Dependency-Curated Large Neighbourhood Search

Frej Knutar Lewander[1]
Department of Information Technology, Uppsala University, Sweden

Pierre Flener
Department of Information Technology, Uppsala University, Sweden

Justin Pearson
Department of Information Technology, Uppsala University, Sweden

Abstract

In large neighbourhood search (LNS), an incumbent initial solution is incrementally improved by selecting a subset of the variables, called the *freeze set*, and fixing them to their values in the incumbent solution, while a value for each remaining variable is found and assigned via solving (such as constraint programming-style propagation and search). Much research has been performed on finding generic and problem-specific LNS selection heuristics that select freeze sets that lead to high-quality solutions. In constraint-based local search (CBLS), the relations between the variables via the constraints are fundamental and well-studied, as they capture dependencies of the variables. In this paper, we apply these ideas from CBLS to the LNS context, presenting the novel dependency curation scheme, which exploits them to find a low-cardinality set of variables that the freeze set of any selection heuristic should be a subset of. The scheme often improves the overall performance of generic selection heuristics. Even when the scheme is used with a naïve generic selection heuristic that selects random freeze sets, the performance is competitive with more elaborate generic selection heuristics.

2012 ACM Subject Classification Computing methodologies → Heuristic function construction

Keywords and phrases Combinatorial Optimisation, Large Neighbourhood Search (LNS), Constraint-Based Local Search (CBLS)

Digital Object Identifier 10.4230/LIPIcs.CP.2025.20

Supplementary Material *Software (Solver)*: `https://github.com/astra-uu-se/gecode-lns` archived at `swh:1:dir:6415a1abc5e2361c3994f6e8a17cfc4716edc89f`
Software (Results, Experiments, and Scheme Generator): `https://github.com/astra-uu-se/gecode-lns-experiments`, archived at `swh:1:dir:099aba93950a85b9ccbb07a750c31877cb21f019`

Funding Supported by grant 2018-04813 of the Swedish Research Council (VR).

Acknowledgements We thank Mikael Zayenz Lagerkvist and Dexter Leander for their help with the Gecode implementation.

1 Introduction

Large neighbourhood search (LNS) [23, 19] is a method that combines systematic search with local search to improve the scalability of the former on constrained optimisation problems using heuristics of the latter. LNS starts from an incumbent solution that is iteratively improved by fixing a subset of the variables to their values in the incumbent solution and a value for each remaining variable is found and assigned via solving (such as constraint programming-style propagation and search). This method has been very successful on a wide variety of problems, such as vehicle routing [23, 2], bin packing [28], and scheduling [8, 24].

[1] Corresponding author

Traditionally, a modeller has to construct a problem-specific selection heuristic for the determination of the subset of variables to fix [8, 22]. However, there are now many variants of LNS that automatically determine that subset with good search performance, such as (but not limited to) (reverse) propagation guided LNS [18], cost impact guided LNS [12], explanation-based LNS [20], self-adaptive LNS [26], and variable-relationship guided LNS [24].

In most combinatorial optimisation solvers, such as for mixed integer linear programming, constraint programming (CP), Boolean satisfiability (SAT), and constraint-based local search (CBLS), when some variable x becomes fixed, the values of some other variables can be found and assigned via search-free unique solving (via inference such as CP propagation, SAT unit propagation, and CBLS invariant propagation). Therefore, the relations between x and the assigned variables are functional dependencies. In CBLS, these functional dependencies are fundamental and heavily studied [15, 16, 27, 10]. We have not found generic LNS selection heuristics or CP branching heuristics that automatically exploit these functional dependencies that are exploited in CBLS. However, they are often exploited manually by the modeller when for example creating a problem-specific CP branching heuristic that guides search.

Our contributions are:

- applying from CBLS to a CP context (and hence to CP-based LNS) the idea of a directed possibly cyclic graph that is induced by the functional dependencies between variables via the constraints of the model;
- designing a scheme that exploits the induced directed graph to remove variables from the LNS space that are functionally defined by others;
- describing how our scheme can be automated;
- showing that state-of-the-art generic LNS selection heuristics are easily extended to make use of our scheme;
- showing that our scheme often improves the overall performance when used with state-of-the-art generic selection heuristics; and
- showing that our scheme, when used with a naïve generic LNS selection heuristic that fixes a set of variables selected at random inside each LNS iteration, is competitive with more elaborate state-of-the-art generic LNS selection heuristics, even when they also use our scheme.

We first give related work from CBLS in Section 2, before we present how to apply it to a CP context (and hence to CP-based LNS) in Section 3. We then give an example of functional dependencies for a CP model in Section 4. We describe generic LNS selection heuristics from the literature in Section 5. We present our scheme in Section 6. We give our experiment setup in Section 7 and discuss the results in Section 8. Finally, we conclude in Section 9.

2 Constraint-Based Local Search

In constraint-based local search [15, 16, 27, 10] (CBLS), each constraint of the problem is either made to always implicitly hold via the initialisation and neighbourhood, or replaced by one or more invariants, where an *invariant* (also called a *one-way constraint*) defines a subset of the constrained variables, called its *output variables*, of the replaced constraint by the remaining variables, called its *input variables*. Invariants that replace a constraint of the problem are called violation invariants, where a *violation invariant* introduces and defines a *violation variable*, absent in the problem, which takes the value 0 if the replaced constraint holds, else a positive value, usually proportional to the amount of violation. The replaced constraint is therefore softened, but only an assignment of the variables where all violation variables take the value 0 is a solution, as it satisfies all the constraints of the problem.

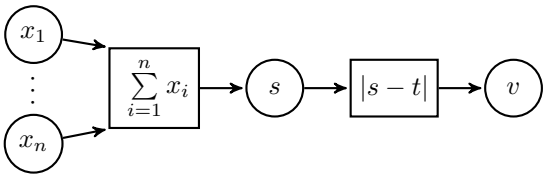

Figure 1 An invariant graph for the subset-sum constraint satisfaction problem of size n with $n+2$ integer variables and two invariants. Note that t is not a variable.

The invariants and variables induce a directed graph, called the *invariant graph*, with the invariants and variables as vertices and the definitions of variables via invariants as arcs, where each variable has an outgoing arc to each invariant that it is an input variable to and an incoming arc from the invariant that defines it (if any). Note that in any CBLS model, any variable is defined by at most one invariant. All the possibly multiple (if not zero) source vertices and possibly multiple (if not zero) sink vertices are variables, where the source variables transitively define the remaining variables via the invariants.

As an example of an invariant graph, consider the subset-sum constraint satisfaction problem, where given a set of n integers v_i and an integer t, a subset of the set is to be found that sums to exactly t. A corresponding invariant graph contains n integer variables, where each variable x_i takes the value v_i if v_i is in the subset and 0 otherwise; one invariant with as inputs the variables x_i and as output the integer variable s, which takes as value the sum of the integers in the subset; and one violation invariant with as input s and as output the violation variable v, which is to be minimised and takes as value the absolute difference between s and t. This amounts to softening the constraint $\sum_{i=1}^{n} x_i = t$. The source variables are the variables x_i, and they transitively define the other variables via the two invariants. Any assignment of the variables x_i where s takes the value t is a solution, as $v = 0$ then, which is minimal. The invariant graph is shown in Figure 1, where circled vertices are variables and boxed vertices are invariants.

