Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint

Adaptive large-neighbourhood search for optimisation in answer-set programming [☆]

Thomas Eiter [a], Tobias Geibinger [a], Nelson Higuera Ruiz [a], Nysret Musliu [a,b], Johannes Oetsch [c,*], Dave Pfliegler [a], Daria Stepanova [d]

[a] Institute for Logic and Computation, TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria
[b] CD-Lab Artis, TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria
[c] Department of Computing, Jönköping University, Gjuterigatan 5, 551 11 Jönköping, Sweden
[d] Bosch Center for AI, Robert Bosch Campus 1, 71272 Renningen, Germany

A B S T R A C T

Answer-set programming (ASP) is a prominent approach to declarative problem solving that is increasingly used to tackle challenging optimisation problems. We present an approach to leverage ASP optimisation by using large-neighbourhood search (LNS), which is a meta-heuristic where parts of a solution are iteratively destroyed and reconstructed in an attempt to improve an overall objective. In our LNS framework, neighbourhoods can be specified either declaratively as part of the ASP encoding or automatically generated by code. Furthermore, our framework is self-adaptive, i.e., it also incorporates portfolios for the LNS operators along with selection strategies to adjust search parameters on the fly. The implementation of our framework, the system ALASPO, currently supports the ASP solver clingo, as well as its extensions clingo-dl and clingcon that allow for difference and full integer constraints, respectively. It utilises multi-shot solving to efficiently realise the LNS loop and in this way avoids program regrounding. We describe our LNS framework for ASP as well as its implementation, discuss methodological aspects, and demonstrate the effectiveness of the adaptive LNS approach for ASP on different optimisation benchmarks, some of which are notoriously difficult, as well as real-world applications for shift planning, configuration of railway-safety systems, parallel machine scheduling, and test laboratory scheduling.

1. Introduction

Answer-set programming (ASP) [3–5] is a declarative problem solving paradigm with roots in logic-based knowledge representation and non-monotonic reasoning. Efficient solver technology and a simple modelling language have put ASP at the forefront of approaches to declarative problem solving with a growing number of applications in academia and industry [6,7]. To solve a problem with ASP, the search space and properties of problem solutions are described by means of a logic program such that its models, called *answer sets*, encode the problem's solutions. Many practical applications require optimisation of some objective function. This often is a challenge as making ASP encodings scale and perform well for the problem instances encountered can be tricky. While the performance of ASP can be improved by various means like manual or automatic tuning of solver parameters [8], adding domain-specific heuristics [9,10], or manual code rewriting to exploit symmetries or to achieve a smaller program grounding [5], these approaches often may require considerable time or expertise. While meta-heuristics that use incomplete methods like local search are popular in combinatorial problem solving, there are very few attempts to use them to improve ASP [11]. A problem that hinders local search

[☆] This article consolidates and extends preliminary conference papers on large-neighbourhood search for answer-set programming presented at AAAI 2022 [1] and KR 2022 [2].

[*] Corresponding author.
E-mail addresses: thomas.eiter@tuwien.ac.at (T. Eiter), tobias.geibinger@tuwien.ac.at (T. Geibinger), nelson.ruiz@tuwien.ac.at (N. Higuera Ruiz), nysret.musliu@tuwien.ac.at (N. Musliu), johannes.oetsch@ju.se (J. Oetsch), dave@pfliegler.at (D. Pfliegler), daria.stepanova@de.bosch.com (D. Stepanova).

https://doi.org/10.1016/j.artint.2024.104230
Received 6 July 2023; Received in revised form 13 September 2024; Accepted 17 September 2024

Available online 23 September 2024

methods for ASP is that its semantics is more "global" in nature than, e.g., the propositional satisfiability problem, where a local search step can be based on, e.g., the number of additional clauses that become satisfied.

Large Neighbourhood Search (LNS) [12,13] is a meta-heuristic that proceeds in iterations by successively destroying and reconstructing parts of a given solution with the goal to obtain better values for an objective function. For the reconstruction part, complete solvers can be used, and it is in fact common to effectively combine LNS with, e.g., mixed integer programming (MIP) [14,15] and constraint programming (CP) [12,16–18]. For ASP however, this potential is, by and large, untapped. Previous work [19] touched LNS using it with the solver `clingcon` for a solution of a specific problem. However, a principled and systematic use of LNS in ASP has not been explored. This is however of particular interest, as ASP is offering problem-solving capacities beyond other paradigms such as MIP and CP [20,21].

We believe that an LNS framework for ASP should fulfil the following desiderata:

(D1) ASP is well reputed for its effective declarative modelling capabilities. Setting up LNS for ASP requires to define neighbourhoods and this should fit, if possible, seamlessly into the overall ASP framework.
(D2) ASP solvers are in general easy to use for non-experts as they work out of the box without any need for configuration. Their performance can however often be improved with suitable parameter settings. A similar philosophy should be adopted for an LNS optimiser for ASP: the complexity of tuning parameters for the meta-heuristic should be hidden under the hood as far as possible to make the system attractive for ASP novices. More advanced users should be able to customise the system if desired.
(D3) For many optimisation problems, extensions of ASP like ASP with integer constraints are better suited, and respective systems should be supported in the context of LNS as well.
(D4) LNS creates a certain overhead for the ASP solver in terms of grounding and solving. This overhead should be kept minimal by exploiting current features of ASP solvers for an efficient implementation.

In this article, we tackle the above issues by introducing a novel *framework for adaptive LNS optimisation for answer-set solving* which makes LNS easily accessible to the ASP user. To address (D1), different neighbourhoods can be specified in a declarative way as part of the ASP encoding itself. To this end, dedicated predicates are used and no language extension is needed. As an alternative, we introduce default neighbourhoods that can be automatically generated by code. Inspired by *adaptive* variants of LNS search [22–24], neighbourhoods and search configurations can be bundled together into portfolios so that effective combinations can be learnt during search. Our framework provides strategies for adaptiveness from the literature as well as a new one that is specifically designed for ASP problems. Default portfolios together with adaptive selection strategies often give the user excellent performance without further configuration, thereby addressing (D2): support for ASP uses at varying degrees of expertise.

While the underlying ideas are generic, we present the system ALASPO (Adaptive Large-neighbourhood search for ASP Optimisation) that implements our framework for the state-of-the-art solver `clingo` [25,26] and its extension `clingo-dl` [27] for constraints in difference logic, as well as `clingcon` [28] for ASP with full integer constraints from the Potassco[1] family; hence we achieve (D3). To effectively explore different neighbourhoods, ALASPO utilises the solver features of *multi-shot solving* and *solving under assumptions* [25]. Multi-shot solving allows us to embed ASP in a more complex workflow that involves tight control over the solver's grounding and solving process. Learnt heuristics and constraints can be kept between solver calls, and repeated program grounding is effectively avoided. Solving under assumptions is a mechanism by which one can temporally fix parts of a solution between solver calls. This minimises the grounding and solving overhead in the LNS loop and addresses (D4).

We emphasise again that ALASPO features predefined neighbourhood definitions that can be used by an ASP novice out of the box without any further configuration, but it can also work with custom neighbourhoods provided by a more advanced user. The latter can be either operationally defined in code or, more conveniently, declaratively described as part of the ASP encoding. If no neighbourhood is specified, a default portfolio containing variants of random neighbourhoods is used, which turns out to be quite effective for many problems. Switching between operators from LNS portfolios can be done during search on the fly, but the system also provides full self-adaptive modes where no user intervention is required. Furthermore, the tool is self-adaptive and learns to use effective modes through different selection strategies.

We demonstrate ALASPO, which is publicly available under the MIT licence,[2] and show its effectiveness for different optimisation problems. In particular, we use the well-known ASP benchmark problems Social Golfer [29], Travelling Salesman Problem [30], generating clues for Sudoku [31], and an optimisation variant of the Strategic Companies problem [32]. Furthermore, we consider four real-world inspired problems from shift design [33], configuration of railway-safety systems [34], test laboratory scheduling [19], and parallel machine scheduling for semiconductor production [35]. The latter application represents an advanced showcase where neighbourhood definitions are easier to specify in an imperative language, and obtaining an initial solution from a construction heuristic instead of the ASP solver is beneficial. For this more customised search variant, we instantiate abstract Python classes that realise the basic LNS loop for a solver and implement an efficient construction heuristic to start the search. Throughout, LNS with ASP yields improved bounds compared to plain ASP on all benchmarks with no or little extra effort.

Main contributions Our main contributions can be summarised as follows:

[1] https://potassco.org/.
[2] https://gitlab.tuwien.ac.at/kbs/BAI/alaspo.

- We present an LNS framework for ASP optimisation that involves different mechanisms for defining neighbourhoods: declaratively, as part of the ASP encoding, or in code. An initial solution is obtained either via ASP or a (fast) construction heuristic.
- Our LNS framework is (self-)adaptive and supports portfolios of LNS operators with different selection strategies to change parameters during search. These strategies comprise one inspired by the literature which uses reinforcement learning [23,36], but also a novel strategy that is designed for ASP problems and dynamically adjusts parameters to avoid timeouts.
- We present the system ALASPO which implements our LNS approach for the solvers `clingo`, `clingo-dl`, and `clingcon`. It is highly customisable for advanced users but also works out of the box with effective default settings suitable for ASP novices.
- We provide a discussion on methodological aspects that details how to set up LNS for ASP in a systematic way and how to avoid common pitfalls. Existing LNS literature commonly deals with the meta-heuristic itself or describes it for particular problems. Our discussion covers however the entire process from beginning to end from a user's perspective and thereby complements LNS literature that is more sparse regarding such practically important general guidelines.
- We experimentally evaluate LNS for ASP optimisation and its adaptive features on different optimisation problems including both artificial benchmarks and real-world applications.

