Solver-independent Large Neighbourhood Search

Jip J. Dekker[1,2][0000-0002-0053-6724], Maria Garcia de la Banda[1][], Andreas Schutt[2][], Peter J. Stuckey[1,2][0000-0003-2186-0459], and Guido Tack[1][]

[1] Monash University, Melbourne, Australia
`{jip.dekker,maria.garciadelabanda,peter.stuckey,guido.tack}@monash.edu`
[2] Data61, CSIRO, Melbourne, Australia
`andreas.schutt@data61.csiro.au`

Abstract. The combination of large neighbourhood search (LNS) methods with complete search methods has proved to be very effective. By restricting the search to (small) areas around an existing solution, the complete method is often able to quickly improve its solutions. However, developing such a combined method can be time-consuming: While the model of a problem can be expressed in a high-level solver-independent language, the LNS search strategies typically need to be implemented in the search language of the target constraint solvers. In this paper we show how we can simplify this process by (a) extending constraint modelling languages to support solver-independent LNS search definitions, and (b) defining small solver extensions that allow solvers to implement these solver-independent LNS searches. Modellers can then implement an LNS search to be executed in any extended solver, by simply using the modelling language constructs. Experiments show that the resulting LNS searches only introduce a small overhead compared to direct implementations in the search language of the underlying solvers.

1 Introduction

Large neighbourhood search (LNS [20]) is a meta-search that has proved to be very successful for scaling complete search methods, such as Constraint Programming (CP), to large optimisation problem sizes. LNS iteratively applies a particular search method to a neighbourhood surrounding a given solution. The neighbourhoods are chosen to be as large as possible, while being small enough for the search to quickly find a higher quality solution. The LNS meta-search then selects a new neighbourhood around this new solution and repeats the process.

The combination of LNS and CP has proven to be crucial for improving solving performance in a range of hard optimisation problems. While simple, unstructured neighbourhoods often work surprisingly well, the performance of many problems can be further improved if the neighbourhoods exploit the problem structure [4]. For example, in Vehicle Routing Problems, Shaw [20] proposes to remove *related* customer visits (such as those assigned to the same vehicle) from the tours of a given solution, and then re-insert them using a CP solver. Pacino and Van Hentenryck [12] solve Job Shop Scheduling problems by repeatedly re-scheduling all activities on a single machine, or in a particular time window. See [15] for a good overview of LNS and its applications.

All these combinations of LNS and CP have been either implemented from scratch, or within a particular CP system, in order to be able to program the interaction between the meta-search, the CP search and the constraint model. This close coupling makes it difficult to experiment with different solvers as backends for LNS, and it either precludes the use of high-level solver-independent modelling languages, or it introduces a gap between the problem model (expressed in a high-level language) and the definition of the neighbourhoods (expressed at the solver level).

The aim of this paper is to **lift LNS from the solver level to the modelling level**. Our main contribution is to show that many problem-specific neighbourhoods can be (a) expressed in a very natural way in a high-level, solver-independent constraint modelling language; and (b) compiled into efficient solver-level specifications that only require a small extension of existing solvers. The new approach has been implemented for the MiniZinc [10] modelling language and solvers Gecode [7] and Chuffed [2]. Our experiments show that the approach is expressive, efficient and effective.

2 Background

Constraint Optimisation Problems A *Constraint Optimisation Problem* (COP) is defined as a tuple $P = (C, X, D, f)$, with X a set of variables, C a set of constraints over subsets of X, D a domain such that for each $x \in X$, $D(x)$ is the set of values x may assume, and f an objective function. A *solution* of P is an *assignment a* such that $a(x) \in D(x)$ for all $x \in X$, and a satisfies all $c \in C$. An *optimal solution* is a solution a such that $f(a)$ is minimal. Usually, we are not just interested in solving one particular COP, but rather a whole *parameterised family* of COPs. We usually call these *models*, and an individual COP with fixed parameters, an *instance* of a model.

A CP solver starts from an instance (C, X, D, f) and performs *constraint propagation* of all constraints, pruning inconsistent values from D until it reaches a fixpoint D'. If at this point propagation has emptied the domain D' for any variable, the problem is *failed*. It is a *solution* if there is exactly one value left in $D(x)$ for each $x \in X$. Otherwise, the solver splits the instance into smaller sub-instances $(C \cup c_1, X, D', f) \ldots (C \cup c_k, X, D', f)$ by adding new constraints $c_1 \ldots c_k$, and recursively solves each of them. When the solver finds a solution, it evaluates its quality using f and adds a constraint to the remainder of the search to only allow solutions better than the current one.

Constraint Modelling Languages Most CP solvers take as input a flat list of variables and constraints. While the facilities of the host programming language (e.g., loops and overloading) can be used to make modelling more comfortable, dedicated Constraint Modelling Languages have become popular (e.g., OPL [24], AMPL [5], Essence [6], and MiniZinc [10]), as they support problem specification at a higher level of abstraction, and in a solver-independent way. We use MiniZinc due to its widespread use and support for over 20 different solvers. MiniZinc supports high-level features, such as different variable types, complex Boolean and arithmetic expressions, user-defined functions and predicates, and a comprehensive library of predefined global constraints. A (solver-independent) MiniZinc model is translated into (solver-specific) FlatZinc by the MiniZinc compiler, which is then interpreted by the target solver. The compiler uses a library of predicate and function definitions, written in MiniZinc specifically for the target solver, to generate FlatZinc code that only contains constraints and variable types supported by the target solver. This cleanly de-couples the MiniZinc compiler from the solver.