During a CBLS-style search, only the values of the source variables are changed, while the values of the remaining variables are found via CBLS-style invariant propagation.

In CBLS, the variables and constraints induce the invariant graph of definitions of variables, whether this is done by the CBLS modeller or by automated analysis of, say, a MiniZinc [17] model [4]. In the sequel, we apply the idea of the induced invariant graph to CP models to find functional definitions of variables, which can be exploited during LNS.

3 The Dependency Graph of a CP Model

In a CP model, some constraints are dependency constraints, where a *dependency constraint* c on the variables $\mathcal{X} = \mathcal{I} \cup \mathcal{O}$ determines the values of the variables in $\mathcal{O}$ when the variables in $\mathcal{I}$ become fixed. We can view c as a function that determines the values of output variables $\mathcal{O}$ when the input variables $\mathcal{I}$ become fixed. We say that c is a *dependency constraint* and that $\mathcal{I}$ *functionally defines* $\mathcal{O}$ via c, denoted by $\mathcal{I} \overset{c}{\Longrightarrow} \mathcal{O}$, or simply that $\mathcal{I}$ *functionally defines* $\mathcal{O}$.

For example, consider the integer variables y and i, the array $\mathcal{P}$ of integers, and the constraint Element$(\mathcal{P}, i, y)$, which constrains y to be equal to the integer in $\mathcal{P}$ at index i. When i becomes fixed to some value v, the value of y becomes assigned to the integer in $\mathcal{P}$ at index v. The constraint is a dependency constraint, via which i functionally defines y.

As another example, consider the Boolean variable b, the array of Boolean variables $\mathcal{B}$, and the constraint Exists$(\mathcal{B}, b)$, which constrains b to take the value **True** if any of the variables in $\mathcal{B}$ takes the value **True**, else b takes the value **False**. The constraint is a dependency constraint, via which $\mathcal{B}$ functionally defines b.

Some constraints are not dependency constraints. For example, consider the integer variables x and y, and the constraint LessEqual(x, y), which enforces that x is smaller than or equal to y. This constraint is not a dependency constraint as neither x nor y can be defined via it by the other variable.

For a CP model, the dependency constraints and the variables induce a directed graph, called the *dependency graph* [13], where for each set $\mathcal{I}$ of variables that functionally defines a set $\mathcal{O}$ of variables via some dependency constraint c, there is a vertex d, an arc $x \to d$ for each vertex $x \in \mathcal{I}$, and an arc $d \to y$ for each vertex $y \in \mathcal{O}$. Note that the dependency graph is possibly cyclic: we will see an example of an acyclic dependency graph in Figure 2 in Section 4, and an example of a cyclic dependency graph in Figure 3 in Section 7.2.3. Also note that all the variables and all the dependency constraints are in the dependency graph, while the other (non-dependency) constraints are not.

A dependency graph differs from an invariant graph as follows:

- in a dependency graph, a variable can have any number of incoming arcs, while it can have at most one in an invariant graph;
- a dependency graph does not contain non-dependency constraints, while in an invariant graph, all constraints that are not made to implicitly hold are encoded (as invariants); and
- an invariant graph is required when performing CBLS-style invariant propagation, while a dependency graph is not required when performing CP-style solving (neither in search nor in CP-style propagation).

For any acyclic dependency graph, the source variables transitively functionally define all remaining variables. Therefore, for any acyclic dependency graph, the minimum-cardinality set of such variables is the set of source variables. However, this does not always hold for a cyclic dependency graph (as we will see for the dependency graph shown in Figure 3 in Section 7.2.3). In Section 6, we present a greedy scheme that, given a CP model (inducing a cyclic or acyclic dependency graph), finds a low-cardinality set of such variables, which can be exploited to guide LNS.

4 Example: Relaxed Car Sequencing (RCS)

We now give an example to show how the dependency graph is constructed from a CP model. The car sequencing (constraint satisfaction) problem [5] has a set of car classes, each class with a set of mandatory features (called options in [5]), each feature with a capacity on how many cars of that feature can be produced in a subsequence of given size, and an order stating how many cars of each class to produce. The problem is to find a production sequence for all ordered cars, such that the capacity restrictions on features are satisfied and all ordered cars are produced. Since LNS can only be performed on constrained optimisation problems, we transform the problem into one by relaxing it into the relaxed car sequencing (RCS) problem by (i) including an additional dummy class of car that has no features; (ii) relaxing the constraint that all ordered cars are to be produced by replacing it by a constraint that each car is to be produced at most once; and (iii) introducing an objective variable that is to be minimised and is the number of dummy cars that are produced. Note that for any solution to the relaxed problem, if the objective variable takes the value 0, then the solution satisfies the constraints of the original problem, as all cars are produced.

A model for RCS in the solver-independent MiniZinc language [17] is in Listing 1. Lines 1–7 declare the set of cars to be produced, the set of features, the set of car classes, the array of the sizes of the feature subsequences, the array of the maximum capacities for

Listing 1 A MiniZinc model for the relaxed car sequencing (RCS) problem.

```
1  set of int: Cars;
2  set of int: Features;
3  set of int: Classes;
4  array[Features] of int: sequenceSize;
5  array[Features] of int: sequenceCapacity;
6  array[Classes] of int: numOrdered;
7  array[Classes, Features] of bool: classHasFeature;
8  int: dummyClass = max(Classes) + 1;
9  set of int: ClassesAndDummy = Classes union {dummyClass};
10 array[ClassesAndDummy, Features] of bool: classHasFeatureExt =
11   array2d(ClassesAndDummy, Features,
12     array1d(classHasFeature) ++ [false | _ in Features]);
13 array[Cars] of var ClassesAndDummy: class;
14 array[Cars, Features] of var bool: carHasFeature = array2d(
15   Cars, Features, [classHasFeatureExt[class[car], f]
16                    | car in Cars, f in Features]);
17 array[ClassesAndDummy] of var int: numProduced =
18   global_cardinality_closed(class, ClassesAndDummy);
19 constraint forall(c in Classes)(numProduced[c]<=numOrdered[c]);
20 constraint forall(f in Features)(
21   sliding_sum(0, sequenceCapacity[f],
22               sequenceSize[f], carHasFeature[.., f]));
23 solve minimize numProduced[dummyClass];
```

the feature subsequences, the array of the numbers of ordered cars for each class, and the two-dimensional matrix of the features that the car classes use respectively. The sets of cars to be produced, features, and car classes are required to be intervals. Lines 8–12 declare the dummy car class, the set of car classes including the dummy car class, and the two-dimensional matrix of the features that the car classes and dummy class use, respectively. On line 13, the array `class` of variables is declared, where `class[c]` denotes the class of car `c`. On lines 14–16, the two-dimensional matrix `carHasFeature` of variables is declared and is functionally defined by `class`, where `carHasFeature[c,f]` denotes if feature `f` is to be installed on car `c`. On lines 17–18, the array `numProduced` of variables is declared and is functionally defined by `class`, where `numProduced[c]` denotes the number of cars of class `c` that are actually produced. Line 19 enforces that no car class is overproduced. Lines 20–22 enforce that the feature capacity is not violated for any subsequence. Finally, Line 23 declares that the number of dummy cars is to be minimised.