In summary, we introduce a novel and comprehensive framework for ASP optimisation with adaptive LNS. We cover on the one hand formal foundations, and, on the other hand, we introduce the ALASPO system which implements our LNS framework for ASP. Furthermore, we demonstrate the versatility of ALASPO and hence of our approach on several challenging applications. Our contribution is relevant for ASP users who seek for performance improvements in problem solving using ASP tools, while avoiding to get into lower-level solver tuning and programming. In fact, it leverages ASP optimisation capabilities and therefore expands the range of problems for which ASP can be fruitfully applied. Notably, it does so while allowing to stay within the declarative ASP paradigm when setting up the meta-heuristic. Furthermore, we provide the user with a methodology for how to proceed with the LNS approach in problem solving. While the discussion of methodological aspects is formulated in the context of ASP, it is relevant beyond as it contains also guidelines for using LNS in general.

Previous work and novelty This article consolidates and extends two previous conference contributions on the same topic: "Large-Neighbourhood Search for Optimisation in Answer-Set Solving" (AAAI 2022) [1] and "ALASPO: An Adaptive Large-Neighbourhood ASP Optimiser" (KR 2022) [2]. The former paper introduces the basic LNS framework for ASP and experimentally evaluates it, while the latter extends the LNS framework with strategies to make the algorithm adaptive.

This article extends the aforementioned papers in several regards, namely by considerably more thorough and detailed discussions of technical aspects, by further applications and experiments, and by addressing usability issues of the approach. In particular, it includes (i) a further application of the approach to a real-world test laboratory scheduling problem that involves ASP with integer constraints and thereby demonstrates LNS search with ALASPO in combination with the constraint-ASP solver `clingcon` (previous work only involved `clingo` and `clingo-dl`), (ii) a further application where we tackle the partner units problem, a challenging and well-researched configuration problem with important industrial applications, and (iii) a detailed section on methodological aspects of the approach as discussed above.

Organisation The remainder of this article is organised as follows. First, we present background on ASP optimisation and adaptive LNS in Section 2. Then, we describe our framework for LNS with ASP in Section 3. Details on the LNS optimiser ALASPO are given in Section 4, and methodological aspects on how to use LNS with ASP are discussed in Section 5. Afterwards, we show how to solve different optimisation benchmarks with LNS and `clingo` in Section 6 and present more advanced applications with `clingo`, `clingo-dl`, and `clingcon` in Section 7. Related work is discussed in Section 8, and we conclude in Section 9.

2. Background

We next provide some background on ASP and LNS.

2.1. Answer-set programming

Answer-set programming (ASP) [4,5,3] is a declarative problem solving paradigm, where a problem is encoded as a logic program such that its *answer sets* (which are special models) correspond to the solutions of the problem and are computable using ASP solvers, e.g., `clingo` from potassco.org or `DLV` from www.dlvsystem.com. We focus in this work on the multi-shot solver `clingo` and its extensions for theories [25,26,28,27]. For a thorough introduction to the modelling languages, we refer to the language standard [37] as well as a respective user guide.[3]

An *ASP program* is a finite set P of *rules* r of the form

$$a_1 \mid \dots \mid a_k \;\; :- \;\; b_1, \dots,\ b_m,\ not\ c_1, \dots,\ not\ c_n, \quad k, m, n \geq 0 \tag{1}$$

where all a_i, b_j, c_l are first-order atoms and *not* is *default negation*; we denote by $H(r) = \{a_1, \dots, a_k\}$ and $B(r) = B^+(r) \cup \{not\ c_j \mid c_j \in B^-(r)\}$ the head and body of r, respectively, where $B^+(r) = \{b_1, \dots, b_m\}$ and $B^-(r) = \{c_1, \dots, c_n\}$. Intuitively, r says that if all atoms

[3] https://github.com/potassco/guide/releases/.

```
 1  player(1..g*p).
 2  group(1..g).
 3  week(1..w).
 4
 5  { plays(P,W,G): group(G)  } = 1 :- player(P), week(W).
 6  { plays(P,W,G): player(P) } = p :- week(W), group(G).
 7
 8  meets(P1,P2,W) :- plays(P1,W,G), plays(P2,W,G), P1 < P2.
 9  :~ #count { W: meets(P1,P2,W) } > 1,
10     player(P1), player(P2), P1 < P2. [1,P1]
11
12  #show plays/3.
```

Fig. 1. Encoding for the Social Golfer Problem.

in $B^+(r)$ are true and there is no evidence that some atom in $B^-(r)$ is true, then some atom in $H(r)$ must be true. We assume that rules are *safe*, i.e., each variable that occurs in a rule r also occurs in $B^+(r)$. If $m = n = 0$ and $k = 1$, then r is a *fact* (with $:-$ omitted); if $k = 0$, r is a *constraint*.

An *Herbrand interpretation* is a set $I \subseteq HB_P$ of ground (i.e., variable-free) atoms that effectively specifies which atoms are true under I, where HB_P is the Herbrand base of program P, i.e., the set of all ground (variable-free) atoms that can be formed with predicate and function symbols as well as constants of P. More specifically, I satisfies a ground atom a, denoted $I \vDash a$, if $a \in I$; I satisfies a negated ground atom *not* a, denoted $I \vDash \textit{not}\ a$, if $a \notin I$. Furthermore, I satisfies a set $L = \{\ell_1, \dots, \ell_n\}$ of possibly negated atoms ℓ_i if $I \vDash \ell_i$ for each $\ell_i \in L$, and I satisfies a ground rule r if either $H(r) \cap I \neq \emptyset$ or $I \nvDash B(r)$. Then, I is a model of a ground program P if I satisfies each $r \in P$. An interpretation I is an *answer set* of P, if it is a subset-minimal model of the Gelfond-Lifschitz reduct of P wrt. I [38], which is given by $P^I = \{H(r) :- B^+(r) \mid r \in P, I \vDash B^-(r)\}$.[4]

Models and answer sets of a program P with variables are defined in terms of the *grounding* of P, denoted $grd(P)$, which consists of all possible ground instances of the rules in P over the Herbrand universe, i.e., all variables in $r \in P$ are substituted with ground terms built using the constant and function symbols of P.

We will also use *choice rules* and *weak constraints*, which are of the respective forms

$$i\,\{a_1; \dots; a_k\}\,j :- b_1, \dots, b_m,\ \textit{not}\ c_1, \dots,\ \textit{not}\ c_n \tag{2}$$

$$:\sim b_1, \dots, b_m,\ \textit{not}\ c_1, \dots,\ \textit{not}\ c_n.\ [w, t] \tag{3}$$

Informally, (2) says that when the body is satisfied, at least i and at most j atoms from $\{a_1, \dots, a_k\}$ must be true in an answer set I, while (3) contributes tuple t with costs w, which is an integer number, to a cost function $cost(I)$, when the body is satisfied in I, rather than to eliminate I; the answer set I is optimal, if the total cost of all such tuples, i.e., $cost(I) = \sum_w \{(w, t) \mid :\sim B(r).[w, t] \in P, I \vDash B(r)\}$, is minimal.

Example 1 *(Social Golfer Problem (SGP)).* As an example for optimisation with clingo, consider the *Social Golfer Problem* (SGP) [29]: the task is to schedule $g \times p$ golfers in g groups of p players for w weeks such that no two golfers play in the same group more than once. An instance of the SGP is denoted by the triple g-p-w. We want to minimise the number of players that meet more than once.

An ASP encoding for SGP in the modelling language of clingo is given in Fig. 1. A problem instance g-p-w is defined in lines 1–3, where we use consecutive numbers to denote the players, groups, and weeks, respectively. The search space of feasible schedules is defined by rules 5 and 6: the former states (reading from right to left) that, for any player P and for any week W, the number of groups player P is assigned to in week W is one. In other words: every player plays in every week in precisely one group. Rule 6 ensures that the size of any group in any week is precisely p. Rule 8 derives meets(P1,P2,W) if P1 and P2 meet in group G in week W. Line 9 is a weak (soft) constraint to give a penalty of 1 for any player P1 who meets another player P2 more than once. The last line is a solver directive to output only atoms over predicate plays/3. Atoms mentioned by this #show directive are the program's *visible atoms*.

Theory solving is a feature of clingo that allows extending the formalism by external theories like integer constraints in the style of SMT [26]. Using integer constraints can immensely help to avoid a large ground program as the integer constants no longer directly contribute to its size. The solver clingo-dl extends clingo by difference constraints which are expressions of form $u - v \leq d$, where u and v are integer variables and d is an integer constant. They can be used in an encoding in the form of theory atoms &diff{u-v}<=d. In contrast to systems of unrestricted integer constraints, systems of difference constraints are solvable in polynomial time [39].

A number of recent ASP applications feature difference constraints for problems that involve timing constraints [35,40–42]. For unrestricted integer constraints, clingcon [28] or other constraint ASP systems [43,44] can be used.

[4] Equivalently, instead of P^I the FLP reduct $fP^I = \{r \in P \mid I \vDash B(r)\}$ may be used.

