Automatic Relaxation and Multi-Armed Bandit Learning for Large Neighbourhood Search

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

Pierre Flener
Uppsala University, Department of Information Technology, Uppsala, Sweden

Justin Pearson
Uppsala University, Department of Information Technology, Uppsala, Sweden

Peter J. Stuckey
Monash University, Melbourne, Australia
OPTIMA ARC Industrial Training and Transformation Centre, Melbourne, Australia

Abstract

Inspired by concepts of constraint-based local search, we present a novel scheme for automatically relaxing a given high-level model into an optimisation model that is better suited for large neighbourhood search (LNS). By exploiting the variable sharing and semantics of the constraints in a model, our scheme (1) identifies constraints that can easily be satisfied simultaneously and can thus constrain the neighbourhood, and (2) relaxes the remaining constraints. As a side effect, our scheme enables the LNS solving of a constraint satisfaction problem, by transforming it into an optimisation problem, and the faster solving of a difficult-to-satisfy constrained optimisation problem, by finding the initial incumbent faster. This scheme can be used with any CP-based LNS solver. We tested a portfolio of CP-based LNS variants running in parallel, with a multi-armed bandit to select which LNS variant to run. Our results show that this approach is very competitive.

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.2026.35

Supplementary Material *Software (Solver)*: `https://github.com/astra-uu-se/gecode-lns` *Software (Results, Experiments, and Scheme Generator)*: `https://github.com/astra-uu-se/gecode-lns-experiments`, archived at `swh:1:dir:d78bdce218c11727ee85927791160d7e313c7df8`

Funding Supported by grants 2018-04813 and 2025-039039 of the Swedish Research Council (VR), and the Australian Research Council OPTIMA ITTC, Project ID IC200100009. The first author visited the fourth author on stipends by Anna Maria Lundin's Scholarship Committee (Smålands Nation) and by the Swedish Operations Research Association (SOAF).

Acknowledgements We thank Guido Tack for his help with the Gecode implementation.

1 Introduction

Given a problem model in a high-level modelling language, such as MiniZinc [24], we want to specialise it automatically into a model suitable for solving with large neighbourhood search (LNS) [32, 26] or for an LNS asset of an information-sharing portfolio of solvers. Note that local search, such as by LNS, has modelling concepts, such as neighbourhood, that are not available in such technology-independent modelling languages, so that a good such specialisation is far from trivial.

[1] Corresponding author

LNS hybridises local search (that is a resource-bounded iteration of attempts to improve an incumbent solution) by using systematic search, such as by a constraint programming (CP) solver, at every iteration for probing the neighbourhood. LNS has been shown to exhibit better scalability than systematic search for a wide variety of problems, such as vehicle routing [32, 3], bin packing [36], and scheduling [13, 33].

However, LNS is only defined for a constrained optimisation problem (COP) and thus requires prior model specialisation for a constraint satisfaction problem (CSP). For example, a relaxation allows some of the CSP constraints to be violated, with the objective of minimising their total violation, ideally down to zero. Also, finding the first incumbent is sometimes difficult for a COP, even though it suffices to satisfy its constraints and ignore its objective function, so that the start of the LNS iterations is delayed. One can avoid that by, for example, performing some prior relaxation. Relaxation is an art, and excellent problem-specific relaxations have been manually identified [25, 27, 33, 15], but our aim is automatic relaxation.

In this paper, we adapt and extend ideas from constraint-based local search (CBLS) [35], where automatic model relaxation and the fast finding of the first incumbent are well-studied [6].

Our contributions are as follows, when given a problem model in a high-level modelling language:

- designing a scheme that exploits the variable sharing and semantics of the constraints in order to identify constraints that should not be relaxed (Section 3.1);
- describing how our scheme can be used to automatically relax for LNS a given model (Section 3.2); and
- showing that extending a state-of-the-art LNS solver with our scheme and a multi-armed bandit (Section 4) improves performance (Section 5).

We relax the constraints in the same way as in [5], but we identify suitable constraints not manually, but automatically by further exploiting the graph of [15], induced by the variables and constraints of the given model.

The rest of the paper is organised as follows. In Section 2, we present existing concepts and techniques that will be useful in Section 3 to describe our novel scheme for identifying constraints that should not be relaxed for a CSP or COP, and how a high-level model can be automatically relaxed for an LNS solver. In Section 4, we describe our LNS solver. In Section 5, we describe our experimental setup and present the results from the experiments. Finally, we conclude in Section 6.

2 Toolkit for Model Relaxation for Large Neighbourhood Search

We discuss existing concepts and techniques that we will use in Section 3. In Section 2.1, we explain LNS in more detail. In Section 2.2, we recall the useful concept of non-failing propagators. In Section 2.3, we discuss the functional dependencies encoded by some constraints, called dependency constraints. In Section 2.4, we describe how to soften a constraint c into a dependency constraint that functionally defines a new variable denoting how violated c is. In Section 2.5, we show how the dependency constraints induce a directed graph, called the dependency graph. Finally, in Section 2.6, we relate all these concepts to their historic origins within the CBLS context.

2.1 Large Neighbourhood Search

Given a model L for a COP and a global computation resource (such as running time or iteration count), large neighbourhood search (LNS) first attempts to solve the CSP part of L in order to find a solution, which becomes the first incumbent if one is found before the global resource is exhausted. This is usually done by using a solving technology that performs systematic search, such as CP or mixed-integer programming.

LNS then iteratively attempts to improve the incumbent, if any was found, until either the global resource is exhausted or, rarely, optimality is trivially proven by the incumbent reaching a known bound on the objective function. At every LNS iteration i, the incumbent is partially destroyed, thus creating the neighbourhood of solutions that are repairs to the destroyed incumbent. Instead of probing the neighbours one by one, as in classical local search, LNS probes them collectively by attempting to solve a model L_i under systematic search before an allocated iteration-level resource (such as running time or failure count) is exhausted. For example, one can destroy the incumbent and design the repair model L_i by copying L and assigning some of its variables to their values in the incumbent (this is also known as *freezing*), with the remaining variables retaining their domains of L (this is also known as *thawing*), and adding the betterness constraint that the solution be no worse than the incumbent. Thawing is usually done for more variables than for classical local search, which typically changes the values of only 1 to 4 variables, so that the neighbourhood is quite large and hence the name LNS. If the systematic-search solver finds a solution to L_i before exhausting its allocated resource, then that solution becomes the new incumbent, else the incumbent is unchanged.