This model was retrieved from the MiniZinc Benchmark repository and relaxed from a constraint satisfaction model into a constrained optimisation model as well as rewritten by renaming parameters and variables and replacing each constraint predicate with the corresponding constraint function where possible.[2]

The acyclic dependency graph for RCS with three cars to produce, two features, and two car classes induced by the MiniZinc model in Listing 1 is shown in Figure 2, where circled vertices are variables and boxed vertices are dependency constraints.

[2] The MiniZinc Benchmark repository: `https://github.com/MiniZinc/minizinc-benchmarks`

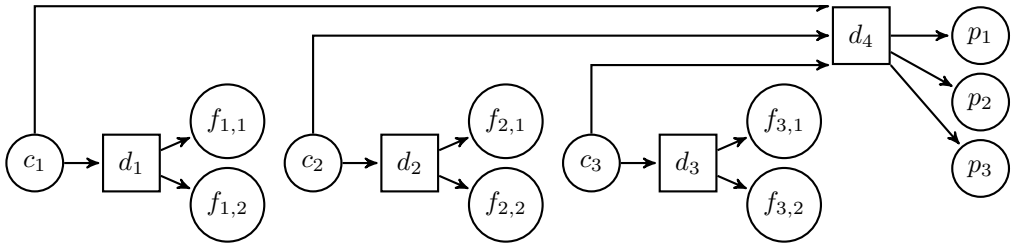

Figure 2 The acyclic dependency graph induced by the RCS MiniZinc model of Listing 1 with 3 cars, 2 features, and 2 non-dummy car classes. Integer variable c_i denotes `class[`i`]`, Boolean variable $f_{i,j}$ denotes `carHasFeature[`i`,`j`]`, and integer variable p_ℓ denotes `numProduced[`ℓ`]`. Vertices d_1, d_2, and d_3 correspond to the dependency constraints on lines 14–16 and vertex d_4 corresponds to lines 17–18. The constraints in lines 19–22 are non-dependency constraints and are therefore absent in the dependency graph.

5 Large Neighbourhood Search

LNS [23, 19] is an iterative method for optimisation problems, starting from an incumbent feasible solution that is improved during search. In every iteration, first a subset of the variables, called the *freeze set*, is selected and the values of these variables are kept from the incumbent solution, then search (for example CP-style branch and bound search) is performed until some stopping criterion is met, and finally, if a better solution was found, then the incumbent solution is replaced by it and a constraint that all future solutions must be better than the new incumbent solution is added. The selection of a freeze set is done via a *selection heuristic*, which can be either problem-specific or generic. Typically, problem-specific selection heuristics (such as [8, 22]) cannot easily be reused between problems.

In Section 5.1 we give a naïve generic selection heuristic that selects the freeze set at random, and in Sections 5.2 to 5.4 we give more elaborate generic selection heuristics from the literature.

5.1 Randomised LNS

Randomised LNS is a naïve generic selection heuristic that inside each LNS iteration creates a freeze set by, for each variable x, selecting uniformly from the continuous closed interval $[0, 1]$ a value p: if p is below some threshold ϕ (in our implementation $\phi \in [0.05, 0.95]$), then x is included in the freeze set. The threshold ϕ is typically updated during search (with initial value 0.6 in our implementation), where it is decreased to escape local minima of the LNS space, while it is increased to focus on exploring a smaller part of the LNS space.

5.2 Propagation Guided LNS and Reverse Propagation Guided LNS

Propagation guided LNS [18] (PG-LNS) has a selection heuristic that gradually expands the freeze set inside each LNS iteration, starting from the empty set, by freezing variables with the largest reduction in domain size. The freeze set is expanded until the size of the CP search space becomes smaller than some predefined size (computed from the sum of the logarithms of the sizes of the current domains of the variables).

Reverse propagation guided LNS [18] (RPG-LNS) differs from PGLNS by having a selection heuristic that gradually excludes variables from the freeze set inside each LNS iteration, starting from the set of all variables, by excluding variables with the smallest reduction in domain size. The freeze set is shrunk until the size of the CP search space becomes greater than some predefined size (which is dynamically updated during search).

5.3 Cost Impact Guided LNS

Cost impact guided LNS [12] (CIG-LNS) has an impact collecting procedure and a selection heuristic, where the latter makes use of information gathered by the former.

During impact collection, several dives are performed, where in each *dive*, first a random permutation of the variables is constructed, then each variable in the permutation is fixed to its value in the incumbent solution, and then the change in the lower (upper) bound of the objective variable (often called *cost variable*) for a minimisation (maximisation) problem is retrieved. For each variable and over the dives, the sum of the retrieved changes in the bound of the objective variable is stored and used by the selection heuristic. For each variable, this sum is called the *impact* of that variable.

The selection heuristic gradually shrinks the freeze set inside every LNS iteration, starting from the set of all variables, by excluding the variables with the greatest impacts as gathered by the impact collecting procedure.

Typically, the impact collecting procedure is performed every 10 LNS iterations and 10 dives are performed during it.

5.4 Variable-Relationship Guided LNS

Given a variable x that is not fixed, the *local cost* of x is the change in the lower (upper) bound of the objective variable for a minimisation (maximisation) problem when x is fixed to its value in the incumbent solution.

Variable-relationship guided LNS [24] (VRG-LNS) has a selection heuristic such that inside each LNS iteration the freeze set is gradually shrunk, starting from the set of all variables, by either (i) first selecting a random subset of the variables and then removing one amongst them that has the greatest local cost, or (ii) removing a variable that is constrained by one or more of the same constraints as the previously excluded variable. The selection heuristic shrinks the freeze set by alternating between (i) and (ii), starting with (i). The number of excluded variables is low in initial iterations but changes during search, where it is increased in some iterations to escape local minima of the LNS space, before it is restored to its original size.

The VRG-LNS selection heuristic (ii) selects a variable x that shares constraints with the previously selected variable y, but does not exploit what we call the dependency graph as the shared constraints do not have to be dependency constraints and there is no requirement that x or y functionally define each other.

6 Dependency Curation for LNS

In the selection heuristics of PG-LNS, RPG-LNS, and VRG-LNS, the freeze set is gradually updated inside each LNS iteration to include variables such that many other variables are assigned values via CP-style propagation [18, 24]. For these selection heuristics, such variables are found either experimentally or heuristically *during* search. We now show such variables can also be found by exploiting the dependency graph *before* search starts and how they can be used by any LNS selection heuristic.

We call any set of variables that transitively functionally define all remaining variables a set of *search variables*, as only their values must be found via search.

