Generating Compound Moves in Local Search by Hybridisation with Complete Search *

Gustav Björdal✉[0000−0002−8032−5774], Pierre Flener[0000−0001−8730−4098], and Justin Pearson[0000−0002−0084−8891]

Uppsala University, Dept of Information Technology, SE – 751 05 Uppsala, Sweden
{Gustav.Bjordal,Pierre.Flener,Justin.Pearson}@it.uu.se

Abstract A black-box local-search *backend* to a solving-technology-independent modelling language, such as MiniZinc, automatically infers from the structure of a declarative model for a satisfaction or optimisation problem a combination of a neighbourhood, heuristic, and metaheuristic. These ingredients are then provided to a local-search *solver*, but are manually designed in a handcrafted local-search *algorithm*. However, such a backend can perform poorly due to model structure that is inappropriate for local search, for example when it considers moves modifying only variables that represent auxiliary information. Towards overcoming such inefficiency, we propose *compound-move generation*, an extension to local-search solvers that uses a complete-search solver in order to augment moves modifying non-auxiliary variables so that they also modify auxiliary ones. Since compound-move generation is intended to be applied to such models, we discuss how to identify them.
We present several refinements of compound-move generation and show its very positive impact on several third-party models. This helps reduce the unavoidable gap between black-box local search and local-search algorithms crafted by experts.

1 Introduction

The aim of technology-independent modelling languages for satisfaction and optimisation problems is to allow many solvers to run for a single problem model and hopefully avoid too early commitment to a solving technology. MiniZinc [13] is such a language where a user designs a model and can then solve the problem using a wide range of backends that call solvers from technologies such as constraint programming (CP), lazy clause generation (LCG), (constraint-based) local search (LS and CBLS), mixed-integer programming (MIP), Boolean satisfiability (SAT), and satisfiability modulo theories (SMT). Given a MiniZinc model, a backend should infer a representation and search strategy that are suitable for its solver and technology.

A *black-box local-search backend* automatically infers from a MiniZinc model, which is purely declarative, a representation required to compute efficiently a

* This work is supported by the Swedish Research Council (VR) through Project Grant 2015-04910.

cost function as well as a combination of a neighbourhood, heuristic, and metaheuristic, which form the search strategy and are then provided to a CBLS solver: these ingredients are manually designed in a handcrafted local-search algorithm, which processes no model. A drawback of technology-independent modelling is that backends of some solving technologies can be sensitive to model structure (which backends of other technologies may be unaffected by). For all these reasons, there will always be a gap between black-box local-search backends and local-search algorithms handcrafted by experts for specific problems.

To help reduce this gap, we here explore a model structure where black-box LS performs poorly, namely when its inferred neighbourhood has moves modifying only variables representing auxiliary information. This model structure can appear when such auxiliary variables are (or seem) not functionally determined by the other variables. The intuition for why this can degrade performance is that the search strategy should consider new values for the non-auxiliary variables and infer (possibly when generating the considered moves) new values for the auxiliary variables from those considered values, and not vice versa.

Towards improving the performance of black-box local search, our contributions are as follows, after defining all required background in Section 2. In Section 3, we propose *compound-move generation* (CMG), an extension to local-search solvers that uses a complete-search solver (based on CP in our implementation) in order to try to augment moves modifying non-auxiliary variables so that they also modify auxiliary ones. In Section 4, we present two approaches for detecting the model structure that CMG is intended for. In Section 5, we experimentally demonstrate the very positive impact of CMG. We discuss related work in Section 6 and conclude in Section 7.

2 Background

We define the relevant concepts of MiniZinc, (CP-style) complete search, and (constraint-based) local search.

2.1 MiniZinc, FlatZinc, Models, and Instances

The constraint-based modelling language MiniZinc [13] for satisfaction and optimisation problems is independent of solving technologies, such as CP, LCG, LS, CBLS, MIP, SAT, and SMT. Its open-source toolchain contains a *flattener*, which translates a model and instance data into a sub-language called FlatZinc, which is amenable to interpretation and analysis by a *backend* that calls a targeted *solver*. We now present a MiniZinc model for our running example.

Example 1. Consider the travelling salesperson problem with time windows (TSPTW). Given are `n` locations, a travel-time matrix `T` (where `T[i,j]` is the travel time from location `i` to location `j` plus the service time at `i`), and a matrix `W` of time windows for each location (where `W[i,1]` is the earliest arrival time and `W[i,2]` the latest arrival time for location `i`). The goal is to find a shortest route that visits each location exactly once and within its time window.

```
1  int: n; set of int: Loc = 1..n; % (number of) locations
2  array[Loc,Loc] of int: T; % travel times
3  array[Loc,1..2] of int: W; % time windows
4  array[Loc] of var Loc: pred; % predecessor locations
5  constraint circuit(pred);
6  int: depot = 1; % location 1 is the depot
7  array[Loc] of var int: A; % arrival times
8  constraint A[depot] = W[depot,1];
9  constraint forall(i in Loc where i != depot)(
10   A[i] = max(A[pred[i]] + T[pred[i],i], W[i,1]));
11 constraint forall(i in Loc)(A[i] <= W[i,2]);
12 solve minimize sum(i in Loc)(T[pred[i],i]);
```

Figure 1. A MiniZinc model for TSP with time windows (TSPTW)

Figure 1 shows a MiniZinc model that is good for most backends. Since the arrival time at a location depends on the departure time at the previous location, the route is modelled using a predecessor array `pred`, where variable `pred[i]` denotes the location visited before location `i`. The `circuit` constraint in line 5 requires `pred` to represent a Hamiltonian circuit. Location `1` is assumed to be the start of the route. The arrival times are modelled using the array `A`, where variable `A[i]` denotes the arrival time at location `i`. Each arrival time is constrained, in lines 8 to 11, to be either the arrival time at the preceding location plus the travel time or the start of its time window, whichever is greater, and at most the end of its time window. The objective is to minimise the travel time of the entire route, which is stated in line 12.