Algorithm 1 Large Neighbourhood Search

```
1: procedure LNS($P = (C, X, D, f)$)
2:     $a \leftarrow \mathit{findsolution}(P)$
3:     while $\neg \mathit{timeout}()$ do
4:         $P' = (C \cup \mathit{nbh}(a) \cup \mathit{obj}(a), X, D, f)$
5:         $a' \leftarrow \mathit{findsolution}(P')$
6:         if $a'$ is a solution then $a \leftarrow a'$
7:     return $a$
```

Large Neighbourhood Search LNS is usually described as a meta-search that starts from a solution to a COP, *relaxes* part of the solution, and then *re-optimises* that part. This process is iterated using a meta-heuristic (e.g., hill-climbing or simulated annealing) to improve the solution, until some stopping criterion is met, such as a time limit or a solution quality. The relaxation step is also described as *freeing up*, *destroying* or *thawing* certain variables in the solution to their original domain. Formally, consider a problem $P = (C, X, D, f)$, a solution a of P, and a domain D' built by selecting a subset of the variables $Y \subset X$ such that $D'(x)$ is set to $a(x)$ for $x \in Y$, and to $D(x)$ otherwise. A *neighbourhood of a* is defined as (C, X, D', f). Depending on the size of Y, this problem is significantly smaller than P, and can thus be solved efficiently using e.g. a CP solver. A more flexible and expressive definition of neighbourhood is $(C \cup \mathit{nbh}(a), X, D, f)$, where given solution a, $\mathit{nbh}(a)$ returns a set of constraints to be added to P for the next iteration of the LNS. This definition subsumes the one above, and allows neighbourhoods that instead of fixing variables, constrain them to take values close to the last solution.

Algorithm 1 shows a simple version of LNS, where $\mathit{findsolution}(P)$ invokes a complete solver to compute a new solution of problem P, and function $\mathit{obj}(a)$ adds constraints to ensure that only solutions better than a (according to f) are returned. Thus, the algorithm implements a simple hill-climbing that is terminated after a timeout. Real implementations would also terminate the complete solver in line 5 after a timeout (in order to balance intensification and exploration). Note that if P' does not lead to an improved solution, the algorithm computes a new neighbourhood $\mathit{nbh}(a)$ for the previous solution. Therefore, function *nbh* is typically non-deterministic (via random number generators) and impure (in programming language terms). More sophisticated versions of this algorithm implement other meta-heuristics, such as simulated annealing, or automatically switch between neighbourhood definitions (as in Adaptive LNS [19]).

3 Modelling of Neighbourhoods and Meta-heuristics

This section introduces a MiniZinc extension that enables modellers to use the more flexible version of $\mathit{nbh}(a)$ described above. As our extension is based on the modelling constructs introduced in MiniSearch [18], we briefly summarise those constructs.

3.1 LNS in MiniSearch

MiniSearch introduced a MiniZinc extension that enables modellers to express meta-searches inside a MiniZinc model. A meta-search in MiniSearch typically solves a given

```
predicate uniformNeighbourhood(array[int] of var int: x, float: destrRate) =
  forall(i in index_set(x))
  (if uniform(0.0,1.0) > destrRate then x[i] = sol(x[i]) else true endif);
```

Listing 1: A simple random LNS predicate implemented in MiniSearch

```
function ann: lns(var int: obj, array[int] of var int: vars,
                  int: iterations, float: destrRate, int: exploreTime) =
  repeat (i in 1..iterations) ( scope(
      if has_sol() then post(uniformNeighbourhood(vars,destrRate))
      else true endif /\
      time_limit(exploreTime, minimize_bab(obj)) /\
      commit() /\ print()
    ) /\ post(obj < sol(obj)) );
```

Listing 2: A simple LNS metaheuristic implemented in MiniSearch

MiniZinc model, performs some calculations on the solution, adds new constraints and then solves again. An LNS definition in MiniSearch consists of two parts. The first part is a declarative definition of a neighbourhood as a MiniZinc predicate that posts the constraints that should be added with respect to a previous solution. This makes use of the MiniSearch function: `function int: sol(var int: x)`, which returns the value that variable `x` was assigned to in the previous solution (similar functions are defined for Boolean, float and set variables). In addition, a neighbourhood predicate will typically make use of the random number generators available in the MiniZinc standard library. Listing 1 shows a simple random neighbourhood. For each decision variable `x[i]`, it draws a random number from a uniform distribution and, if it exceeds threshold `destrRate`, posts constraints forcing `x[i]` to take the same value as in the previous solution. For example, `uniformNeighbourhood(x,0.2)` would result in each variable in the array `x` having a 20% chance of being unconstrained, and an 80% chance of being assigned to the value it had in the previous solution.

The second part of a MiniSearch LNS is the meta-search itself. The most basic example is that of function `lns` in Listing 2. It performs a fixed number of iterations, each invoking the neighbourhood predicate `uniformNeighbourhood` in a fresh scope (so that the constraints only affect the current loop iteration). It then searches for a solution (`minimize_bab`) with a given timeout, and if the search does return a new solution, it commits to that solution (so that it becomes available to the `sol` function in subsequent iterations). The `lns` function also posts the constraint `obj < sol(obj)`, ensuring the objective value in the next iteration is strictly better than that of the current solution.

Limitations of the MiniSearch approach. Although MiniSearch enables the modeller to express *neighbourhoods* in a declarative way, the definition of the *meta-search* is rather unintuitive and difficult to debug, leading to unwieldy code for defining simple restarting strategies. Furthermore, the MiniSearch implementation requires either a close integration of the backend solver into the MiniSearch system, or it drives the solver through the regular text-file based FlatZinc interface, leading to a significant communication overhead. To address these two issues for LNS, we propose to keep modelling neighbourhoods as predicates, but define a small number of additional MiniZinc built-in annotations and functions that (a) allow us to express important aspects of the meta-search in a more convenient way, and (b) enable a simple compilation scheme that requires no additional communication with and only small, simple extensions of the backend solver.