If, as in Section 1, a CSP or COP model M in a high-level modelling language has been specialised (for example, by relaxation) into a COP model L for LNS, then the constraints that are unchanged between M and L are called *neighbourhood constraints*, as they are enforced at every LNS iteration for probing the neighbourhood.

2.2 Propagators and Non-Failing Propagators

During the inference phases of CP-style solving, the *propagator* for each constraint c attempts to prune the current domains of its variables according to the semantics of c. If that pruning empties the domain of some variable, then the propagator *fails*: the search backtracks if there is an unexplored choice point and fails otherwise, as the problem is then unsatisfiable.

For example, consider the constraint $x \leq y$ and the current domains $\mathcal{D}_x = \{0, 2, 3, 5, 7\}$ and $\mathcal{D}_y = \{0, 1, 4, 6\}$. Its propagator should prune 7 from $\mathcal{D}_x$, because $7 \not\leq \max(\mathcal{D}_y)$, resulting in $\mathcal{D}_x = \{0, 2, 3, 5\}$. If search or the propagator of some other constraint prunes the even values from both domains, resulting in $\mathcal{D}_x = \{3, 5\}$ and $\mathcal{D}_y = \{0, 1\}$, then the propagator should empty $\mathcal{D}_x$ or $\mathcal{D}_y$ (or both), because $\min(\mathcal{D}_x) \not\leq \max(\mathcal{D}_y)$, and then fail.

A propagator can be made *non-failing* [5]: it prunes the current domains of its variables, but *only* until it would empty the domain of some variable; at that point, instead of emptying that domain and failing, it is withdrawn from the subsequent inference phases, but restored on backtracking. It is at the discretion of the implementer to decide how many values to prune before the propagator is withdrawn. In the previous example, instead of emptying $\mathcal{D}_x$ or $\mathcal{D}_y$ (or both), a non-failing propagator must leave at least one value in both $\mathcal{D}_x$ and $\mathcal{D}_y$ before being withdrawn. There is no need to design non-failing propagators from scratch for every constraint: under minimal conditions on the inference engine, every existing propagator can be made non-failing upon changing just a few lines of the code of the inference engine [5].

2.3 Dependency Constraints

A *dependency constraint* c on the variables $\mathcal{I} \cup \mathcal{O}$ fixes the variables in $\mathcal{O}$, called its *output variables*, when the variables in $\mathcal{I}$, called its *input variables*, become fixed. We say that c *functionally defines* $\mathcal{O}$ *from* $\mathcal{I}$.

For example, consider the integer variables y and i, the array $\mathcal{P}$ of (fixed) integers, and the constraint $\textsc{Element}(\mathcal{P}, i, y)$, which requires $\mathcal{P}[i] = y$. When i becomes fixed to some value v, the value of y becomes fixed to the integer in $\mathcal{P}$ at index v. This is a dependency constraint that functionally defines y from i. Consider now the constraint $x \leq y$ over the integer variables x and y. This is not a dependency constraint, as it functionally defines neither x nor y from its other variable.

Depending on the initial (or current) domains of its variables, every constraint can be (or become) a dependency constraint. For example, $\textsc{AllDifferent}(\mathcal{X})$ functionally defines each variable $x \in \mathcal{X}$ from its other variables $\mathcal{X} \setminus \{x\}$ when they all have the same domain of size $|\mathcal{X}|$.

2.4 Constraint Softening

A constraint c can be reformulated as a *soft constraint* $\check{c}$, where $\check{c}$ is a dependency constraint that functionally defines from the variables of c a new integer variable, called a *violation variable*. The latter takes value 0 when c is satisfied, otherwise a positive value, usually proportional to the amount of violation of c. We say that we *soften* c into $\check{c}$, and that $\check{c}$ is a *softening* of c.

For example, a softening of $x \leq y$ is $\max(0, x - y) = v$, which introduces and functionally defines the violation variable v as the difference $x - y$ between x and y when $x \not\leq y$. Similarly, a softening of $x = y$ is $|x - y| = v$, which introduces and functionally defines the violation variable v as the absolute difference between x and y. One can also soften any constraint γ by what is called *reification* into $1 - [\gamma] = v$, where $[\gamma]$ is 1 when γ holds and 0 otherwise, so that the violation variable v is 0 (for satisfaction) or 1 (for violation).

Each time we soften a constraint by introducing a violation variable v_i, the objective of the model must be changed. A model for a CSP becomes a model for a COP, where the objective is the minimisation of the sum $\sum_i v_i$, called the *total violation*. A model for a COP with objective variable o, which is to be minimised (without loss of generality), remains a model for a COP, but its objective becomes the lexicographic minimisation of the pair $\langle \sum_i v_i,\ o \rangle$. Whenever we mention that we soften a constraint, we consider this transformation of the objective to be included. A *solution* has $\sum_i v_i = 0$, so that it represents a solution to the original model. A *pseudo-solution* [5] has $\sum_i v_i > 0$. Whenever we want to emphasise that the word "solution" does not refer to a pseudo-solution, we write *non-pseudo-solution*.

2.5 Dependency Graphs

The dependency constraints induce a not necessarily unique directed graph, called a *dependency graph*, where all variables and dependency constraints are the nodes, and there are arcs to each dependency constraint from its input variables, as well as arcs from each dependency constraint to its output variables [15]. A *source* (respectively *sink*) of a dependency graph is a node with no incoming (respectively no outgoing) arcs and is necessarily a variable, as each dependency constraint has one or more incoming arcs and one or more outgoing arcs: we speak of *source variables* and *sink variables*.

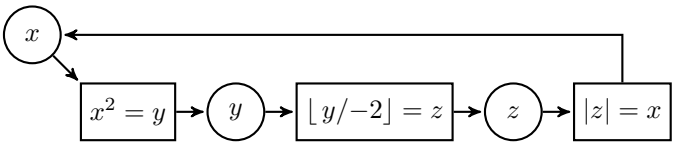

Figure 1 A cyclic dependency graph.

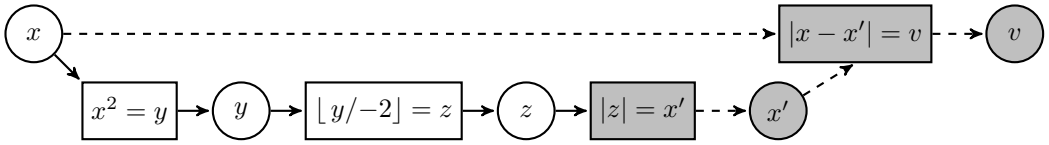

Figure 2 An acyclic dependency graph corresponding to the cyclic one in Figure 1, where the variables x' and v are added, the output variable x of constraint $|z| = x$ is replaced by x', and the dependency constraint $|x - x'| = v$ is added. The white nodes and solid arcs are the same as in Figure 1, while the grey nodes and dashed arcs are not in Figure 1.