Consider a CP model with the set $\mathcal{V}$ of variables. Finding a set $\mathcal{S} \subseteq \mathcal{V}$ of search variables is trivial as $\mathcal{V}$ itself is a set of search variables, though of maximum cardinality. Our idea is that the smaller the set $\mathcal{S}$ is, the more CP-style propagation will occur, as the value of each

variable in $\mathcal{V} \setminus \mathcal{S}$ is found and assigned via only propagation (as $\mathcal{S}$ transitively functionally defines $\mathcal{V} \setminus \mathcal{S}$). Additionally, for any (generic or problem-specific) selection heuristic, if the freeze set is forced to become a subset of $\mathcal{S}$, then the LNS space is reduced, no data has to be stored, and no operations have to be performed on any variable in $\mathcal{V} \setminus \mathcal{S}$ by the selection heuristic, potentially improving its memory footprint and running time.

Note that if the dependency graph is acyclic, then the minimum-cardinality set of search variables is the set of source variables. Otherwise, the dependency graph contains at least one strongly connected component (SCC) with two or more vertices, and finding a low-cardinality set of search variables is a subtle issue, as discussed next.

As the variables and dependency constraints are known up-front, a low-cardinality set of search variables can be constructed before search starts. A (generic or problem-specific) selection heuristic can use the constructed set of search variables throughout search by forcing the freeze set to be a subset of that set.

Our scheme, called the *dependency curation scheme* (DCS), finds a low-cardinality set of search variables given the variables, vertices, and arcs of a dependency graph. It is shown in greedy Algorithm 1, where:

- function stronglyConnectedComponents($\mathcal{N}, \mathcal{A}$) returns an ordered list of SCCs for vertices $\mathcal{N}$ and arcs $\mathcal{A}$, such that each SCC is before all other SCCs that are reachable from it, like [25] but where the returned list is reversed (as the graph is explored in depth-first order);
- function in($\mathcal{A}, v$) returns all incoming arcs to vertex v in the set $\mathcal{A}$ of arcs;
- function out($\mathcal{A}, v$) returns all outgoing arcs from vertex v in the set $\mathcal{A}$ of arcs;
- function stack() returns an empty stack;
- function empty($\mathcal{Z}$) returns **True** if the stack $\mathcal{Z}$ contains no elements, else **False**;
- function pop($\mathcal{Z}$) removes the top-most element from the stack $\mathcal{Z}$ and returns it; and
- procedure push($\mathcal{Z}, v$) pushes element v onto stack $\mathcal{Z}$.

First, the sets $\mathcal{W}$ of visited vertices and $\mathcal{S}$ of search variables are initialised, lines 2–3. Each SCC of the dependency graph is iterated over, where SCC $\mathcal{X}$ is iterated over before all other SCCs that are reachable from $\mathcal{X}$ are iterated over, line 4. While there are unvisited variables in $\mathcal{X}$, line 5, a variable x with the lexicographically lowest priority is retrieved from the unvisited variables in $\mathcal{X}$, where the *priority* of a variable y is (i) the in-degree of y and (ii) the opposite of the out-degree of y, line 6. Variable x is added to the set $\mathcal{S}$ of search variables, line 7, and to the set $\mathcal{W}$ of visited vertices, line 8. The empty stack $\mathcal{Z}$ is created, line 9, before x is added as the top-most element to $\mathcal{Z}$, line 10. While $\mathcal{Z}$ is not empty, line 11, its top-most element u is removed, line 12. The outgoing arcs $\mathcal{A}_u$ from u are retrieved, line 13, and removed from $\mathcal{A}$, line 14. For each arc $(u, v) \in \mathcal{A}_u$, where v has not been visited and either (i) is a variable or (ii) has no incoming arcs in $\mathcal{A}$, line 15, vertex v is added to the set $\mathcal{W}$ of visited vertices, line 16; its incoming arcs are removed from $\mathcal{A}$, line 17; and v is added as the top-most element to $\mathcal{Z}$, line 18. Finally, the found set $\mathcal{S}$ of search variables is returned, line 19.

Note that Algorithm 1 is greedy as the selection of a search variable x, line 6, potentially makes the set $\mathcal{S}$ *not* a minimal-cardinality set of search variables.

For example, consider the car sequencing problem given in Section 4 with three cars, two features, and two car classes; its MiniZinc model in Listing 1; and its dependency graph in Figure 2. The dependency graph contains 16 SCCs: one for each vertex. Algorithm 1 first creates the initially empty sets of search variables and visited vertices, lines 2–3. The sets $\{c_1\}$, $\{c_2\}$, and $\{c_3\}$ of vertices are the first three SCCs that are iterated over, as c_1, c_2, and c_3 are the only vertices without incoming arcs, line 4. The remaining lines then explore

Algorithm 1 The dependency curation scheme for finding a low-cardinality set of search variables.

Data: $\mathcal{V}$ is the set of variables, $\mathcal{N} \supseteq \mathcal{V}$ is the set of nodes in the dependency graph, and $\mathcal{A}$ is the set of arcs in the dependency graph.
Result: A set $\mathcal{S} \subseteq \mathcal{V}$ of search variables, which transitively functionally define the remaining variables $\mathcal{V} \setminus \mathcal{S}$ via the constraints.

function dependency-curation($\mathcal{V}, \mathcal{N}, \mathcal{A}$)**:**
  $\mathcal{W} \coloneqq \emptyset$ // The set of visited vertices
  $\mathcal{S} \coloneqq \emptyset$ // The set of search variables
  forall $\mathcal{X} \in$ stronglyConnectedComponents($\mathcal{N}, \mathcal{A}$) **do**
    while $(\mathcal{X} \cap \mathcal{V}) \setminus \mathcal{W} \neq \emptyset$ **do**
      $x \coloneqq \arg\operatorname{lex}\min_{y \in (\mathcal{X} \cap \mathcal{V}) \setminus \mathcal{W}} (\langle |\text{in}(\mathcal{A}, y)|, -|\text{out}(\mathcal{A}, y)| \rangle)$
      $\mathcal{S} \coloneqq \mathcal{S} \cup \{x\}$
      $\mathcal{W} \coloneqq \mathcal{W} \cup \{x\}$
      $\mathcal{Z} \coloneqq$ stack() // The stack used for depth-first search
      push($\mathcal{Z}, x$)
      while ¬empty($\mathcal{Z}$) **do**
        $u \coloneqq$ pop($\mathcal{Z}$)
        $\mathcal{A}_u \coloneqq \text{out}(\mathcal{A}, u)$
        $\mathcal{A} \coloneqq \mathcal{A} \setminus \mathcal{A}_u$
        forall $u \to v \in \mathcal{A}_u$ **where** $v \notin \mathcal{W}$ **and** $(v \in \mathcal{V}$ **or** $\text{in}(\mathcal{A}, v) = \emptyset)$ **do**
          $\mathcal{W} \coloneqq \mathcal{W} \cup \{v\}$
          $\mathcal{A} \coloneqq \mathcal{A} \setminus \text{in}(\mathcal{A}, v)$
          push($\mathcal{Z}, v$)
  return $\mathcal{S}$

the dependency graph in a fashion similar to depth-first search, starting from these source variables, where traversed outgoing arcs are removed and outgoing arcs from any non-variable vertex are only traversed if all its incoming arcs have been traversed and removed. The dependency graph is acyclic, and therefore the variables c_1, c_2, and c_3 transitively functionally define the remaining variables and actually form (in this case) a minimum-cardinality set of search variables.