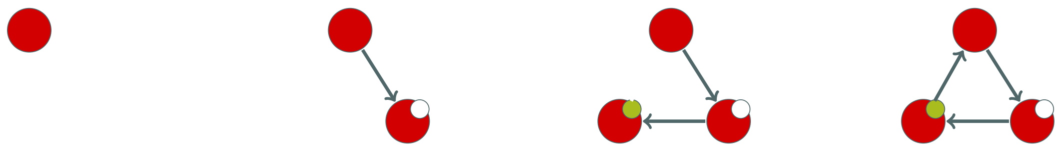

(a) Initial solution. (b) Relax. (c) Restore. (d) Repeat.

Fig. 2. Schematic visualisation of Large-Neighbourhood Search.

Algorithm 1 LNS optimisation for a minimisation problem.

```
1: $s^* \leftarrow$ feasible solution
2: repeat
3:     $s' \leftarrow search(relax(s^*))$
4:     $\Delta c \leftarrow cost(s^*) - cost(s')$
5:     if $\Delta c > 0$ then
6:         $s^* \leftarrow s'$
7:     end if
8: until stop criterion met
9: return $s^*$
```

The solver `clingo` supports hierarchical optimisation criteria and uses a range of model-guided methods [45] as well as core-guided techniques [46] that work by identifying and relaxing sets of unsatisfiable weak constraints until a solution is found. While `clingcon` also supports optimisation statements for integer variables, this is not the case for `clingo-dl`, where only minimisation of a single integer variable is directly supported by iteratively adding a constraint to enforce a smaller value on the integer variable.

2.2. Large-neighbourhood search

Large-Neighbourhood Search (LNS) is a meta-heuristic optimisation method introduced by Shaw [12]. Whereas it was originally designed for a vehicle routing problem, the underlying concept was quickly adopted as a problem-independent technique [13]. Intuitively, LNS aims at gradually improving a solution iteratively through alternating a destruction and a recreation phase.

A visualisation of the basic LNS procedure is given in Fig. 2. In general, LNS starts with a feasible initial solution. Then a part of that solution is “destroyed” or “relaxed”, for example, by unassigning a specified percentage of all decision variables. Afterwards the partially relaxed solution is “repaired” or “restored”, where the parts of the solution that were not relaxed stay the same. If the solution obtained by this repair is better than the previous one, it is adopted as the new incumbent solution, which will be relaxed in the next iteration. In general, LNS is an incomplete search method, i.e., at no step is it known whether the incumbent solution is optimal and the procedure is repeated until a stop criterion, e.g., a global time limit, is met. The term “large neighbourhood” stems from the fact that the destroy operation generally relaxes a big part of the incumbent solution generating a large number of possible repairs. It is thus usually not feasible to traverse every possible restoration and often LNS relies on an exact method, like CP or MIP, to repair the solution.

The pseudo-code of a simple LNS implementation is given in Algorithm 1. The procedure $search(\cdot)$ restores the partial solution obtained through applying $relax(\cdot)$ to the incumbent solution. If the cost improves, we change the incumbent to the new solution.

Often, there are different destroy and repair operations inside an LNS implementation. For instance, one could consider different sizes for the parts of the solution that will be relaxed or different restoration methods. An example for the latter would be to consider multiple time limits for the exact method used to repair or different construction heuristics. *Adaptive LNS* [22–24] expands on this by incorporating a portfolio of destroy and repair operators. At each iteration of the LNS procedure, the algorithm then decides which operators to use for this step. This decision is generally influenced by the past performance of the operators thus resulting in a selection strategy which adapts to the current instance. Some adaptive LNS implementations feature general problem-independent operators and the strategy thus aims to adapt to both the problem and the specific instance [24].

3. An LNS framework for ASP

We next define the relevant terminology and constituents for LNS in the ASP context.

Definition 1 *(Solution).* Let P be an ASP program that includes optimisation statements. A *solution* for P is an answer set I of P with associated costs $cost(I)$. The set of all solutions of P is denoted by $sol(P)$.

An LNS search operator ($search(\cdot)$ in Algorithm 1) takes a partial solution and maps it into a complete solution. It is formally defined as follows:

Definition 2 *(LNS search operator).* Let P be a program and I be a solution of P. An *LNS search operator* is a function $f_I : 2^I \rightarrow sol(P)$ that maps each set $I' \subseteq I$ of atoms to some solution $J = f_I(I)$ of P such that $I' \subseteq J$.

Algorithm 2 LNS with multi-shot solving and assumptions.

Input: ASP program P and input facts I
Parameter: global time limit t, neighbourhood timeout t^*

```
1: $c \leftarrow$ initialise clingo based solver
2: $c.ground(P \cup I)$
3: $s \leftarrow getInitialSolution(P \cup I)$
4: $c.addBound(cost(s) - 1)$
5: repeat
6:     $s' \leftarrow c.solve(t^*, getMoveAssumptions(s))$
7:     if SAT then
8:         $s = s'$
9:         $c.addBound(cost(s) - 1)$
10:    end if
11: until time passed $> t$
12: return $s$
```

We will use ASP solvers to realise LNS search operators.

The purpose of $relax(\cdot)$ from Algorithm 1 is to remove parts of a complete solution. We realise it by selecting, uniformly and at random, a partial solution from the neighbourhood of a given complete solution. The neighbourhood of a solution is a set of partial solutions obtained by applying an LNS neighbourhood operator to a complete solution.

Definition 3 *(LNS neighbourhood operator).* Let P be a program. An *LNS neighbourhood operator* $ln : sol(P) \rightarrow 2^{2^{HB_P}}$ maps each solution I of P to a subset $ln(I)$ of the power set of I. We refer to $ln(I)$ as the *neighbourhood* for I.

Intuitively, LNS neighbourhood operators in ASP define ways to remove atoms from an answer set. For example, relaxing a solution by removing 20% of all atoms is achieved by a neighbourhood that contains, for any answer set I, a collection of all sets $I' \subseteq I$ that contain 20% less atoms compared to I.

We will refer to both LNS search and LNS neighbourhood operators simply as *LNS operators*, and we will also omit "LNS" if clear from the context.

3.1. The LNS loop

In the LNS loop, an incumbent solution is repeatedly relaxed and reconstructed by an ASP solver to continuously obtain better objective values for the optimisation problem at hand.

An *initial solution* is generated by the ASP solver as either the first solution without optimisation, or after letting the solver *pre-optimise* the problem for a specified amount of time, i.e., run the solver in optimisation mode for a specified time before LNS takes over. Pre-optimisation is useful if the ASP solver is already good at finding optimal or near-optimal solutions for many instances. Alternatively, an initial solution can be obtained by a custom procedure using a *construction heuristic*, e.g., a fast greedy procedure implemented in Python. If such a heuristic is available and worth the extra effort is however problem specific.

In each iteration of the LNS loop, the currently best solution I is relaxed using an LNS neighbourhood operator. For instance, it could pick and relax a certain percentage of the visible atoms at random. Then, the resulting partial solution is reconstructed using an LNS search operator realised with an ASP solver. This reconstruction step depends on a search configuration S which defines solver options and a time limit. If a better solution is found within the time limit, it becomes the new incumbent solution; otherwise (i.e., the solver runs into a timeout or yields unsatisfiability), the old solution I remains the best known one.

So far, our discussion of LNS with ASP has been *solver agnostic*. Relaxing a solution can always be implemented by temporarily adding a constraint `:- not a`, for every atom `a` in a solution that we do not want to relax, to the program at hand. Searching for a better solution can be realised by adding a constraint that enforces a strictly better solution than the current one. However, if the used ASP solver supports *multi-shot solving* [25], the LNS heuristic can be implemented more efficiently.

Multi-shot solving allows us to ground an encoding only once and then explore neighbourhoods in subsequent solver calls with potentially further constraints added to enforce better solutions. Besides avoiding the overhead of repeated grounding, we can keep learnt heuristics and constraints.

If the ASP solver used supports *solving under assumptions* [25], this feature can be used to realise the LNS search operator more efficiently. Assumptions temporarily fix truth values of atoms in a solver call without the need to add explicit constraints. Between solver calls, we fix all atoms that are part of the solution that is not relaxed.

Our implementation of this framework for LNS (more details will follow in the next section) indeed relies on multi-shot solving and solving under assumptions. The way in which those features are used to implement LNS is shown in Algorithm 2. The given ASP program is first grounded in line 2. Afterwards, we obtain an initial solution in the next line. This initial solution is generated with the specified solver. By default, it is the first solution found. Alternatively, pre-optimisation or a construction heuristic (as discussed above) is used. Overriding *getInitialSolution* provides the ability to specify the initial solution used. Hence, if the ASP solver struggles with finding an initial solution, it is a good idea to use (if available) a fast construction heuristic to start the search.

After an initial solution was obtained, a bound is given to the ASP solver to enforce that the next solution has strictly better cost.[5] There are two ways in which this bound can be specified for solvers from the Potassco family: either by adding a hard constraint that enforces a better solution and using plain search or by using the ASP solver's optimisation features and providing an improved bound `b` as an initial value (option `--opt-mode=opt,b`), which has the same effect and also forces the next solution to be strictly better. The latter is preferred, since we can use advanced strategies like core-guided optimisation. The first option is needed if the solver type does not support this, e.g., `clingo-dl` if we want to directly minimise an integer variable.

At each iteration of the loop, the algorithm calls the ASP solver with assumptions generated for this iteration and the given neighbourhood timeout. Those assumptions specify which parts of the current solution are fixed in this iteration (or move) based on the selected neighbourhood. If the solver finds a solution, then incumbent and bound are updated. The algorithm continues with such LNS iterations and different assumptions until the global time limit is reached.