A dependency graph can be cyclic. For example, consider the integer variables x, y, and z and the constraints $x^2 = y$, $\lfloor y/-2 \rfloor = z$, and $|z| = x$. Since each constraint defines the variable on its right-hand side, but not vice versa, the dependency graph has three dependencies and there is one cycle, namely $x \to x^2 = y \to y \to \lfloor y/-2 \rfloor = z \to z \to |z| = x \to x$, and zero source and sink variables: see Figure 1.

Any cyclic dependency graph can be transformed as follows into an equivalent acyclic one, but with additional violation variables [6]. As long as the dependency graph is cyclic, pick a dependency constraint c, with output variables $\mathcal{O}$, that is in some cycle with variables $\mathcal{C}$, replace an output variable $x \in \mathcal{O} \cap \mathcal{C}$ by a new output variable x' of c, and add to the model (and hence the graph) a softening of $x = x'$, say $|x - x'| = v$, which introduces and functionally defines the violation variable v. For example, the cyclic dependency graph in Figure 1 can be transformed into the acyclic one in Figure 2.

2.6 Constraint-Based Local Search

For a constraint-based local search (CBLS) solver [21, 22, 35, 16], each constraint of the given high-level model must be formulated by either an implicit constraint or some invariants.

An *implicit constraint* is the CBLS counterpart of the LNS concept of neighbourhood constraint, except that it is *not* in a CBLS model, but is rather to be satisfied by (1) the algorithmic construction of the initial incumbent, and (2) the algorithmic enumeration of neighbours to the current solution. In other words, it constrains the neighbourhood but is not enforced by the inference engine of a CBLS solver, which only processes invariants.

An *invariant*, also known as a *one-way constraint*, is the CBLS counterpart of a dependency constraint. Each constraint that is not algorithmically made implicit either is already a dependency constraint or is softened into a dependency constraint and thus becomes an invariant. The inference engine of a CBLS solver propagates variable updates through the *invariant graph*, which is the CBLS counterpart of the dependency graph. The search engine of a CBLS solver aims at minimising the sum of the violation variables introduced by having softened some constraints.

It is through the study of CBLS solvers that we were inspired to lift CBLS concepts into the LNS context, such as the cycle elimination in Section 2.5.

3 Model Relaxation for Large Neighbourhood Search

We now describe how an LNS model can be relaxed automatically from a given high-level model of a CSP or COP. There are problem-specific relaxations [25, 27, 33, 15], but we here aim at a relaxation scheme that can be automated. In Section 3.1, we show how to identify, by exploiting the dependency graph of the given model, a set of constraints that are easy to satisfy simultaneously and thus should not be relaxed. In Section 3.2, we relax the given model into an LNS model by eliminating all cycles of its dependency graph and relaxing all its constraints that are neither dependency constraints nor the constraints found by the scheme in Section 3.1.

Note that in Section 2.3 we allowed a constraint to be a dependency constraint depending on the initial (or current) domains of its variables. However, we consider only dependency constraints whose syntax or semantics contain a dependency, regardless of the domains of their variables. For the models that we have encountered, this restriction has not been detrimental to the automatic relaxation of high-level models into LNS models.

3.1 Identification of Neighbourhood Constraints

Consider the dependency graph of the given high-level model of a CSP or COP. Most dependency constraints are by definition easy to satisfy and should thus not be relaxed for the LNS model. Among its non-dependency constraints, the ones that involve at least one non-source variable are poor candidates for relaxation for the LNS model because we will build on the competitive LNS scheme of only freezing some of the source variables [15], so that all non-source variables are fixed by CP-style solving during neighbourhood probing of the LNS model. This leaves us with the non-dependency constraints that only involve source variables: some of them may be easy to satisfy simultaneously, and should thus *not* be relaxed for the LNS model. They are neighbourhood constraints, as they are present during every neighbourhood probing.

As a motivating example, consider the travelling salesperson problem (TSP), where a minimal-duration Hamiltonian tour of n locations is to be found. A TSP model has an n by n matrix Δ of non-negative integers, with $\Delta[i, j]$ denoting the travel duration from location i to location j; the variables $\mathcal{L} = \{\ell_1, \ldots, \ell_n\}$, with ℓ_i denoting the i^{th} location in the tour, which starts at ℓ_1 and ends at $\ell_{n+1} = \ell_1$; the variables $\mathcal{D} = \{d_1, \ldots, d_n\}$, with d_i denoting the travel duration from ℓ_i to ℓ_{i+1}; the objective variable o, denoting the total travel duration; the non-dependency constraint AllDifferent($\mathcal{L}$), enforcing that each location is visited exactly once, using propagator α; the n dependency constraints $\Delta[\ell_i, \ell_{i+1}] = d_i$, each defining a d_i; and the dependency constraint $\sum_i d_i = o$, defining o, which is to be minimised. The dependency graph has $\mathcal{L}$ as source variables, variable o as sink variable, and $\mathcal{D}$ as non-source, non-sink variables. If CP-style solving is performed by branching on the variables in $\mathcal{L}$, then α never fails if it always prunes all impossible domain values. Therefore, finding an initial incumbent that can be used for LNS iterations can be done here without backtracking. There is thus no need to relax the non-dependency constraint AllDifferent($\mathcal{L}$) for the LNS model.

In general now, we show that a set of non-dependency constraints that need not be relaxed for the LNS model can be found automatically by exploiting the dependency graph *before* solving starts.

Let scope(c) be the set of variables of constraint c. Inspired by impact-based search [28], we chose an integer, denoted rank(p), for each available propagator p of each constraint, so that rank(p_a) $\geq$ rank(p_b) means that propagator p_a of constraint a is expected to prune at

Algorithm 1 Identification of neighbourhood constraints.