If each strongly connected component only has a single vertex, then the dependency graph is acyclic and the algorithm can be bypassed, as the minimum-cardinality set of search variables is the set of source variables.

For a CP model, DCS finds a low-cardinality set of search variables that can be used with any (generic or problem-specific) selection heuristic by forcing its freeze set to be a subset of that set of search variables. Note that DCS itself is not a selection heuristic and must be used together with one, such as any of the generic selection heuristics in Sections 5.1 to 5.4.

VRG-LNS (Section 5.4) [24] is similar to DCS as both find the information of relations on variables via constraints before LNS iterations. However, they differ from each other, as VRG-LNS uses *both* dependency and non-dependency constraints that each pair of variables share, while DCS uses the functional definitions of variables via *only* dependency constraints.

7 Experiments

In Section 7.1, we give the details of our extensions to the solver that was used for the experiments. In Section 7.2, we describe how we selected the problems and for each problem, its specification and how its initial incumbent solutions are generated.

7.1 Setup

We extended the Gecode-based information-sharing portfolio solver in [11] – containing amongst other assets the CP solver Gecode [6]; Gecode-based LNS for randomised LNS, PG-LNS, and RPG-LNS; and Gecode-based LNS inspired by CIG-LNS and VRG-LNS – by:

- reimplementing CIG-LNS and VRG-LNS by following [12] and [24] more closely;
- allowing for supplying a set of search variables;
- enforcing the freeze set to be a subset of the set of search variables, if one was supplied;
- allowing for running a single generic LNS selection heuristic without any parallelism instead of simultaneously running multiple communicating assets in parallel (effectively making it a non-portfolio solver); and
- allowing for supplying an initial solution to be used as the first incumbent solution.

Note that we could have chosen any open-source CP solver, such as MiniCP [14], but we chose to extend the solver in [11], as it is very competitive; already has support for randomised LNS, PG-LNS, and RPG-LNS; as well as already has support for a variation each of CIG-LNS and VRG-LNS.

We decided not to implement the generic selection heuristics of explanation-based LNS [20] and self-adaptive LNS [26], as there is no Gecode-based LNS or one inspired thereof for either of them in the portfolio solver [11].

Additionally, since variable objective LNS [21] does not have a generic selection heuristic and therefore must be paired with (a generic or problem-specific) one, we decided not to use the generic selection heuristic that makes use of it [11].

There are problem-specific LNS selection heuristics, for example [8] for the job shop problem and [22] for steel mill slab design. However, we here only compare the performance of generic LNS selection heuristics, both with and without the use of DCS, as DCS can be used with any (generic or problem-specific) LNS selection heuristic. Additionally, we believe that the use of DCS for most problem-specific LNS selection heuristics gives little to no improvement in performance, as for any such LNS selection heuristic, the functional definitions via constraints are typically exploited by it by design.

For each run on a problem instance, we supply the same initial incumbent solution in order to make a fair comparison between selection heuristics on the same instance, as we can then compare the quality of their found solutions.

7.2 Problem Selection

We selected four problems: relaxed car sequencing (Section 4) [5] (RCS), steel mill slab design [9, 22] (SMSD), job shop (JSP), and the travelling salesperson problem with time windows [7] (TSPTW). RCS and SMSD were selected since their models contain many dependency constraints, and therefore our conjecture was that DCS improves performance. Conversely, JSP was selected since its model contains few dependency constraints, and therefore our conjecture was that DCS does not improve performance. Lastly, TSPTW was selected since the induced dependency graph is cyclic (as will be seen in Section 7.2.3), because of cyclic dependency constraints. SMSD was used both in [12] and [24], while a version of RCS was also used in the latter.

We used models in the solver-independent MiniZinc language [17] and instances for RCS and SMSD from the MiniZinc Benchmark repository, where we rewrote each model and instance for ease of readability, by renaming parameters and variables, moving some parameters from the models to the instances, and replacing each constraint predicate with the corresponding constraint function where possible. As the JSP model in the MiniZinc benchmark repository does not make use of suitable global constraints (such as `disjunctive`), we handcrafted a new MiniZinc model for JSP but used MiniZinc instances from the MiniZinc benchmark repository. We handcrafted a MiniZinc model for TSPTW, starting from [3]. We translated published instances for TSPTW into MiniZinc-readable instances.[3]

We created for each problem a DCS generator that, for each problem instance, finds a low-cardinality set of search variables.

We gave the specification for RCS in Section 4. The dependency graph for each RCS instance is acyclic, and the minimum-cardinality set of search variables is the set of classes of the cars, as shown in Section 6. The initial incumbent solution for each RCS instance is where the class of each car is the dummy class, as that fulfils the capacity restrictions on the features.

We give the specifications, the found sets of search variables, and how the initial incumbent solutions are created for the remaining problems in Sections 7.2.1 to 7.2.3.

7.2.1 Steel Mill Slab Design (SMSD)

The steel mill slab design problem is a constrained optimisation problem that has a set of orders, each with a size and colour; a set of slabs, each taking one of a set of capacities; and a colour capacity c. Orders are to be assigned to slabs, such that for each slab, the number of differently coloured orders does not exceed c and the sizes of the fitted orders do not exceed the capacity of the slab. The objective is to minimise the unused capacity (called *slack*) of the slabs (unused slabs have zero slack).

The SMSD MiniZinc model also contains symmetry-breaking constraints that (i) for any pair of slabs s_j and s_k, with $j < k$, required the total size of the fitted orders of s_j to be at most the total size of the fitted orders of s_k and (ii) for any pair of orders o_ℓ and o_m, with $\ell < m$ and where o_ℓ and o_m both have the same colour and size, require order o_ℓ to be placed in the same slab as o_m or a previous slab than o_m.

The dependency graph for each instance is acyclic, and the minimum-cardinality set of search variables is the set of slabs that the orders are placed in.

The initial supplied incumbent solution for each instance is automatically generated by assigning each order to a distinct slab.

7.2.2 Job Shop (JSP)

The job shop problem is a constrained optimisation problem that has a set $\mathcal{M}$ of machines and a set of jobs, where each job consists of a sequence of $|\mathcal{M}|$ tasks, each having a discrete duration, and the tasks of a job each require a distinct machine. For any job j, before the task at index i may be started, all previous tasks $1\,..\,(i-1)$ of j must have been completed. No two tasks requiring the same machine can be performed simultaneously. Once a machine has started working on a task, it must run it until completion. The objective is to minimise the latest completion time of all tasks.

[3] TSPTW instances: `https://myweb.uiowa.edu/bthoa/tsptwbenchmarkdatasets.htm`

The dependency graph for each instance is acyclic, and the minimum-cardinality set of search variables is the set of starting times of the tasks.

The initial supplied incumbent solution is to perform the tasks of each job sequentially, without any two tasks of any two jobs being performed simultaneously.