Each variable `A[i]` represents auxiliary information and seems at first sight to be functionally determined by lines 8 and 10. However, since `A[pred[i]]` on the right-hand side of the equality in line 10 defines `A[i]` possibly in terms of itself, a backend should infer that `A[i]` cannot be functionally determined by lines 8 and 10 alone. This is an example of a model that, somewhat unexpectedly, a backend can see as having non-functionally determined variables representing auxiliary information.[1] Furthermore, the equality constraints in lines 8 and 10 are here only correct because in a minimal solution the salesperson always arrives as early as possible. If this assumption cannot be made, possibly due to additional side constraints, then lines 8 and 10 have to be expressed using inequalities, thus making the auxiliary `A[i]` variables *necessarily* non-functionally determined. □

[1] Note that upon *also* considering the semantics of the `circuit` constraint in line 5, a backend that *only* explores assignments satisfying that constraint can infer that the `A[i]` are in fact functionally determined by line 10. However, to the best of our knowledge, no backend to MiniZinc performs such a semantic analysis. Also, doing so would not address all cases where a model can be seen as having non-functionally determined variables representing auxiliary information.

Without loss of generality, we explain everything for minimisation problems: a maximisation or satisfaction problem can be transformed into a minimisation problem by minimising the negated objective function or a constant, respectively. In order to emphasise the independence from MiniZinc of our method, a FlatZinc model for a minimisation problem *instance* is here abstracted as a tuple $\langle \mathcal{V}, \mathcal{D}, \mathcal{C}, o \rangle$, where $\mathcal{V}$ is the set of variables; $\mathcal{D}$ is the function mapping each variable to its set of possible values, called its *domain*; $\mathcal{C}$ is the set of constraints over variables in $\mathcal{V}$; and $o \in \mathcal{V}$ is the variable that is constrained in $\mathcal{C}$ to take the value of the objective function, which is to be minimised.

2.2 Complete Search and Constraint Programming

Given a minimisation instance $\langle \mathcal{V}, \mathcal{D}, \mathcal{C}, o \rangle$, a *solution* is an assignment of all variables $\mathcal{V}$ to values allowed by the domain mapping $\mathcal{D}$ such that all the constraints $\mathcal{C}$ are satisfied. We denote the value assigned to a variable v in a solution by $\mathrm{sol}(v)$ and a solution by $\mathrm{sol}(\mathcal{V}) := \{v \mapsto \mathrm{sol}(v) \mid v \in \mathcal{V}\}$.

Given enough time, a *complete* solver is guaranteed either to return a proven *minimal solution*, which is a solution where $\mathrm{sol}(o)$ is minimal, or to prove unsatisfiability otherwise. If a complete solver is stopped early, then it returns either the *best-found solution*, without proof of minimality, or nothing, meaning it is not known if the instance is satisfiable or not.

Many solving technologies offer complete solvers, such as CP, LCG, MIP, SAT, and SMT. Our main ideas are independent of which complete technology is used, but some refinements exploit features of CP solvers, defined next.

A *CP solver* builds a search tree by interleaving propagation and search. It modifies the *current domain*, which maps each variable $v \in \mathcal{V}$ to a set $\mathrm{dom}(v)$, initialised to $\mathcal{D}(v)$. *Propagation* computes the fixpoint of the propagators, one for each constraint in $\mathcal{C}$: a *propagator* for a constraint c deletes (not necessarily all) values from the current domain of each variable in c that are impossible under c. The current domain for the *root node* of the search tree is computed by a first run of propagation. If the current domain of some variable becomes empty at some node, then there is a *failure* and backtracking occurs. If the current domain of each variable becomes a singleton at some node, then the instance is proven satisfiable, under the assignment $\{v \mapsto d \mid v \in \mathcal{V} \wedge \mathrm{dom}(v) = \{d\}\}$, which is $\mathrm{sol}(\mathcal{V})$, and the constraint $o < \mathrm{sol}(o)$ is added to $\mathcal{C}$ before backtracking in order to search for a better solution. If at least one current domain has at least two values, then a *child node* is created for each part of a partition of $\mathrm{dom}(v)$ into at least two non-empty disjoint subsets for some variable v, guided by a *branching strategy*. Solving (by propagation and search) recursively continues for each child node, under usually a depth-first exploration order. Solving either returns a minimal solution or reports unsatisfiability.

2.3 (Constraint-Based) Local Search

Other solving technologies offer non-complete solvers. For example, *local search* (LS), say [11], initialises and iteratively modifies the *current assignment*, which

maps each variable $v \in \mathcal{V}$ to a value $\text{val}(v)$, called its *current value*, in its domain $\mathcal{D}(v)$. The current assignment need not satisfy all the constraints $\mathcal{C}$. The *initial assignment* is built under some amount of randomisation. At every *iteration*, a two-step search *heuristic* is followed. First, a set of *candidate moves* is considered, each being a set of reassignments $v \mapsto d$ for at least one variable $v \in \mathcal{V}$ and value $d \in \mathcal{D}(v)$. We assume that each candidate move is *probed* by (i) tentatively performing its reassignments, (ii) estimating the proximity $\hat{p}$ of the resulting *tentative assignment* to some assignment satisfying $\mathcal{C}$ and computing the resulting value $\hat{o}$ of the objective variable o, and (iii) undoing the tentatively performed reassignments and returning the pair $\langle \hat{p}, \hat{o} \rangle$. The set of probed candidate moves is called the *neighbourhood*, which is said to be *explored*, and its elements are called *neighbours*. Second, among the candidate moves, the heuristic *selects* one based on a *cost function* applied to each pair $\langle \hat{p}, \hat{o} \rangle$ and actually *commits* it, yielding the new current assignment. A *meta-heuristic*, such as tabu search, say [10], can be used to escape local optima of the cost function. Together, the neighbourhood, heuristic, and meta-heuristic form a *local-search strategy*.