Data: A set $\mathcal{C}$ of non-dependency constraints.
Result: An ideally max-total-priority subset of $\mathcal{C}$ with constraints of disjoint scopes.

```
1  function neighbourhood-constraints(C):
2      N := ∅ % The set of neighbourhood constraints
3      V := ∅ % The set of variables covered by N
4      while C ≠ ∅ do
5          n := arg lex max_{c∈C} (⟨rank(c), |scope(c)|⟩)
6          C := C \ {n}
7          if scope(n) ∩ V = ∅ then
8              N := N ∪ {n}
9              V := V ∪ scope(n)
10     return N
```

least as many domain values than propagator p_b of constraint b. This ranks propagators for the same constraint ($a = b$, starting from the same scope and domains) and propagators for different constraints ($a \neq b$, where a has a more complex semantics than b, say `cumulative` is more complex than `circuit`, which is more complex than `all_different`), and thus induces a total order on propagators.

Let $\mathcal{S}$ be the set of source variables of the dependency graph for the given high-level model. We call a *set of neighbourhood constraints* a set $\mathcal{N}$ of constraints that have scopes within $\mathcal{S}$ and can be easily satisfied simultaneously, under a criterion given below. Our idea is that the higher-ranked the constraints of $\mathcal{N}$ and the larger the total scope for $\mathcal{N}$, the more CP-style propagation will occur. So we let the *priority* of a constraint c be the pair $\langle \text{rank}(c), |\text{scope}(c)| \rangle$ and we aim at maximising the total priority of the constraints of $\mathcal{N}$.

A simple criterion for easy simultaneous satisfaction of a constraint set $\mathcal{N}$ is that its constraints have disjoint scopes. This implies that they can be satisfied without backtracking if all their propagators prune all impossible values. A more elaborate criterion would be that $\mathcal{N}$ induces a Berge-acyclic hypergraph [2], where the scopes can overlap under some conditions and yet the pruning of all impossible values yields backtrack-free satisfaction. We have not yet encountered a model where this would yield a better set of neighbourhood constraints, so we only design an algorithm for the simple criterion.

The greedy randomised Algorithm 1 is fed the set of non-dependency constraints that constrain only source variables of the dependency graph, and returns an ideally max-total-priority set of neighbourhood constraints. First, the set $\mathcal{N}$ of neighbourhood constraints and the set $\mathcal{V}$ of variables covered by $\mathcal{N}$ are both initialised to the empty set, lines 2 and 3. While there are unvisited constraints in $\mathcal{C}$, line 4, a constraint c with the lexicographically highest priority is found, under random tie breaking, line 5. Constraint c is removed from $\mathcal{C}$, line 6. If the variables of c are disjoint from $\mathcal{V}$, line 7, then c is added to $\mathcal{N}$, line 8, and its variables are added to $\mathcal{V}$, line 9. Finally, the found set $\mathcal{N}$ is returned, line 10. The algorithm is greedy and randomised in line 5, potentially making the set $\mathcal{N}$ *not* of the maximum total priority. For models that we have encountered, Algorithm 1 finds a set of neighbourhood constraints with high ranks and where all search variables are covered.

Let us run Algorithm 1 on a slightly more complex example than the motivating example above (for which it indeed identifies AllDifferent($\mathcal{L}$) as a neighbourhood constraint). Consider the magic square problem, where all the integers 1 until n^2 are to be placed into an n

Listing 1 A MiniZinc model for the magic square problem.

```
1 int: n; % width (and height) of the magic square
2 int: magicSum = (n^3+n) div 2; % sum of each row, column, major diagonal
3 array[1..n,1..n] of var 1..n*n: Magic; % magic square
4 constraint all_different(Magic); % distinct values
5 constraint forall(r in 1..n)(sum(Magic[r,..]) = magicSum); %   row sums
6 constraint forall(c in 1..n)(sum(Magic[..,c]) = magicSum); %   col sums
7 constraint sum([Magic[    i,i] | i in 1..n])  = magicSum;  % diag1 sum
8 constraint sum([Magic[n+1-i,i] | i in 1..n])  = magicSum;  % diag2 sum
9 solve satisfy;
```

by n matrix such that the integers on each row, column, and major diagonal sum to $(n^3+n)\div 2$, called the *magic sum*. A magic square for $n = 3$ is $[[2,7,6],[9,5,1],[4,3,8]]$, with 15 as magic sum. A model for the magic square problem in the solver-independent MiniZinc language is in Listing 1. Line 1 declares the parameter `n`. Line 2 computes the magic sum. Line 3 declares the array `Magic` of `n` by `n` variables, where `Magic[r,c]` denotes the integer in row `r` and column `c` of the magic square. The constraint in line 4 requires that the integers in the array are distinct. The constraints in lines 5 to 8 require that the integers in each row, column, and major diagonal sum to the magic sum. Under the assumptions of Section 2.3, the dependency graph contains no dependency constraints and only the variables `Magic[r,c]`, which are thus all source variables: all the constraints are candidates for becoming neighbourhood constraints. Assume that we have an `all_different` propagator that prunes all impossible values, and a linear-equality propagator that only tightens the lower and upper bounds of all domains (as pruning all impossible values is NP-hard), and is thus lower-ranked. Algorithm 1 starts from the initially empty sets of neighbourhood constraints and their covered variables. The first constraint to be visited is `all_different(Magic)`, because it has the strictly highest rank and thus the strictly highest priority (and incidentally the largest scope). It is transferred from the set of unvisited constraints to the set of neighbourhood constraints, and its variables are added to the set of covered variables. Each remaining constraint is visited, but not made a neighbourhood constraint, because all variables are covered by the already identified neighbourhood constraint `all_different(Magic)`.

3.2 Automatic Relaxation for LNS of a High-Level Model

We now give a novel scheme for automatically relaxing into an LNS model L a given high-level model M of a CSP or COP. The scheme handles as a side effect the potential difficulty of satisfying the CSP part of a COP during LNS initialisation.

All neighbourhood constraints identified within M using Algorithm 1 are added to L. The dependency constraints of M, if need be upon transformation so that their dependency graph becomes acyclic, are added to L, but each dependency constraint that gets a new output variable under that transformation (such as $|z| = x$ from Figure 1 to Figure 2) is also added to L, with a non-failing propagator, so as to help prune additional values during propagation. Indeed, cycles tend to be difficult to satisfy, especially if there are other constraints on the variables in a cycle.

Each remaining constraint $c(\vec{x})$ of M is *relaxed* in the following two-fold way [5]: (1) we add $c(\vec{x})$ to L with a non-failing propagator until a solution is found and a normal propagator thereafter, *and* (2) we add a softening $\check{c}(\vec{x}, v)$ of c to L with a normal propagator. Indeed,

satisfying only $c(\vec{x})$ in L, with a normal propagator, would tend to be difficult. Part (2) simplifies this by allowing $c(\vec{x})$ to be violated, and its shared variables $\vec{x}$ with part (1) drive the satisfaction attempt into a good direction.

