Exploring Declarative Local-Search Neighbourhoods with Constraint Programming

Gustav Björdal✉[0000−0002−8032−5774]1, Pierre Flener[0000−0001−8730−4098]1, Justin Pearson[0000−0002−0084−8891]1, and Peter J. Stuckey[0000−0003−2186−0459]2

[1] Uppsala University, Department of Information Technology, Uppsala, Sweden
{Gustav.Bjordal,Pierre.Flener,Justin.Pearson}@it.uu.se
[2] Monash University, Faculty of Information Technology, Melbourne, Australia
Peter.Stuckey@monash.edu

Abstract Using constraint programming (CP) to explore a local-search neighbourhood was first tried in the mid 1990s. The advantage is that constraint propagation can quickly rule out uninteresting neighbours, sometimes greatly reducing the number actually probed. However, a CP model of the neighbourhood has to be handcrafted from the model of the problem: this can be difficult and tedious. That research direction appears abandoned since large-neighbourhood search (LNS) and constraint-based local search (CBLS) arose as alternatives that seem easier to use. Recently, the notion of declarative neighbourhood was added to the technology-independent modelling language MiniZinc, for use by any backend to MiniZinc, but currently only used by a CBLS backend. We demonstrate that declarative neighbourhoods are indeed technology-independent by using the old idea of CP-based neighbourhood exploration: we explain how to encode automatically a declarative neighbourhood into a CP model of the neighbourhood. This enables us to lift any CP solver into a local-search backend to MiniZinc. Our prototype is competitive with CP, CBLS, and LNS backends to MiniZinc.

1 Introduction

Technology-independent modelling is a paradigm where we model a problem and choose among solvers of several technologies in order to solve it for given data. This helps avoid early commitment to a technology and solver, and enables the easy comparison of technologies and solvers on the same model. MiniZinc [16] is a technology-independent modelling language, supported by solvers of many technologies, such as constraint programming (CP), lazy clause generation (LCG), integer programming (IP), Boolean satisfiability (SAT), satisfiability modulo theories (SMT), constraint-based local search (CBLS [23]), and hybrids.

Many solvers can work in a black-box way, where we only need to provide a model and the instance data. Some technologies, notably CP and LCG, also allow a search strategy to be attached to a model; in practice, a good search strategy is often key to an efficient solving process. MiniZinc has had from its inception a

notation for declaratively indicating a search strategy for CP and LCG solvers. Recently, MiniZinc was extended with a notation [3] for declaratively specifying a local-search *neighbourhood*, that is a set of candidate moves that re-assign some variables within the current valuation of a local-search method. At present, these *declarative neighbourhoods* are only supported by the CBLS backend fzn-oscar-cbls [4]: experiments showed that, as expected, CBLS via MiniZinc can be accelerated via a neighbourhood specification.

In this paper, we revisit the idea of encoding a local-search neighbourhood for a CP solver so that, given the current valuation of the variables in a local-search method, the CP solver finds by systematic search a best neighbour of that current valuation, under some heuristic. The local-search method then moves to that neighbour, using some meta-heuristic for escaping local optima, such as simulated annealing or tabu search. The idea of encoding a neighbourhood was first proposed in [18] and then refined in [21]: exploring a neighbourhood by using a CP solver on such a *neighbourhood model* can lead to the efficient pruning of both infeasible and sub-optimal neighbours, sometimes greatly reducing the number of actually probed neighbours. However, this idea was presented as a methodology, where a neighbourhood model has to be handcrafted from the problem model, and there is limited reusability of encodings between neighbourhoods.

Large-neighbourhood search (LNS) [20] is another popular method for performing local search by using a CP solver. An LNS neighbourhood is constructed by *freezing* some variables, that is fixing them to their values in the current valuation of a local-search method, and it is explored by performing systematic search on the problem model in order to find an improving valuation for the remaining variables. However, this is fundamentally different from the neighbourhoods classically explored by CBLS solvers and ad hoc local-search methods. For example, a relocation neighbourhood (e.g., [**?**, Chapter 23]) cannot be explored by LNS without also exploring a very large number of neighbours that are not obtainable by relocation moves: LNS does not have the classical notion of move. Another crucial difference is that LNS uses *one* copy of the variables of the problem model and freezes *some* variables in each move, whereas a neighbourhood model uses *two* copies of the variables and freezes *most* variables in each move.

Putting all these ingredients together, the organisation and contributions of this paper are as follows:

- an encoding of a declarative neighbourhood [3] for any CP solver;
- the new global constraint Writes for encoding local-search moves;
- a good definition of Writes using constraints available in most CP solvers;
- a recipe for building a local-search backend to MiniZinc from any CP solver;
- evidence that declarative neighbourhoods are technology-independent.

We wrap the paper up with an experimental evaluation of our prototype against CP, CBLS, and LNS backends to MiniZinc, as well as directions for future work.

2 Background

After discussing in Section 2.1, with the example of a vehicle routing problem, the technology-independent modelling language MiniZinc and its extension with the notion of declarative neighbourhood [3], conceived for local-search backends, we briefly summarise in Section 2.2 the principles of local search (e.g., [14]).

For brevity, we discuss constrained minimisation problems: the maximisation of an objective function amounts to minimising its opposite, and the satisfaction of constraints amounts to satisfying them while minimising a constant.

2.1 MiniZinc and Declarative Neighbourhoods

Conceptually, a MiniZinc model for a constrained minimisation problem is in this paper a tuple $\langle \mathcal{V}, \mathcal{C}, o, \mathcal{S}, \mathcal{N} \rangle$, where $\mathcal{V}$ is the set of variables; $\mathcal{C}$ is the set of constraints on these variables, including their domain membership constraints; the variable $o \in \mathcal{V}$ is the *objective variable*, whose value is to be minimised; $\mathcal{S}$ is the optional annotation for suggesting a systematic-search branching strategy on the variables; and $\mathcal{N}$ is the optional annotation for suggesting a declarative neighbourhood [3].

As a running example, we use the model for the travelling salesperson problem with time windows (TSPTW) used in [2], but extended with a relocation neighbourhood. Given are `n` locations; an array `TravTime`, where `TravTime[i,j]` is the travel time from location `i` to location `j` plus the service time at `i`; and an array `ArrWin` of arrival-time windows, where `ArrWin[i,1]` is the earliest arrival time and `ArrWin[i,2]` the latest arrival time at location `i`. The objective is to find a shortest Hamiltonian circuit that visits each location exactly once and within its arrival-time window.

Listing 1 has a MiniZinc model for TSPTW with a relocation neighbourhood, with the data above declared in lines 1 to 3. The route is modelled in line 4 by an 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 in line 6 to be the depot, that is the start of the route. The arrival times are modelled in line 7 using the array `ArrTime`, where variable `ArrTime[i]` denotes the arrival time at location `i`. Each arrival time is constrained, in lines 8 to 11, to be at least either the arrival time at the preceding location plus the travel time, or the start of its arrival-time window, whichever is greater, and at most the end of its arrival-time window. The objective is to minimise the travel time of the entire circuit, which is stated in lines 12 and 15.