3.2. *Defining the neighbourhood*

The LNS neighbourhood operator defines which parts of a solution are kept and which are destroyed in each iteration. The operator is usually problem specific but generic ones can also be effective. A good neighbourhood is large enough to contain a better solution but sufficiently small for the solver to actually find one. In our framework, we provide two predefined neighbourhood operators. Custom ones can be defined either in a purely declarative way, as part of the encoding and orthogonal to the problem specification, or by using a Python plugin.

As an example, consider the Social Golfer Problem from Section 2. There, a solution is a weekly schedule that defines which golfer plays in which group; consider a solution for the 3-3-3 instance:

	Week 1	Week 2	Week 3
Group 1	$(1,2,3)$	$(1,4,7)$	$(1,5,7)$
Group 2	$(4,5,6)$	$(2,5,8)$	$(2,6,8)$
Group 3	$(7,8,9)$	$(3,6,9)$	$(3,4,9)$

This schedule can be further optimised as some players meet more than once, e.g., 1 and 7 meet in both Week 2 and Week 3. A potential neighbourhood could be to unassign random positions in the above schedule. Another one could be to destroy entire groups or even weeks.

Predefined neighbourhoods We provide two problem-independent predefined neighbourhood types: random-atoms and random-constants.

- *Random-atoms* relaxes a random sample of visible atoms from an answer set specified by the `clingo` directive `#show`. Although quite simple, our experiments, see Section 6, confirm that this method is often surprisingly effective.
- *Random-constants* selects a random sample of all constant symbols of the visible atoms and relaxes all atoms that contain any constants from that selection. The underlying idea here is that often atoms are not independent and shared constants help to relax groups of related atoms together. As a small example, consider the following solution to some optimisation problem:

  ```
  sol(a,1), sol(a,2), sol(b,1), sol(b,4), sol(b,5)
  ```

  The set of constants is given by `a,b,1,2,4,5`. If we select `a`, the resulting relaxation would be

  ```
  sol(b,1), sol(b,4), sol(b,5)
  ```

  Hence, we remove all atoms that relate to `a`. Relaxation rates are used to define the size of the random sample for both types.

Declarative neighbourhoods To define a neighbourhood directly in ASP, we introduce two dedicated predicates `_lns_select/1` and `_lns_fix/2`:

- `_lns_select/1` is a unary predicate to define a set S of terms. In the LNS loop, a random sample is taken from the terms identified by this select predicate.
- `_lns_fix/2` is a binary predicate that serves to define a mapping from the terms in S to atoms that should be fixed with assumptions between solver calls. The first argument is the atom to fix and the second one is the corresponding term from S.

We illustrate this for different neighbourhood candidates for our running example.

Example 2 *(SGP continued).* For the Social Golfer Problem, we define the following four large neighbourhoods *pos, week, group, and group-p*:

[5] While enforcing a strictly better bound is how LNS is usually defined, tolerating a bit worse solutions with small (and decreasing) probability like in simulated annealing could be considered as well. We leave this as a direction for future work.

(*pos*) If we want to fix random positions of the schedule and therefore relax the rest, we can use:

```
_lns_select((P,W,G)) :- plays(P,W,G).
_lns_fix(plays(P,W,G),(P,W,G)) :- _lns_select((P,W,G)).
```

The selection is made on positions of the schedule, and atoms over `plays/3` are fixed if they match the selected position. This definition corresponds to the predefined neighbourhood "random-atoms".

(*week*) We can fix entire weeks of the schedule:

```
_lns_select(W) :- week(W).
_lns_fix(plays(P,W,G),W) :- _lns_select(W), plays(P,W,G).
```

(*group*) Similarly, we can fix random groups as follows:

```
_lns_select((W,G)) :- week(W), group(G).
_lns_fix(plays(P,W,G),(W,G)) :- _lns_select((W,G)),
                                plays(P,W,G).
```

(*group-p*) We may fix all groups containing a selected player:

```
_lns_select(P) :- player(P).
_lns_fix(plays(P1,W,G),P) :- _lns_select(P),
                             plays(P,W,G), plays(P1,W,G).
```

The size of the neighbourhoods defined via `_lns_select/1`, as for the predefined types, is controlled using respective relaxation rates.

Customised neighbourhoods in Python An alternative to the declarative specification is to define the neighbourhood in Python code. This is in particular valuable if a definition by rules would be cumbersome or not efficient.

Example 3 *(SGP continued).* Suppose that we want to alternate between different neighbourhoods in the Social Golfer example and pick each with a specified probability. The method *getMoveAssumptions* that constructs the neighbourhoods in Algorithm 2 can easily be overloaded with customised versions to implement this behaviour. We give examples for this later for an advanced parallel machine scheduling problem, which can serve as a blue-print for more customised applications.

3.3. LNS search and neighbourhood portfolios

The effectiveness of LNS is predicated upon suitable choices for search and neighbourhood operators. Different such operators can be bundled together into portfolios to choose from. A search configuration typically involves different time limits used in the reconstruction step. Likewise, a neighbourhood operator specifies the type of the neighbourhood, which corresponds to the structure and selection strategy of atoms, as well as different rates to control the size of the selection. Portfolios can contain both predefined neighbourhoods as well as custom ones.

3.4. Strategies for adaptive LNS

While LNS already works well with the right choice of search and neighbourhood operators, it seems suggestive to make changes when the search gets stuck. We provide to this end several *selection strategies* for LNS operators in portfolios that do not require any user intervention. After each iteration of the LNS loop, they take information like change in objective value, elapsed time, and whether a new solution was found, and then select from a portfolio some potentially more suitable LNS operator for the next iteration. Specifically, we discuss here three such selection strategies.

(1) Roulette-wheel selection In the *self-adaptive roulette-wheel strategy* [23,36], each pair (S, N) of search and neighbourhood operators from the used portfolio gets an initial weight. After each iteration of the LNS loop, the weight of the currently used pair (S, N) is updated according to the reinforcement learning formula

$$weight_{new} = (1 - \alpha) \cdot weight_{old} + \alpha \cdot r$$

where r is the *effectiveness score* of the operator, which is defined as the ratio of cost improvement over time needed for the operator to produce the result, and α is the *learning rate*. Then, a new pair of operators is selected with probability proportional to the weights.

(2) Uniform roulette-wheel selection The *uniform roulette-wheel strategy* is a simple non-adaptive version of the previous strategy, where all pairs of operators have equal weight, and are thus selected with equal probability in each iteration.

(3) Dynamic strategy The third strategy, which we call *dynamic strategy*, attempts to escape a stuck search by varying relaxation rates and time limits. Operators as well as neighbourhoods are not changed as long as they give some improvement. If the solver reports unsatisfiability three times in a row, we increase the relaxation rate of the currently used neighbourhood to the next largest rate; after the largest one is reached, new operators are randomly selected and search is resumed with the smallest relaxation rate for the new neighbourhood. If the solver times out however, it decides by coin flip to either decrease the relaxation rate to its lowest setting or to increase the time limit of the search operator; if this is not possible, new LNS operators are randomly chosen.

4. The ALASPO system

The system ALASPO,[6] standing for Adaptive Large-neighbourhood search for ASP Optimisation, is our implementation of the LNS framework from Section 3. It provides ASP optimisation with support for different ASP solvers, search configurations, and neighbourhood definitions. Fig. 3 gives on overview of its architecture and how its components play together to realise the system's functionality.

- *ASP Encoding and JSON Configuration File*: The optimisation problem is formulated in ASP and stored in one or multiple input files. Further input to the system can be provided in a JSON configuration file for the strategy to be used during the search.
- *Strategy*: The strategy component determines an LNS search operator, together with parameters for the search, resulting in a search configuration S, and an LNS neighbourhood operator for creating a neighbourhood N. It has access to the configuration file (if present), the search and neighbourhood portfolios, and it receives information about the change in costs and time needed for generating new solutions.
- *Search and Neighbourhood Portfolios*: These portfolios bundle different search and neighbourhood operators together to choose from. A search configuration typically involves different time limits used in the reconstruction step. Similarly, a neighbourhood operator specifies the type of neighbourhood, which corresponds to the structure and selection strategy of the atoms, as well as different rates to control the size of the selection. The portfolios contain some predefined neighbourhoods, but custom ones can be added.
- *Construction Heuristic*: This component allows to provide an initial solution by a custom procedure, which, however, is problem specific and must be provided by the user as an implementation of an abstract class in Python 3.
- *ASP Solver*: The ASP solver component is in charge of computing the solutions to the problem and reconstructing better solutions from a relaxed partial solution, respecting a search configuration S. Currently, the ASP solvers `clingo`, `clingo-dl`, and `clingcon` from the Potassco[7] family are supported. They all feature multi-shot solving and solving under assumptions [25]. The former is used to keep learnt heuristics and constraints between solver calls and to avoid unnecessary grounding; the latter is used to technically realise the relaxation step by fixing the parts of the solution that are not relaxed with assumptions prior to the next solver call.
- *Control & Interaction*: The control component (not shown) orchestrates the working of the other components and also takes care of the interaction with the user.

Upon invocation, the system sets up the LNS portfolios and ASP solver based on information from the command line and, if present, the JSON configuration file. Afterwards, it grounds the ASP encoding. If a construction heuristic is specified, it is used to generate the initial solution which is given to the ASP solver via assumptions. If none is specified, the first solution is generated with the ASP solver itself. Unless this first solution is already optimal, in which case search stops immediately, ALASPO proceeds with the LNS loop. Progress information is continuously logged and presented to the user. The system updates the search and neighbourhood operators according to the used strategy, and the search continues until a global time limit is reached or the search is interrupted by the user. Finally, ALASPO outputs the best solution that was found and its cost.

