the average CPU time it took the Gecode, Chuffed and Gurobi [13] solvers to find an optimal solution and prove optimality for 10 randomly generated instances.

CPU time (secs)	Gecode 6.1.1	Chuffed 0.10.4	Gurobi 7.5.2
original objective	57	236	232
reformulated objective	4	3	152

The last row shows the time required after (a) adding to the model a new variable xa_i for each decision variable x_i, and (b) reformulating its objective as $\sum_i xa_i$. To define xa_i, let tv_1 be the first variable in the t^{th} triple; ta_1 be its first matching value; val_t be the value of this triple in an assignment; and $xa_i(b)$ be $\sum_{t:tv_1=x_i \wedge ta_1=b} val_t$. Then $xa_i = max(xa_i(1), xa_i(0))$.
□

4 Methodology

The examples presented in the previous section clearly show the potential benefits of introducing new variables to a model. However, picking an effective reformulation is very difficult, since the number of possible new variables is huge. This is true even for our reduced scope, where these new variables must be formed from any (iterative) combination of the variables in the objective. The challenge then is to chose those variables that will achieve good improvements in a large number of instances.

Our core-guided reformulation method is designed to address this challenge. To achieve this, it takes two inputs: an optimisation model whose objective function is additive, and a set of input data files. It then performs the following main steps:

1. Use a core-guided solver to find, for each model instance, cores that are candidates for new variables;
2. Select some of these candidates, based on their likely reduction in solving time;
3. Modify the model to add new variables for the selected cores, constrain them, and use them to reformulate the objective, without changing the optimal solutions.

The rest of this section discusses the above three steps in more detail, using the Resource-Constrained Project Scheduling Problem with Weighted Earliness and Tardiness cost to illustrate the method. This problem tries to schedule tasks that have a given duration and desired start time, subject to precedence constraints and cumulative resource restrictions. The objective is to find an optimal schedule that minimises the weighted cost of the earliness and tardiness of any task that is not completed by their desired deadline.

The model we use (`rcpsp-wet` in the MiniZinc benchmarks) has the objective:

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

that is, the sum of the earliness and tardiness costs for every task `i` in input set of `Tasks`, where parameter `deadline[i,1]` gives the desired start time for i, parameters `deadline [i,2]` and `deadline[i,3]` give the cost per time unit for task `i` to start before or after its desired time, respectively, and variable `s[i]` represents the start time for task `i`.

4.1 Step 1: Finding core candidates

Step 1.1 Solver instrumentation: As we will see below, we currently find new variables by manually interpreting the cores found by the solver. Therefore, the solver needs to be instrumented to output them in human readable form. The Geas solver, which connects to the MiniZinc system and is the core-guided solver we use, already produces verbose output for debugging purposes. This includes, for each iteration in which the objective is modified, the value of the objective function at the end of each iteration, and all cores found together with their individual impact on the lower bound (for a minimising objective). While our current manual method simply uses this output, any future automation of the method will require a formal protocol for communicating with the core-guided solver, similar to that used in the profiling of CP solvers [21].

Step 1.2 Collect the cores: To collect the cores for a given problem, we run Geas in core-opt mode on a subset of the model instances we have, and record its verbose output. Currently, the subset selected corresponds to at most 2 small instances, as this will simplify the remaining manual steps. For the case of `rcpsp-wet`, we used the instances obtained by instantiating the model with data files `j30_1_3.dzn` and `j30_43_10.dzn`. Once this step is automated, better results will be obtained by using a large and diverse set of instances. Note that we disabled Geas core hardening for this step, as we do not want any literals (including those made false by hardening) to be omitted from the reported cores.

Step 1.3 Rename the cores: Solvers express cores in terms of the variables created when compiling the instance, or those created by the solver itself. These names are generic, making them difficult for humans to interpret. For example, the following shows an extract from the verbose output created by GEAS when solving the `j30_1_3.dzn` instance:

```
Found core of size 2, new lb: 5
CORE: X_INTRODUCED_261_ >= 7,
      X_INTRODUCED_217_ >= 1
```

indicating that the core has 2 literals, resulted in a new lower bound of 5 for the objective function, and contains instance variables `X_INTRODUCED_261_` and `X_INTRODUCED_217_`. Typically, solver writers who want to interpret such names, must examine the compiled instance output to see what these variables might refer to. In [24], the authors used a source map produced by the MINIZINC compiler, to map instance variables back to variables and expressions in the original model. Herein, we use the same method to link back the core variables, which for the above core results in:

```
Found core of size 2, new lb: 5
CORE: 'max(0, deadline[16, 1] - s[16])' >= 7,
      'max(0, s[25] - deadline[25, 1])' >= 1
```

The variables can now be easily recognised (from the objective) as the earliness of task 16, and the tardiness of task 25.