The relocation neighbourhood is used in the annotation (prefixed by `::`) of line 13 and declared in lines 16 to 22. It considers in line 18 all combinations of two locations `i` and `j` such that they are distinct and the predecessor of `i` is not `j`: this is prescribed by the `where` pre-condition in line 18 on the elements of the `moves` set comprehension of candidate moves. Each *candidate move* consists of the composition of three parallel re-assignments that relocates the predecessor of `i` so that it goes between `j` and the predecessor of `j`, as prescribed

```
1  int: n;  set of int: Loc = 1..n; % number and set of locations
2  array[Loc,Loc] of int: TravTime; % travel times
3  array[Loc,1..2] of int: ArrWin; % arrival-time windows: earliest, latest
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: ArrTime; % arrival times
8  constraint ArrTime[depot] = ArrWin[depot,1];
9  constraint forall(i in Loc where i != depot)(
10   ArrTime[i] = max(ArrTime[Pred[i]]+TravTime[Pred[i],i], ArrWin[i,1]));
11 constraint forall(i in Loc)(ArrTime[i] <= ArrWin[i,2]);
12 var int: time = sum(i in Loc)(TravTime[Pred[i],i]); % objective variable
13 solve :: use_neighborhood(relocate())
14       :: int_search(Pred,first_fail,indomain_min,complete)
15   minimize time;
16 function ann: relocate() :: neighborhood_definition =
17   initially(circuit(Pred)) /\
18   moves(i, j in Loc where i != j /\ Pred[i] != j)(
19     Pred[i] := Pred[Pred[i]] /\
20     Pred[j] := Pred[i] /\
21     Pred[Pred[i]] := Pred[j] % /\ ensuring(circuit(Pred)) % implied
22   );
```

Listing 1. A MiniZinc model for TSPTW and a relocation neighbourhood.

by lines 19 to 21. The initialisation post-condition of the neighbourhood is that `Pred` forms a Hamiltonian circuit, as prescribed by the `initially` condition in line 17: every (re-)start must be from a valuation satisfying this condition. Together, this initialisation post-condition, the pre-condition on candidate moves, and the nature of the candidate moves imply that each candidate move reaches a valuation of `Pred` that forms a Hamiltonian circuit, so we do not need to include the commented-out `ensuring` post-condition on candidate moves in line 21.

The `initially`, `where`, and `ensuring` conditions of a declarative neighbourhood are constraint satisfaction problems, expressed on the data and variables of the problem model, using the existing and full MiniZinc syntax.

2.2 Local Search

Given a model $\langle \mathcal{V}, \mathcal{C}, o, \mathcal{S}, \mathcal{N} \rangle$ for a constrained minimisation problem, a *local-search method* iteratively maintains a *current valuation* θ that maps each variable in $\mathcal{V}$ to a value in its domain prescribed in $\mathcal{C}$, and that is initialised under some amount of randomisation so as to satisfy the initialisation condition of $\mathcal{N}$, usually a subset of the constraints in $\mathcal{C}$. At each iteration, the local-search method considers the set of candidate moves defined by the neighbourhood $\mathcal{N}(\theta)$: it *selects* under some amount of randomisation a candidate move, as specified by some *heuristic* such as best-improving or first-improving, and *makes* the selected

candidate move by updating the current valuation θ accordingly. The idea is that each move made should reduce the value of some *cost function* $\text{cost}(\theta)$, which does not necessarily return the current objective value $\theta(o)$, as seen below.

In order to escape local optima of $\text{cost}(\theta)$, a *meta-heuristic*, such as simulated annealing or tabu search [12], is used. Together, the neighbourhood $\mathcal{N}$, the heuristic, and the meta-heuristic form the *local-search strategy* of the local-search method. A local-search method typically also involves *restarts*, where periodically the search may be begun again from scratch, in order to avoid being trapped in local minima. It also may use *intensification*, where more search effort is applied around a current valuation that seems promising.

All constraints in $\mathcal{C}$ are to be satisfied. Existing local-search backends to MiniZinc automatically choose for each constraint among three ways of handling it:

- A constraint $c \in \mathcal{C}$ can be satisfied when initialising the current valuation θ and its satisfaction can be preserved by all candidate moves. Hence c is *hard*: it cannot be violated during search.
- A constraint c that functionally defines some variable $v \in \mathcal{V}$ in terms of other variables $W \subset \mathcal{V}$ can be made hard by extending every made move on at least one variable in W into also re-assigning v accordingly.
- A constraint c can be made *soft*, meaning it can be violated during search but should be satisfied in the final valuation, by using a *violation function* giving 0 if c is satisfied under θ, and otherwise a positive value that indicates how violated c is under θ. For example, for linear expressions x and y, the linear constraints $x = y$ and $x \leq y$ are softened [23] into $v_1 = |x - y|$ and $v_2 = \textbf{if } x \leq y \textbf{ then } 0 \textbf{ else } x - y \textbf{ endif}$, respectively, defining an introduced *violation variable* v_i.

For example, for the model in Listing 1, a typical way of handling its constraints is: the `circuit` constraint in line 5 is satisfied by every valuation explored during the search; the constraint that functionally defines `time` in line 12 is made hard and moves do not consider changing this variable; the time-window end constraint in line 11 is made soft; finally, although the constraints in lines 8 to 10 define the `ArrTime[i]` variables functionally, it is hard to detect that the definition is not circular: hence they are made soft and moves must consider changing these variables.

Let $\text{soft}(C, g)$ denote the constraint set where some constraints in C are softened under some scheme, including a new variable g, denoting the *global violation*, constrained to be the sum of all the introduced violation variables. We assume that the individual violation variables and the global violation variable g are implicitly added to the variable set of the model containing C. Replacing C by $\text{soft}(C, g)$ requires changing the model containing C to minimising both the objective variable o and the global violation variable g. For example, in fzn-oscar-cbls [4], a weighted sum $\alpha \cdot o + \beta \cdot g$ is used as the cost function $\text{cost}(\theta)$, where the values of α and β are dynamically tuned during search.

If too many constraints are made hard, then this may *disconnect* the search space, since local search only moves from one valuation to another via a move of the neighbourhood: it may be that no sequence of moves in the declarative

neighbourhood are able to move between two given valuations. If this is the case, then we can seriously weaken the local-search capability to find good valuations.