3.2 Restart annotations

Instead of the complex MiniSearch definitions, we propose to add support for simple meta-searches that are purely based on the notion of *restarts*. A restart happens when a solver abandons its current search efforts, returns to the root node of the search tree, and begins a new exploration. Many CP solvers already provide support for controlling their restarting behaviour, e.g. they can periodically restart after a certain number of nodes, or restart for every solution. Typically, solvers also support posting additional constraints upon restarting (e.g Comet [9]) that are only valid for the particular restart (i.e., they are "retracted" for the next restart). In its simplest form, we can therefore implement LNS by specifying a neighbourhood predicate, and annotating the `solve` item to indicate the predicate should be invoked upon each restart:

```
solve ::on_restart(myNeighbourhood) minimize cost;
```

Note that MiniZinc currently does not support passing functions or predicates as arguments. Calling the predicate, as in `::on_restart(myNeighbourhood())`, would not have the correct semantics, since the predicate needs to be called for *each* restart. As a workaround, we currently pass the name of the predicate to be called for each restart as a string (see the definition of the new `on_restart` annotation in Listing 3).

The second component of our LNS definition is the *restarting strategy*, defining how much effort the solver should put on each neighbourhood (i.e., restart), and when to stop the overall search. We propose adding new search annotations to MiniZinc to control this behaviour (see Listing 3). The `restart_on_solution` annotation tells the solver to restart immediately for each solution, rather than looking for the best one in each restart, while `restart_without_objective` tells it not to add branch-and-bound constraints on the objective. The other `restart_X` annotations define different strategies for restarting the search when no solution is found. The `timeout` annotation gives an overall time limit for the search, whereas `restart_limit` stops the search after a fixed number of restarts.

3.3 Neighbourhood selection

It is often beneficial to use several neighbourhood definitions for a problem. Different neighbourhoods may be able to improve different aspects of a solution, at different phases of the search. Adaptive LNS [19, 14], which keeps track of the neighbourhoods that led to improvements and favours them for future iterations, is the prime example for this approach. A simpler scheme may apply several neighbourhoods in a round-robin fashion.

In MiniSearch, adaptive or round-robin approaches can be implemented using *state variables*, which support destructive update (overwriting the value they store). In this way, the MiniSearch strategy can store values to be used in later iterations. We use the *solver state* instead, i.e., normal decision variables, and define two simple built-in functions to access the solver state *of the previous restart*. This approach is sufficient for expressing neighbourhood selection strategies, and its implementation is much simpler.

```
% post predicate "pred" whenever the solver restarts
annotation on_restart(string: pred);
% restart after fixed number of nodes
annotation restart_constant(int: nodes);
% restart with scaled Luby sequence
annotation restart_luby(int: scale);
% restart with scaled geometric sequence (scale*base^n in the n-th iteration)
annotation restart_geometric(float: base, int: scale);
% restart with linear sequence (scale*n in the n-th iteration)
annotation restart_linear(int: scale);
% restart on each solution
annotation restart_on_solution;
% restart without branch-and-bound constraints on the objective
annotation restart_without_objective;
% overall time limit for search
annotation timeout(int: seconds);
% overall limit on number of restarts
annotation restart_limit(int: n_restarts);
```

Listing 3: New annotations to control the restarting behaviour

```
% Report the status of the solver (before restarting).
enum STATUS = {START, UNKNOWN, UNSAT, SAT, OPT}
function STATUS: status();
% Provide access to the last assigned value of variable x.
function int: lastval(var int: x);
```

Listing 4: Functions for accessing previous solver states

State access and initialisation The state access functions are defined in Listing 4. Function `status` returns the status of the previous restart, namely: `START` (there has been no restart yet); `UNSAT` (the restart failed); `SAT` (the restart found a solution); `OPT` (the restart found and proved an optimal solution); and `UNKNOWN` (the restart did not fail or find a solution). Function `lastval` (which, like `sol`, has versions for all basic variable types) allows modellers to access the last value assigned to a variable (the value is undefined if `status()=START`). In order to be able to initialise the variables used for state access, we reinterpret `on_restart` so that the predicate is also called for the initial search (i.e., before the first "real" restart) with the same semantics, that is, any constraint posted by the predicate will be retracted for the next restart.

Parametric neighbourhood selection predicates We define standard neighbourhood selection strategies as predicates that are parametric over the neighbourhoods they should apply. For example, since `on_restart` now also includes the initial search, we can define a strategy `basic_lns` that applies a neighbourhood only if the current status is not `START`:

```
predicate basic_lns(var bool: nbh) = (status()!=START -> nbh);
```

In order to use this predicate with the `on_restart` annotation, we cannot simply pass `basic_lns(uniformNeighbourhood(x,0.2))`. First of all, calling `uniformNeighbourhood`

```
array[1..n] of var 1..n: x;  % decision variables
var int: cost;               % objective function
% ... some constraints defining the problem
% The user-defined LNS strategy
predicate my_lns() = basic_lns(uniformNeighbourhood(x,0.2));
% Solve using my_lns, restart every 500 nodes, overall timeout 120 seconds
solve ::on_restart("my_lns") ::restart_constant(500) ::timeout(120)
      minimize cost;
```

Listing 5: Complete LNS example

```
1 predicate round_robin(array[int] of var bool: nbhs) =
2       let { int: N = length(nbhs);
3             var -1..N-1: select; % Neighbourhood selection
4       } in  if status()=START then select= -1
5             else select= (lastval(select) + 1) mod N
6             endif /\
7             forall(i in 1..N) (select=i-1 -> nbhs[i]);
```

Listing 6: A predicate providing the round robin meta-heuristic

like that would result in a *single* evaluation of the predicate, since MiniZinc employs a call-by-value evaluation strategy. Furthermore, the `on_restart` annotation only accepts the name of a nullary predicate. Therefore, users have to define their overall strategy in a new predicate. Listing 5 shows a complete example of a basic LNS model.