For example, reconsider the magic square problem, with only the `all_different(Magic)` constraint on line 4 of Listing 1 identified for the set of neighbourhood constraints of L. Each remaining constraint is not a dependency constraint, so it is relaxed as defined above.

This model relaxation scheme differs from the existing scheme in [5] in that we *automatically* identify neighbourhood constraints (and relax the remaining ones), instead of relaxing *manually* annotated constraints (and letting the remaining ones be neighbourhood constraints), and that we eliminate cycles in the dependency graph.

4 Multi-Armed Bandit LNS

We only consider incumbent destruction by thawing (recall Section 2.1), because that is where almost all work on generic choices for LNS is done. The set of thawed variables is the complement of the set of frozen variables, called the *freeze set*, which is suggested by what we call the *freeze heuristic*.

LNS approaches can involve multiple choices of how to choose which freeze heuristic to use next. Adaptive LNS [29] is a common approach that, given problem-specific freeze heuristics, uses the heuristic that has recently been the most successful in improving the incumbent. Self-adaptive LNS [34] is inspired by adaptive LNS and uses generic freeze heuristics, but does not exploit the dependency graph and does not use multi-armed bandits. Another related approach is the Balans framework [8], which employs a multi-armed bandit to select generic freeze heuristics for MIP.

In Section 4.1, we point to existing freeze heuristics. In Section 4.2, we recall an existing scheme of exploiting the dependency graph in order to identify a good set of variables within which to select the freeze set. In Section 4.3, we explore a multi-armed bandit approach to choosing the best freeze heuristic to try next.

4.1 LNS Freeze Heuristics

A freeze heuristic, also known as a *neighbourhood heuristic* [9], is either problem-specific or generic. By construction, problem-specific freeze heuristics, such as [13, 31], cannot easily be reused between problems. Fortunately, there are many generic freeze heuristics that can be easily automated and yield good search performance, such as those of propagation-guided LNS (PG-LNS) [25], reverse propagation-guided LNS (RPG-LNS) [25], reinforced adaptive LNS (RA-LNS) [19], cost-impact-guided LNS (CIG-LNS) [18], explanation-based LNS (EB-LNS) [27], and variable-relationship-guided LNS (VRG-LNS) [33]. Variable-objective LNS (VO-LNS) [30] also has a generic freeze heuristic, but it requires the objective variable to be the sum of other variables. Finally, folklore has what we here call Random-LNS, namely the freezing of a random subset of all variables, each variable being frozen under some probability p (with typically $p \approx 80\%$). In our implementation, p is dynamically updated during search, ranging from 5% to 95% [15].

4.2 The Dependency Curation Scheme for LNS

In generic freeze heuristics, most variables can be in the freeze set. However, the source variables of the dependency graph transitively functionally define all the remaining variables, so the latter should all be thawed and a subset of the source variables should be frozen, so

that the thawing of the remaining source variables drives an LNS iteration [15]. This insight is the *dependency curation scheme* (DCS), which is by itself not a freeze heuristic but can be imposed on every freeze heuristic, typically improving the performance of LNS [15].

4.3 LNS Configurations and a Multi-Armed Bandit

We have two LNS hyperparameters, forming what we call an *LNS configuration*: (1) the LNS freeze heuristic, and (2) whether DCS is used (because it does not always pay off [15]). Since the best LNS configuration is unknown and may differ between LNS iterations of the same run, good LNS configurations need to be found during search. This is where we use the multi-armed bandit (MAB) problem [4]: a bandit has multiple arms, each arm yielding an unknown numerical reward when pulled, and the problem is to find the arm that currently yields the greatest reward.

We model the choice of an LNS configuration as an MAB problem as follows: each arm of the MAB corresponds to an LNS configuration and the yielded *reward* from pulling an arm is the number of non-worsening (pseudo-)solutions found during a time interval (which is 5 seconds in our implementation). When selecting an arm, we use as probability distribution the normalised exponential function (often called softmax [7]) on the yielded rewards of the arms.

Consider a high-level model M. Our solver is explained in Algorithm 2. First, the relaxed model L is created (as in Section 3.2), line 2. Then, a CP solver is used on the CSP part of L in order to find the first incumbent (pseudo-)solution I, line 3. The set $\mathcal{K}$ is created, where each tuple $\langle f, d\rangle \in \mathcal{K}$ is an LNS configuration, where f is the freeze heuristic and d is whether DCS is to be used, line 4. The multi-armed bandit b is created, with an arm for each LNS configuration in $\mathcal{K}$, line 5, The reward of each arm of b is initialised to 0, line 6. While time remains and I is not a solution, line 7, first an arm a is selected from b, line 8; then LNS iterations are performed on L, starting from I and using a for an interval of 5 seconds, potentially updating I into a better incumbent, and returning the number r of non-worsening (pseudo-)solutions found during the time interval, line 9; and finally arm a of b yields reward r, line 10. If time remains, M is for a COP and the incumbent I now is a non-pseudo-solution, line 11, then the arms of b are reset by resetting all rewards to 0, line 12, and while time remains and I is not trivially optimal, line 13, LNS iterations are run again like in lines 8 to 10, but now on M [5], since the incumbent is now a solution and all future incumbents necessarily are solutions, lines 14 to 16. The final incumbent I is returned, line 17.

Our implementation is actually parallelised in lines 8 to 10 and lines 14 to 16, having multiple threads that perform simultaneous LNS, all using the same MAB and sharing a set (of maximum cardinality 3 in our implementation) of equally good incumbents, where if the set is at capacity when an equally good incumbent is found, then the oldest incumbent is replaced. However, the threads differ as different LNS configurations might be selected for them by the MAB. Additionally, we also have an asset that performs CP-style search on M, sharing any solutions found with the LNS assets.

5 Experiments

In Section 5.1, we give the details of our extensions to the solver that was used for the experiments. In Section 5.2, we describe the solvers that our solver is compared against. In Section 5.3, we specify the problems that we have selected. In Section 5.4, we present and discuss the experiment results.

Algorithm 2 Solver for multi-armed bandit LNS.