Given a neighbourhood $\mathcal{N}$, we can partition the variables $\mathcal{V}$ of a model into three sub-sets: $\mathcal{V}_{\text{targ}}$ is the set of variables that are targeted by the moves of $\mathcal{N}$ (such as the array `Pred` in line 4 of Listing 1); $\mathcal{V}_{\text{func}}$ is the set of non-targeted variables that are each functionally defined by some constraint (such as `time` being functionally defined by `Pred` in line 12); and $\mathcal{V}_{\text{aux}}$ is the set of the remaining variables, which we call *auxiliary variables* (such as the array `ArrTime` in line 7). Search must be over $\mathcal{V}_{\text{targ}} \cup \mathcal{V}_{\text{aux}}$, but local search over $\mathcal{V}_{\text{targ}} \cup \mathcal{V}_{\text{aux}}$ was shown in [2] to degrade greatly the performance of CBLS solvers, unless every move on $\mathcal{V}_{\text{targ}}$ is somehow automatically extended by a corresponding re-assignment of $\mathcal{V}_{\text{aux}}$.

3 Encoding a Declarative Neighbourhood as a CP Model

We show how to encode automatically a declarative neighbourhood, specified in MiniZinc, as a CP model, which we call the *neighbourhood model*. We show in Section 3.1 how to encode two states of a local-search method, namely its current and next valuations, using variables. We explain in Section 3.2 how to encode a move as constraints on these variables. The exploration of the neighbourhood then amounts to solving the neighbourhood model, as discussed in Section 3.3.

3.1 Encoding the Current and Next Valuations

Given a MiniZinc model $\langle \mathcal{V}, \mathcal{C}, o, \mathcal{S}, \mathcal{N} \rangle$, we extract the following sets:

- $\mathcal{V}_{\text{gen}}$ has variables for the generators of the `moves` set comprehension of $\mathcal{N}$;
- $\mathcal{M}$ has the move expressions of the `moves` comprehension of $\mathcal{N}$;
- $\mathcal{V}_{\text{targ}} \subseteq \mathcal{V}$ has the *targeted variables* of $\mathcal{V}$, that is those re-assigned in $\mathcal{M}$;
- $\mathcal{C}_{\text{where}}$ has the constraints on $\mathcal{V} \cup \mathcal{V}_{\text{gen}}$ of the `where` pre-condition of $\mathcal{N}$; and
- $\mathcal{C}_{\text{ensure}}$ has the constraints on $\mathcal{V} \cup \mathcal{V}_{\text{gen}}$ of the `ensuring` post-condition of $\mathcal{N}$.

For example, for the model in Listing 1, the set $\mathcal{V}_{\text{gen}}$ has variables, called *generator variables*, for the generators `i` and `j` in line 18; the set $\mathcal{M}$ has the three re-assignments in lines 19 to 21; the set $\mathcal{V}_{\text{targ}}$ has the entire array `Pred` since any variable thereof can be referred to in the left-hand sides of the re-assignments in $\mathcal{M}$; the set $\mathcal{C}_{\text{where}}$ has the two constraints of the `where` pre-condition in line 18; and the set $\mathcal{C}_{\text{ensure}}$ is empty since there is no `ensuring` post-condition.

Since our encoding must reason on the current and next valuations of the variables in a local-search method, we must use in the neighbourhood model two copies of some variables of the given problem model: for a set X of variables, we denote by X^{c} the set of variables corresponding to X in the current valuation, and by X^{n} the set of variables corresponding to X in the next valuation. We use the same notation for individual variables.

The variable set of the *neighbourhood model* is $\mathcal{V}^{\text{c}} \cup \mathcal{V}^{\text{n}} \cup \mathcal{V}^{\text{c}}_{\text{gen}}$. We give its constraint set in Section 3.3, after focussing on the constraints encoding a move.

3.2 Encoding a Move

A move is a transition from $\mathcal{V}^{\mathrm{c}}_{\mathrm{targ}}$ to $\mathcal{V}^{\mathrm{n}}_{\mathrm{targ}}$, so we must constrain each variable in $\mathcal{V}^{\mathrm{n}}_{\mathrm{targ}}$ to take the same value as its corresponding variable in $\mathcal{V}^{\mathrm{c}}_{\mathrm{targ}}$, except those re-assigned by the move, which are constrained to take new values accordingly.

Towards encoding this, we introduce the constraint $\textsc{Writes}(O, I, P, V)$ on two arrays O and I of the same number n of variables and two arrays P and V of the same number m of variables: it holds if and only if O, called the *output array*, is point-wise equal to I, called the *input array*, except that $O[P[j]]$ is constrained to be equal to $V[j]$ for each j in $\{1, \ldots, m\}$. We assume that all indexing in this paper starts from 1.

We encode using Writes the set $\mathcal{M}$ of move expressions. The *basic moves* are x `:=` y, $X[i]$ `:=` y, x `:=:` y, and $X[i]$ `:=:` $Y[j]$, specified and encoded as follows:

- x `:=` y means re-assign to x the current value of y, which is encoded as either $x^{\mathrm{n}} = y^{\mathrm{c}}$ or $\textsc{Writes}([x^{\mathrm{n}}], [x^{\mathrm{c}}], [1], [y^{\mathrm{c}}])$;
- $X[i]$ `:=` y means re-assign to $X[i]$ the current value of y, which is encoded as $\textsc{Writes}(X^{\mathrm{n}}, X^{\mathrm{c}}, [i^{\mathrm{c}}], [y^{\mathrm{c}}])$;
- x `:=:` y means swap the current values of x and y, which is encoded as either $x^{\mathrm{n}} = y^{\mathrm{c}} \wedge y^{\mathrm{n}} = x^{\mathrm{c}}$ or $\textsc{Writes}([x^{\mathrm{n}}, y^{\mathrm{n}}], [x^{\mathrm{c}}, y^{\mathrm{c}}], [1, 2], [y^{\mathrm{c}}, x^{\mathrm{c}}])$;
- $X[i]$ `:=:` $Y[j]$ means swap the current values of $X[i]$ and $Y[j]$, which is encoded the way the compound move $X[i]$ `:=` $Y[j]$ `/\` $Y[j]$ `:=` $X[i]$ is; see below.

The first and third Writes-based encodings are only useful when we merge them with others in order to preserve the semantics of moves, as discussed next.

A *compound move* is the parallel composition of basic moves, which is written by overloading the `/\` logical-and connective. The composition of basic moves that always re-assign different variables, such as $X[i]$ `:=` u `/\` $Y[j]$ `:=` v when the arrays X and Y share no variables, is the conjunction of the encodings of the basic moves. However, the composition of basic moves that can re-assign the same variable, such as $X[i]$ `:=` u `/\` $X[j]$ `:=` v, *must* be encoded by merging the encodings of the basic moves, since $\textsc{Writes}(X^{\mathrm{n}}, X^{\mathrm{c}}, [i^{\mathrm{c}}], [u^{\mathrm{c}}])$ requires $\forall k \neq i^{\mathrm{c}} : X^{\mathrm{n}}[k] = X^{\mathrm{c}}[k]$, which prevents any value other than $X^{\mathrm{c}}[j^{\mathrm{c}}]$ from being written at index j^{c} by $\textsc{Writes}(X^{\mathrm{n}}, X^{\mathrm{c}}, [j^{\mathrm{c}}], [v^{\mathrm{c}}])$, unless $j^{\mathrm{c}} = i^{\mathrm{c}}$.