In *constraint-based local search* (CBLS) [19], a declarative model is coupled with either a user-defined LS strategy, yielding a *white-box LS solver* (such as Comet [19] and OscaR.cbls [6]), or a solver-inferred LS strategy, yielding a *black-box LS solver* (such as LocalSolver [1] and fzn-oscar-cbls [2]). For each built-in constraint c, a predefined *violation function* $\text{viol}(c)$, which returns the value 0 when c is satisfied and otherwise a positive value, can be used for estimating the proximity of a tentative assignment to an assignment satisfying c. One can then estimate the proximity $\hat{p}$ as the *violation* $\text{viol}(\mathcal{C}) := \sum_{c \in \mathcal{C}} \text{viol}(c)$. *Note that objective function, cost function, and violation function are here* not *synonyms.*

A CBLS model has two categories of *explicit constraints*. *Soft constraints* have a violation function and may be violated during search but must be satisfied in a solution. *One-way constraints*, such as `z <== x * y` in OscaR.cbls syntax and called *invariants* in Comet, are impossible to violate by candidate moves: in `z <== x * y`, the functionally determined variable `z` cannot undergo a move, since its value is maintained by the solver to be the product of the variables `x` and `y`, which can undergo moves. An *implicit constraint* in a CBLS model is satisfied by the initial assignment and preserved by all committed moves: this can be done by using a *constraint-specific neighbourhood* [2].

For each constraint of a problem, a CBLS modeller must choose whether to make it soft, one-way, or implicit. *Note that implicit and one-way constraints do* not *exist as such in MiniZinc and FlatZinc.*

2.4 A Local-Search Backend to MiniZinc

Our fzn-oscar-cbls [2] LS backend to MiniZinc conceptually performs three steps. First, the constraints of a given flattened MiniZinc model are categorised into the three CBLS constraint categories (soft, one-way, and implicit) by using a structure identification scheme (see [2] for full details). Second, an LS strategy for the CBLS solver OscaR.cbls [6] is inferred: the neighbourhood is the union of the constraint-specific neighbourhoods for all identified implicit constraints and

a default neighbourhood for all variables that are not part of any constraint-specific neighbourhood (note that variables identified as functionally determined, and thus maintained by one-way constraints, are not in any neighbourhood); the heuristic selects a random best candidate move from the neighbourhood; and the meta-heuristic is a variation of tabu search [10]. Third, OscaR.cbls is invoked. *Note that backend and solver are here* not *synonyms.*

Example 2. For the model in Figure 1, fzn-oscar-cbls categorises the `circuit` constraint in line 5 as implicit, since a constraint-specific neighbourhood (namely 3-opt) is available in fzn-oscar-cbls. The `A[i]` variables are mistakenly conjectured not to be functionally determined, as the structure identification scheme does not take the semantics of `circuit` into account, hence the `A[i]` seem defined possibly in terms of themselves and are not maintained by one-way constraints: a default neighbourhood is inferred for them. The objective variable (introduced by line 12) is maintained by a one-way constraint. The soft constraints are the equality constraints in lines 8 and 10. □

Two major burdens for an LS backend to a technology-independent modelling language such as MiniZinc are the identification of an LS-appropriate structure of a model, which is non-trivial as models need not be written with LS in mind, and the ensuing neighbourhood inference, which depends on the identified structure. We now address these two burdens by trying to make LS backends more robust to models without an identifiable LS-appropriate structure and by making the moves of the inferred neighbourhoods more suitable to such models.

3 Compound-Move Generation

We present compound-move generation (CMG), an extension to local search (LS) that hybridises LS with complete solving and is geared for models where an LS solver is forced to make moves over what we will call *auxiliary variables*, which we will demonstrate in Section 5.2 to greatly degrade performance. The main idea is to use a complete solver, in our implementation a CP solver, to try to augment each move probed by the LS solver in order to generate what we will call a *compound move* that also reassigns auxiliary variables. We first explain the basic CMG algorithm and then discuss implementation-specific refinements.

3.1 Basic Algorithm

Consider a flattened MiniZinc model $\langle \mathcal{V}, \mathcal{D}, \mathcal{C}, o \rangle$, partitioned a priori such that $\mathcal{V} = \mathcal{V}_\mathrm{c} \cup \mathcal{V}_\mathrm{a}$, where the variables of $\mathcal{V}_\mathrm{c}$ are called *core variables* and those of $\mathcal{V}_\mathrm{a}$ are called *auxiliary variables*; and $\mathcal{C} = \mathcal{C}_\mathrm{c} \cup \mathcal{C}_\mathrm{a} \cup \mathcal{C}_\ell$ where the constraints of $\mathcal{C}_\mathrm{c}$ are called *core constraints* and are all the constraints over only variables in $\mathcal{V}_\mathrm{c}$, those of $\mathcal{C}_\mathrm{a}$ are called *auxiliary constraints* (also known as *side constraints*) and are all the constraints over only variables in $\mathcal{V}_\mathrm{a}$, and those of $\mathcal{C}_\ell$ are called *linking constraints*. Note that o need not be in $\mathcal{V}_\mathrm{c}$.

In Example 1, the `pred[i]` variables are ideally in $\mathcal{V}_c$ and the `A[i]` in $\mathcal{V}_a$: the constraint sets $\mathcal{C}_c$, $\mathcal{C}_a$, and $\mathcal{C}_\ell$ follow by their definitions. We discuss in Section 4 how to guess this partition automatically.

A model for $\langle \mathcal{V}, \mathcal{D}, \mathcal{C}, o \rangle$ is created for the CBLS solver and a neighbourhood is inferred for the non-functionally determined variables in $\mathcal{V}_c$ but *not* for any variables in $\mathcal{V}_a$; the values of all the variables in $\mathcal{V}_c$ identified as functionally determined are maintained by one-way constraints (see Section 2.3) in the CBLS model.

Further, a model for $\langle \mathcal{V}', \mathcal{D}', \mathcal{C}'_a \cup \mathcal{C}'_\ell, o' \rangle$ is created for the complete solver: we add the prime symbol to the corresponding objects for the CBLS solver.