Step 1.4 Collect new variable core candidates: Cores containing more than one literal are candidates for new variables to be introduced in the model (singleton cores contain a variable that already exists in the model and are, therefore, not useful for our method). We collect all such cores by performing the previous two steps for the selected subset of model instances, and recording the results.

4.2 Step 2: Selecting good candidates

Step 2.1 Find patterns among the cores: Once all candidate cores are collected, the next main step in our method involves interpreting these cores to determine subsets that are likely to reduce solving time for many instances. To achieve this, we first try to find *patterns* among the different cores found. The following details three of the patterns we have often found in our experiments. Importantly, we focus on finding patterns for the most effective cores, i.e., those with greater impact on the objective function value and its lower bound. In GEAS, these are often the cores found early in the search.

Identical up to renaming: Many of the cores collected differ only in the name of the parameters present in the core's variables, and their bounds. For example, core:

```
Found core of size 2, new lb: 10
CORE: 'max(0, s[14] - deadline[14, 1])' >= 4,
      'max(0, deadline[8, 1] - s[8])' >= 1
```

and the one in Step 1.3, have two variables with pattern **max**`(0,deadline[i,1]-s[i])`, **max**`(0,s[j]-deadline[j,1])`, where `i` represents tasks 16 in the first core and 8 in the second, and `j` represents tasks 25 and 14. Note that we always ignore the literals' bound (e.g., 4 and 1 in the above core). While we currently find these patterns manually, the method described in [24] for finding nogood patterns across instances, can be easily adapted to cores.

Simple ordering: A simple but common pattern consists of pairs or triplets of variables that appear "near" each other in some ordering in the model. For example, variables representing the state of some object at time points t and $t + 1$, or variables representing two tasks where one is a successor of the other, as task 14 is of task 8 in the above core for `rcpsp-wet`.

Element constraints: Sometimes cores have literals assigning all (or most of) the possible values of a variable, e.g., $1 * (x = 1) + 5 * (x = 2) + 6 * (x = 3) \ldots$. This often occurs when the variable's contribution to the objective is non-linear. These cores reconstruct an `element` global constraint (see, for example, the reformulation for the `jp-encoding` model in Section 5.1).

Step 2.2 Interpret the patterns: We now look for reasons for the patterns to appear, that is, for the associated variables to appear often together in effective cores. This usually requires in-depth knowledge regarding the relationship between these variables. For example, for the pattern `max(0,deadline[i,1]-s[i])`, `max(0,s[j]-deadline[j,1])` mentioned above, we must understand what connects the earliness of the tasks represented by `i` (16 or 8) to the tardiness of those represented by `j` (25 or 14, respectively). Visualising the input data using a variation of a Gantt chart helped us realise, for example, that task `j` is often the direct successor of task `i`, they overlap in time, and have the highest earliness and tardiness costs. In other cores `j` is often a non-direct successor of `i`, and the penalty for scheduling the chain of tasks is also very high due to overlaps.

4.3 Step 3: Reformulating the model

Step 3.1 Reformulate the objective: Once the patterns are interpreted, we reformulate the objective using this information. The aim is to group objective components that are expected to form effective cores. We have observed that patterns often suggest an *ordering* of components that places them in cores together. For example, for `rcpsp-wet` we can sort the earliness/tardiness components of direct successor tasks that overlap, based on the cost of enforcing their precedence (we call this ordering *direct*), leaving the remaining components unchanged. Alternatively, we can use an ordering obtained by simply sorting the earliness/tardiness components based on the desired start time of their task (we call this ordering *start*).

For any such ordering of the components in the objective, our method recursively creates new variables for each disjoint pair of adjacent components, and replaces them in the objective function with the new variable. We achieve this by using the following group function, which we have implemented in MiniZinc and added to its library:

$$\mathsf{group}([x]) = x$$
$$\mathsf{group}([x_1, \ldots, x_{2n}]) = \mathsf{group}([z_1, \ldots, z_n])$$
$$\quad \mathbf{s.t.}\ z_i \geq x_{2i-1} + x_{2i},\ \forall.\, i \in 1 \ldots n$$
$$\mathsf{group}([x_1, \ldots, x_{2n}, x_{2n+1}]) = \mathsf{group}([z_1, \ldots, z_n, x_{2n+1}])$$
$$\quad \mathbf{s.t.}\ z_i \geq x_{2i-1} + x_{2i},\ \forall.\, i \in 1 \ldots n$$

The function receives as its argument a list with the objective components in the given ordering $[x_1, \ldots, x_m]$, and creates a new variable z_i for each pair of adjacent components $x_{2i-1} + x_{2i}$. It then recursively calls itself with a list of the new variables in the order they were created as input, while appropriately dealing with the case of m being odd or even. The recursion ends when the list contains a single component x, simply returning x as the new objective to minimize. Note that the best performance for GEAS occurs when z_i is bound from below. However, other solvers may perform better when z_i is defined as $z_i = x_{2i-1} + x_{2i}$. Therefore, in practice we use the bounding strategy that is best for a given solver. Note also that the group function uses recursion to combine more distant components. We do this to compensate for the locality brought by the fact we currently only consider the early cores of a very small selected subset of small instances, since these are the ones that are easier to interpret by hand. Once better support for the interpretation step is achieved, this should be reconsidered.