We can also define round-robin and adaptive strategies using these primitives. Listing 6 defines a round-robin LNS meta-heuristic, which cycles through a list of `N` neighbourhoods `nbhs`. To do this, it uses the decision variable `select`. In the initialisation phase (`status()=START`), `select` is set to `-1`, which means none of the neighbourhoods is activated. In any following restart, `select` is incremented modulo `N`, by accessing the last value assigned in a previous restart (`lastval(select)`). This will activate a different neighbourhood for each restart (line 7). For adaptive LNS, a simple strategy is to change the size of the neighbourhood depending on whether the previous size was successful or not. Listing 7 shows an adaptive version of the `uniformNeighbourhood` that increases the number of free variables when the previous restart failed, and decreases it when it succeeded, within the bounds $[0.6, 0.95]$.

3.4 Meta-heuristics

The LNS strategies we have seen so far rely on the default behaviour of MiniZinc solvers to use branch-and-bound for optimisation: when a new solution is found, the solver adds a constraint to the remainder of the search to only accept better solutions, as defined by the objective function in the `minimize` or `maximize` clause of the `solve` item. When combined with restarts and LNS, this is equivalent to a simple hill-climbing meta-heuristic.

We can use the constructs introduced above to implement alternative meta-heuristics such as simulated annealing. In particular, we use `restart_without_objective` to tell the solver not to add the branch-and-bound constraint on restart. It will still use the declared

```
predicate adaptiveUniform(array[int] of var int: x, float: initialDestrRate) =
  let { var float: rate; } in
  if     status() = START then rate = initialDestrRate
  elseif status() = UNSAT then rate = min(lastval(rate)-0.02,0.6)
  else                         rate = max(lastval(rate)+0.02,0.95)
  endif /\
  forall(i in index_set(x))
        (if uniform(0.0,1.0) > rate then x[i] = sol(x[i]) else true endif);
```

Listing 7: A simple adaptive neighbourhood

objective to decide whether a new solution is the globally best one seen so far, and only output those (to maintain the convention of MiniZinc solvers that the last solution printed at any point in time is the currently best known one). With `restart_without_objective`, the restart predicate is now responsible for constraining the objective function. Note that a simple hill-climbing (for minimization) can still be defined easily in this context as:

```
predicate hill_climbing() =
  if status()=START then true
  else _objective < sol(_objective) endif;
```

It takes advantage of the fact that the declared objective function is available through the built-in variable `_objective`. A simulated annealing strategy is also easy to express:

```
predicate simulated_annealing(float: initTemp, float: coolingRate) =
  let { var float: temp; } in
  if status()=START then temp = initTemp
  else
    temp = lastval(temp)*(1-coolingRate) /\ % cool down
    _objective < sol(_objective) - ceil(log(uniform(0.0,1.0)) * temp)
  endif;
```

4 Compilation of Neighbourhoods

The neighbourhoods defined in the previous section can be executed with MiniSearch by adding support for the `status` and `lastval` built-in functions, and by defining the main restart loop. The MiniSearch evaluator will then call a solver to produce a solution, and evaluate the neighbourhood predicate, incrementally producing new FlatZinc to be added to the next round of solving. While this is a viable approach, our goal is to keep the compiler and solver separate, by embedding the entire LNS specification into the FlatZinc that is passed to the solver. This section introduces such a compilation approach. It only requires simple modifications of the MiniZinc compiler, and the compiled FlatZinc can be executed by standard CP solvers with a small set of simple extensions.

4.1 Compilation overview

The neighbourhood definitions from the previous section have an important property that makes them easy to compile to standard FlatZinc: they are defined in terms of standard

```
1 predicate status(var int: stat);
2 function var STATUS: status() =
3     let { var STATUS: stat;
4           constraint status(stat);
5     } in stat;
```

Listing 8: MiniZinc definition of the `status` function

MiniZinc expressions, with the exception of a few new built-in functions. When the neighbourhood predicates are evaluated in the MiniSearch way, the MiniSearch runtime implements those built-in functions, computing the correct value whenever a predicate is evaluated. Instead, the compilation scheme presented below uses a limited form of *partial evaluation*: parameters known at compile time will be fully evaluated; those only known during the solving, such as the result of a call to any of the new functions (`sol`, `status` etc.), are replaced by decision variables. This essentially **turns the new built-in functions into constraints** that have to be supported by the target solver. The neighbourhood predicate can then be added as a constraint to the model. The evaluation is performed by hijacking the solver's own capabilities: It will automatically perform the evaluation of the new functions by propagating the new constraints.

To compile an LNS specification to standard FlatZinc, the MiniZinc compiler performs four simple steps:

1. Replace the annotation `::on_restart("X")` with a call to predicate `X`.
2. Inside predicate `X` and any other predicate called recursively from `X`: treat any call to built-in functions `sol`, `status`, and `lastvar` as returning a `var` instead of a `par` value; and rename calls to random functions, e.g., `uniform` to `uniform_nbh`, in order to distinguish them from their standard library versions.
3. Convert any expression containing a call from step 2 to `var` to ensure the functions are compiled as constraints, rather than statically evaluated by the MiniZinc compiler.
4. Compile the resulting model using an extension of the MiniZinc standard library that provides declarations for these built-in functions, as defined below.