4.1. Running ALASPO in novice mode

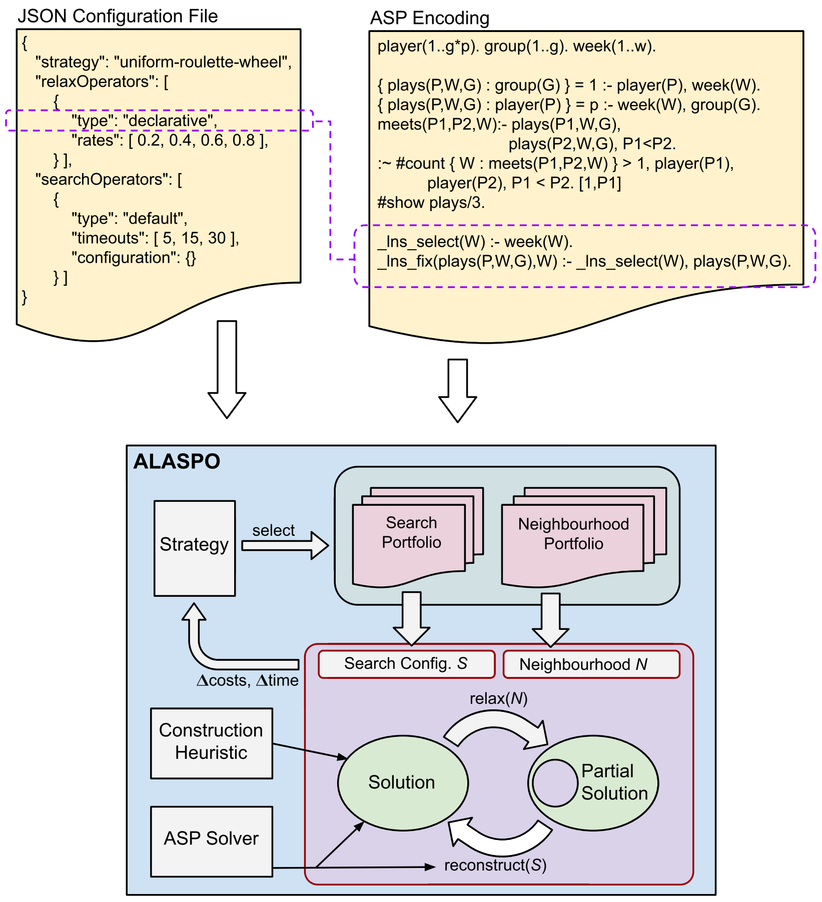
ALASPO can be used out of the box with little or no configuration. If needed, however, it can also be customised through either the command line or a configuration file as described above. The tool is called in standard configuration for a problem stored in `program.lp` as follows:

```
 >  alaspo -i program.lp
```

where it runs the system with its *default portfolio* using neighbourhood relax-atoms with relaxation rates 0.1, 0.2, 0.4, 0.6, and 0.8, neighbourhood relax-constants with rates 0.1, 0.2, 0.3, 0.4, 0.5, time limits of 5, 15, 30, and 60 seconds for the search configurations,

[6] https://gitlab.tuwien.ac.at/kbs/BAI/alaspo.

[7] https://potassco.org/.


JSON Configuration File

```
{
  "strategy": "uniform-roulette-wheel",
  "relaxOperators": [
    {
      "type": "declarative",
      "rates": [ 0.2, 0.4, 0.6, 0.8 ],
    } ],
  "searchOperators": [
    {
      "type": "default",
      "timeouts": [ 5, 15, 30 ],
      "configuration": {}
    } ]
}
```

ASP Encoding

```
player(1..g*p). group(1..g). week(1..w).

{ plays(P,W,G) : group(G) } = 1 :- player(P), week(W).
{ plays(P,W,G) : player(P) } = p :- week(W), group(G).
meets(P1,P2,W):- plays(P1,W,G),
                 plays(P2,W,G), P1<P2.
:~ #count { W : meets(P1,P2,W) } > 1, player(P1),
      player(P2), P1 < P2. [1,P1]
#show plays/3.

_lns_select(W) :- week(W).
_lns_fix(plays(P,W,G),W) :- _lns_select(W), plays(P,W,G).
```

Fig. 3. Adaptive LNS for ASP in the system ALASPO.

and the dynamic selection strategy. These defaults cover a range of neighbourhood sizes and time limits that are reasonable for many optimisation problems. During execution, the solver will continuously report when new solutions are found.

4.2. *Further command-line arguments and neighbourhoods*

Additional command-line arguments can be used to change the global time limit and to select an ASP solver other than `clingo` and extensions of it (`clingo-dl` and `clingcon` are currently supported as well). Sometimes, it is beneficial to let the ASP solver optimise the problem for some time before the LNS loop starts; to this end, a time limit for pre-optimisation can be specified.

Instead of the default portfolio, the user can select a single neighbourhood type from the predefined ones (random-atoms or random constants) or specify that one declaratively defined by the ASP program should be used instead. In the command line, one can set a relaxation rate, an argument list to be passed to the ASP solver, and a time limit for reconstruction steps as well. In the mode where single search and neighbourhood operators are used, a new neighbourhood in the LNS loop is only selected if the current one stops to yield improvements and is fully exploited. Option `-help` gives a complete description of the command line.

4.3. *Defining portfolios and search strategies*

The features described so far allow one to run the tool with a simple default portfolio or with single LNS operators. With a good relaxation rate and time limit for LNS steps, better solutions can indeed be found starting from the relaxed solution while the search space is small enough for the solver to actually succeed within the time limit. Such calibration needs in general some trials; alternatively user-defined portfolios with different selection strategies can be employed.

Several declarative neighbourhoods can be specified for use in the operator portfolio of the system. Alternative definitions are disambiguated in the encoding by giving them a name as additional first argument of `_lns_select` and `_lns_fix`. Examples of how to define neighbourhoods in ASP can be found in the Section 3 for Social Golfer, one of which is also shown in Fig. 3. The encoding together with the neighbourhood definition (either as one or several files) is given as input to ALASPO.

Portfolios can be specified in an additional JSON file as depicted in Fig. 3. There, the user lists all LNS operators that should be used. For each neighbourhood, a type (random-atoms, random-constants, or declarative), an optional name in case there are several

```
{
    "strategy": {
        "name": "dynamic",
        "unsatStrikes": 3,
        "timeoutStrikes": 1
    },
    "relaxOperators": [
        {
            "type": "randomAtoms",
            "sizes": [ 0.1, 0.2, 0.4, 0.6, 0.8 ]
        },
        {
            "type": "randomConstants",
            "sizes": [ 0.1, 0.2, 0.3, 0.5 ]
        }
    ],
    "searchOperators": [
        {
            "name": "default",
            "timeouts": [ 5, 15, 30, 60 ],
            "configuration": {}
        }
    ]
}
```

Fig. 4. ALASPO's default portfolio.

declarative neighbourhoods in the encoding, and a list of relaxation rates for instances of the operator are given. Search configurations can be defined with a list of time limits on LNS iterations. The selection strategy for the portfolio (roulette-wheel or dynamic) can be specified in the JSON file as well. The portfolio in Fig. 3 selects the uniform roulette-wheel strategy and "declarative" as the type of the LNS neighbourhood operator; the respective definition as ASP rules appears as part of the ASP encoding on the right side of the figure. The next example explains the default portfolio of ALASPO.

Example 4. ALASPO's default portfolio is shown in Fig. 4. It uses the dynamic strategy with additional parameters to change LNS operators if the ASP solver yields "UNSAT" three times in a row or if it times out once. The neighbourhood operators are random-atoms and random-constants with a range of relaxation rates. The ASP solver itself is used in its default configuration with time limits ranging from 5 to 60 seconds.

4.4. *Monitoring the optimisation progress*

During the optimisation process, the solver gives continuous feedback to the user that allows to monitor progress. In particular, this feedback consists of

- the currently used LNS neighbourhood operator,
- the currently used LNS search operator,
- the size of the neighbourhood in terms of atoms that have been fixed by assumptions, and
- the outcome of the LNS iteration which is either "SAT" (in which case a new objective value is reported), "UNSAT" (no solution with better objective value exists within the current neighbourhood), or "TIMEOUT" (no solution has been found within the time limit of the search operator).

This information helps to find out why search stalls so that the user can change the configuration or take other actions as discussed in the next subsection.

4.5. *Running ALASPO in interactive mode*

In *interactive mode*, prompted by command-line option `--interactive`, the user has the option to change LNS operators manually whenever search is interrupted.

When intervention seems necessary, e.g., there are only timeouts and no progress, the user can interrupt search by pressing `Ctr+C` on the keyboard. The system will then show available configurations for search and neighbourhood operators from the portfolios on the terminal, giving the user the option to select a better fitting configuration before search is resumed.

4.6. Custom applications with python plugins

The ALASPO system can be extended for custom applications by defining new neighbourhood types in code. To give an idea how this works, we show Python code to implement random-atoms in the following example.

Example 5. Implementation of the random-atoms neighbourhood in Python: `incumbent` contains the current solution, `asm` is a list of the atoms that should be fixed via assumptions, `max_selection_sz` represents the number of atoms in the current solution, and `self._size` is the relaxation rate.

```
def get_move_assumptions(self, incumbent):
        asm = []
        max_selection_sz = len(incumbent.model.shown)
        selection_sz = round(max_selection_sz * (1 - self._size))
        asm = random.sample(incumbent.model.shown, selection_sz)
        return asm
```

Furthermore, if an ASP solver cannot produce a good first solution, construction heuristics, e.g., a simple greedy procedure, can be specified by instantiating a respective abstract class. An example developed for a challenging parallel machine scheduling problem is described in more detail in Section 7.