Our method adds to the model both the function that produces the ordering and the group function which uses it (in the case of the group function, this is done by adding it to the MINIZINC library, but it has the same effect). As a result, the time needed to compute this ordering is an overhead to the execution of the instance. Therefore, care needs to be taken when defining orderings that might be too time consuming to compute. The same can be said for the group function, although in this case reducing the time overhead might not be as important as reducing the number of variables created. We therefore experimented with a version of the group function that only performs k iterations, with the aim of introducing fewer new variables. The resulting models did not yield noticeable improvements in our experiments (data not shown).

Step 3.2 Add bounds for new variables: The reformulation of the objective can improve the solving time of any clause-learning solver (such as LCG and MaxSAT). This is because it introduces new variables that can be used by these solvers to learn new clauses and, thus, reduce the search space. However, the reformulation would not usually help CP solvers, as they will be unable to infer tighter bounds on the introduced variables.

To counter this, we modify group to add a bound to the new z_i variables it creates. For example, for `rcpsp-wet`, if the first iteration creates the following variable:

```
new_var = deadline[i,2]*max(0,deadline[i,1]-s[i]) +
          deadline[j,3]*max(0,s[j]-deadline[j,1])
```

we also add to the model the constraint:

```
new_var >= min(deadline[i,2],deadline[j,3])*
           (deadline[i,1]+d[i]-deadline[j,1]);
```

ensuring `new_var` is greater or equal than the minimum cost to enforce the precedence, that is, the minimum of the earliness cost of task `i` and the tardiness cost of task `j`, multiplied by their overlap (`d[i]` is a parameter of the model representing `i`'s duration).

4.4 Automating the method

The initial stages of our method (all substeps in step 1: finding core candidates) are automatic, thanks to the use of core-guided optimisation to generate the cores and their information, and the use of existing MINIZINC infrastructure to collect and rename the cores, and identify the core candidates. While substep 2.1 (finding patterns among the cores) is currently done manually, it can be automated using similar technology to that used by [24] to identify patterns among nogoods.

The most difficult manual stages to automate are the analytical ones: pattern interpretation (step 2.2), and designing the reformulation (step 3). As these rely on insights regarding the underlying model structure, full automation is quite challenging. However, it is possible to automate certain processes to make these stages easier. For example, interpreting the patterns requires understanding *why* the cores represented by the pattern hold. Since each such core typically only relates to a small fragment of the model, identifying this fragment can often immediately reveal the meaning of the core. And *this* identification is something that can be automated: given a model M and core C, we know $M \wedge \neg C$ is unsatisfiable. Thus, we can use tools such as FINDMUS [16] to identify a minimal unsatisfiable subset of M that causes the failure.

Example 3. Consider the following core which was part of the output created by GEAS for the `rcpsp-wet` model, with the instance obtained from data file `j30_1_3.dzn`.

```
CORE: 'max(0, s[27] - deadline[27, 1])' >= 3,
      'max(0, deadline[17, 1] - s[17])' >= 1
```

We update the model with name annotations that explain what each constraint means.

```
constraint forall ( i in Tasks, j in suc[i] ) (
  ( s[i] + d[i] <= s[j] )
    :: "Task \(i) must finish before task \(j) starts" );
```

We then add the negation of the core to the MINIZINC model as follows:

```
constraint :: "Core 5"
           not (    max(0, s[27] - deadline[27, 1]) >= 3
                 \/ max(0, deadline[17, 1] - s[17]) >= 1);
```

With just these modifications, FINDMUS is able to output the following:

```
MUS:  Core 5
      Task 17 must finish before task 24 starts
      Task 24 must finish before task 27 starts
```

Note that, while the core only mentions tasks 17 and 27, FINDMUS is able to identify the chain of precedence constraints required for explaining the negated core. This makes the task of interpreting the cores much easier.