7.2.3 Travelling Salesperson with Time Windows (TSPTW)

Consider n locations that are to be visited, where one location is called the *depot*, with a travelling duration between each directed pair of locations, and for each location u an earliest visiting time e_u and a latest visiting time ℓ_u. A TSPTW *tour* is a Hamiltonian cycle of the weighted directed graph induced by the n locations as nodes, starting at the depot, visiting each location u exactly once and between times e_u and ℓ_u, with a zero visit duration. If the salesperson arrives at location u before e_u, then they have to wait until e_u before departing. The total duration of the tour is to be minimised.

Note that for other TSPTW problems, the objective function to be minimised is typically the total travel time, such as in [3]. However, and only for realism, we here instead minimise the total duration of the tour, including the wait times.

A MiniZinc model for TSPTW is in Listing 2. Lines 1–6 declare the number of locations, the depot location, the set of locations to be visited, the array of the earliest visiting times of the locations, the array of the latest visiting times of the locations, and the two-dimensional matrix with the travelling durations between the locations, respectively. On line 7, the array `pred` of variables is declared, where `pred[l]` denotes the location that is visited just before location `l`. On lines 8–9, the array `durFromPred` of variables is declared and is functionally defined by `pred`, where `durFromPred[l]` denotes the travelling duration from location `pred[l]` to location `l`. On line 10, the array `arrival` of variables is declared, where `arrival[l]` denotes the arrival time at location `l`. The arrival time back at the depot is the total duration of the tour. On lines 11–12, the array `departure` of parameters and variables is declared and is functionally defined by the arrival times at non-depot locations in `arrival`, where `departure[l]` denotes the departure time from location `l`; it differs from `arrival[l]` if `l` is the depot or the salesperson arrives at `l` before `early[l]` and has to wait there until `early[l]` before departing. Note that `departure[depot]` is the earliest visiting time of the depot, which is a parameter. On lines 13–14, the array `departurePred` of variables is declared and is functionally defined by `pred` and `departure`, where `departurePred[l]` is the departure time from location `pred[l]`. On lines 15–16, the arrival time at each location is functionally defined by `durFromPred` and `departurePred`. Line 17 enforces that the `pred` variables form a Hamiltonian cycle. Line 18 enforces that each location is visited before its latest visiting time. Finally, line 19 declares that `arrival[depot]` is to be minimised, which is the total duration of the tour.

For each location `l` variable `pred[l]` defines variable `durFromPred[l]`. The variables in the array `departure` are transitively functionally defined by themselves and the variables in the array `pred`, and the same is true for the variables in the arrays `departurePred` and `arrival`. Therefore, there is a cycle in the induced dependency graph containing the variables in the arrays `departure`, `departurePred`, and `arrival`. Using DCS, the greedily found low-cardinality set of search variables has the variables in the arrays `pred` and `departure`. The cyclic dependency graph for TSPTW with three locations corresponding to the MiniZinc model in Listing 2 is shown in Figure 3.

Listing 2 A MiniZinc model for the travelling salesperson with time windows (TSPTW) problem.

```
1  int: n;
2  int: depot = 1;
3  set of int: Locs = 1..n;
4  array[Locs] of int: early;
5  array[Locs] of int: late;
6  array[Locs, Locs] of int: duration;
7  array[Locs] of var Locs: pred;
8  array[Locs] of var 0..max(duration): durFromPred = [
9      duration[l, pred[l]] | l in Locs];
10 array[Locs] of var 0..max(late): arrival;
11 array[Locs] of var min(early)..max(late): departure =
12   [early[depot]] ++ [max(early[l], arrival[l]) | l in 2..n];
13 array[Locs] of var 0..sum(duration): departurePred = [
14     departure[pred[l]] | l in Locs];
15 constraint forall (l in Locs) (
16     arrival[l] = departurePred[l] + durFromPred[l]);
17 constraint circuit(pred);
18 constraint forall (l in Locs) (departure[l] <= late[l]);
19 solve minimize arrival[depot];
```

In order to generate initial incumbent solutions for the TSPTW instances, we created a MiniZinc model for a constrained satisfaction problem from Listing 2 by replacing line 19 with `solve satisfy`. We used the Gecode [6] CP solver (without LNS) with a timeout of 10 minutes to find for each instance a solution to be used as the initial incumbent solution. For some instances, no solution was found before timing out, and they are therefore not used.

8 Results

We ran our experiments on a desktop computer with an ASUS PRIME Z590-P motherboard, a 3.5 GHz Intel Core i9 11900K processor, and four 16 GB 3200 MT/s DDR4 memories, running Ubuntu 22.04.4 LTS with GCC (the GNU Compiler Collection) 11.

For each selection heuristic, both with DCS and without DCS, and each problem instance, we ran 10 independent runs, each under a timeout of 3 minutes from the same initial incumbent solution.

We used the scoring function $100 \cdot (o - b) \div i$ proposed in [1] and used in [24], where for each instance, the mean objective value over the 10 independent runs is denoted by o, the best found objective value over any run for the instance is denoted by b, and the objective value of the initial incumbent solution is denoted by i. Therefore, the scores range over the closed continuous interval $[0, 100]$, where a low score is better than a high one.

The results are shown in Table 1 and Figure 4. For all generic selection heuristics, using DCS improves performance for RCS, SMSD, and TSPTW, but worsens performance for JSP. These results support our conjectures that using DCS improves performance for RCS and SMSD, but does not improve performance for JSP. For JSP and each generic selection heuristic except VRG-LNS, using DCS has significantly worse performance than not using it. For JSP and VRG-LNS, using DCS has similar performance to not using it. However, there are a few JSP instances where using DCS with VRG-LNS, PG-LNS, and RPG-LNS

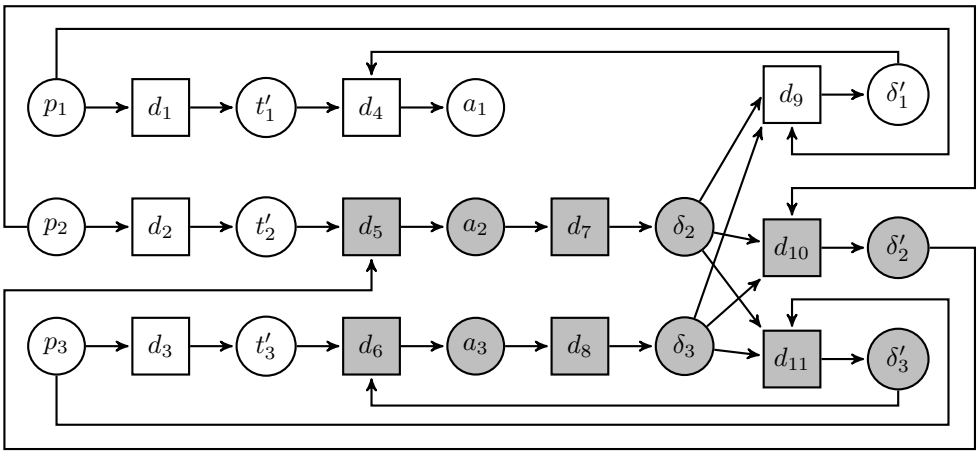

Figure 3 The cyclic dependency graph for the TSPTW MiniZinc model of Listing 2 with 3 locations (for ease of readability). Variable p_ℓ denotes `pred[`ℓ`]`, variable t'_ℓ denotes `durFromPred[`ℓ`]`, variable a_ℓ denotes `arrival[`ℓ`]`, variable δ_ℓ denotes `departure[`ℓ`]`, and variable δ'_ℓ denotes `departurePred[`ℓ`]`. The depot is location 1 and the objective variable is a_1. Vertices d_1, d_2, and d_3 correspond to the dependency constraints on lines 8–9; vertices d_4, d_5, and d_6 correspond to lines 15–16; vertices d_7 and d_8 correspond to lines 11–12; vertices d_9, d_{10}, and d_{11} correspond to lines 13–14; and the vertices with grey backgrounds together form an SCC, while each other SCC has a single vertex. The constraints on lines 17 and 18 are non-dependency constraints and are therefore absent in the graph.