5. Methodology for using LNS in ASP

While LNS has the potential to substantially improve the optimisation performance of an ASP solver, its effectiveness depends on how the meta-heuristic is set up, i.e., what LNS operators are used and how they are configured. We next discuss a methodology of LNS for ASP in more depth and give pointers distilled from our practical experience. Others have been discussing aspects such as variants of LNS itself and how to use it for specific problems [13], but a more holistic methodology has not been presented, in particular not for ASP. Some aspects of our discussion are indeed more specific to ASP and ALASPO, like finding good ASP solver configurations or search parameters for ALASPO, while other aspects are guidelines for LNS that are useful more generally, like how to define good neighbourhoods, and are therefore of interest beyond ASP.

5.1. How to find good LNS search configurations?

To fine-tune the solver for a given optimisation problem, it is in general advisable to have a couple of problem instances of representative size available to experiment with. As a first step, one can just run the ASP solver, say `clingo`, on these instances to create a baseline. This will also tell one whether an advanced optimisation heuristic like LNS is even necessary. This is also the time to try different command-line options and to observe their impact on optimisation performance. Recall that LNS search operators in ALASPO are determined by command-line options for the ASP solver and a time limit for the search step of the LNS loop. It is our experience that command-line options that do well in this preliminary step are also suitable for the LNS search. For example, in the shift design application from Section 7.1, the solver option for unsatisfiable-core based optimisation was chosen for ALASPO as it performed best with plain `clingo` in previous work [33].

The time limit for the search operator depends on what neighbourhood operators are used and in particular the size of the neighbourhoods as larger ones will in general require longer run times. As a starting point, a portfolio configuration with a wide range of time limits, for example from one second to a minute in 5 second steps, can be used and later pruned to obtain a more efficient setup. We will be more specific when we discuss neighbourhood portfolios.

5.2. How to obtain an initial solution?

LNS requires an initial solution that is iteratively improved. By default, this initial solution is the first solution found by the ASP solver. This works reasonably well when a first solution can be computed fast even if the optimisation problem is hard, and we indeed use this approach for all benchmarks from Section 6.

This option is however not ideal if the ASP solver often finds optimal solutions and only fails to produce improvements for some instances after some time. Then, and we discuss a shift design problem in Section 7.1 where this is the case, the option to allow ALASPO to pre-optimise the problem for a set time is preferable. We then still get the optimal solutions for the instances where the plain ASP solver is able to find them, but if search gets stuck, LNS helps to find potential further improvements that are missed otherwise.

If ASP is rather bad at finding an initial solution, i.e., the solver is slow and the quality of the first solution is poor, using a custom construction heuristic can become the option of choice. We exemplify this for a hard machine scheduling problem in Section 7.3, where the first solution is obtained by greedily assigning jobs to machines that incur the least increase of the solution cost. For classical search problems, well-known heuristics exist and can be applied; a more in-depth discussion of this topic is however outside the scope of this article.

5.3. How to define good LNS neighbourhoods?

A good neighbourhood operator is essential for LNS to work: on the one hand, solutions need to be relaxed in a way that makes it likely that better solutions can be found in the reconstruction step. On the other hand, we cannot make the neighbourhood too large as otherwise the advantage of focusing search on fewer decision variables is diminished and the reconstruction step can become infeasible. In general, the type of neighbourhood depends on the problem and requires domain knowledge. The size of the neighbourhood needs to be dynamically adjusted during search in accordance with the time limit of the LNS search operator.

Random-atoms as starting point The predefined LNS operator *random-atoms* is often a good starting point provided that the visible atoms filter answer sets to atoms that represent problem solutions. Similar to the time limits for the LNS search operator, a portfolio with a range of percentages, for example $10\%, 20\%, \dots, 90\%$, for the neighbourhood size can be used. For all benchmarks except for Social Golfer from Section 6, random-atoms was used as the single-best LNS neighbourhood operator.

When does random-atoms fail Random-atoms can be a bad choice if there are lots of dependencies between the visible atoms. If many relaxed atoms are implied by fixed ones, it becomes likely that the relaxed solution will be restored into the old one unless the neighbourhood size is excessively large. This can happen when dealing with, e.g., nested assignments that are encoded with dependent choices.

Example 6 *(Dependent choice in ASP).* Consider the following example program:

```
{ choice(c1); choice(c2) } = 1.
{ dep_choice(1); ...; dep_choice(n)   } :- choice(c1).
{ dep_choice(n+1); ...; dep_choice(m) } :- choice(c2).
```

Each answer set will contain either `choice(c1)` or `choice(c2)` which constrains the rest of the answer set to either contain atoms `dep_choice/1` over $\{1, \dots, n\}$ or $\{n+1, \dots, m\}$. Note that the `dep_choice/1` atoms "imply" the `choice/1` atoms: if an answer set I contains any atom `dep_choice/2` over $\{1, \dots, n\}$, then it contains `choice(c1)`. Likewise, if I contains any atom `dep_choice/2` over $\{n+1, \dots, m\}$, then it also contains `choice(c2)`.

For large m, the probability that the choice between `c1` and `c2` ever changes in the reconstruction step quickly approaches zero if random-atoms is the neighbourhood strategy.

A remedy is to either define a custom neighbourhood or to use the predefined LNS neighbourhood *random-constants*. This neighbourhood allows us to use constant symbols to deal with atom dependencies. We only need to rewrite the last two rules from the example slightly to make dependencies explicit and accessible:

```
{ dep_choice(c1,1); ...; dep_choice(c1,n)   } :- choice(c1).
{ dep_choice(c2,n+1); ...; dep_choice(c2,m) } :- choice(c2).
```

Assume that random-constants selects `c1`. Then `choice(c1)` together with all atoms over `dep_choice/2` dependent on `c1` are removed from the solution. Hence, we are not stuck with `choice(c1)` anymore and the choice can change indeed from `c1` to `c2` in the reconstruction step.

A more complex example where random-constants is a better choice than random-atoms due to issues with dependencies between atoms is discussed in Section 7.3. There, jobs have to be assigned to machines. On each machine, a processing order of the jobs has to be determined. Similar to the example from above, the latter is dependent on the former which is disadvantageous for the random-atoms operator.

Another scenario where random-atoms is not a good choice is when only few visible atoms in a solution are violating a weak constraint. A relaxed solution cannot be improved if no atoms are involved in such conflicts, hence one has to make sure that this is unlikely by using more customised neighbourhood definitions that will include a sufficient number of conflict atoms.

Neighbourhood definitions that correspond to relaxed problems Some problems are composed out of simpler but mostly mutually dependent sub-problems. As an example, consider the task of assigning n jobs to m machines with the objective to minimise the total completion time. Each subset of the m machines induces a sub-problem. Good neighbourhood operators are often obtained by focusing search on simpler sub-problems in each LNS step. This could mean to remove all jobs from k out of m machines while fixing the other assignments. Here, the size of the neighbourhood is controlled via k. A custom procedure that implements this idea is used for a similar scheduling problem in Section 7.3. Another example for this type of neighbourhoods is removing entire weeks from solutions in our Social Golfer example, cf. Example 2.

Interplay between neighbourhood size and time limits for LNS search Assume we have found a suitable command-line configuration for the ASP solver as well as a good neighbourhood type for the optimisation problem at hand by following these guidelines so far. Also assume that we use a portfolio with a wide range of ratios for the size of the neighbourhood as well as for the time limits used in each reconstruction step. One option to proceed is to just run ALASPO with this portfolio so it can self-adapt to effective LNS operators. Experiments that involve portfolios and ALASPO in self-adaptive mode are discussed in Section 6.2. If the goal is however to run the system with a single pair of search and neighbourhood operators, the interactive mode can help to learn which combinations are most effective. This is also helpful to gain more general insights into how the LNS search performs. We present experiments that use ALASPO with single LNS operators in Section 6.1. The interactive mode allows us to interrupt search and select new LNS operators from the portfolios. We next discuss different cases when such an intervention is useful.

1. *The solver returns UNSAT most of the time.* In this case, the neighbourhood is too small and the solver can exhaust the search space without being able to find a better solution. This implies that the neighbourhood of the current size is not useful anymore and we can switch to a larger one. If this happens already at the beginning, this neighbourhood size can be removed from the portfolio altogether.
2. *The solver frequently times out.* This is due to one of the following two reasons: either (i) there is a better solution within the current neighbourhood but the time limit for the solver to find it is too small, or (ii) the neighbourhood does not contain a better solution but showing that the instance is UNSAT is not feasible within the current time limit. Then the user can increase the time limit, and, if reason (ii) is suspected, also increase the size of the neighbourhood.

5.4. *How to define good LNS portfolios?*

So far, we outlined how to define good LNS search and neighbourhood operators, and how to identify good combinations of size and time limits. It is however not necessary to nail down everything to a single pair of LNS operators. Instead, the adaptive modes can be used to great advantage.

A portfolio can contain any LNS operator that is considered promising. The range of the time limits and neighbourhood sizes in use should be generous but not excessive. If there are too many options, the selection strategies will spend too much time on ineffective combinations and the overall performance can suffer. The interactive mode, as discussed previously, can help to prune away settings that are not useful.

Limitations of the selection strategies ALASPO offers currently three different selection strategies: (1) adaptive roulette-wheel selection, (2) uniform roulette-wheel selection, and (3) the dynamic strategy. It is instructive to discuss the conditions when each of these strategies performs sub-optimally.