Rules 1, 2, and 3 below show how to merge Writes constraints; we only give rules for the cases that can appear in our prototype backend to MiniZinc:

Rule 1. The constraints $\textsc{Writes}(O, I, P, V)$ and $\textsc{Writes}([x], [y], [1], [v])$, where for some constant index p we have that $O[p]$ is x and $I[p]$ is y, are merged into $\textsc{Writes}(O, I, P +\!\!+ [p], V +\!\!+ [v])$, where $+\!\!+$ denotes array concatenation.

Rule 2. The constraints $\textsc{Writes}(O, I, P_1, V_1)$ and $\textsc{Writes}(O, I, P_2, V_2)$ on the same output and input arrays are merged into $\textsc{Writes}(O, I, P_1 +\!\!+ P_2, V_1 +\!\!+ V_2)$.

Rule 3. Consider $\textsc{Writes}(O_1, I_1, P_1, V_1)$ and $\textsc{Writes}(O_2, I_2, P_2, V_2)$, where a non-empty set J has the indices j for which there exists an index i such that $O_2[j]$ is $O_1[i]$ and $I_2[j]$ is $I_1[i]$. Let O_2 and I_2 have length n. Let $O' = O_1 +\!\!+ [O_2[k] \mid$

$k \in \{1,\ldots,n\} \setminus J]$ and $I' = I_1 +\!\!+ [I_2[k] \mid k \in \{1,\ldots,n\} \setminus J]$ be the non-redundant mergers of the two O_ℓ arrays and the two I_ℓ arrays, respectively. Let M be the array that maps indices of I_2 to indices of I' defined so that $\forall i \in \{1,\ldots,n\} : O_2[i] = O'[M[i]] \wedge I_2[i] = I'[M[i]]$. Let P_2 and V_2 have length m. Let $P' = P_1 +\!\!+ [M[P_2[i]] \mid i \in \{1,\ldots,m\}]$. The two Writes constraints above are merged into $\textsc{Writes}(O', I', P', V_1 +\!\!+ V_2)$.

For example, for the model in Listing 1, the compound move is encoded, after maximal merging, as the single constraint $\textsc{Writes}(\texttt{Pred}^{\text{n}}, \texttt{Pred}^{\text{c}}, [\texttt{i}^{\text{c}}, \texttt{j}^{\text{c}}, \texttt{Pred[i]}^{\text{c}}], [\texttt{Pred[Pred[i]}^{\text{c}}\texttt{]}^{\text{c}}, \texttt{Pred[i]}^{\text{c}}, \texttt{Pred[j]}^{\text{c}}])$.

Let the maximally merged encodings of the move expressions in $\mathcal{M}$, together with the constraint $\exists v \in \mathcal{V}_{\text{targ}} : v^{\text{n}} \neq v^{\text{c}}$ requiring at least one variable in $\mathcal{V}^{\text{n}}_{\text{targ}}$ to be different from the corresponding one in $\mathcal{V}^{\text{c}}_{\text{targ}}$, form the constraint set $\mathcal{C}_{\text{move}}$.

3.3 The Neighbourhood Model and Neighbourhood Exploration

The *neighbourhood model* has the variable set $\mathcal{V}^{\text{c}} \cup \mathcal{V}^{\text{n}} \cup \mathcal{V}^{\text{c}}_{\text{gen}}$ mentioned in Section 3.1 and the following constraint set for channelling between $\mathcal{V}^{\text{c}}$ and $\mathcal{V}^{\text{n}}$:

- the set $\mathcal{C}_{\text{where}}\{\mathcal{V}/\mathcal{V}^{\text{c}}, \mathcal{V}_{\text{gen}}/\mathcal{V}^{\text{c}}_{\text{gen}}\}$, for meeting the `where` pre-condition;
- the set $\mathcal{C}_{\text{move}}$ defined at the end of Section 3.2, for encoding a move;
- the set $\text{soft}(\mathcal{C}, g)\{\mathcal{V}/\mathcal{V}^{\text{n}}\}$, for evaluating and pruning neighbours; and
- the set $\mathcal{C}_{\text{ensure}}\{\mathcal{V}/\mathcal{V}^{\text{n}}, \mathcal{V}_{\text{gen}}/\mathcal{V}^{\text{c}}_{\text{gen}}\}$, for meeting the `ensuring` post-condition.

where $R\{X/Y\}$ denotes the copy of the constraint set R where the variables of the set X are point-wise substituted by those of the same-sized set Y of variables.

A declarative neighbourhood can have the union of several `moves` set comprehensions with possibly different pre- and post-conditions, effectively giving the union of sub-neighbourhoods. In order to encode such a neighbourhood, we propose that each sub-neighbourhood be separately encoded in its own neighbourhood model, each being explored under its own instantiation of a CP solver. We believe that the disjunctive encoding of the sub-neighbourhoods would be at most as efficient as encoding and exploring the sub-neighbourhoods separately.

Thus, given the current valuation θ of a local-search method, exploring its neighbourhood amounts to solving the neighbourhood model, but with the additional constraints $\{v^{\text{c}} = \theta(v) \mid v \in \mathcal{V}\}$ for enforcing θ, using a CP solver: either we apply systematic search in order to find one or all neighbours, or we add to the neighbourhood model the objective function that corresponds to the cost function of the local-search method and apply systematic branch-and-bound search in order to find a best neighbour and prune sub-optimal ones on-the-fly.

4 Implementing a Local-Search Solver Using a CP Solver

Since we can explore a declarative neighbourhood using a CP solver, we now show how to lift any CP solver into a local-search backend for MiniZinc. We use OscaR.cp [17] in order to implement our prototype backend, called LS(cp).

```
predicate writes(array[int] of var int: O, array[int] of var int: I,
                 array[int] of var int: P, array[int] of var int: V) =
  forall(j in index_set(P))(O[P[j]] = V[j]) /\
  forall(i in index_set(I) where forall(j in index_set(P))(P[j]!=i))(
    O[i]=I[i]);
```

Listing 2. A straightforward definition of the WRITES constraint in MiniZinc syntax.

```
predicate writes(array[int] of var int: O, array[int] of var int: I,
                 array[int] of var int: P, array[int] of var int: V) =
let { int: k = min(index_set(P));
      array[index_set(I)] of var 0 .. length(P): S;
} in forall(i in index_set(I))(S[i] = 0 -> O[i] = I[i] /\
            forall(j in 1..length(P))(S[i] = j -> P[j+k-1] = i)) /\
  alldifferent_except_0(S) /\forall(j in index_set(P))(O[P[j]] = V[j]);
```

Listing 3. An improved definition of the WRITES constraint in MiniZinc syntax.