Data: A high-level model M, a set $\mathcal{F}$ of freeze heuristics, and a timeout t.
Result: The best (pseudo-)solution to M found within timeout t.

```
function bandit-LNS(M, F, t):
    L := relax(M)
    I := satisfy(L) % the incumbent (pseudo-)solution
    K := {⟨f, d⟩ | f ∈ F, d ∈ {false, true}} % the LNS configurations
    b := create-mab(K) % create an MAB with K as arms
    initialise-rewards(b) % set the reward of each arm to 0
    while ¬timeout(t) ∧ total-violation(I) > 0 do
        a := select-arm(b) % arm a = ⟨f_a, d_a⟩ is an LNS configuration a ∈ K
        r := run-LNS-iterations(L, I, a, 5s) % perform LNS with a 5 second
          timeout, potentially updating I as a side effect
        give-reward(b, a, r) % the yielded reward from pulling a is r
    if ¬timeout(t) ∧ is-COP(M) ∧ total-violation(I) = 0 then
        initialise-rewards(b) % reset the reward of each arm to 0
        while ¬timeout(t) ∧ ¬is-trivially-optimal(I) do
            a := select-arm(b)
            r := run-LNS-iterations(M, I, a, 5s)
            give-reward(b, a, r)
    return I
```

5.1 Solver Setup

We extended the Gecode-based information-sharing portfolio solver in [15] – introducing DCS and containing as assets the CP solver Gecode [11], PG-LNS [25], RPG-LNS [25], VO-LNS [30] (when applicable), CIG-LNS [18], VRG-LNS [33], and Random-LNS [15] – by adding the non-failing propagators and constraint relaxation from [5], using the described neighbourhood identification scheme, implementing the MAB that chooses LNS configurations, as well as making it solution-sharing, sharing the best and up to two additional equally good (pseudo-)solutions among the portfolio assets. We decided not to add any freeze heuristics not in [15], such as RA-LNS [19] and EB-LNS [27], which were not in the underlying Gecode-Dexter [17] either. We here call the resulting solver Gecode-depLNS-MAB.[2] For all three solvers and each LNS thread, we use a static failure count of 3000 as the iteration-level resource.

5.2 Comparison Study

As an ablation study, we compare with Gecode-depLNS, which is Gecode-depLNS-MAB with the MAB dropped. Gecode-depLNS performs CP-style search using one thread, and LNS on the remaining seven, using the following LNS configurations: PG-LNS with DCS, RPG-LNS with DCS, CIG-LNS with DCS, VRG-LNS with DCS, Random-LNS with DCS, PG-LNS

[2] The source code to our solver is at `https://github.com/astra-uu-se/gecode-lns/tree/feature/self-subsuming` and models, instances, and experiment logs are at `https://github.com/astra-uu-se/gecode-lns-experiments/tree/non-failing`.

without DCS, and RPG-LNS without DCS. We do not compare here with a dropping of the exploitation of the dependency graph, because all LNS assets would then be idle until the first incumbent non-pseudo-solution is found by the CP asset (since it would not benefit from any relaxation). Furthermore, we compare with the parallelised solver Gecode-Par, which makes use of both Random-LNS (run on one thread) and parallelised CP-style search (run on the remaining threads).[3] Gecode-Par is quite competitive (but not eligible for medals) at the annual MiniZinc Challenge.

We are in this paper not interested in where Gecode-depLNS-MAB beats other existing solvers or finds new best-known results: by limiting our experiments to all Gecode-based portfolio solvers, we have a common baseline and can demonstrate the viability of our ideas from that baseline. Otherwise, it would be unknown if differences in performance are due to our ideas or to other ideas or improvements in the solvers.

5.3 Problem Selection

We selected six problems: car sequencing (CS), nurse rostering (NR), rotating workforce rostering (RWR), the job shop problem (JSP), steel mill slab design (SMSD), and the travelling salesperson problem with time windows (TSPTW). CS was selected because handmade relaxations for LNS were used in [25, 5, 33, 15], but we here relax it automatically. NR and RWR were selected because their models have multiple constraints only on source variables that have overlapping scopes. JSP was selected because its instances are typically easy to satisfy and since it is used in [18, 15]. SMSD is a widely used problem that has been tackled with LNS [18, 5, 15]. Finally, TSPTW was selected because the dependency graph induced from its model is cyclic and finding (non-pseudo-)solutions is typically difficult.

We used the models and instances in the solver-independent MiniZinc language [24] for CS, JSP, SMSD, and TSPTW from [15], but we used a CSP model for CS (rather than their relaxed COP model). We used the MiniZinc model and instances for NR and RWR from [5], but we removed their manually added `soften` annotations, since suitable neighbourhood constraints can automatically be identified (see Section 3.1).

Car sequencing (CS) [10] is a CSP with a set of car classes, each class with a set of mandatory features, each feature with a capacity on how many cars of that feature can be produced in a subsequence of a 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. The neighbourhood constraint is the `global_cardinality_closed` constraint that enforces that all ordered cars are produced.

The *nurse rostering* (NR) problem of [5] is a CSP with a set of nurses, a period of days, and the work shifts "day", "night", and "off". The problem is to schedule the nurses such that each nurse is assigned a shift each day, is assigned an "off" shift in each four-day period, and is assigned at most two "night" shifts in a row. The neighbourhood constraints are the `regular` constraints that enforce that the schedule for each nurse is valid.

Rotating workforce rostering (RWR) [23] is a CSP that aims to schedule workers satisfying shift sequence constraints and ensuring enough shifts are covered on each day, where every worker completes the same schedule, just starting at different days in the schedule. The neighbourhood constraint is the `regular` constraint that enforces that the schedule is valid.

[3] Gecode-Par is at `https://github.com/guidotack/gecode/tree/multi_asset_flatzinc`.

The *job shop problem* [20] is a COP and has a set $\mathcal{M}$ of machines and a set $\mathcal{J}$ of jobs, where each job $j \in \mathcal{J}$ is a sequence of $|\mathcal{M}|$ tasks, each with a discrete duration and requiring a distinct machine in $\mathcal{M}$. For any job j, a task can only be started if the machine it requires is vacant and all previous tasks in the task sequence of j have been completed. A machine can only work on one task at a time. Once a machine has started working on a task, it is run until completion. The objective is to minimise the latest completion time of all tasks. The neighbourhood constraints are the `disjunctive` constraints that enforce that each machine only works on one task at a time.