These transformations will not change the code of many neighbourhood definitions, since the built-in functions are often used in positions that accept both parameters and variables. For example, the `uniformNeighbourhood` predicate from Listing 1 uses `uniform(0.0,1.0)` in an `if` expression, and `sol(x[i])` in an equality constraint. Both expressions can be translated to FlatZinc when the functions return a `var`.

4.2 Compiling the new built-ins

We can compile models that contain the new built-ins by extending the MiniZinc standard library as follows.

`status` Listing 8 shows the definition of the `status` function. It simply replaces the functional form by a predicate `status` (declared in line 1), which constrains its local variable argument `stat` to take the status value.

```
predicate int_sol(var int: x, var int: xi);
function int: sol(var int: x) = if is_fixed(x) then fix(x)
    else let { var lb(x)..ub(x): xi;
               constraint int_sol(x,xi);
         } in xi;
    endif;
```

Listing 9: MiniZinc definition of the `sol` function for integer variables

```
predicate float_uniform(var float:l, var float: u, var float: r);
function var float: uniform_nbh(var float: l, var float: u) :: impure =
  let { var lb(l)..ub(u): rnd;
        constraint float_uniform(l,u,rnd):
  } in rnd;
```

Listing 10: MiniZinc definition of the `uniform_nbh` function for floats

`sol` and `lastval` Since `sol` is overloaded for different variable types and FlatZinc does not support overloading, we produce type-specific built-ins for every type of solver variable (`int_sol(x, xi)`, `bool_sol(x, xi)`, etc.). The resolving of the `sol` function into these specific built-ins is done using an overloaded definition like the one shown in Listing 9 for integer variables. If the value of the variable in question becomes known at compile time, we use that value instead. Otherwise, we replace the function call with a type specific `int_sol` predicate, which is the constraint that will be executed by the solver. To improve the compilation of the model further, we use the declared bounds of the argument (`lb(x)..ub(x)`) to constrain the variable returned by `sol`. This bounds information is important for the compiler to be able to generate the most efficient FlatZinc code for expressions involving `sol`. The compilation of `lastval` is similar to that for `sol`.

Random number functions Calls to the random number functions have been renamed by appending `_nbh`, so that the compiler does not simply evaluate them statically. The definition of these new functions follows the same pattern as for `sol`, `status`, and `lastval`. The MiniZinc definition of the `uniform_nbh` function is shown in Listing 10.[3] Note that the function accepts variable arguments `l` and `u`, so that it can be used in combination with other functions, such as `sol`.

4.3 Solver support for LNS FlatZinc

We will now show the minimal extensions required from a solver to interpret the new FlatZinc constraints and, consequently, to execute LNS definitions expressed in MiniZinc. First, the solver needs to parse and support the restart annotations of Listing 3. Many solvers already support all this functionality. Second, the solver needs to be able to parse the new constraints `status`, and all versions of `sol`, `lastval`, and random number functions like `float_uniform`. In addition, for the new constraints the solver needs to:

[3] Random number functions need to be marked as `::impure` for the compiler not to apply Common Subexpression Elimination (CSE) [23] if they are called multiple times with the same arguments.

- `status(s)`: record the status of the previous restart, and fix `s` to the recorded status.
- `sol(x,sx)` (variants): constrain `sx` to be equal to the value of `x` in the incumbent solution. If there is no incumbent solution, it has no effect.
- `lastval(x,lx)` (variants): constrain `lx` to take the last value assigned to `x` during search. If no value was ever assigned, it has no effect. Note that many solvers (in particular SAT and LCG solvers) already track `lastval` for their variables for use in search. To support LNS a solver must at least track the *lastval* of each of the variables involved in such a constraint. This is straightforward by using the `lastval` propagator itself. It wakes up whenever the first argument is fixed, and updates the last value (a non-backtrackable value).
- random number functions: fix their variable argument to a random number in the appropriate probability distribution.

Importantly, these constraints need to be propagated in a way that their effects can be undone for the next restart. Typically, this means the solver must not propagate these constraints in the root node of the search.

Modifying a solver to support this functionality is straightforward if it already has a mechanism for posting constraints during restarts. We have implemented these extensions for both Gecode (110 new lines of code) and Chuffed (126 new lines of code).

Example 1. Consider the model from Listing 5 again. Listing 11 shows a part of the FlatZinc that arises from compiling `basic_lns(uniformNeighbourhood(x,0.2))`, assuming that `index_set(x) = 1..n`. Lines 1–4 define a Boolean variable `b1` that is true iff the status is not `START`. The second block of code (lines 6–15) represents the decomposition of the expression `(status()!=START /\ uniform(0.0,1.0)>0.2) -> x[1]=sol(x[1])`, which is the result of merging the implication from the `basic_lns` predicate with the `if` expression from `uniformNeighbourhood`. The code first introduces and constrains a variable for the random number, then adds two Boolean variables: `b2` is constrained to be true iff the random number is greater than 0.2; while `b3` is constrained to be the conjunction `status()!=START /\ uniform(0.0,1.0)>0.2`. Line 13 constrains `x1` to be the value of `x[1]` in the previous solution. Finally, the half-reified constraint in line 15 implements `b3 -> x[1]=sol(x[1])`. We have omitted the similar code generated for `x[2]` to `x[n]`. Note that the FlatZinc shown here has been simplified for presentation.

The first time the solver is invoked, it sets `s` to 1 (`START`). Propagation will fix `b1` to `false` and `b3` to `false`. Therefore, the implication in line 15 is not activated, leaving `x[1]` unconstrained. The neighbourhood constraints are effectively switched off.