We use only existing components of OscaR.cp (including the FlatZinc parser of the CBLS solver OscaR.cbls [7,4] of the same OscaR framework), provide a good implementation of WRITES (Section 4.1), motivate a particular constraint softening scheme (Section 4.2), and discuss the control flow (Section 4.3).

4.1 Implementation of the WRITES Global Constraint

A straightforward definition (or: decomposition) of the WRITES(O, I, P, V) constraint of Section 3.2 is given in Listing 2 using MiniZinc syntax. We also propose the improved definition in Listing 3, which reasons on a matching between the variables of P and O: an array S denotes for each index i of I if its element is unchanged in O (when $S_i = 0$) or denotes the value at index j of P that determines its change (when $S_i = j$). The improved definition propagates WRITES$([1..3, 1..3, 1..3], [4, 4, 4], [1..3, 2..3, 2..3], [1, 1, 1])$, where $\ell..u$ denotes a variable of that domain, to WRITES$([1, 1..3, 1..3], [4, 4, 4], [1, 2..3, 2..3], [1, 1, 1])$, whereas the first definition propagates nothing. However, neither achieves domain consistency, namely WRITES$([1, 1, 1], [4, 4, 4], [1, 2..3, 2..3], [1, 1, 1])$. Given that the max-clique problem reduces to achieving domain consistency on a WRITES constraint, domain-consistent propagation is NP-hard and we do not investigate this further in this paper. In practice, we provide special cases in the definition when O and I have length $n \in \{1, 2\}$, capturing the WRITES-free encodings of the x `:=` y and x `:=:` y moves shown in Section 3.2.

Algorithm 1 Control flow of a CP-based local-search backend to MiniZinc.

while no time-out **do**
  $\theta \coloneqq \text{initialise}(\tau_{\text{init}})$ {create a new current valuation}
  while no time-out and no restart **do** {the meta-heuristic decides when to restart}
    $\theta' \coloneqq \text{explore}(\theta, \tau_{\text{explo}})$ {select a neighbour}
    $\theta \coloneqq \text{intensify}(\theta', \tau_{\text{intens}})$ {improve the selected neighbour}

4.2 Constraint Softening Scheme

As hard constraints decrease the neighbourhood size and exploration time [18], it can be beneficial to soften only a few constraints, if any. We argue that one should soften neither constraints that functionally define variables, nor constraints on auxiliary variables. The former rule is what the MiniZinc CBLS backend fzn-oscar-cbls [4] does, and the latter rule allows us to leverage propagation for determining values of the auxiliary variables, which was shown to be beneficial in [2]. In our LS(cp), the soft(C, g) operator of Section 2.2 softens each linear (in)equality constraint that neither functionally defines some variable nor constrains auxiliary variables, and we change the objective function into the cost function $\alpha \cdot o + \beta \cdot g$, as in [4], but currently statically with $\alpha = 1 = \beta$.

4.3 Control Flow

Given a MiniZinc model $\langle \mathcal{V}, \mathcal{C}, o, \mathcal{S}, \mathcal{N} \rangle$ with a declarative neighbourhood $\mathcal{N}$, our local-search backend consists of three major components — initialisation, exploration, and intensification — which are used under the control flow in Algorithm 1. Each component has its own instantiation of a CP solver for its own model, but the exploration has several in case of sub-neighbourhoods.

Initialisation. Let $\mathcal{C}_{\text{init}}$ denote the constraint set in the initialisation post-condition of $\mathcal{N}$: the *initialisation model* is $\langle \mathcal{V},\ \text{soft}(\mathcal{C}, g) \cup \mathcal{C}_{\text{init}},\ o + g,\ \mathcal{S}_{\text{init}},\ \_ \rangle$, where the systematic-search strategy $\mathcal{S}_{\text{init}}$ is a randomisation of $\mathcal{S}$, if present, and otherwise a randomising default strategy. The CP solver is limited to return the best solution found within τ_{init} seconds, or, if no solution was found yet, then to return the first solution found thereafter. The CP solution returned by initialise(τ_{init}) is used as the initial or re-start valuation θ by the local search.

Exploration. Let $\mathcal{C}_{\text{explo}}$ denote the four constraint sets at the start of Section 3.3: the *neighbourhood model* is $\langle \mathcal{V}^{\text{c}} \cup \mathcal{V}^{\text{n}} \cup \mathcal{V}^{\text{c}}_{\text{gen}},\ \mathcal{C}_{\text{explo}},\ o^{\text{n}} + g^{\text{n}},\ \mathcal{S}_{\text{explo}},\ \_ \rangle$. We describe everything for a trail-based CP solver. Before applying the constraints $\{v^{\text{c}} = \theta(v) \mid v \in \mathcal{V}\}$ for enforcing the given current valuation θ of the local search, a choice point (recording the current state of the CP solver) is pushed onto the trail of the CP solver. This allows us to backtrack, when the local search has a new current valuation θ, to the choice point before the variables were fixed, by popping the trail, thus reusing rather than re-building the

neighbourhood model in the next iteration of local search. Note that in an LCG solver this also allows us to keep all generated nogoods, since any dependence on the current valuation θ is included in the nogood.

The search strategy $\mathcal{S}_{\text{explo}}$ is similar to the one of [21], which argues that branching by domain bisection on the generator variables $\mathcal{V}^{\text{c}}_{\text{gen}}$ propagates better. However, in general this only guarantees fixing the targeted variables $\mathcal{V}^{\text{n}}_{\text{targ}} \subseteq \mathcal{V}^{\text{n}}$ by propagation. We therefore then also branch on the remaining variables.

Many local-search heuristics for selecting a neighbour are easy to implement. For example, for the first-improving heuristic used by our LS(cp), we limit the CP branch-and-bound search to stop after finding a solution within τ_{explo} seconds where $\langle g^{\text{n}}, o^{\text{n}}\rangle$ is lexicographically strictly less than $\langle g^{\text{c}}, o^{\text{c}}\rangle$; if no such solution was found yet, then the best solution found so far is returned, if any. Further, the best-improving heuristic can be implemented by searching exhaustively.

We implement a greedy local-search phase by enforcing g^{c} and o^{c} as upper bounds on g^{n} and o^{n}: this prunes all non-improving neighbours to θ. After some iterations, once θ is a local minimum, these bounds will empty the neighbourhood and we end the greedy local-search phase by no longer using these bounds. This allows the best-found CP solution to become a non-improving neighbour.

Let σ be the solution returned by the CP solver, if any: explore$(\theta, \tau_{\text{explo}})$ returns the valuation $\theta' = \{v \mapsto \sigma(v^{\text{n}}) \mid v \in \mathcal{V}\}$ if σ is defined, otherwise $\theta' = \theta$.