Steel mill slab design [14] is a COP with a set of orders, each with a size and a colour; a set of slabs, each taking one of a set of capacities; and a colour capacity c. The problem is to assign all orders to the slabs and assign to each slab a capacity from a set of capacities, such that for each slab, the total sizes of its orders do not exceed its capacity and the number of distinct colours among its orders does not exceed c. The objective is to minimise the total unused capacity, called *slack*, of the slabs, where unused slabs have zero unused capacity. Since the total slack is the sum of other variables, the freeze heuristic of VO-LNS is applicable to this problem, and Gecode-depLNS-MAB had it in its LNS configurations. For an SMSD instance, if there are identical orders, then the neighbourhood constraints are the `increasing` symmetry-breaking constraints; else there are no neighbourhood constraints.

The *travelling salesperson problem with time windows* (TSPTW) [12] has n locations that are to be visited, with a travel 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 route is a Hamiltonian path in the weighted directed graph induced by the n locations as nodes, 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 departure time at the last-visited location is to be minimised. The neighbourhood constraint is the `circuit` constraint that enforces the Hamiltonian path.

5.4 Experiment 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 24.04.4 LTS with GCC (the GNU Compiler Collection) 13.

For each solver – Gecode-Par, Gecode-depLNS, and Gecode-depLNS-MAB – and each problem instance, we made 10 independent runs, each under a time limit of 3 minutes and with 8 threads.

For the CSP models, we measure the time taken to solve an instance and use the scoring function $100 \cdot (o - b) \div w$ proposed in [1] and used in other LNS papers [33, 15]: for each solver s and instance, the mean objective value over the 10 independent runs by s is denoted by o, the best and worst found objective value over all (non-pseudo-)solutions of all runs by all solvers are denoted by b and w respectively. Therefore, the scores range over the closed continuous interval $[0, 100]$, where a low score is better than a high one. Note that the scoring function cannot be used for a CSP model, as it does not have an objective value and we only score (non-pseudo-)solutions.

The results are shown in Figures 3 and 4 and Table 1. Both Gecode-depLNS and Gecode-depLNS-MAB have overall better performance than Gecode-Par, as each finds better optima for COP instances and solves more CSP instances than Gecode-Par. For JSP, SMSD, and CS, Gecode-depLNS-MAB has overall better performance than Gecode-depLNS, as Gecode-depLNS-MAB finds better optima for JSP and SMSD instances and solves more CS instances. However, for TSPTW and RWR, Gecode-depLNS has overall better performance

Table 1 The mean scores for JSP, SMSD, and TSPTW instances of COPs, each over 10 independent runs, for Gecode-Par, Gecode-depLNS-MAB, and Gecode-depLNS. Boldface indicates the best performance on each row.

	Gecode-Par	Gecode-depLNS	Gecode-depLNS-MAB
JSP	9.94	11.30	**4.18**
SMSD	43.39	11.40	**11.08**
TSPTW	1.61	**0.39**	1.27

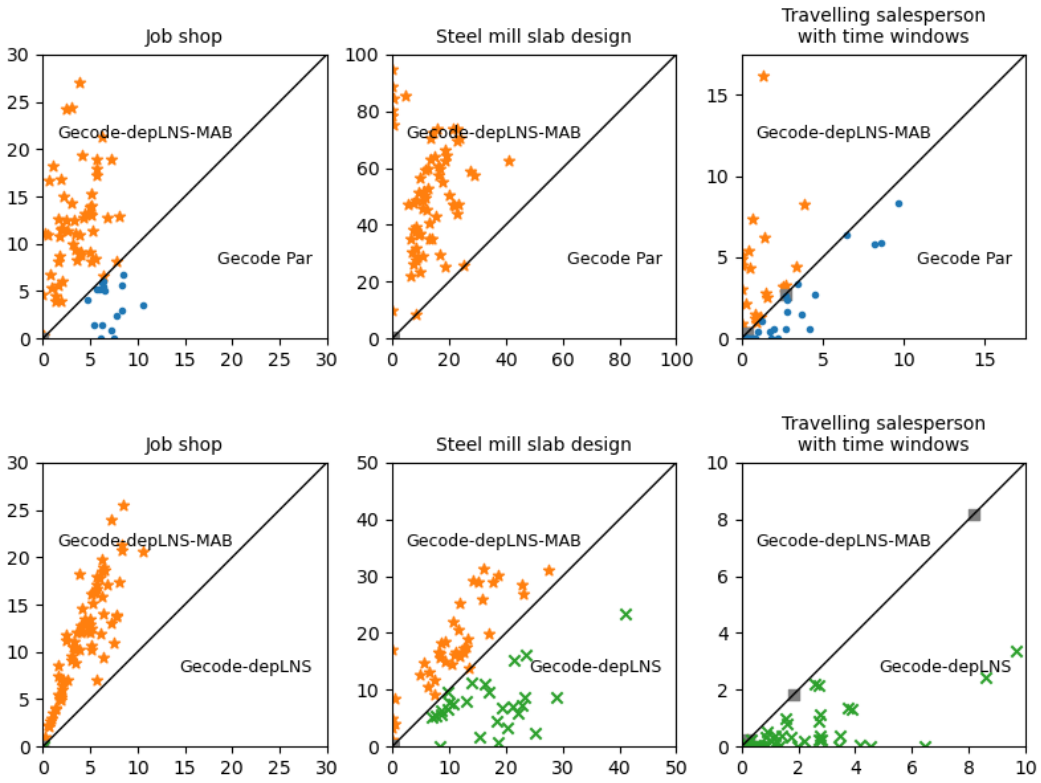

Figure 3 The scores for JSP, SMSD, and TSPTW instances of COPs for Gecode-Par, Gecode-LNS, and Gecode-depLNS-MAB. For each graph, the x axis is for the scores of Gecode-depLNS-MAB. For each of the top three graphs, the y axis is for the scores of Gecode-Par. For each of the bottom three graphs, the y axis is for the scores of Gecode-depLNS. Each marker denotes the mean score over 10 independent runs for Gecode-depLNS-MAB versus either Gecode-Par or Gecode-depLNS, where a low score corresponds to a better (non-pseudo-)solution than a high score. Each star over the diagonal corresponds to Gecode-depLNS-MAB outperforming the other solver. Each disk under the diagonal corresponds to Gecode-Par outperforming Gecode-depLNS-MAB. Each cross under the diagonal corresponds to Gecode-LNS outperforming Gecode-depLNS-MAB. Each square on the diagonal corresponds to the two compared solvers having equal performance.