When the solver restarts, all of the special propagators are re-executed. Now `s` is not 1, and `b1` will be set to `true`. The `float_random` propagator assigns `rnd1` a new random value and, depending on whether it is greater than `0.2`, the Boolean variables `b2`, and consequently `b3` will be assigned. If it is `true`, the constraint in line 15 will become active and assign `x[1]` to its value in the previous solution. □

5 Experiments

We will now show that a solver that evaluates the compiled FlatZinc LNS specifications can (a) be effective and (b) incur only a small overhead compared to a dedicated

```
var 1..5: s;
constraint status(s);
var bool b1;
constraint int_ne_reif(s,1,b1); % b1 <-> status()!=START

var 0.0..1.0: rnd1;
constraint float_uniform(0.0,1.0,rnd1);
var bool: b2;
constraint float_gt_reif(rnd1,0.2,b2);
var bool: b3;
constraint bool_and(b1,b2,b3);
var 1..3: x1;
constraint int_sol(x[1],x1);
% (status()!=START /\ uniform(0.0,1.0)>0.2) -> x[1]=sol(x[1])
constraint int_eq_imp(x[1],x1,b3);
...
```

Listing 11: FlatZinc that results from compiling `basic_lns(uniformNeighbourhood(x,0.2))`.

implementation of the neighbourhoods. We ran experiments for three models from the MiniZinc challenge [21, 22] (`gbac`, `steelmillslab`, and `rcpsp-wet`) using three versions of Gecode [7] and two versions of Chuffed [2]: gecode, the current Gecode release (6.0); gecode-fzn, Gecode performing LNS with FlatZinc neighbourhoods. gecode-replay, Gecode replaying the effects of gecode-fzn on the original model; chuffed, the development version of Chuffed; and chuffed-fzn, Chuffed performing LNS with FlatZinc neighbourhoods.[4] The overhead of gecode-fzn can be established by compariing it to gecode-replay, as the latter is an LNS implementation without any overhead. The effectiveness of the two solvers can be established by comparing gecode-fzn and chuffed-fzn to their respective base versions gecode and chuffed.

All experiments were run on a single core of an Intel Core i5 CPU @ 3.4 GHz with 4 cores and 16 GB RAM running MacOS High Sierra. LNS benchmarks are repeated with 10 different random seeds and the average is shown. For each solving method we measured the average integral of the model objective after finding the initial solution ($\int$), the average best objective found (min), and the standard deviation of the best objective found in percentage (%), which is shown as the superscript on min when running LNS. The underlying search strategy used is the fixed search strategy defined in the model. For each model we use a round robin evaluation (Listing 6) of two neighbourhoods: a neighbourhood that destroys 20% of the main decision variables (Listing 1) and a structured neighbourhood for the model (described below). The optimal objective value is shown for every instance. The restart strategy is `::restart_constant(250)` `::restart_on_solution`. The overall timeout for each run is 120 seconds.

gbac The Generalised Balanced Academic Curriculum problem comprises courses having a specified number of credits and lasting a certain number of periods, load limits

[4] Our implementations are avaible at `https://github.com/Dekker1/{libminizinc,gecode,chuffed}` on branches containing the keyword `on_restart`.

Table 1: `gbac` benchmarks

	optimal	gecode		gecode-fzn		gecode-replay		chuffed		chuffed-fzn	
Instance	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min
UD2-gbac	**146**	1502k	12515	93k	376^{16}	**92k**	$\mathbf{362}^{15}$	1494k	12344	**207k**	$\mathbf{598}^{54}$
UD4-gbac	**396**	1517k	12645	121k	$\mathbf{932}^{24}$	**120k**	$\mathbf{932}^{24}$	1151k	9267	**160k**	$\mathbf{1142}^{5}$
UD5-gbac	[5]**222**	2765k	23028	283k	$\mathbf{2007}^{39}$	**281k**	$\mathbf{2007}^{39}$	2569k	21233	**483k**	$\mathbf{2572}^{22}$
UD8-gbac	**40**	1195k	9611	21k	$\mathbf{53}^{26}$	**20k**	$\mathbf{53}^{26}$	1173k	9559	**114k**	$\mathbf{76}^{26}$
reduced_UD4	**949**	629k	4917	**114k**	$\mathbf{950}^{0}$	**114k**	$\mathbf{950}^{0}$	715k	5491	**117k**	$\mathbf{950}^{0}$

Table 2: `steelmillslab` benchmarks

	optimal	gecode		gecode-fzn		gecode-replay		chuffed		chuffed-fzn	
Instance	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min
bench_13_0	**0**	3247	27	20	$\mathbf{0}^{0}$	**19**	$\mathbf{0}^{0}$	1315	9	**50**	$\mathbf{0}^{0}$
bench_14_1	**0**	1248	**0**	32	$\mathbf{0}^{0}$	**31**	$\mathbf{0}^{0}$	**72**	**0**	79	$\mathbf{0}^{0}$
bench_15_11	**0**	4458	30	27	$\mathbf{0}^{0}$	**26**	$\mathbf{0}^{0}$	143	**0**	**65**	$\mathbf{0}^{0}$
bench_16_10	**0**	2446	**0**	**19**	$\mathbf{0}^{0}$	**19**	$\mathbf{0}^{0}$	122	**0**	**51**	$\mathbf{0}^{0}$
bench_19_5	**0**	3380	28	12	$\mathbf{0}^{0}$	**11**	$\mathbf{0}^{0}$	3040	19	**31**	$\mathbf{0}^{0}$

of courses for each period, prerequisites for courses, and preferences of teaching periods for professors. A detailed description of the problem is given in [1]. The main decisions are to assign courses to periods, which is done via the variables `period_of` in the model. Listing 12 shows the neighbourhood chosen, which randomly picks one period and frees all courses that are assigned to it.