Intensification. Let θ' be the local-search valuation of $\mathcal{V} = \mathcal{V}_{\text{targ}} \cup \mathcal{V}_{\text{func}} \cup \mathcal{V}_{\text{aux}}$ returned by the exploration; recall the end of Section 2.2 for the semantics of these variable sets. The projection of θ' onto only $\mathcal{V}_{\text{targ}}$ may have several extensions for $\mathcal{V}_{\text{func}}$ and $\mathcal{V}_{\text{aux}}$ that have a better objective value than under θ'. This may happen for example upon a first-improving heuristic. In order to try and improve θ', the function intensify$(\theta', \tau_{\text{intens}})$ calls a CP solver on the *intensification model* $\langle \mathcal{V},\ \text{soft}(\mathcal{C}, g) \cup \{v = \theta'(v) \mid v \in \mathcal{V}_{\text{targ}}\},\ o + g,\ \mathcal{S},\ \_\rangle$, where $\mathcal{S}$ is from the original model. The CP solver is limited to return a best solution found within τ_{intens} seconds, if any. This intensification is essentially a single LNS iteration where $\mathcal{V}_{\text{targ}}$ is frozen and values for $\mathcal{V}_{\text{func}} \cup \mathcal{V}_{\text{aux}}$ are sought.

Let σ be the solution returned by the CP solver, if any: intensify$(\theta', \tau_{\text{intens}})$ returns the valuation $\theta = \{v \mapsto \sigma(v) \mid v \in \mathcal{V}\}$ if σ is defined, otherwise $\theta = \theta'$.

Meta-Heuristic: Tabu Search, Aspiration, and Restarts. In order to help local search escape local minima, we improve on Algorithm 1 by implementing a tabu search meta-heuristic by extending explore$(\theta, \tau_{\text{explo}})$ to return also which variables in $\mathcal{V}_{\text{targ}}$ were re-assigned in the selected neighbour: after the latter is intensified into the new current valuation, each re-assigned variable in $\mathcal{V}_{\text{targ}}$ is called *tabu* for $\delta + u$ local-search iterations, where u is taken uniformly at random between 0 and the *tabu tenure* δ. Before starting exploration, each tabu variable in $\mathcal{V}^{\text{n}}_{\text{targ}}$ is required to be equal to its corresponding variable in $\mathcal{V}^{\text{c}}_{\text{targ}}$. We need not do this algorithmically: the constraint $\forall v \in \text{tabu}(\mathcal{V}_{\text{targ}}) : v^{\text{n}} = v^{\text{c}}$, where tabu$(\mathcal{V}_{\text{targ}})$ denotes the set of tabu variables, is added to the neighbourhood model before starting CP search on it.

We improve this tabu search by adding the *aspiration criterion* that allows re-assigning tabu variables if this yields a new overall best valuation, under lexicographic order on $\langle g, o \rangle$. This is achieved by instead posting the constraint $g^{\mathrm{n}} < \mathrm{best}(g) \vee (g^{\mathrm{n}} = \mathrm{best}(g) \wedge o^{\mathrm{n}} < \mathrm{best}(o)) \vee \forall v \in \mathrm{tabu}(\mathcal{V}_{\mathrm{targ}}) : v^{\mathrm{n}} = v^{\mathrm{c}}$, where $\mathrm{best}(v)$ denotes the value of variable v in the overall best valuation.

In order to further help the local search escape local minima, we also implement a *restart mechanism*: if the exploration step does not return a new valuation for γ iterations of local search or does not improve the overall best valuation since the last restart for λ iterations, then a restart is made. Our LS(cp) performs restarts from the initialisation step. Recall that the initialisation step uses randomisation in its branching strategy.

5 Experimental Evaluation

Our aim is to show that declarative neighbourhoods are technology independent and enable lifting any CP solver into also being a MiniZinc local-search backend.

We evaluate our prototype LS(cp) against fzn-oscar-cbls [4], a CBLS backend that uses declarative neighbourhoods (but can be run black-box); Yuck,[3] a CBLS backend that only runs black-box; Gecode [11], a CP backend that uses CP search annotations such as line 14 of Listing 1; and Gecode-lns, an LNS backend that uses Gecode upon adding the `relax_and_reconstruct` annotation, with an 80% probability of freezing for each variable, to the MiniZinc model and the `-restart luby` flag when running the backend. These settings for Gecode-lns were decided upon after observing robust performance in initial experiments. We use two configurations of LS(cp) in order to see the impact of better neighbour pruning at the cost of possibly disconnecting the search space: LS(cp)-soft uses the constraint softening scheme of Section 4.2, and LS(cp)-hard sets $\mathrm{soft}(C, g) = C$. For both configurations, we use the parameters $\tau_{\mathrm{init}} = 10\mathrm{s}$, $\tau_{\mathrm{explo}} = 30\mathrm{s}$, $\tau_{\mathrm{intens}} = 10\mathrm{s}$, $\delta = 0.1 \cdot |\mathcal{V}_{\mathrm{targ}}|$, $\gamma = 3 \cdot \delta$, and $\lambda = 1000$ and the first-improving heuristic during exploration, as initial experiments showed these settings were robust. For all backends that use randomisation, we report the best-found and median objective values of 10 independent runs, as well as, in prefixed superscript if non-zero, the number of runs where feasibility was not established. For Gecode, we report the best-found objective of a single run. For each run we use a 15-minute timeout, under Linux Ubuntu 18.04 (64 bit) on an Intel Xeon E5520 of 2.27 GHz, with 4 processors of 4 cores each, with 24 GB RAM.

Since we must handcraft declarative neighbourhoods for each model, which requires a good understanding of both the model and the underlying problem, we evaluated LS(cp) on the models, instances, and declarative neighbourhoods of [3], namely steel-mill slab design [19], generalised balanced academic curriculum design (GBAC) [6], car sequencing [8], and community detection [10]. For space reasons, we omit the last two, which give results similar to GBAC. We also evaluated on the TSPTW model in Listing 1, and added declarative

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

neighbourhoods to models used in the MiniZinc Challenge [22] for a capacitated vehicle routing problem (CVRP), a time-dependent travelling salesperson problem (TDTSP), and the seat moving problem. Table 1 has the results, where boldface indicates the best objective value by all backends for the instance of the corresponding row and a "–" indicates that no feasible valuation was found in any run by the backend in the corresponding column.

CVRP. We used the model and instances of the MiniZinc Challenge 2015, except the easiest instance, 'simple2'. We extended the model with a declarative neighbourhood similar to the one in Listing 1, as CVRP is here also modelled using a `circuit` constraint. Gecode-lns was never worse and usually much better than both versions of LS(cp), but they significantly outperformed fzn-oscar-cbls, Yuck, and Gecode on all instances. Yuck and fzn-oscar-cbls here make moves on auxiliary variables (recall the end of Section 2.2): this explains why LS(cp) was better. LS(cp)-hard gave better valuations than LS(cp)-soft, which suggests that softening constraints is not important for this model. Hence it is unsurprising that LNS was best.