has significantly better performance than when not using it. For RCS and each selection heuristic, the performance of using DCS is significantly better than when not using it. For both SMSD and TSPTW and each selection heuristic, the performance of using DCS is better than when not using it. Additionally, for RCS, SMSD, and TSPTW, using DCS makes the naïve randomised LNS competitive with the remaining more elaborate generic selection heuristics, even when they use DCS.

9 Conclusion and Future Work

We have presented our dependency curation scheme (DCS), which can be used with any (generic or problem-specific) LNS selection heuristic. We have compared the performance of a naïve generic randomised selection heuristic and more elaborate state-of-the-art generic selection heuristics from the literature both with and without DCS, revealing overall improved

Table 1 The mean score for 72 JSP instances, 79 RCS instances, 80 SMSD instances, and 112 TSPTW instances, each over 10 independent runs, both with and without the dependency curation scheme (DCS), for CIG-LNS, PG-LNS, randomised LNS, RPG-LNS, and VRG-LNS, where the "no" and "yes" columns denote if DCS was used or not. Boldface indicates the best performance on each row.

	CIG-LNS		PG-LNS		Randomised LNS		RPG-LNS		VRG-LNS	
DCS	no	yes	no	yes	no	yes	no	yes	no	yes
JSP	**0.13**	7.45	7.30	28.53	0.18	18.37	32.68	18.35	32.20	33.19
RCS	22.11	0.47	9.94	0.48	24.72	**0.34**	9.93	0.43	20.35	0.63
SMSD	1.62	0.87	4.06	**0.67**	1.14	0.85	5.64	0.73	6.45	0.81
TSPTW	4.65	1.82	3.67	**1.36**	4.87	2.18	3.71	1.80	5.41	2.20

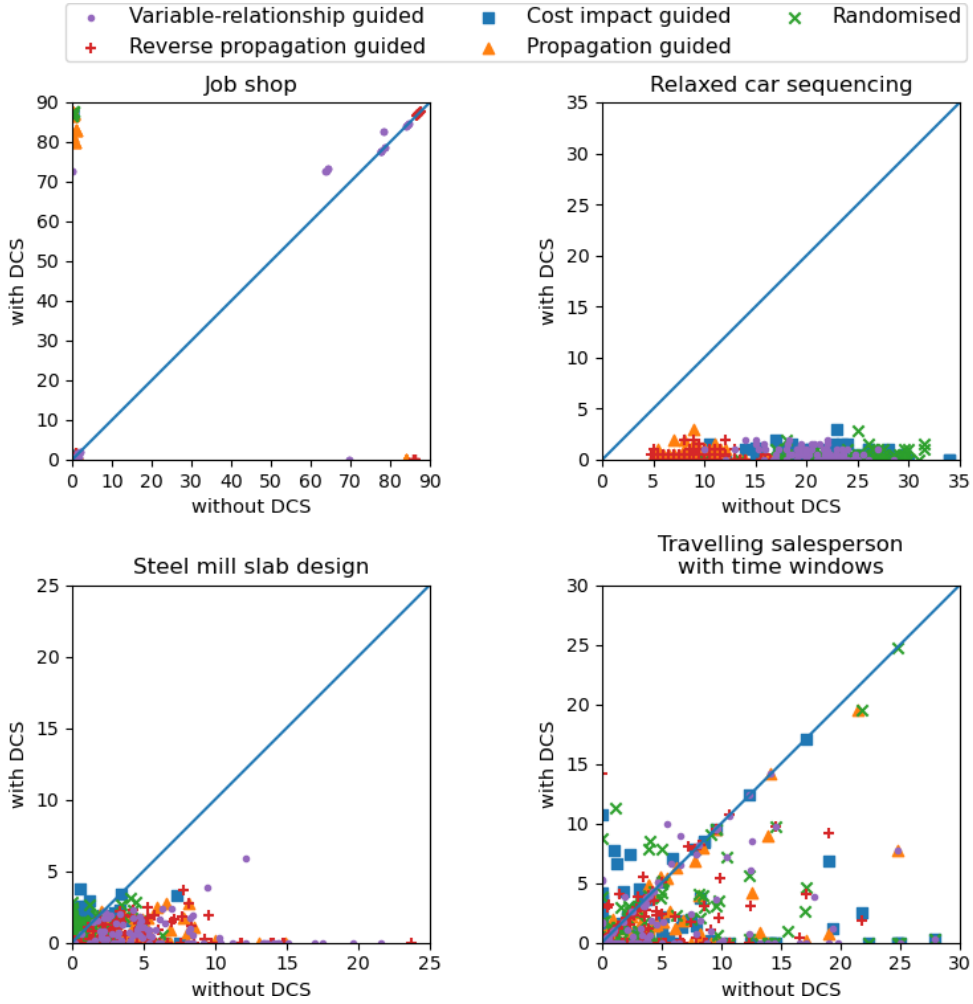

Figure 4 The scores for 72 JSP instances, 79 RCS instances, 80 SMSD instances, and 112 TSPTW instances, each over 10 independent runs, both with and without the dependency curation scheme (DCS), for VRG-LNS, RPG-LNS, CIG-LNS, PG-LNS, and randomised LNS. Each marker is the mean score over 10 independent runs for a selection heuristic, where a low score corresponds to a high-quality solution. Each marker over the diagonal line corresponds to DCS worsening performance. Conversely, each marker under the diagonal line corresponds to DCS improving performance.

performance when using DCS. Our experiments show that the performance of using DCS with the naïve randomised selection heuristic is competitive with the more elaborate state-of-the-art generic selection heuristics, even when they use DCS.

In future work, we intend to implement for our extension of the Gecode-based portfolio solver [11] our dependency curation scheme, so that the solver works without the modeller needing to provide (a generator that finds) a set of search variables. Once our scheme is implemented, we intend to compare the performance of the generic selection heuristics, both with and without DCS, on all optimisation problems in the MiniZinc [17] benchmark repository, to verify the results of this paper and to compare our extended Gecode-based portfolio solver with other state-of-the-art optimisation solvers. Additionally, we intend to extend the (non-portfolio) Gecode [6] CP solver by implementing our scheme and a naïve randomised generic LNS selection heuristic, thereby allowing a modeller to easily use competitive LNS for any MiniZinc constrained optimisation model.