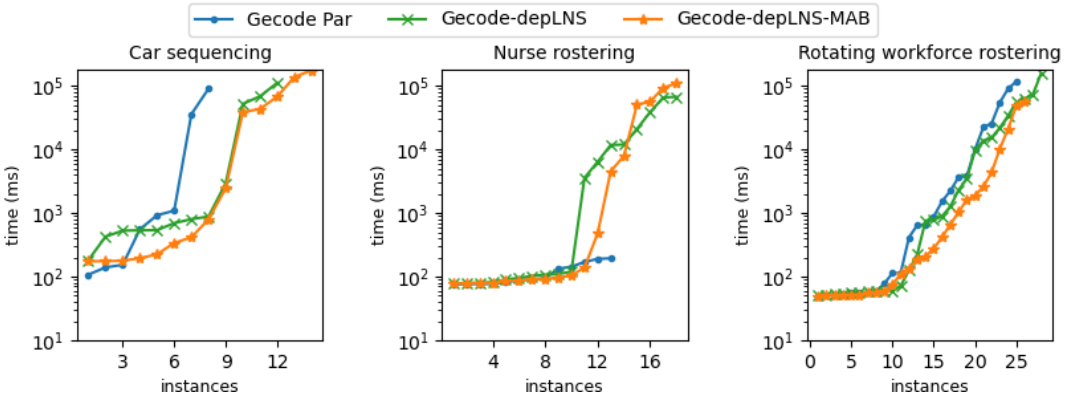

Figure 4 The results for CS, NR, and RWR instances of CSPs, each over 10 independent runs, with a timeout of 3 minutes, for Gecode-Par, Gecode-LNS, and Gecode-depLNS-MAB. For each graph, each marker gives the solution time, upon sorting (separately for each solver) the instances by increasing time.

than Gecode-depLNS-MAB, as Gecode-depLNS finds better optima for TSPTW instances and solves more RWR instances. For NR, Gecode-depLNS and Gecode-depLNS-MAB have comparable performance.

6 Conclusion

We presented a novel scheme that exploits the dependency graph and the semantics of constraints of a high-level model to automatically identify which constraints should be softened for an LNS model. We showed how our scheme can automatically relax a given high-level model into a relaxed optimisation model, which is better suited for LNS. For the experiments, we implemented our scheme and extended a Gecode-based solver with an MAB to find good freeze heuristics and whether DCS [15] should be used during search. The experiments show that our solver is competitive, as it outperforms the parallel Gecode-Par solver by solving additional CSP instances and finding better optima for COP instances compared to Gecode-Par. Additionally, our experiments show that if MABs are used, then performance is improved for some of the problems. The results indicate that our approach, whether MABs are used or not, can improve performance on CSP and COP instances.

In future work, we intend to try different variations of MABs, different reward strategies, and different LNS durations than 5 seconds. We also intend to try different strategies for setting the failure count, the iteration-level resource for LNS iterations, to improve the performance of our solver on CSPs. We also intend to implement softened constraints for all constraints in our solver, thereby allowing a modeller to easily use competitive LNS for any MiniZinc model. Once implemented, we intend to compare our performance on problems in the MiniZinc [24] benchmark repository, to verify our experimental results and to compare our solver against other state-of-the-art solvers.

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):1–17, 2016. doi:10.1007/s10479-016-2203-7.

2 Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, July 1983. doi:10.1145/2402.322389.

3 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.

4 Donald A. Berry and Bert Fristedt. *Bandit Problems: Sequential Allocation of Experiments*. Springer, 1985. doi:10.1007/978-94-015-3711-7.

5 Gustav Björdal, Pierre Flener, Justin Pearson, Peter J. Stuckey, and Guido Tack. Solving satisfaction problems using large-neighbourhood search. In Helmut Simonis, editor, *CP 2020*, volume 12333 of *LNCS*, pages 55–71. Springer, 2020. doi:10.1007/978-3-030-58475-7_4.

6 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.

7 John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Françoise Fogelman-Soulié and Jeanny Hérault, editors, *Neurocomputing: Algorithms, Architectures and Applications*, volume 68 of *NATO ASI Series*, pages 227–236. Springer, 1989. doi:10.1007/978-3-642-76153-9_28.

8 Junyang Cai, Serdar Kadıoğlu, and Bistra Dilkina. Balans: Multi-armed bandits-based adaptive large neighborhood search for mixed-integer programming problems. In James Kwok, editor, *IJCAI-25*, pages 2566–2574. International Joint Conferences on Artificial Intelligence Organization, 2025. doi:10.24963/ijcai.2025/286.

9 Tom Carchrae and J. Christopher Beck. Principles for the design of large neighborhood search. *Journal of Mathematical Modelling and Algorithms*, 8(3):245–270, 2009. doi:10.1007/s10852-008-9100-2.

10 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.

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

12 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.

13 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: https://aaai.org/papers/icaps-05-009.

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

15 Frej Knutar Lewander, Pierre Flener, and Justin Pearson. Dependency-curated large neighbourhood search. In Maria Garcia de la Banda, editor, *CP 2025*, volume 340 of *LIPIcs*, pages 20:1–20:17. Dagstuhl Publishing, 2025. doi:10.4230/LIPIcs.CP.2025.20.

16 Frej Knutar Lewander, Pierre Flener, and Justin Pearson. Invariant graph propagation in constraint-based local search. *Journal of Artificial Intelligence Research*, 83:10:1–10:42, August 2025. doi:10.1613/jair.1.17482.

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

18 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.

19 Jean-Baptiste Mairy, Yves Deville, and Pascal van Hentenryck. Reinforced adaptive large neighbourhood search. In *Doctoral Program of the 17th International Conference on Principles and Practice of Constraint Programming (DPCP'11)*, pages 55–60, 2011. URL: https://www.dmi.unipg.it/cp2011/downloads/dp2011/DP_at_CP2011.pdf.

20 Alan S. Manne. On the job-shop scheduling problem. *Operations Research*, 8(2):219–223, 1960.

21 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.

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

23 Nysret Musliu, Andreas Schutt, and Peter J. Stuckey. Solver independent rotating workforce scheduling. In Willem-Jan van Hoeve, editor, *CPAIOR 2018*, volume 10848 of *LNCS*, pages 429–445. Springer, 2018. doi:10.1007/978-3-319-93031-2_31.

24 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.

25 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.

26 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.

27 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.

28 Philippe Refalo. Impact-based search strategies for constraint programming. In Mark Wallace, editor, *CP 2004*, volume 3258 of *LNCS*, pages 557–571. Springer, 2004. doi:10.1007/978-3-540-30201-8_41.

29 Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006. doi:10.1287/trsc.1050.0135.

30 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.

31 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.

32 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.

33 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.

34 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.

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

36 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.