The probing (recall that we assume it consists of (i) tentatively performing a candidate move m; (ii) computing the resulting value of the cost function; and (iii) undoing the candidate move) in the CBLS solver is modified such that between (i) and (ii) an extra step of calling the complete solver is added, divided into three sub-steps:

1. Each variable v' in $\mathcal{V}'_c$ of the model in the complete solver is fixed to the tentative value of its corresponding variable v in the CBLS solver by adding the constraint $v' = \text{val}(v)$ to the model in the complete solver. The search of the complete solver is then launched in order to find an assignment of the variables $\mathcal{V}'_a$ that satisfies all constraints in $\mathcal{C}'_a \cup \mathcal{C}'_\ell \cup \{v' = \text{val}(v) \mid v' \in \mathcal{V}'_c\}$.
2. There are two possible outcomes: either (a) the complete solver reports unsatisfiability, whether at the root node or through search, and the normal probing of m continues; or (b) the complete solver returns a minimal solution $\text{sol}(\mathcal{V}')$ and the normal probing continues for the candidate move $m \cup \{v \mapsto \text{sol}(v') \mid v \in \mathcal{V}_a\}$, which we call a *compound move*, instead of m.
3. Sub-step 1 is undone in the complete solver.

The other aspects of the search in the CBLS solver, such as the heuristic and the meta-heuristic, remain unchanged.

3.2 Refinements and Implementation

We have implemented CMG for our black-box local-search backend fzn-oscar-cbls [2] to MiniZinc, calling the OscaR.cbls solver [6], by using as the complete solver the OscaR.cp solver of the same OscaR framework [14], thereby exploiting the felicitous co-existence of CP and CBLS solvers within the OscaR toolkit.

In its basic form, CMG can be very slow or memory-intensive. We here describe several refinements that improve the performance of CMG, sometimes modifying parts of fzn-oscar-cbls. Some refinements have parameters (denoted by Greek letters), for which we propose values in Section 5. We refer to "the complete solver" when refinements or concepts are technology-agnostic, and to "the CP solver" when refinements or concepts are dependent on CP technology.

A. Incomplete Solving. Since finding a minimising assignment for the variables in $\mathcal{V}_a$ can be NP-hard, we can limit the complete solver in the total runtime τ, the number ϕ of failures (if it is a CP solver), and the number σ of intermediate solutions.

B. Always Modifying Auxiliary Variables. If, for a large number of consecutive committed moves, the complete solver has failed to augment them into compound ones, then the auxiliary variables $\mathcal{V}_\mathrm{a}$ remain unchanged in the CBLS solver during those iterations. Since the current assignment of $\mathcal{V}_\mathrm{a}$ contributes to the violation in the CBLS solver, as $\mathcal{C}_\mathrm{a} \cup \mathcal{C}_\ell$ is part of its model, and thus can affect which candidate move is committed, this can result in the local search getting stuck in some region of the search space. So fzn-oscar-cbls can be modified to infer also a neighbourhood for $\mathcal{V}_\mathrm{a}$: whenever the CBLS solver commits a move for which the complete solver has failed, a move from the neighbourhood of $\mathcal{V}_\mathrm{a}$ is also committed, in the same iteration.

C. Calling the Complete Solver Again Before Committing a Move. If Refinement A is used, then the assignment of the auxiliary variables $\mathcal{V}_\mathrm{a}$ returned by the complete solver may not be minimal with respect to the objective function. So, between selecting and committing a candidate move, the complete solver can be run again for that move in order to get a possibly better assignment of $\mathcal{V}_\mathrm{a}$. This second solving can be either complete or, as done in our experiments in Section 5, a deeper incomplete solving, depending on the new parameters $\tau^\uparrow$, $\phi^\uparrow$, and $\sigma^\uparrow$, which have the semantics of their counterparts in Refinement A.

D. Only Calling the Complete Solver Before Committing a Move. Refinement C can be taken to the extreme where the complete solver is not used at all while probing, but only after selecting a candidate move: instead of modifying the probing step, we can call the complete solver as a post-processing step to selecting a move. For large neighbourhoods, this will significantly speed up the probing, but at the cost of possibly missing good candidate moves.

E. Only Returning the Objective Value. The cost function minimised by fzn-oscar-cbls is $\alpha \cdot \mathrm{val}(o) + \beta \cdot \mathrm{viol}(\mathcal{C})$, where o is the objective variable, $\mathrm{viol}(\mathcal{C})$ is the violation of the constraints in $\mathcal{C} = \mathcal{C}_\mathrm{c} \cup \mathcal{C}_\mathrm{a} \cup \mathcal{C}_\ell$, and α, β are non-negative weights that are tuned during search. An assignment $\mathrm{sol}(\mathcal{V}'_\mathrm{a})$ returned by the complete solver satisfies all constraints in $\mathcal{C}'_\mathrm{a} \cup \mathcal{C}'_\ell$ and $\mathrm{sol}(o')$ is the same value as o will have in the CBLS solver if the corresponding reassignment is made. Therefore, if using Refinement C, then it is enough, *while probing*, for the complete solver to return $\mathrm{sol}(o')$ since $\alpha \cdot \mathrm{val}(o) + \beta \cdot \mathrm{viol}(\mathcal{C}) = \alpha \cdot \mathrm{sol}(o') + \beta \cdot \mathrm{viol}(\mathcal{C}_\mathrm{c})$ in this case. By maintaining $\mathrm{viol}(\mathcal{C}_\mathrm{c})$ in a separate constraint system [19] in the underlying CBLS solver, we can compute the cost function faster while probing.

F. Exploiting Conflicting Assignments. We say that the current assignment of some variables in the CBLS solver is *conflicting* if they cause the current domain of at least one variable to become empty in the CP solver due to root-node propagation. For efficiency reasons, we limit this definition to root-node failure, but one can generalise it to any failure.

For example, consider the constraints $x < a$ and $a < y$, where $x, y \in \mathcal{V}_\mathrm{c}$ and $a \in \mathcal{V}_\mathrm{a}$, and the conflicting assignment $\{x \mapsto 2, y \mapsto 1\}$. Until x or y is reassigned by a CBLS move, the CP solver empties the domain of its variable a' and fails at the root node upon adding the constraints $\{x' = 2,\ y' = 1\}$ in sub-step 1 of the basic CMG. However, for this conflicting assignment and most values of a, one of the two inequality constraints is always satisfied in the CBLS solver and

the other one might only make a small contribution to viol($\mathcal{C}$). So there might be no strong indication for the LS strategy that x or y needs to be reassigned.