The results for `gbac` in Table 1 show that the overhead introduced by gecode-fzn w.r.t. gecode-replay is quite low, and both their results are much better than the baseline gecode. Since learning is not very effective for `gbac`, the performance of chuffed is inferior to Gecode. However, LNS again significantly improves over standard Chuffed.

`steelmillslab` The Steel Mill Slab design problem consists of cutting slabs into smaller ones, so that all orders are fulfilled while minimising the wastage. The steel mill only produces slabs of certain sizes, and orders have both a size and a colour. We have to assign orders to slabs, with at most two different colours on each slab. The model uses the variables `assign` for deciding which order is assigned to which slab. Listing 13 shows a structured neighbourhood that randomly selects a slab and frees the orders assigned to it in the incumbent solution. These orders can then be freely reassigned to any other slab.

[5] Best objective found during the MiniZinc Challenge; not proved to be the optimal value.

```
let { int: period = uniform(periods) } in
forall(i in courses where sol(period_of[i]) != period)
     (period_of[i] = sol(period_of[i]));
```

Listing 12: `gbac`: neighbourhood freeing all courses in a period.

```
predicate free_slab() =
    let { int: slab = uniform(1, nbSlabs) } in
    forall(i in 1..nbSlabs where slab != sol(assign[i]))
        (assign[i] = sol(assign[i]));
```

Listing 13: `steelmillslab`: Neighbourhood that frees all orders assigned to a selected slab.

```
predicate free_timeslot()  =
    let { int: slot = max(Times) div 10;
          int: time = uniform(min(Times), max(Times) - slot); } in
    forall(t in Tasks)
          ((sol(s[t]) < time \/ time+slot > sol(s[t])) -> s[t] = sol(s[t]));
```

Listing 14: `rcpsp-wet`: Neighbourhood freeing all tasks starting in the drawn interval.

For this problem a solution with zero wastage is always optimal. The use of LNS makes these instances easy, as all the LNS approaches find optimal solutions. As Table 2 shows, gecode-fzn is again slightly slower than gecode-replay (the integral is slightly larger). While chuffed significantly outperforms gecode on this problem, once we use LNS, the learning in chuffed-fzn is not advantageous compared to gecode-fzn or gecode-replay. Still, chuffed-fzn outperforms chuffed by always finding an optimal solution.

`rcpsp-wet` The Resource-Constrained Project Scheduling problem with Weighted Earliness and Tardiness cost, is a classic scheduling problem in which tasks need to be scheduled subject to precedence constraints and cumulative resource restrictions. The objective is to find an optimal schedule that minimises the weighted cost of the earliness and tardiness for tasks that are not completed by their proposed deadline. The decision variables in array s represent the start times of each task in the model. Listing 14 shows our structured neighbourhood for this model. It randomly selects a time interval of one-tenth the length of the planning horizon and frees all tasks starting in that time interval, which allows a reshuffling of these tasks.

Table 3 shows that gecode-replay and gecode-fzn perform almost identically, and substantially better than baseline gecode for these instances. The baseline learning solver chuffed is best overall on the easy examples, but LNS makes it much more robust. The poor performance of chuffed-fzn on the last instance is due to the fixed search, which limits the usefulness of nogood learning.

Table 3: `rcpsp-wet` benchmarks

	optimal	gecode		gecode-fzn		gecode-replay		chuffed		chuffed-fzn	
Instance	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min	$\int$	min
j30_1_3-wet	**93**	20k	161	**11k**	**93**^{0}	**11k**	**93**^{0}	**3k**	**93**	13k	**93**^{0}
j30_43_10-wet	**121**	19k	158	15k	**121**^{0}	**14k**	**121**^{0}	**10k**	**121**	15k	**121**^{0}
j60_19_6-wet	**227**	54k	441	**29k**	**235**^{3}	**29k**	**235**^{3}	63k	487	**29k**	**227**^{0}
j60_28_3-wet	**266**	94k	770	**33k**	**273**^{0}	**33k**	**273**^{0}	79k	604	**35k**	**272**^{1}
j90_48_4-wet	**513**	199k	1653	72k	**535**^{2}	**71k**	**535**^{2}	201k	1638	**109k**	**587**^{2}

Summary The results show that LNS outperforms the baseline solvers, except for benchmarks where we can quickly find and prove optimality. However, the main result from these experiments is that the overhead introduced by our FlatZinc interface, when compared to an optimal LNS implementation, is relatively small. We have additionally calculated the rate of search nodes explored per second and, across all experiments, gecode-fzn achieves around 3% fewer nodes per second than gecode-replay. This overhead is caused by propagating the additional constraints in gecode-fzn. Overall, the experiments demonstrate that the compilation approach is an effective and efficient way of adding LNS to a modelling language with minimal changes to the solver.

6 Related Work and Conclusion

Large neighbourhood search is straightforward to implement using a scripting language and a separate modelling language. Scripting languages like MiniSearch [18], OPL Script [25] and AMPL Script can be used transparently with their underlying modelling language. However, this form of LNS requires either the solver to be restarted from scratch for every solve, or the scripting language to be tightly tied to a particular solver.

Many CP systems such as Choco [16], Comet [9], Objective CP [26], OSCAR [11], or or-tools [8] have an `onRestart` or `onSolution` event (or similar APIs) to which arbitrary code can be attached. This makes LNS easy to implement, although it typically mixes declarative and procedural aspects. It is also much more expressive than MiniSearch but relies on a tight relationship with the underlying solvers. Support for our compiled LNS specifications will be easy to implement for these solvers.