Also, for VRG-LNS, a variable is selected for exclusion from the freeze set if it shares a constraint with a previously excluded variable x. When used with DCS, each variable y that does not share a constraint with x but that transitively functionally defines a variable that does so should be eligible for selection. Our idea is to make this modification to VRG-LNS and empirically test if it improves performance when used with DCS.

Finally, we intend to create a generic branching heuristic for CP-style tree search, where the variables that are selected to be branched over must be a subset of the low-cardinality set of search variables found by DCS.

References

1 H. Murat Afsar, Christian Artigues, Eric Bourreau, and Safia Kedad-Sidhoum. Machine reassignment problem: The ROADEF/EURO Challenge 2012. *Annals of Operations Research*, 242:1–17, 2016. doi:10.1007/s10479-016-2203-7.

2 Russell Bent and Pascal Van Hentenryck. A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. *Computers and Operations Research*, 33(1):875–893, January 2006. doi:10.1016/j.cor.2004.08.001.

3 Gustav Björdal, Pierre Flener, and Justin Pearson. Generating compound moves in local search by hybridisation with complete search. In Louis-Martin Rousseau and Kostas Stergiou, editors, *CP-AI-OR 2019*, volume 11494 of *LNCS*, pages 95–111. Springer, 2019. doi:10.1007/978-3-030-19212-9_7.

4 Gustav Björdal, Jean-Noël Monette, Pierre Flener, and Justin Pearson. A constraint-based local search backend for MiniZinc. *Constraints*, 20(3):325–345, July 2015. doi:10.1007/s10601-015-9184-z.

5 Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In Yves Kodratoff, editor, *ECAI 1988*, pages 290–295. Pitman, 1988.

6 Gecode Team. Gecode: A generic constraint development environment, 2019. The Gecode solver and its MiniZinc backend are available at https://www.gecode.org.

7 Michel Gendreau, Alain Hertz, Gilbert Laporte, and Mihnea Stan. A generalized insertion heuristic for the traveling salesman problem with time windows. *Operations Research*, 46(3):330–335, 1998. doi:10.1287/opre.46.3.330.

8 Daniel Godard, Philippe Laborie, and Wim Nuijten. Randomized large neighborhood search for cumulative scheduling. In Susanne Biundo, Karen L. Myers, and Kanna Rajan, editors, *ICAPS 2005*, pages 81–89. AAAI Press, 2005. URL: http://www.aaai.org/Library/ICAPS/2005/icaps05-009.php.

9 Jayant Kalagnanam, Milind Dawande, Mark Trumbo, and Ho Lee. Inventory matching problems in the steel industry. *IBM Research Report, Computer Science/Mathematics*, April 1999.

10 Frej Knutar Lewander, Pierre Flener, and Justin Pearson. Invariant graph propagation in constraint-based local search. *Journal of Artificial Intelligence Research*, 2025. Forthcoming.

11 Dexter Leander. Building portfolio search in Gecode for MiniZinc. Master's thesis, Department of Information Technology, Uppsala University, Sweden, September 2024. Available at https://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-533041, with the code at https://github.com/DLeander/gecode-dexter.

12 Michele Lombardi and Pierre Schaus. Cost impact guided LNS. In Helmut Simonis, editor, *CP-AI-OR 2014*, volume 8451 of *LNCS*, pages 293–300. Springer, 2014. doi:10.1007/978-3-319-07046-9_21.

13 Toni Mancini and Marco Cadoli. Exploiting functional dependencies in declarative problem specifications. *Artificial Intelligence*, 171(16–17):985–1010, November 2007. doi:10.1016/j.artint.2007.04.017.

14 Laurent Michel, Pierre Schaus, and Pascal Van Hentenryck. MiniCP: A lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021. The source code and teaching materials are available at `http://minicp.org`. `doi:10.1007/s12532-020-00190-7`.

15 Laurent Michel and Pascal Van Hentenryck. Localizer: A modeling language for local search. In Gert Smolka, editor, *CP 1997*, volume 1330 of *LNCS*, pages 237–251. Springer, 1997. `doi:10.1007/BFb0017443`.

16 Laurent Michel and Pascal Van Hentenryck. Localizer. *Constraints*, 5(1–2):43–84, 2000. `doi:10.1023/A:1009818401322`.

17 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessière, editor, *CP 2007*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007. The MiniZinc toolchain is available at `https://www.minizinc.org`. `doi:10.1007/978-3-540-74970-7_38`.

18 Laurent Perron, Paul Shaw, and Vincent Furnon. Propagation guided large neighborhood search. In Mark Wallace, editor, *CP 2004*, volume 3258 of *LNCS*, pages 468–481. Springer, 2004. `doi:10.1007/978-3-540-30201-8_35`.

19 David Pisinger and Stefan Ropke. Large neighborhood search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, volume 272 of *ORMS*, chapter 4, pages 99–127. Springer, 2019. `doi:10.1007/978-3-319-91086-4_4`.

20 Charles Prud'homme, Xavier Lorca, and Narendra Jussien. Explanation-based large neighborhood search. *Constraints*, 19(4):339–379, October 2014. `doi:10.1007/s10601-014-9166-6`.

21 Pierre Schaus. Variable objective large neighborhood search: A practical approach to solve over-constrained problems. In Alexander Brodsky, editor, *ICTAI 2013*, pages 971–978. IEEE Computer Society, 2013. `doi:10.1109/ICTAI.2013.147`.

22 Pierre Schaus, Pascal Van Hentenryck, Jean-Noël Monette, Carleton Coffrin, Laurent Michel, and Yves Deville. Solving steel mill slab problems with constraint-based techniques: CP, LNS, and CBLS. *Constraints*, 16(2):125–147, April 2011. `doi:10.1007/s10601-010-9100-5`.

23 Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In Michael Maher and Jean-François Puget, editors, *CP 1998*, volume 1520 of *LNCS*, pages 417–431. Springer, 1998. `doi:10.1007/3-540-49481-2_30`.

24 Filipe Souza, Diarmuid Grimes, and Barry O'Sullivan. An investigation of generic approaches to large neighbourhood search. In Paul Shaw, editor, *CP 2024*, volume 307 of *LIPIcs*, pages 39:1–39:10. Dagstuhl Publishing, 2024. `doi:10.4230/LIPIcs.CP.2024.39`.

25 Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972. `doi:10.1137/0201010`.

26 Charles Thomas and Pierre Schaus. Revisiting the self-adaptive large neighborhood search. In Willem-Jan van Hoeve, editor, *CP-AI-OR 2018*, volume 10848 of *LNCS*, pages 557–566. Springer, 2018. `doi:10.1007/978-3-319-93031-2_40`.

27 Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

28 Gerhard Wäscher and Thomas Gau. Heuristics for the integer one-dimensional cutting stock problem: A computational study. *OR Spektrum*, 18:131–144, 1996. `doi:10.1007/BF01539705`.