So one should try to identify which variables in $\mathcal{V}'_{\mathrm{c}}$ have caused a root-node failure in the CP solver, and then force the CBLS solver to commit moves on its corresponding core variables. This can be done with a solver that provides explanations for failures, such as any LCG solver, say Chuffed [3]. Most CP solvers (e.g., OscaR.cp, which we use in our implementation) do not provide explanations for failures, so we try to identify which variables have caused such a failure by extending the basic CMG algorithm:

a. The constraints $v' = \mathrm{val}(v)$ for each $v' \in \mathcal{V}'_{\mathrm{c}}$ are, in a random order, iteratively added to the model of the CP solver in sub-step 1, triggering root-node propagation each time. If a failure occurs, then the last variable that was fixed (i.e., that triggered the failure) is returned, say u'. Otherwise, there is no conflicting assignment.
b. Each such variable $u' \in \mathcal{V}'_{\mathrm{c}}$ is recorded in a map that maintains its number of triggered failures, which we call its *conflict count*. The counter is reset to zero for a variable in $\mathcal{V}'_{\mathrm{c}}$ whenever a move reassigning its corresponding variable is committed by the CBLS solver.

If at least one variable has a conflict count that is at least a parameter ω, we force the search heuristic in the CBLS solver to make a move on one or more variables in $\mathcal{V}_{\mathrm{c}}$ with a conflict count of at least another parameter $\underline{\omega}$, by exploiting the tabu search of fzn-oscar-cbls: we make all variables in $\mathcal{V}_{\mathrm{c}}$ with a conflict count under $\underline{\omega}$ tabu. We recommend $\omega > \underline{\omega}$ to avoid making too many variables tabu.

4 Partitioning a Model for CMG

To use compound-move generation (CMG), one must first partition a model instance in order to get the sets $\mathcal{V}_{\mathrm{c}}$, $\mathcal{V}_{\mathrm{a}}$, $\mathcal{C}_{\mathrm{c}}$, $\mathcal{C}_{\mathrm{a}}$, and $\mathcal{C}_{\ell}$. By $\mathcal{V} = \mathcal{V}_{\mathrm{c}} \cup \mathcal{V}_{\mathrm{a}}$ and the definitions of $\mathcal{C}_{\mathrm{c}}$, $\mathcal{C}_{\mathrm{a}}$, $\mathcal{C}_{\ell}$, all these sets can be inferred given either $\mathcal{V}_{\mathrm{c}}$ or $\mathcal{V}_{\mathrm{a}}$. We do not impose any semantics to $\mathcal{V}_{\mathrm{c}}$ and $\mathcal{V}_{\mathrm{a}}$: CMG can be applied to any such partition, as done at the end of Section 5.1. However, we conjecture that CMG is most efficient when $\mathcal{V}_{\mathrm{c}}$ is the set of variables that model the combinatorial sub-structure of the problem, and $\mathcal{V}_{\mathrm{a}}$ has the variables whose values can easily be determined (ideally by CP-style propagation) given an assignment of $\mathcal{V}_{\mathrm{c}}$.

We present two ways of making this partition, namely by user-provided hints in a MiniZinc model and automatically through a heuristic: in a black-box setting, a user should not have to provide a hint to use CMG and most third-party MiniZinc models are not written with a method such as CMG in mind.

4.1 Hint-Based Partitioning

MiniZinc allows modellers to provide hints to a backend through *annotations* to parts of a model. We introduce the `search_variables(array of var int: V)` annotation, which is attached to the `solve` statement of a model and indicates

that the modeller wants search to be performed on the variables in `V`: an LS backend with CMG can then use the `V[i]` as the core variables.

In Example 2 we saw that both the `pred[i]` and `A[i]` variables of the model in Figure 1 are searched on by fzn-oscar-cbls. Upon annotating the `solve` statement in line 12 with `search_variables(pred)`, fzn-oscar-cbls with CMG can compute that $\mathcal{V}_a$ = `{A[i] | i in Loc}` because $\mathcal{V}_c$ = `{pred[i] | i in Loc}`.

MiniZinc officially supports search annotations for CP and LCG solvers in order to specify branching strategies. Unfortunately, in general, those search annotations cannot be used in place of our here introduced `search_variables` annotation, as their semantics does not hint at distinguishing core and auxiliary variables. One could make the (often incorrect) assumption that all variables appearing in a branching strategy are core variables: however, in practice, many MiniZinc modellers specify a branching strategy for *all* variables of a model, and our aim includes good performance on *third-party* models.

4.2 Heuristic-Based Partitioning

Based on our conjecture that $\mathcal{V}_c$ should have the variables that model the combinatorial sub-structure of the problem, we can try to detect such model structure automatically by using a heuristic to guess a partition. Since the global constraints in MiniZinc (such as `circuit` in Figure 1) capture combinatorial substructures of a problem and fzn-oscar-cbls has constraint-specific neighbourhoods for some global constraints, we can use the following heuristic to decide which variables belong to $\mathcal{V}_c$: if fzn-oscar-cbls infers that constraint-specific neighbourhoods can be used, then all variables that belong to those neighbourhoods are guessed to be in $\mathcal{V}_c$, and all other variables (which would have been put into a default neighbourhood) are therefore in $\mathcal{V}_a$. Otherwise, the heuristic will decide that CMG cannot be used.

This heuristic leads to the same partition for the model in Figure 1 as when annotating its `solve` statement by `search_variables(pred)`, but without having to modify the MiniZinc model. Furthermore, this heuristic is able to guess a good partition for some third-party models used in the MiniZinc Challenges, as we will see in Section 5.1. However, the heuristic can guess bad partitions, so the modeller currently has to say at the command line if CMG should be used.

5 Experimental Evaluation

We believe the strength of CMG lies in dealing with solver-independent models, say in MiniZinc, where non-functionally-determined auxiliary variables can appear naturally and where the modeller need not be familiar with local search (LS). Therefore, we evaluate CMG on third-party MiniZinc models to see its impact on the robustness of an LS backend to MiniZinc across a variety of models in Section 5.1. In order to see how other LS solvers are affected by the presence of non-functionally-determined auxiliary variables, we modify Example 1 to force their presence and evaluate the impact in Section 5.2.