There are a number of approaches to automatically defining neighbourhoods, such as random [3], propagation guided [13], and explanation based [17] neighbourhoods. If the solver supports these then they can be used to build LNS solutions using this strategy straightforwardly. They are orthogonal to user defined neighbourhoods.

In this paper we have shown how we can take a high level model and LNS definition and communicate that to a CP solver, which then completes the LNS search. The additions to the CP solver are minor, by hijacking the use of propagators to do expression evaluation, and adding a few simple (pseudo-)propagators. The result is a solver independent approach to LNS that does not rely on repeated calls to the solver. While we have concentrated on LNS, it seems that more of MiniSearch [18] can be compiled into FlatZinc in the same way. This opens up interesting possibilities for further research.

References

1. Chiarandini, M., Gaspero, L.D., Gualandi, S., and Schaerf, A.: The balanced academic curriculum problem revisited. J. Heuristics 18(1), 119–148 (2012)
2. Chu, G.: Improving Combinatorial Optimization. Department of Computing and Information Systems, University of Melbourne (2011).
3. Cipriano, R., Di Gaspero, L., and Dovier, A.: “Gelato: A Multi-paradigm Tool for Large Neighborhood Search”. In: Hybrid Metaheuristics. Ed. by E.-G. Talbi. Springer Berlin Heidelberg, 2013, pp. 389–414.

4. Danna, E., and Perron, L.: Structured vs. Unstructured Large Neighborhood Search: A Case Study on Job-Shop Scheduling Problems with Earliness and Tardiness Costs. In: Rossi, F. (ed.) Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings. LNCS, vol. 2833, pp. 817–821. Springer, Heidelberg (2003)
5. Fourer, R., Gay, D., and Kernighan, B.: AMPL: A Mathematical Programming Language. Management Science 36, 519–554 (1990)
6. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., and Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints 13(3), 268–306 (2008)
7. Gecode Team: *Gecode: A Generic Constraint Development Environment*. 2016. `http://www.gecode.org`.
8. Google: *or-tools*. 2017. `https://developers.google.com/optimization/`.
9. Michel, L., and Hentenryck, P.V.: The Comet Programming Language and System. In: Beek, P. van (ed.) Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings. LNCS, vol. 3709, pp. 881–881. Springer, Heidelberg (2005)
10. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., and Tack, G.: MiniZinc: Towards a Standard CP Modelling Language. In: Bessière, C. (ed.) Principles and Practice of Constraint Programming – CP 2007, pp. 529–543. Springer Berlin Heidelberg (2007)
11. OscaR Team: *OscaR: Scala in OR*. Available from `https://bitbucket.org/oscarlib/oscar`. 2012.
12. Pacino, D., and Van Hentenryck, P.: Large Neighborhood Search and Adaptive Randomized Decompositions for Flexible Jobshop Scheduling. In: Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three. IJCAI'11, pp. 1997–2002. AAAI Press, Barcelona, Catalonia, Spain (2011)
13. Perron, L., Shaw, P., and Furnon, V.: Propagation Guided Large Neighborhood Search. In: Wallace, M. (ed.) Principles and Practice of Constraint Programming – CP 2004, pp. 468–481. Springer Berlin Heidelberg (2004)
14. Pisinger, D., and Ropke, S.: A General Heuristic for Vehicle Routing Problems. Comput. Oper. Res. 34(8), 2403–2435 (2007)
15. Pisinger, D., and Ropke, S.: "Large Neighborhood Search". In: Handbook of Metaheuristics. Ed. by M. Gendreau and J.-Y. Potvin. Boston, MA: Springer US, 2010, pp. 399–419. ISBN: 978-1-4419-1665-5. DOI: `10.1007/978-1-4419-1665-5_13`. `https://doi.org/10.1007/978-1-4419-1665-5_13`.
16. Prud'homme, C., Fages, J.-G., and Lorca, X.: *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. 2017. `http://www.choco-solver.org`.
17. Prud'homme, C., Lorca, X., and Jussien, N.: Explanation-based large neighborhood search. Constraints 19(4), 339–379 (2014)
18. Rendl, A., Guns, T., Stuckey, P.J., and Tack, G.: MiniSearch: A Solver-Independent Meta-Search Language for MiniZinc. In: Pesant, G. (ed.) Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings, pp. 376–392 (2015)
19. Ropke, S., and Pisinger, D.: An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation science 40(4), 455–472 (2006)
20. Shaw, P.: Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In: Maher, M., and Puget, J.-F. (eds.) Principles and Practice of Constraint Programming — CP98, pp. 417–431. Springer Berlin Heidelberg (1998)
21. Stuckey, P.J., Becket, R., and Fischer, J.: Philosophy of the MiniZinc challenge. Constraints 15(3), 307–316 (2010)

22. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., and Fischer, J.: The MiniZinc Challenge 2008-2013. AI Magazine 35(2), 55–60 (2014)
23. Stuckey, P.J., and Tack, G.: MiniZinc with Functions. In: Gomes, C., and Sellmann, M. (eds.) Proceedings of the 10th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming. LNCS,vol. 7874, pp. 268–283. Springer, Heidelberg (2013)
24. Van Hentenryck, P.: The OPL optimization programming language. MIT Press (1999)
25. Van Hentenryck, P., and Michel, L.: OPL Script: Composing and Controlling Models. In: New Trends in Constraints. Ed. by K. Apt, E. Monfroy, A. Kakas, and F. Rossi, pp. 75–90. Springer Berlin Heidelberg(2000)
26. Van Hentenryck, P., and Michel, L.: The Objective-CP Optimization System. In: Schulte, C. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 8124, pp. 8–29. Springer, Heidelberg (2013)