GBAC. Yuck and fzn-oscar-cbls overall outperformed the other backends, and fzn-oscar-cbls was best, possibly due to the declarative neighbourhoods. For the UD3, UD7, and UD9 instances, LS(cp) did not find any feasible valuation in any run, while Gecode-lns and Gecode found very bad valuations. This indicates that finding a feasible valuation for these instances is difficult using CP-style search, which LS(cp) and Gecode-lns use for initialisation. Clearly, this is an example where softening many constraints is important to find reasonable valuations.

Seat Moving. We used the model and instances of the MiniZinc Challenge 2018, except the hardest instance, 15-12-00, for which no solutions were found by any backend. We extended the model with a declarative neighbourhood defining moves that either swap two variables in a row of a 2D array or re-assigns one. LS(cp) and Gecode-lns performed best, with LS(cp)-soft being the sole best on one instance. Yuck and fzn-oscar-cbls failed to find a feasible valuation for the instances 10-12-00 and 20-20-00, and found poor valuations for the other ones. It would therefore appear that CP-style search is here more suitable for finding initial valuations, and that keeping more (but not all) constraints hard improves local search.

Steel Mill. We used the `hard_steelmill` neighbourhood of [3]. Arguably, fzn-oscar-cbls was best, followed by LS(cp)-hard, LS(cp)-soft, and then Gecode-lns, but each of these approaches wins on some instances. There is no clear pattern here, illustrating the importance of trying multiple technologies on the same model for each instance.

TDTSP. We used the model and instances of the MiniZinc Challenge 2017. We extended the model with a declarative neighbourhood defining moves preserves the satisfaction of an `inverse` constraint. Gecode-lns significantly outperformed all the other backends, and both versions of LS(cp) outperformed fzn-oscar-cbls, Yuck, and Gecode. As with the CVRP model, fzn-oscar-cbls and Yuck here make moves on auxiliary variables, which explains why LS(cp) was better.

Table 1. Experimental results on various minimisation problems.

	declarative neighbourhood						black-box		CP search annotation		
	LS(cp)-soft		LS(cp)-hard		fzn-oscar-cbls		Yuck		Gecode-lns		Gecode
CVRP	best	med.	best	med.	best	med.	best	med.	best	med.	best
A-n37-k5	773	920	722	773	2530	2736	–	–	**693**	**693**	1673
A-n64-k9	3187	3273	3113	3202	–	–	–	–	**1617**	**1617**	3544
B-n45-k5	1833	2019	1633	1848	3633	4004	–	–	**769**	**769**	2408
P-n16-k8	**450**	455	**450**	**450**	–	–	559	559	**450**	**450**	530
GBAC											
UD1	8577	9776	7635	9722	**438**	624	944	944	31263	31264	45420
UD2	**174**	213	189	217	189	206	289	289	354	376	12305
UD3	–	–	–	–	**191**	267	413	413	37576	37654	57681
UD4	470	1190	974	1190	**401**	472	485	485	904	904	11925
UD5	386	427	368	489	**272**	327	626	626	2039	2244	23028
UD6	124	135	125	144	122	153	154	154	**55**	**55**	9846
UD7	–	–	–	–	**519**	639	745	761	27330	27330	44044
UD8	65	87	74	107	63	86	105	105	**48**	**48**	9472
UD9	–	–	–	–	**463**	572	692	692	29213	29213	44010
UD10	126	176	107	162	81	91	138	138	**53**	**53**	12101
Seat Moving											
10-12-00	**463**	465	464	467	–	–	–	–	735	735	555
10-20-05	**90**	**90**	**90**	**90**	130	132	132	132	**90**	**90**	139
15-20-00	**199**	**199**	**199**	**199**	209	[7]210	–	–	**199**	**199**	207
20-20-00	**262**	**262**	**262**	**262**	–	–	–	–	**262**	**262**	286
Steel Mill											
bench_3_0	11	13	11	14	12	15	629	629	**8**	**8**	64
bench_3_1	31	46	29	42	**22**	31	–	–	77	77	167
bench_3_2	31	43	**28**	43	42	60	–	–	33	33	83
bench_3_3	36	48	**38**	54	**38**	76	896	896	70	70	326
bench_3_4	20	41	23	31	71	111	–	–	**14**	**14**	38
bench_3_5	**40**	50	46	54	54	56	925	925	98	98	270
bench_3_6	**27**	48	33	58	54	106	–	–	55	55	292
bench_3_7	81	103	82	103	**79**	101	1039	1039	109	109	203
bench_3_8	128	173	132	161	**116**	205	1400	1400	183	183	341
bench_3_9	212	230	**183**	260	205	252	1678	1678	240	240	331
TDTSP											
10_35_20	9114	9764	**9055**	9279	14424	17217	10847	10847	**9055**	**9055**	**9055**
10_42_00	**8421**	**8421**	**8421**	**8421**	15866	18751	9248	9248	**8421**	**8421**	**8421**
10_58_20	11043	11435	10800	10986	15581	19021	12319	12323	**10306**	**10306**	13799
20_26_00	16124	17677	15105	17427	22752	[5]22961	17906	19956	**12741**	**12741**	18197
20_36_10	15272	16045	15028	15772	22020	[1]22602	17727	17727	**12308**	**12308**	15051
TSPTW											
n40w120	**434**	**434**	**434**	**434**	–	–	–	–	**434**	**434**	490
n40w140	**328**	**328**	**328**	**328**	–	–	–	–	**328**	**328**	380
n40w160	**348**	**348**	**348**	**348**	–	–	–	–	**348**	**348**	425
n60w120	**384**	**384**	**384**	**384**	–	–	–	–	387	387	513
n100w80	720	[8]856	**679**	[8]694	–	–	–	–	–	–	772

TSPTW. We used the model in Listing 1 and medium-sized GendreauDumas-Extended *.001 instances.[4] LS(cp) and Gecode-lns outperformed all the other backends on all instances. In fact, the best-found objective values on all but the n100w80 instance are optimal. For the latter, Gecode-lns did not find any feasible valuation, but both versions of LS(cp) found feasible valuations in $10 - 8 = 2$ runs. Like with the CVRP and TDTSP models, both fzn-oscar-cbls and Yuck here make moves on auxiliary variables, namely the `ArrTime[i]` ones.

6 Conclusion, Related Work, and Future Work

Conclusion. We have demonstrated that declarative neighbourhoods, which were originally conceived for CBLS backends to MiniZinc, can also be used in order to generate automatically an LS(cp) method for local search, where the neighbourhood is initialised and explored using CP models and a CP solver. In fact, since we propose a decomposition for our new Writes constraint, nothing in our recipe for lifting any CP solver into a local-search backend to MiniZinc is specific to CP: our recipe equally applies to any IP, SAT, or SMT solver.

While we see a wide variety of behaviour across the benchmarks against CP, CBLS, and LNS backends to MiniZinc, our prototype LS(cp) backend finds the best solutions for the seat moving and TSPTW benchmarks and is competitive on all others, except GBAC, where it sometimes even struggles to find initial valuations. Hence there seems to be a sweet spot for the CP-based exploration of local-search neighbourhoods, where auxiliary variables make CBLS backends slow and non-LNS neighbourhoods are important for local search.