Automating the model reformulation (step 3) is more involved, as it requires the design of higher-level groupings using the *reasons* for the cores found in step 2.2. However, as we will see in the next section, the reformulations we produce are usually structurally simple: either an ordering or hierarchical clustering, based on proximity with respect to numeric parameters (i.e. `jp-encoding` and `seat-moving`), or constraint structures (i.e. `spot5` and `rcpcsp-wet`). It is thus possible (although non-trivial) to use structural analysis methods, such as [15], to do this, since they are able to identify subsets of the model constraints that correspond to pre-defined constraint structures.

5 Experimental Evaluation

This section illustrates how to apply our core-guided model reformulation method to five models, and experimentally evaluates the efficiency of the reformulations obtained for different orderings.

5.1 Models and their reformulation

To evaluate the effectiveness of our method, we require models of optimisation problems with an additive objective function, and for which core-guided solvers can obtain better results than branch and bounds ones (otherwise, the method has no chance of success). Therefore, we selected the top five models in the MiniZinc annual competition [22], for which core-guided Geas performs drastically better on at least one instance, than branch-and-bound Geas. This yielded the `rcpsp-wet` model used above to illustrate our method, and the four models described below.

For each model, we selected 1–2 instances to analyze (typically, the smallest instance to identify cores, and a moderate one to check that the identified patterns reoccur). After modification, we evaluated the reformulated model over all instances from the challenge. The following describes how our core-guided reformulation method was applied to the other four selected models.

The `seat-moving` model: Given a set of seats and the people sitting in them (some may be empty), the problem is to find the minimum number of moves and time-steps needed to reach a target seating plan. Some people can swap seats with anyone in one move; the rest must first move to an empty seat to make way. The objective is defined as:

```
cost = sum(i in 1..MAX_STEP-1,p in 1..P)
         (person[i,p]!=person[i+1,p]);
objective = cost + step*MAX_STEP*P;
```

where variable `cost` counts, for each time-step `i` and person `p`, the seats where `p` is at time `i` but not at `i+1` (note that boolean `person[i,p]!=person[i+1,p]` is coerced to an integer), and variable `step` is the number of time-steps needed. Therefore, the objective sums all moves performed in every step by any person,

Studying the early cores Geas finds for instances `sm-10-12-00` and `sm-10-20-00`, we realised they contain the moves of a single person, rather than those of a time-step, which is how they appear in the sum that defines variable `cost`. Thus, we grouped the components using a *reverse* ordering that simply reverses the order of the sum indices:

```
cost = group(p in 1..P, i in 1..MAX_STEP-1)
         (person[i,p]!=person[i+1,p]);
```

In addition, we also added a (very weak) bound on the first set of new variables (i.e., those created in the first iteration of the group function), that ensures the number of moves for people not starting in their target seat is ≥ 1.

The `jp-encoding` model: The Japanese Encoding problem tries to find the most likely encoding used for each byte in a byte stream of encoded Japanese text, where multiple encodings may be used. The model considers the ASCII, EUC-JP, SJIS and UTF-8 encodings, each with a scoring table that maps each byte to its penalty score (based on likelihood) for that encoding, plus an "unknown" encoding with a large penalty. The objective to minimise is defined in the model as:

```
objective = 1000*n_unknown + sum(i in 1..len) (
   (encoding[i]==e_euc_jp)*eucjp_score[stream[i]+1]
 + (encoding[i]==e_sjis)*jis_score[stream[i]+1]
 + (encoding[i]==e_utf8)*utf8_score[stream[i]+1]);
```

that is, it sums the penalties (given by parameter tables `eujp_score`, `sjis_score` and `utf8_score`) for the encoding chosen by variable `encoding[i]` for each byte (represented by parameter `stream[i]`) in position `i` of the input stream. Note that the penalty is 0 for the ASCII encoding, and 1000 for an unknown encoding (in this case variable `n_unknown` has been incremented by 1).

Studying the early cores found by GEAS on `data200.dzn`, we realised they refer to all possible encodings of the byte in a given position (it must get *some* encoding). We thus used the `element` global constraint to create new variables `encoding_cost(i)`, representing the encoding penalty of position `i`:

```
function var int: encoding_cost(int: i) =
  array1d(0..4,[0,eucjp_score[stream[i]+1],
                sjis_score[stream[i]+1],
                utf8_score[stream[i]+1],1000])[encoding[i]];
objective = sum(i in 1..len)(encoding_cost(i));
```

We then re-applied our method to the reformulated model and realised that the new cores were *local*, i.e., involved `encoding_cost(i)` and either `encoding_cost(i+1)` or `encoding_cost(i+2)`. Thus, we defined a *local* ordering that simply sorted the encodings by position (which is the same as the original order in the model):

```
objective = group(i in 1..len)(encoding_cost(i));
```