The adaptive roulette wheel selection has the tendency to stick with LNS operators that perform well at the beginning of the search, as this increases their relative weights. This can hinder the search when it becomes harder to make progress as the algorithm is overly optimistic with its initial selection. Hence, this strategy is not ideal when the search goes quickly from easy, where all LNS operators perform well, to hard, where only few are able to produce any progress.

The uniform roulette wheel selection chooses LNS operators randomly in each step. This does not work very well when the portfolio contains lots of operators that only produce improvements in rare cases. If the portfolio is however well pruned, this strategy can avoid some of the pitfalls of the other strategies as it is unbiased and tries every combination of LNS operators over and over. This can work well, provided the global time limit is not too short.

The dynamic strategy is less explorative than the other strategies and sticks more conservatively with one neighbourhood type as long as some progress can be made. It rather adjusts the size of the neighbourhood and the time limit if progress stalls. Progress can be unnecessarily slow if the strategy chooses a neighbourhood type that does not work well with a given problem instance, while a better one would be available.

6. Experiments on ASP benchmark problems

In this section, we experimentally demonstrate the effectiveness of ALASPO on different benchmark problems that originate from ASP competitions [47–49]. The problems vary in their complexity which ranges (for their decision variants) from NP-complete, over strongly NP-complete, to Σ_2^P-complete. First, we consider a more controlled setup where we evaluate the performance of ALASPO for a single pair of LNS operators that has been manually determined for each benchmark problem. Then, we evaluate ALASPO with portfolios and adaptive selection strategies on the same benchmarks to demonstrate the ability of the system to automatically find effective LNS operators.[8]

6.1. *Experiments with single LNS operators*

Unless stated otherwise, `clingo` was called with no additional command-line parameters, i.e., it uses a single solving thread and employs branch-and-bound-based optimisation.

Social Golfer problem For Social Golfer, we compared `ALASPO` against plain `clingo` as a baseline with a time limit of 1 800 seconds for each run. As instances, we considered problems with 8 groups of 4 golfers over 7 to 12 weeks. The optimisation goal is to minimise the number of times two players meet each other more than once. We used `ALASPO` with the different neighbourhood definitions from

[8] All experiments were run on a cluster with 13 nodes, each having 2 Intel Xeon CPUs E5-2650 v4 (max. 2.90GHz, 12 physical cores, no hyperthreading), with memory limit 20 GB. We used `clingo` v 5.5.1, `clingo-dl` v 1.4.1, and `clingcon` v 5.2.0. All encodings, instances, logs, and random seeds are available at https://doi.org/10.5281/zenodo.11058964.

Table 1
clingo vs. ALASPO for instances 8-4-w of the Social Golfer Problem with different neighbourhoods. For ALASPO, we report the best and worst penalties, i.e., number of times golfers meet more than once, over 5 runs.

w	clingo	ALASPO			
		pos	*week*	*group*	*group-p*
7	**0**	**0**	**0**	**0**	**0**
8	3	**1**–2	2–3	3	2–4
9	7	6–7	**4**–6	6–7	5–7
10	11	9–10	**7**–9	9–10	8–9
11	13	12–13	**11**–12	12–13	12
12	15	**14**–15	**14**	**14**–15	**14**

```
{ cycle(X,Y): edge(X,Y); cycle(X,Y): edge(Y,X) }=1 :- vtx(X).
{ cycle(X,Y): edge(X,Y); cycle(X,Y): edge(Y,X) }=1 :- vtx(Y).

reached(1).
reached(Y) :- reached(X), cycle(X,Y).
:- vtx(X), not reached(X).
:~ cycle(X,Y), edgewt(X,Y,C). [C,X,Y]

#show cycle/2.
```

Fig. 5. Encoding for the Travelling Salesman Problem.

Section 3. We report the best and worst solution found with ALASPO in five runs. The time limit to explore individual neighbourhoods was 20 seconds. The size of each neighbourhood was set to relax about 80% of the atoms over plays/3. This is rather large compared to our other experiments, but necessary to find better solutions while still helping the solver by restricting the search space. The results are shown in Table 1.

Social Golfer is known to be notoriously hard for symbolic solvers due to symmetries, and optimal solutions are still out of reach for many instances where optimal bounds are known. Yet, any improvement for ASP can be considered as an important step forward. For instances with 7–10 weeks, conflict-free schedules exist in principle; this is not the case for instances with 11 and 12 weeks. LNS with ALASPO is able to find better solutions than plain clingo in many cases with all neighbourhood settings. Fixing a number of weeks entirely turns out to work best for this experiment, where it gives improvements most consistently.

Travelling Salesman problem We next consider the well-known Travelling Salesman Problem (TSP) [30]. The encoding in Fig. 5 is an optimisation variant of the one from the Asparagus platform, from which also the instances were taken.[9]

The overall time limit was set to 300 seconds, and we limited search within any neighbourhood to 5 seconds. We used ALASPO with random relaxation of the cycle/2 atoms and a neighbourhood size of 30%. The results are shown in Table 2, where we report the cost of the best round trip found by clingo as well as the average, best, and worst costs found by ALASPO in 5 runs.

The LNS approach finds better bounds than clingo throughout. Even the worst solutions found with LNS give an improvement of 34% on average. The default neighbourhood is advantageous for this problem since cycle/2 atoms indicate the next element in the Hamiltonian tour, and relaxing them resembles k-opt moves from local search, where, in each step, k links of the current tour are replaced by links such that a shorter tour is achieved.

Sudoku puzzle generation ASP can be used for optimisation problems where checking feasible solutions is beyond NP; in fact, uniform ASP encodings can solve decisional variants of such problems with complexity up to Δ_3^P [21]. In particular, checks in coNP are expressible (e.g., a TSP instance has no solution) with a *saturation technique* [50] that uses minimality of answer sets.

Suppose we want to compute Sudoku puzzles [31] that give a smallest number of hints. Fig. 6 shows an encoding for this problem with variable grid size. Roughly speaking, we guess a set of hints subject to minimisation (lines 1,23) and check that they can be completed to a fully filled-in Sudoku S (lines 3–9). As each Sudoku puzzle must have a unique completion, we check, using saturation, that no different completion S' exists, i.e., every assignment S' of numbers to the grid is either not a valid completion or equal to S (lines 11–21).

We compared clingo and ALASPO with options --configuration=many and -t 4 for clingo, which is the default portfolio for multi-threaded solving and four threads. For ALASPO, the neighbourhood operator was random relaxation of 20% of the visible atoms. We used the same clingo options for the LNS search operator and a time limit of 15 seconds. While clingo finds a solution

[9] https://asparagus.cs.uni-potsdam.de/.

Table 2
`clingo` vs. `ALASPO` for 20 instances of the Travelling Salesman Problem with average, best, and worst cost among 5 runs for `ALASPO`.

	clingo	ALASPO	
01	601	390.4	(**384**–394)
02	563	332.0	(**327**–337)
03	580	408.2	(**403**–413)
04	649	435.2	(**430**–440)
05	602	369.8	(**365**–373)
06	643	406.0	(**399**–409)
07	569	393.2	(**385**–399)
08	549	369.6	(**367**–374)
09	606	393.6	(**391**–399)
10	540	345.0	(**338**–357)
11	567	353.4	(**349**–357)
12	721	409.8	(**401**–420)
13	598	422.6	(**414**–430)
14	695	434.2	(**429**–440)
15	745	469.2	(**463**–474)
16	696	426.4	(**424**–429)
17	725	444.0	(**441**–449)
18	667	502.2	(**394**–513)
19	740	450.4	(**446**–456)
20	683	420.2	(**413**–426)

```
{ hint(R,C,N): R=1..grid_sz, C=1..grid_sz, N=1..grid_sz }.

  a(R,C,N) :- hint(R, C, N).
  { a(R,C,N): N = 1..grid_sz } = 1 :- R = 1..grid_sz,
              C = 1..grid_sz.
  :- a(R,C1,N), a(R, C2, N), C1 != C2.
  :- a(R1,C,N), a(R2, C, N), R1 != R2.
  :- a(R,C,N), a(R1, C1, N), R != R1, C != C1,
     (((R-1)/subgrid_sz)*subgrid_sz + (C-1)/subgrid_sz) =
     (((R1-1)/subgrid_sz)*subgrid_sz + (C1-1)/subgrid_sz).

  b(R,C,N) : N = 1..grid_sz :- R = 1..grid_sz, C = 1..grid_sz.
  saturate :- b(R,C,N1), hint(R, C, N2), N1 != N2.
  saturate :- b(R,C1,N), b(R, C2, N), C1 != C2.
  saturate :- b(R1,C,N), b(R2, C, N), R1 != R2.
  saturate :- b(R,C,N), b(R1, C1, N), R != R1, C != C1,
        (((R-1)/subgrid_sz)*subgrid_sz + (C-1)/subgrid_sz) =
        (((R1-1)/subgrid_sz)*subgrid_sz + (C1-1)/subgrid_sz).
  saturate :- equals(R,C) : (R,C) = (1..grid_sz, 1..grid_sz).
  equals(R,C) :- a(R,C,N), b(R,C,N).
  b(R,C,N) :- saturate, R = 1..grid_sz, C = 1..grid_sz,
              N = 1..grid_sz.
  :- not saturate.

  :~ hint(R, C, N). [1,R,C,N]

  #show hint/3.
```

Fig. 6. Encoding for Sudoku Puzzle Generation.

with 21 hints for the standard 9×9 grid within 10 minutes, we found puzzles with 19 hints using ALASPO. This is a significant improvement that reduces the gap between the baseline and the known minimal bound 17 by 50%.