5.1 Benchmark Problems

We compare two configurations of CMG in fzn-oscar-cbls with our original fzn-oscar-cbls and Yuck,[2] which is also a CBLS backend to MiniZinc. As a point of reference, we also run the LCG backend Chuffed [3].

The first configuration, called config1, uses Refinements A, B, C, E, F of Section 3.2, while the second one, called config2, uses A, B, D, F, but not E, which is meaningless with D. Indeed, initial experiments showed that using D instead of C can both improve and degrade performance, depending on the model, while each other refinement individually seems to improve performance. For both configurations, we set the parameters $\tau = \tau^{\uparrow} = 30$ seconds, $\phi = 10000$ failures, $\sigma = 2$ solutions, $\omega = 3$ conflicts, $\underline{\omega} = 1$ conflict, $\phi^{\uparrow} = 100000$ failures, and $\sigma^{\uparrow} = \infty$ solutions: initial experiments showed that all those are good values.

We do not compare with the basic CMG algorithm: initial experiments showed that it is too slow. We do not compare with the black-box local-search solver LocalSolver [1] as it offers no backend to MiniZinc. Reformulating models in LocalSolver's modelling language LSP would not yield a meaningful performance comparison as it does *not* have all the global constraints of MiniZinc and as LSP has features that MiniZinc does *not* have.

We evaluate CMG on models and instances for a capacitated vehicle routing problem (CVRP) and a time-dependent travelling salesperson problem (TDTSP), which are taken from the MiniZinc Challenges [18] of 2015 and 2017, as well as our model in Figure 1 for our running example, the travelling salesperson problem with time windows (TSPTW). Furthermore, we run CMG on all instances of all models of the MiniZinc Challenge 2018 where the heuristic-based partitioning of Section 4.2 detects that CMG can be used (in a competition setting, the original fzn-oscar-cbls would run on the other instances). Finally, to showcase the hint-based partitioning of Section 4.1, we also perform experiments on a group scheduling problem (GFD), used in the MiniZinc Challenge 2018 with a model where the partitioning heuristic of Section 4.2 does not detect that CMG can be used. All models except the one of Figure 1 and all instances are third-party.

For the local-search backends, we made 10 independent runs with a 600-second-timeout each. For the complete-search backend Chuffed, which is deterministic, we report the objective value of one run with the same timeout. The results are reported in Table 1: note that all problems are minimisation or satisfaction problems and that all chosen instances happen to be satisfiable.

TSPTW. The heuristic-based partitioning of Section 4.2 detects auxiliary variables in our model in Figure 1. We selected five ".001" instances of the GendreauDumasExtended benchmark[3] around the instance size where Chuffed and the original fzn-oscar-cbls stopped establishing satisfiability. We see in Table 1 that both CMG configurations improved the best-found and median values of the original fzn-oscar-cbls. Both configurations established satisfiability for all instances in at least 50% of the runs, whereas the original fzn-oscar-cbls only did

[2] `https://github.com/informarte/yuck`

[3] `http://lopez-ibanez.eu/tsptw-instances`

so in at most 20% of the runs. The best-found objective value by config1 is for each instance equal to the best-known objective value reported at the benchmark site. Yuck was not able to establish satisfiability for any instance.

CVRP. The heuristic-based partitioning detects auxiliary variables in the model `cvrp`. We used all except the toy instance of the MiniZinc Challenge 2015. We see in Table 1 that config1 performed worse than the original fzn-oscar-cbls, except for winning on the P-n16-k8 instance, whereas config2 otherwise outperformed all other backends. On all but P-n16-k8, the best-found solutions by config2 are better than those at the MiniZinc Challenge 2015 for any challenge category.

TDTSP. The heuristic-based partitioning detects auxiliary variables in the model `tdtsp`. We used the four largest instances among MiniZinc Challenges 2015 and 2017. In Table 1 we see that both Yuck and Chuffed outperformed the original fzn-oscar-cbls, but that config1 outperformed all other backends.

MiniZinc Challenge 2018. The heuristic-based partitioning of Section 4.2 finds CMG to be applicable to 3 models and 14 instances of the 20 models and 100 instances in the MiniZinc Challenge 2018, namely `elitserien`, `soccer-computational`, and `vrplc`. For `elitserien` and `vrplc`, neither the original fzn-oscar-cbls, nor config1, nor config2, nor Yuck found any solution within the given timelimit to any instance, whereas Chuffed solved all five `elitserien` instances and two `vrplc` instances to optimality within the given timelimit.

For `soccer-computational`, which is the only satisfaction problem in our evaluation, the heuristic-based partitioning determines that CMG is not applicable to the xIGData_22_12_22_5 instance, as a global constraint for which fzn-oscar-cbls has a neighbourhood is removed during flattening. For the other four instances, Yuck found a solution in all runs, while the original fzn-oscar-cbls found a solution in at most half the runs, and Chuffed found a solution to only one instance. Both configurations of CMG had a negative impact on fzn-oscar-cbls, as they did not find any solution in any run.

GFD schedule. The model `gfd-schedule2` is for a scheduling problem with multiple levels of decisions: items are allocated to groups, groups are allocated to factories, and groups are scheduled to be processed on a day. The heuristic-based partitioning does not detect auxiliary variables since the model does not use any global constraint for which fzn-oscar-cbls has a neighbourhood. We therefore use this model to showcase the hint-based partitioning of Section 4.1 by annotating the variables representing the allocation of groups to factories as core variables: it is then inferred that all other variables are auxiliary, and their values will be sought by the complete solver instead. Note that not all of the here inferred auxiliary variables are actually auxiliary, and that CMG will here behave similarly to a decomposition where a master problem of allocating groups to factories is solved by LS and a sub-problem of allocating items to groups and scheduling groups is solved by CP. We use the five instances of the MiniZinc Challenge 2018. We see in Table 1 that both config1 and config2 greatly improved over the original fzn-oscar-cbls. Chuffed found and proved minimal solutions to three instances, though config2 found the best objective value for the second instance. On the largest instance, none of the backends found any solution.