The `rel2onto` model: The Relational-To-Ontology Mapping problem takes as input (a) an *alignment* graph formed by `cnodes`, corresponding to the ontology's classes, `dnodes` corresponding to the data properties of the classes, and weighted edges, (b) the set of attributes of a relational database, and (c) the set of `dnodes` each attribute might be matched to with a given cost. The problem is to find a single match for all attributes and a Steiner Tree for the alignment graph, such that the matched `dnode` of every attribute is in the tree, and the cost of the edges in the Steiner Tree and of the matched attributes is minimised. The objective to minimise is defined in the model as:

```
w = sum(i in edges)(es[i]) * ws[i]);
wm = sum(a in atts)(match_costs[a, match[a]]);
objective = w + wm;
```

that is, the sum `w` of the weight (given by parameter `ws[i]`) of every edge `i` in the graph that appears in the Steiner tree (true if variable `es[i]` holds), plus the sum `wm` of the cost for every attribute `a` of the match (given by variable `match[a]`).

When studying the cores inferred by GEAS for `5_5.dzn` and `5_28.dzn`, we discovered many cores involve two variables `es[i]` and `es[j]`, where `i` and `j` are edges of the alignment graph. Moreover, the cores were generated for attributes that could only be associated with two possible `dnodes`, indicating that, since each attribute must be in the matching, some edge adjacent to that attribute must be in the Steiner tree. Based on this, we constructed an *adjacent* ordering that groups edges associated with a given attribute. As this did not perform as well as hoped, we looked deeper into the cores and realised that, when an attribute's matches overlapped with a previous one, the solver would *merge* the old variable with the new adjacent edges. Thus, we encoded a similar iterative *merging* strategy: starting with each edge in a singleton partition, we repeatedly select a new attribute a, and introduce a fresh variable for the sum of all partitions containing edges adjacent to a. Once all attributes are processed, the objective sums the resulting cost variables.

The `spot5` model: The SPOT5 earth observation satellite management problem [3] tries to find a subset of a given set of photographs to take, given many different constraints, including minimum distance, non-overlapping, and recording capacity. The model encodes these constraints as a set of binary and ternary `table` constraints. The objective to minimise is defined in the model as:

```
objective = sum(j in 1..num_variables)(costs[j]*(p[j]=0));
```

which sums the cost (given by parameter `costs[j]`) of each given photograph `j` (`j in 1..num_variables`) that is *not* taken (given by variable `p[j]` having value 0).

While the table encoding of constraints makes interpretation difficult, we did observe for `54.dzn` that the early cores have two variables connected by some binary table, constraining one of them to be 0. This ensures at most one of two photos is taken. Later cores also contain 2 or 3 variables, connecting a new variable to existing reformulated costs, or grouping some new variables together. Moreover, the variables in these expanded cores formed cliques connected by non-zero cost tables.

Our first reformulation used a *merging* strategy that, given a set of (initially singleton) clusters, iteratively merged the two with greatest inter-cluster cost (creating a new cost variable). This yielded effective reformulations but was too slow to compute. Visu-

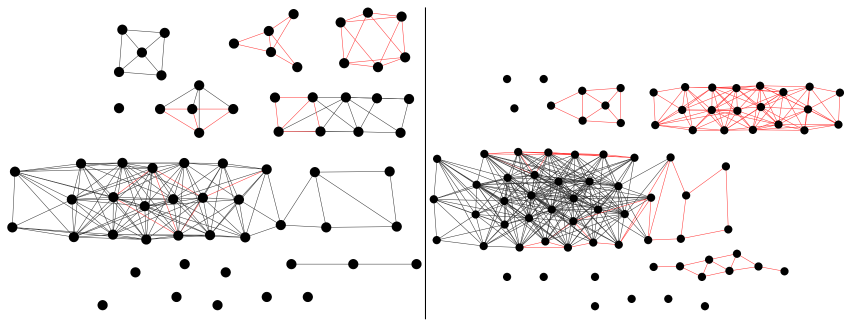


Fig. 1: Constraint structure of two `spot5` instances (left|right). Nodes represent variables; edges binary constraints (red if non-zero cost).

alising the constraint graph of two instances, Figure 1, we realised that they form almost (but not always) *interval graphs*: those where vertices are real intervals and edges their intersection [14]. [2] If they form an interval graph, there must be an ordering where members of any maximal clique appear sequentially. This *interval* ordering would be a good candidate for grouping, as it keeps related vertices nearby. For interval graphs, the ordering can be obtained using lexicographic breadth-first searches (LexBFSs) [7]. We used this procedure, expecting suitable orderings even for constraints that do not form interval graphs. The bounds for group G are computed by a greedy vertex cover of the subgraph of non-zero cost tables containing only leaves of G.