Related Work. We have already discussed the differences between LS(cp) and LNS (large-neighbourhood search, [20]) near the end of Section 1

Structured neighbourhood search (SNS, [1]) is a local-search framework for models written in Essence [9]. In SNS, a set of neighbourhoods is automatically inferred from the variables of a model and their types. This is done via a set of predefined rules for each basic variable type and predefined rules for variables of nested types, such as lists of sets of integers. Although the connection is not explicitly made in [1], the SNS framework is defined using the ideas in [18]. Specifically, in SNS, the variables of the given model of a problem are referred to as *active variables*. For each active variable, a corresponding *primary variable* is introduced to represent the next valuation. Each neighbourhood is then expressed as a neighbourhood model connecting the active and primary variables, which is essentially the same approach as in [18] and in this paper. Because multiple neighbourhoods are inferred, a multi-armed-bandit algorithm is used for selecting which neighbourhood to use.

The StoreElement(p, v, I, O) constraint used in [13] for constraint-based testing, with a propagator in [5], is the particular case Writes$(O, I, [p], [v])$, which is equivalent to $O = \text{write}(I, p, v)$ in the theory of arrays [15]. The Writes(O, I, P, V) constraint encodes a *parallel* series of writes on an array

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

I giving an array $O = \text{write}(\cdots \text{write}(\text{write}(I, p_1, v_1), p_2, v_2) \cdots p_m, v_m)$ with $\forall i, j : p_i = p_j \Rightarrow v_i = v_j$. However, encoding WRITES as a sequence of STOREELEMENT generates m copies of the input array in the encoding, but we only need the initial input array and the final output array.

Future Work. More advanced versions of LNS, such as propagation-guided LNS [**?**], cost-impact-guided LNS [**?**], and explanation-guided LNS [**?**], should be compared with LS(cp) to better understand the problems suitable for LS(cp) and LNS respectively.

Our LS(cp) is limited by initialisation being able to find an initial valuation, just like LNS. While the softening of some constraints makes initialisation more likely to succeed, the finding of an initial valuation is not guaranteed. Furthermore, the initialisation, exploration, and intensification steps all rely on appropriate CP search strategies being used, and our initial experiments indicate that these branching strategies have a significant impact on the performance. Clearly there is scope here for further investigation.

We also need to determine how best to soften constraints and which ones to soften, trading search over larger neighbourhoods for better robustness. Ideally, some form of dynamic adjustment, automatically softening when it is difficult to find improving moves, seems attractive to pursue.

Acknowledgements. We would like to thank the anonymous reviewers for their constructive feedback that helped improve this paper. This work is supported by the Swedish Research Council (VR) through Project Grant 2015-04910.

References

1. Akgün, O., Attieh, S., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P., Salamon, A.Z., Spracklen, P., Wetter, J.: A framework for constraint based local search using Essence. In: Lang, J. (ed.) IJCAI 2018. pp. 1242–1248. IJCAI Organization (2018)
2. Björdal, G., Flener, P., Pearson, J.: Generating compound moves in local search by hybridisation with complete search. In: Rousseau, L.M., Stergiou, K. (eds.) CP-AI-OR 2019. LNCS, vol. 11494. Springer (forthcoming)
3. Björdal, G., Flener, P., Pearson, J., Stuckey, P.J., Tack, G.: Declarative local-search neighbourhoods in MiniZinc. In: Alamaniotis, M., Lagniez, J.M., Lallouet, A. (eds.) ICTAI 2018. pp. 98–105. IEEE Computer Society (2018)
4. Björdal, G., Monette, J.N., Flener, P., Pearson, J.: A constraint-based local search backend for MiniZinc. Constraints **20**(3), 325–345 (July 2015), the fzn-oscar-cbls backend is available at `http://optimisation.research.it.uu.se/software`
5. Charreteur, F., Botella, B., Gotlieb, A.: Modelling dynamic memory management in constraint-based testing. Journal of Systems and Software **82**(11), 1755–1766 (November 2009)
6. Chiarandini, M., Di Gaspero, L., Gualandi, S., Schaerf, A.: The balanced academic curriculum problem revisited. Journal of Heuristics **18**(1), 119–148 (February 2012), also see `http://satt.diegm.uniud.it/projects/gbac`
7. 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 https://www.orbel.be/orbel27/pdf/abstract293.pdf; the OscaR.cbls solver is available at https://bitbucket.org/oscarlib/oscar/branch/CBLS
8. Dincbas, M., Simonis, H., Van Hentenryck, P.: Solving the car-sequencing problem in constraint logic programming. In: Kodratoff, Y. (ed.) ECAI 1988. pp. 290–295. Pitman (1988)
9. Frisch, A.M., Grum, M., Jefferson, C., Martinez Hernandez, B., Miguel, I.: The design of Essence: A constraint language for specifying combinatorial problems. In: IJCAI 2007. pp. 80–87. Morgan Kaufmann (2007)
10. Ganji, M., Bailey, J., Stuckey, P.J.: A declarative approach to constrained community detection. In: Beck, J.C. (ed.) CP 2017. LNCS, vol. 10416, pp. 477–494. Springer (2017)
11. Gecode Team: Gecode: A generic constraint development environment (2018), the Gecode solver and its MiniZinc backend are available at https://www.gecode.org
12. Glover, F., Laguna, M.: Tabu search. In: Modern Heuristic Techniques for Combinatorial Problems, pp. 70–150. John Wiley & Sons (1993)
13. Gotlieb, A., Botella, B., Watel, M.: INKA: Ten years after the first ideas. In: Pollet, Y. (ed.) ICSSEA 2006 (2006), available at https://www.semanticscholar.org/author/Mathieu-Watel/96187134
14. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations & Applications. Elsevier / Morgan Kaufmann (2004)
15. McCarthy, J.: Towards a mathematical science of computation. In: Proceedings of IFIP Congress (1962)
16. 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
17. OscaR Team: OscaR: Scala in OR (2012), available at https://oscarlib.bitbucket.io
18. Pesant, G., Gendreau, M.: A constraint programming framework for local search methods. Journal of Heuristics **5**(3), 255–279 (October 1999), extends a preliminary version at CP 1996, LNCS, vol. 1118, pp. 353–366, Springer (1996)
19. Schaus, P., Van Hentenryck, P., Monette, J.N., Coffrin, C., Michel, L., Deville, Y.: Solving steel mill slab problems with constraint-based techniques: CP, LNS, and CBLS. Constraints **16**(2), 125–147 (April 2011), also see http://becool.info.ucl.ac.be/steelmillslab
20. 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)
21. Shaw, P., De Backer, B., Furnon, V.: Improved local search for CP toolkits. Annals of Operations Research **115**(1–4), 31–50 (September 2002)
22. 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
23. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. The MIT Press (2005)