Table 1. Comparison on MiniZinc models and third-party benchmark instances between our original LS backend fzn-oscar-cbls, two configurations enriching it with CMG, the LS backend Yuck, and the complete-search backend Chuffed: best-found objective value over 10 runs (column 'best'), boldface indicating overall best performance for the instance of that row, flagged by '+' if equal to the best-known value, and flagged by '*' if proven optimal by Chuffed; median of the best-found objective values over these runs (column 'median'), a superscript indicating the number of runs establishing satisfiability before timing out, a '–' indicating no such run.

	fzn-oscar-cbls original		fzn-oscar-cbls CMG config1		fzn-oscar-cbls CMG config2		Yuck		Chuffed
TSPTW	best	median	best	median	best	median	best	median	best
n20w180	377	377^{1}	$^{+}\mathbf{253}$	253^{10}	261	263^{10}	–	–	$^{*}\mathbf{253}$
n20w200	347	373^{2}	$^{+}\mathbf{233}$	233^{10}	$^{+}\mathbf{233}$	234^{10}	–	–	$^{*}\mathbf{233}$
n40w120	–	–	$^{+}\mathbf{434}$	439^{5}	437	464^{9}	–	–	536
n40w140	–	–	$^{+}\mathbf{328}$	334^{7}	367	388^{10}	–	–	–
n40w160	–	–	$^{+}\mathbf{348}$	349^{8}	362	393^{10}	–	–	–
CVRP									
A-n37-k5	2614	2870^{9}	2925	2934^{10}	**875**	983^{9}	–	–	1570
A-n64-k9	5431	5659^{4}	5518	5661^{9}	**2868**	3472^{8}	–	–	3667
B-n45-k5	3638	4121^{6}	4201	4207^{10}	**972**	1182^{10}	–	–	2466
P-n16-k8	489	503^{2}	**450**	523^{6}	481	481^{1}	–	–	502
TDTSP									
20_14_10	22546	22883^{9}	**12556**	13506^{10}	13390	15706^{10}	17446	17446^{10}	17024
20_25_00	22924	22961^{2}	**14888**	16024^{10}	15014	16114^{8}	18646	18646^{10}	22328
20_26_00	22901	22930^{2}	**12926**	13917^{10}	13723	14711^{8}	20790	20790^{10}	19076
20_36_10	22611	22946^{6}	**12809**	14559^{10}	13859	16752^{10}	17247	17247^{10}	17054
GFD schedule									
n65f2d50...	12741	18547^{8}	446	552^{10}	343	645^{10}	6861	6861^{10}	$^{*}\mathbf{19}$
n80f7d30...	9952	13338^{10}	665	1062^{10}	**660**	857^{10}	10578	10578^{10}	2023
n90f5d40...	8655	15865^{9}	473	967^{10}	655	880^{10}	15488	15488^{10}	$^{*}\mathbf{11}$
n100f7d5...	29967	40071^{4}	1187	2384^{9}	885	1474^{10}	26397	26397^{10}	$^{*}\mathbf{14}$
n200f5d5...	–	–	–	–	–	–	–	–	–

5.2 Impact of Auxiliary Variables on Local-Search Solvers

We now show the negative impact on CMG-free black-box local search of committing moves on non-functionally-determined variables that represent auxiliary information. Towards this, we reformulate the TSPTW model in Figure 1 such that it *can* be written in LocalSolver's modelling language LSP and, unlike in Example 2, the `A[i]` variables *can* be detected to be functionally determined without a semantic analysis of the entire model. We replace the predecessor array `pred` in line 4 by the array `order`, where variable `order[i]` denotes the `i`th location visited. The `circuit(pred)` constraint in line 5 then becomes `alldifferent(order)`, and the earliest-arrival-time constraint in line 10

becomes `A[i] = max(A[i-1] + T[order[i-1],order[i]], W[order[i],1])`: this means that the `A[i]` *can* now be determined by one-way constraints and there is no need for CMG. The main drawbacks of the new model, called the `alldifferent` model, are that an LS backend can infer a more suitable neighbourhood for the old model, called the `circuit` model, namely 3-opt in the case of fzn-oscar-cbls, and that the `circuit` model is better suited for complete solvers as it captures the combinatorial sub-structure of the problem better.

However, in the `alldifferent` model, we can now artificially make the `A[i]` variables non-functionally determined by replacing the equality constraint above by an `>=` inequality constraint, which will *not* change the minimal objective value. This allows us to measure the negative impact on CMG-free black-box local search of committing moves on non-functionally-determined auxiliary variables. Recall from Example 1 that using an inequality is not only correct but *may also be necessary* in TSPTW variants.

We examined the impact on LocalSolver, Yuck, and the original fzn-oscar-cbls of the `alldifferent` model with either equality (model variant `eq`) or inequality (model variant `ineq`) constraints in the modified line 10. Since LocalSolver does not have a backend to MiniZinc, we wrote equivalent models in LSP.

In Table 2 we see the negative impact of having auxiliary variables that are not functionally determined and thus must undergo moves: both fzn-oscar-cbls and LocalSolver did not find any solutions to `ineq`, where the auxiliary variables are not functionally determined, though Yuck found solutions but with worse objective values than for `eq`. On model `eq`, where the auxiliary variables need not undergo moves, both fzn-oscar-cbls and Yuck found solutions, as opposed to when running the `circuit` model for TSPTW used in Table 1.

This shows that other black-box local-search solvers are adversely affected when moves must be made on non-functionally-determined auxiliary variables.