5.2 Experimental Results

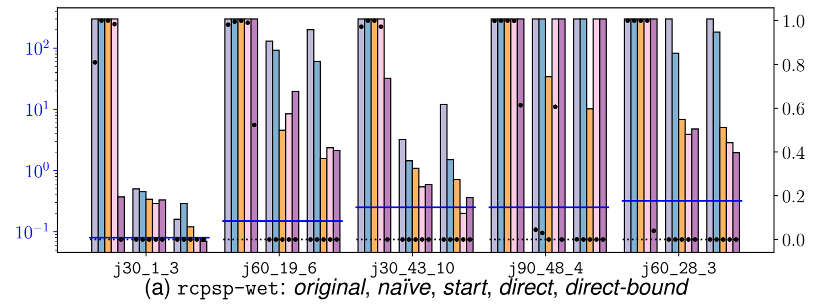
Each of the sub-figures in Figure 2 shows the results obtained for the problem shown in the caption. For each problem, the results are grouped by the data files used to create the instances, as given in the x-axis. The results for each data file are divided into 3 sets of bars separated by spaces. Each set of bars corresponds to the results given by one solver when executing the instances obtained by adding the data file to each of the reformulations named in the caption in the given order. Note the captions always start with the *original* model and a *naïve* grouping of the components in the order they appear in the original objective. The three solvers used are Gecode (set of bars on the left), Chuffed (middle) and the branch-and-bound version of GEAS (right). The values shown per instance are as follows: (a) the solving time as the height of the bar w.r.t. left y-axis in logarithmic scale, and with a 300 sec timeout; (b) the objective value as a black dot on the bar w.r.t. right y-axis, and scaled to the range $[0, 1]$, and (c) the baseline for each instance (core-guided GEAS on the *original* model) as the horizontal blue line for time, and the black dotted line for the objective value. Note that we also obtained results for reformulations with random orderings. These results were consistently worse, and are omitted to improve legibility.

For `rcpsp-wet`, *start*, *direct* and *direct-bound* achieve very good improvements for both LCG solvers (given the logarithmic scale for time), with *direct* able to improve the time of the largest instance, and *direct-bound* often performing best. As expected, Gecode does not benefit from *start* or *direct*, but drastically improves with *direct-bound* on two instances and gets a much better objective bound on the remaining three.

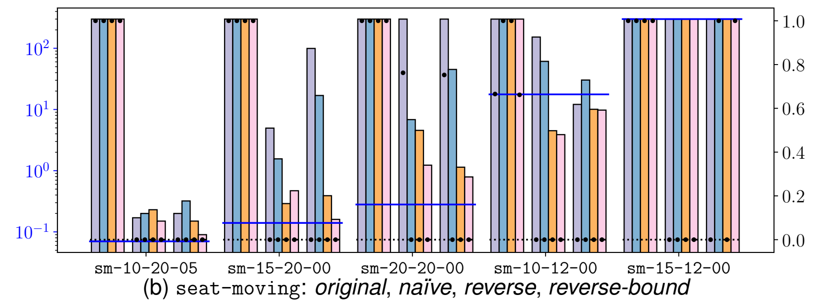
For `seat-moving`, both *reverse* and *reverse-bound* significantly improve the performance of the two LCG solvers for most instances. Instance `sm-10-12-00` is interesting, as the LCS solvers often outperforms the baseline time. The *reverse-bound* reformulation did not improve Gecode (the bound was too weak).

For `jp-encoding` *element* and *local* both yield good performance, with *element* often performing much better for Chuffed, and *local* performing outstandingly for GEAS (often better than the baseline). Interestingly, *naïve* performs badly for Chuffed but, for GEAS, despite never proving optimality, discovers similar or better bounds than *element*. This is because the terms in the implied element constraint are grouped together in the original objective, resulting in similar reformulations.

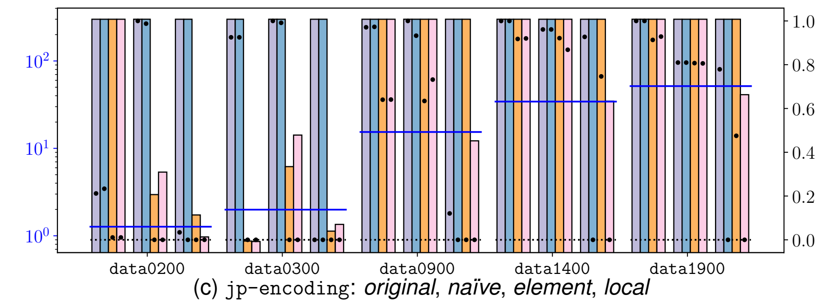
[2] Vertices in the model correspond to observations made along the trajectory of a satellite (have an underlying ordering); edges correspond to observations that are close enough to interfere.