6 Related Work

In the hybridisation context, [8] discusses two categories of hybrids between local search (LS) and constraint programming (CP):

Table 2. Best-found objective values over 5 independent runs for variants of the `alldifferent` model of TSPTW, showing the negative impact on the original fzn-oscar-cbls, on Yuck, and on LocalSolver (neither of them using CMG) when reformulating so that functionally-determined auxiliary variables (column '`eq`') become non-functionally-determined ones (column '`ineq`'). Boldface indicates overall best performance for the instance of that row, flagged by '+' if equal to the best-known value.

	fzn-oscar-cbls		Yuck		LocalSolver	
instance \ model	`eq`	`ineq`	`eq`	`ineq`	`eq`	`ineq`
n40w120	+**434**	–	436	468	+**434**	–
n40w140	+**328**	–	+**328**	391	+**328**	–
n40w160	352	–	+**348**	411	+**348**	–

– **Augmenting LS with CP:** Examples include using only the root-node propagation of a CP solver to try to check the feasibility of side constraints of capacitated vehicle routing problems and thereby try to find values for the auxiliary variables when probing an LS candidate move [5]; modelling an LS neighbourhood as an optimisation problem and using a CP solver to find a best neighbour to the current assignment [15]; and *large-neighbourhood search* (LNS) [17], where some variables in a feasible current LS assignment are fixed for a CP solver to look for a best assignment to the other variables, thereby building an LS move to another feasible current assignment.
– **Augmenting CP with LS:** Examples include performing LS starting from the leaf nodes of the CP search tree in order to improve solutions; performing LS at the internal nodes of the CP search tree in order to repair or improve a node [16]; and using LS in order to guide the CP branching strategy [12].

We have here augmented LS with CP in a manner most similar to [5] and LNS. Unlike [5], we allow the CP solver to *search* for a *best* assignment to the auxiliary variables when *feasible* values cannot be inferred through only *root-node propagation*; furthermore, we refine CMG and make it available in a problem-independent context. Like LNS, a partial assignment is here fixed for a CP solver, and complete search is made over the remaining variables. However, unlike LNS, the complete search is here on a subset of the constraints, always the same variables are here fixed, and we allow moves to *infeasible* current assignments.

In the MiniZinc context, LS-CP hybrids exist for LNS, namely the GELATO framework [4], combining their LS solver EasyLocal++ with the CP solver Gecode [9], and Mini-LNS [7], which is solver-independent, but neither of these are black-box and therefore both require a search strategy to be specified.

7 Conclusion and Future Work

We presented compound-move generation (CMG), an extension to black-box local search, geared for model structure that may cause local-search solvers to make moves reassigning variables representing auxiliary information. We have outlined two methods for detecting such model structure, which can appear naturally, for example for routing problems with side constraints. This means that such solvers without CMG might perform unexpectedly worse than complete solvers and considerably worse than handcrafted local-search algorithms on such problems. Our experiments show that several black-box local-search solvers are adversely affected in the presence of that model structure, and that CMG greatly improves the performance of our fzn-oscar-cbls backend to MiniZinc on such models, without requiring any model reformulation.

Future work includes extracting more information from the complete solver in order to help guide local search. For example, if an LCG solver is used as the complete solver for CMG, then learned clauses could be used to construct the next local-search move. Furthermore, making some constraints soft for the complete solver could help when infeasible solutions are explored and would improve on refinement B in Section 3.2.

References

1. Benoist, T., Estellon, B., Gardi, F., Megel, R., Nouioua, K.: LocalSolver 1.x: a black-box local-search solver for 0-1 programming. 4OR – A Quarterly Journal of Operations Research **9**(3), 299–316 (September 2011), LocalSolver is available at `https://www.localsolver.com`
2. Björdal, G., Monette, J.N., Flener, P., Pearson, J.: A constraint-based local search backend for MiniZinc. Constraints **20**(3), 325–345 (July 2015). https://doi.org/10.1007/s10601-015-9184-z, the fzn-oscar-cbls backend is available at `http://optimisation.research.it.uu.se/software`
3. Chu, G.: Improving Combinatorial Optimization. Ph.D. thesis, Department of Computing and Information Systems, University of Melbourne, Australia (2011), the Chuffed solver and MiniZinc backend are available at `https://github.com/chuffed/chuffed`
4. Cipriano, R., Di Gaspero, L., Dovier, A.: A multi-paradigm tool for large neighborhood search. In: Talbi, E. (ed.) Hybrid Metaheuristics, SCI, vol. 434, chap. 15, pp. 389–414. Springer (2013)
5. De Backer, B., Furnon, V., Shaw, P., Kilby, P., Prosser, P.: Solving vehicle routing problems using constraint programming and metaheuristics. Journal of Heuristics **6**(4), 501–523 (September 2000)
6. De Landtsheer, R., Ponsard, C.: OscaR.cbls: An open source framework for constraint-based local search. In: ORBEL-27, the 27th annual conference of the Belgian Operational Research Society (2013), available as `http://www.orbel.be/orbel27/pdf/abstract293.pdf`; the OscaR.cbls solver is available at `https://bitbucket.org/oscarlib/oscar/branch/CBLS`
7. Dekker, J.J., Garcia de la Banda, M., Schutt, A., Stuckey, P.J., Tack, G.: Solver-independent large neighbourhood search. In: Hooker, J. (ed.) CP 2018. LNCS, vol. 11008, pp. 81–98. Springer (2018)
8. Focacci, F., Laburthe, F., Lodi, A.: Local search and constraint programming. In: Glover, F.W., Kochenberger, G.A. (eds.) Handbook of Metaheuristics, ORMS, vol. 57, chap. 13, pp. 369–403. Springer (2003)
9. Gecode Team: Gecode: A generic constraint development environment (2018), the Gecode solver and MiniZinc backend are available at `http://www.gecode.org`
10. Glover, F., Laguna, M.: Tabu search. In: Modern Heuristic Techniques for Combinatorial Problems, pp. 70–150. John Wiley & Sons (1993)
11. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations & Applications. Elsevier / Morgan Kaufmann (2004)
12. Jussien, N., Lhomme, O.: Local search with constraint propagation and conflict-based heuristics. Artificial Intelligence **139**(1), 21–45 (July 2002)
13. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer (2007), the MiniZinc toolchain is available at `https://www.minizinc.org`
14. OscaR Team: OscaR: Scala in OR (2012), available at `https://oscarlib.bitbucket.io`
15. Pesant, G., Gendreau, M.: A constraint programming framework for local search methods. Journal of Heuristics **5**(3), 255–279 (October 1999)
16. Prestwich, S.: A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, pp. 337–352. Springer (2000)

17. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M., Puget, J.F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer (1998)
18. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc Challenge 2008–2013. AI Magazine **35**(2), 55–60 (summer 2014), see `https://www.minizinc.org/challenge.html`
19. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. The MIT Press (2005)