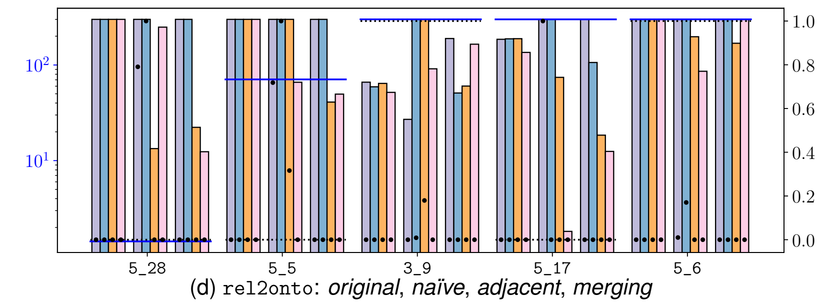
(a) `rcpsp-wet`: *original*, *naïve*, *start*, *direct*, *direct-bound*



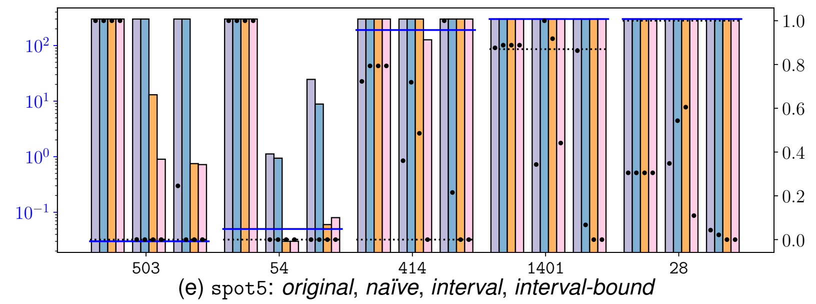
(b) `seat-moving`: *original*, *naïve*, *reverse*, *reverse-bound*



(c) `jp-encoding`: *original*, *naïve*, *element*, *local*



(d) `rel2onto`: *original*, *naïve*, *adjacent*, *merging*



(e) `spot5`: *original*, *naïve*, *interval*, *interval-bound*

Fig. 2: Results for our five models

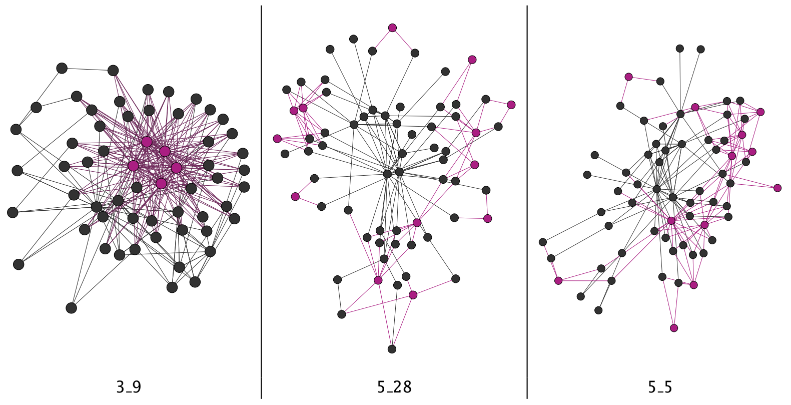


Fig. 3: Visualising three `rel2onto` instances: black edges for ontology structure; magenta for possible attribute/concept matchings.

For `rel2onto`, *adjacent* and *merging* make Chuffed worse for `3_9`, whose structure is quite different from the two instances we used for our cores, as shown in Figure 3). However, they significantly improve Chuffed and GEAS on most other instances except Interestingly, core-guided GEAS on the original model times out for three instances, but branch-and-bound LCG solvers perform much better with the reformulations.

For `spot5`, *interval* and *interval-bound* perform significantly better than *original* or *naïve* for both LCG solvers, while *interval-bound* did not improve Gecode.

6 Conclusions and Future Work

From the above results we conclude the following. First, non-bound core-guided reformulations are often enough to achieve excellent improvements for LCG solvers, as bounds can be learned for the new variables. Second, tight bounds (as for `rcpsp-wet`) can drastically improve CP solvers. Finally, our reformulations of simple models yield great results, but we believe the insights of model owners should enable even better groupings and tighter bounds for all models, as in-depth knowledge is key. Given the scarcity of core-guided (and LCG) solvers, a key contribution is to show modellers the importance of appropriately grouping the objective components and tightly bounding them. In particular, we show that significant speedups can be obtained by simply using our grouping function on orderings of the objective components that are based on “relatedness”. While these orders can be tried speculatively without using core-guided optimization, its use can help to quickly identify where to look for “usefully related” terms (i.e., orderings), and for candidates for analytic bounds to add. We also show how parts of the process can be either automated or supported by automation. We are particularly excited by the idea of using an MUS enumeration tool to identify the *reasons* behind the cores. We are following this approach in our future work, where we aim to further automate our method as much as possible.

References

1. Andres, B., Kaufmann, B., Matheis, O., Schaub, T.: Unsatisfiability-based optimization in clasp. In: Proc. ICLP Technical Communications. LIPIcs, vol. 17, pp. 211–221. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
2. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving SAT-based weighted MaxSAT solvers. In: Proc. CP. Lecture Notes in Computer Science, vol. 7514, pp. 86–101. Springer (2012)
3. Bensana, E., Lemaître, M., Verfaillie, G.: Earth observation satellite management. Constraints **4**(3), 293–299 (1999). https://doi.org/10.1023/A:1026488509554, https://doi.org/10.1023/A:1026488509554
4. Berg, J., Järvisalo, M.: Weight-aware core extraction in SAT-based MaxSAT solving. In: Proc CP. Lecture Notes in Computer Science, vol. 10416, pp. 652–670. Springer (2017)
5. Brand, S., Narodytska, N., Quimper, C.G., Stuckey, P., Walsh, T.: Encodings of the sequence constraint. In: Bessière, C. (ed.) Principles and Practice of Constraint Programming – CP 2007. pp. 210–224. Springer Berlin Heidelberg (2007)
6. Chu, G.: Improving combinatorial optimization. Ph.D. thesis, University of Melbourne, Australia (2011), http://hdl.handle.net/11343/36679
7. Corneil, D.G.: Lexicographic breadth first search - A survey. In: Hromkovic, J., Nagl, M., Westfechtel, B. (eds.) 30th International Workshop on Graph-Theoretic Concepts in Computer Science. Lecture Notes in Computer Science, vol. 3353, pp. 1–19. Springer (2004)
8. Feydy, T., Stuckey, P.J.: Lazy Clause Generation Reengineered. In: Gent, I.P. (ed.) Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 5732, pp. 352–366. Springer (2009)
9. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A mathematical programming language. AT & T Bell Laboratories Murray Hill, NJ 07974 (1987)
10. Frisch, A.M., Grum, M., Jefferson, C., Martínez, B., Miguel, H.I.: The design of ESSENCE: a constraint language for specifying combinatorial problems. In: IJCAI-07. pp. 80–87 (2007)
11. Gange, G., Berg, J., Demirović, E., Stuckey, P.J.: Core-guided and core-boosted search for CP. In: Hebrard, E., Musliu, N. (eds.) Proceedings of Seventeenth International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming (CPAIOR2020). Springer (2020), *to appear*
12. Gecode Team: Gecode: Generic Constraint Development Environment (2006), Available from http://www.gecode.org
13. Gurobi Optimization, Inc.: Gurobi Optimizer Reference Manual Version 7.5. Houston, Texas: Gurobi Optimization (2017)
14. Lekkeikerker, C., Boland, J.: Representation of a finite graph by a set of intervals on the real line. Fundamenta Mathematicae **51**(1), 45–64 (1962)
15. Leo, K., Mears, C., Tack, G., Banda, M.G.d.l.: Globalizing Constraint Models. In: Schulte, C. (ed.) CP. LNCS, vol. 8124, pp. 432–447. Springer (2013)
16. Leo, K., Tack, G.: Debugging unsatisfiable constraint models. In: Salvagnin, D., Lombardi, M. (eds.) CPAIOR 2017. Lecture Notes in Computer Science, vol. 10335. Springer (2017). https://doi.org/10.1007/978-3-319-59776-8
17. Marques-Silva, J., Argelich, J., Graça, A., Lynce, I.: Boolean lexicographic optimization: algorithms & applications. Annals of Mathematics and Artificial Intelligence **62**(3-4), 317–343 (2011)
18. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a Standard CP Modelling Language. In: Bessiere, C. (ed.) CP. LNCS, vol. 4741, pp. 529–543. Springer (2007)

19. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = Lazy Clause Generation. In: Bessiere, C. (ed.) Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. LNCS, vol. 4741, pp. 544–558. Springer (2007)
20. Perron, L., Furnon, V.: OR-Tools (2019), https://developers.google.com/optimization/
21. Shishmarev, M., Mears, C., Tack, G., Garcia de la Banda, M.: Visual search tree profiling. Constraints pp. 1–18 (2015)
22. Stuckey, P., Becket, R., Fischer, J.: Philosophy of the MiniZinc challenge. Constraints **15**(3), 307–316 (2010). https://doi.org/10.1007/s10601-010-9093-0
23. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press, Cambridge, MA, USA (1999)
24. Zeighami, K., Leo, K., Tack, G., de la Banda, M.G.: 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. Lecture Notes in Computer Science, vol. 11008, pp. 403–419. Springer (2018)