Weighted strategic companies A well-known ASP benchmark that is complete for Σ_2^P is Strategic Companies [32]: a company of a holding is strategic if it belongs to a strategic set, i.e., a minimal set of companies of the holding that allow to manufacture all products and maintain control relationships. We consider an optimisation variant here, where we assign random weights to companies, and the objective is to find strategic sets of minimal total weight. The encoding is given in Fig. 7; the instances are those of the 3rd [47], 4th [48] and 5th [49] ASP competition with random weights from $[1, 1000]$ added.

```
  strat(X1) | strat(X2) | strat(X3) | strat(X4) :-
                             produced_by(X,X1,X2,X3,X4).
  strat(W) :- controlled(W,X1,X2,X3,X4),
              strat(X1), strat(X2), strat(X3), strat(X4).

 :~ strat(C), weight(C,W). [W,C]

 #show strat/1.
```

Fig. 7. Encoding for Weighted Strategic Companies.

Table 3
`clingo` vs. `ALASPO` for instances of Weighted Strategic Companies with average, best, and worst weight among 5 runs for `ALASPO`.

	`clingo`	`ALASPO`	
`001`	231 092	209 414.6	(**207 374**–212 612)
`006`	91 221	94 782.2	(**88 925**– 99 116)
`015`	224 472	210 205.6	(**207 467**–212 338)
`018`	134 757	129 645.0	(**124 313**–131 251)
`019`	105 481	103 482.0	(**98 796**–107 197)
`030`	226 653	213 393.8	(**203 136**–217 266)
`033`	230 732	219 070.6	(**217 381**–219 493)
`042`	138 809	125 909.0	(**124 494**–128 524)
`050`	210 771	190 303.6	(**186 975**–192 591)
`051`	170 227	76 929.8	(**69 945**– 82 626)
`052`	207 188	90 402.6	(**84 859**– 98 724)
`053`	161 343	74 111.4	(**69 348**– 81 988)
`054`	224 058	76 589.4	(**72 038**– 85 074)
`055`	205 034	95 254.6	(**82 870**–105 613)
`056`	204 921	84 050.4	(**78 299**– 91 111)
`057`	219 262	79 053.2	(**74 010**– 86 627)
`058`	175 945	73 224.0	(**70 936**– 77 177)
`059`	200 575	73 383.8	(**70 640**– 77 592)
`060`	201 830	87 292.4	(**82 552**– 91 980)
`061`	216 207	74 421.4	(**71 772**– 76 291)

We compare `clingo` (called via the Python API) against `ALASPO`, where we use the default neighbourhood and relax 20% of the companies in each step. The global time limit was 1 800 seconds, and the time limit for LNS steps was 30 seconds. The results are shown in Table 3. Note that we omit instances for which `clingo` does not produce any feasible solution in 30 minutes. The LNS approach improves the bounds from the baseline by up to 65%, while the average solution quality is only worse for a single instance.

6.2. Experiments with portfolios and adaptive strategies

We use the benchmarks Social Golfer (SG), Travelling Salesman (TSP), Sudoku Puzzle Generation (SPG), and Weighted Strategic Companies (WSC) from Section 6.1 to evaluate how the self-adaptive modes of ALASPO perform in comparison with hand-selected search and neighbourhood operators.

We used `clingo` (v. 5.5.1) with the same search configuration (the default configuration if not stated otherwise) as the LNS approach to establish a baseline.

For our comparisons with the adaptive modes of ALASPO with portfolios, we used the solver settings as described in Section 6.1 and time limits from the default portfolio (cf. Section 4). For the neighbourhood operators, we included both random-atoms and random-constants with relaxation rates from the default portfolio. For Social Golfer, we used all the declarative neighbourhoods for that problem from Section 3 together in a portfolio.

Results The results of our experiments are summarised in Table 4. For all problems, we report the relative improvement in objective value for ASP with LNS in comparison with plain `clingo` averaged over five runs. The first row gives the results for LNS with single hand-selected LNS operators that fit the problem best as described in Section 6.1. The remaining rows show results obtained with the portfolio described above for different selection strategies.

Overall, `clingo` with LNS is able to produce better results than plain `clingo` on all benchmarks. The only case where the baseline is not reached is Sudoku Puzzle Generation with roulette-wheel selection and a high learning rate where sub-optimal operators become preferred ones. The setting with single hand-selected LNS operators is the clear winner in performance gain for the Travelling Salesman Problem, while it is outperformed by the portfolio approaches for Social Golfer and Weighted Strategic Companies. Notably, putting LNS operators into a portfolio is less effort for a user than determining manually which ones work best. The self-adaptive roulette-

Table 4
Average percentage of improvement and relative standard deviation over plain `clingo` for ASP with (adaptive) LNS.

	SG	TSP	SPG	WSC
single LNS operators	15.91 ± 0.74	$\mathbf{35.85} \pm 0.10$	$\mathbf{3.33} \pm 0.02$	39.70 ± 0.51
uniform roulette	17.14 ± 0.71	19.65 ± 0.10	0.47 ± 0.02	39.83 ± 0.56
adapt. roul. $\alpha = 0.2$	18.36 ± 0.71	20.75 ± 0.11	0.00 ± 0.02	40.01 ± 0.57
adapt. roul. $\alpha = 0.4$	$\mathbf{20.00} \pm 0.74$	20.54 ± 0.11	2.38 ± 0.03	40.17 ± 0.57
adapt. roul. $\alpha = 0.6$	17.14 ± 0.72	20.74 ± 0.11	1.90 ± 0.00	39.96 ± 0.57
adapt. roul. $\alpha = 0.8$	18.36 ± 0.71	21.17 ± 0.11	-0.95 ± 0.03	$\mathbf{40.25} \pm 0.57$
dynamic	19.18 ± 0.72	29.03 ± 0.11	$\mathbf{3.33} \pm 0.03$	39.44 ± 0.51

wheel strategy performs better than the uniform one in most of the cases, at least with the right learning rate. The dynamic strategy gives most consistently excellent improvements though not in all cases the best ones.

7. Applications of LNS with ASP

We next turn to advanced use cases of ASP with LNS and present three applications of our approach that involve real-world problems from the literature: the practically relevant *shift design problem* from the domain of work force scheduling [33], *job scheduling on parallel machines* from semiconductor production at Bosch [35], and *test laboratory scheduling* [51,19]. For the first problem, `clingo` is well suited as underlying ASP solver for the LNS search. The second and third problem require reasoning over integer domains and are better suited for `clingo-dl`, which extends `clingo` by difference constraints, and `clingcon` for full integer constraints.

While the definition of a single suitable neighbourhood is enough to improve results for the shift design problem, the two scheduling problems serve as a showcase for neighbourhood definitions in code and portfolios, respectively.

7.1. Shift design

The goal in Shift Design is to align shifts so that over- and understaffing of workers is avoided. We refer to related work [33] for a detailed problem description as well as the description of the ASP encoding and the instances.[10] The objective function we use is the hierarchical one from the original paper of first avoiding understaffing, second avoiding overstaffing, and third, minimising the total number of shifts. We consider all instances from DataSet3 and DataSet4, some of which are still quite challenging for ASP. DataSet3 contains instances where over- and understaffing cannot be avoided, and DataSet4 contains a larger instance from a real-world application.

We used the solver `clingo` with options

```
--opt-strat=usc,3 --configuration=handy
```

as baseline. The options run `clingo` with unsatisfiable-core based optimisation and defaults geared towards large problems. This solver configuration was the most effective in the experiments of the original paper. We used the same solver configuration also within the LNS loop, as well as the one hour limit per instance for the experiments. The LNS solver spends at most 30 seconds exploring each neighbourhood, which is set to randomly relax 70% of the assigned shifts in each step. Plain ASP is with the right solver configuration already quite effective for this problem and finds optimal or near optimal solutions in many cases. We thus used pre-optimisation for 50 minutes to let the solver reproduce the old bounds before using LNS on top. We report the best and worst bounds from five runs for the LNS approach. For eight out of 33 instances from both data sets, neither approach could find any solution. For 17 instances `clingo` could find the optimal value and ALASPO reported the same value as `clingo`. Results for the eight remaining instances are given in Table 5, where we get indeed considerable improvements.

Although a single fixed pair of LNS operators is already quite effective, we also evaluated the portfolio approach, as described in Section 6.1, on this problem. The results are shown in Table 6. Interestingly, the uniform selection strategy performs best on average this time. A possible explanation is the long pre-optimisation that we used which makes it difficult for any operator to find further improvements. If it happens, it is highly dependent on luck in the relaxation step. It is thus best to repeatedly try all operators without any bias, while the adaptive roulette wheel selection overestimates the utility of operators once they produced an improvement, and the dynamic strategy sticks too long with the same operator and thereby wastes too much time.

7.2. Partner units problem

The Partner Units Problem (PUP) is a challenging real-world configuration problem with industrial applications such as, e.g., railway-safety systems at Siemens [34,52]. It requires to group sensors into zones and connecting them to control units. Control units

[10] For completeness, the ASP encoding is also part of the appendix.

Table 5
Shift Design instances from DataSet3 and DataSet4 on which `clingo` and ALASPO diverged, where for ALASPO the best and worst objective value from 5 runs is shown. The values correspond to shortage of staff, excess of staff, and number of shifts, respectively.

	`clingo`	ALASPO
`3-04`	(0, 413, 50)	(**0**, **353**, **45**)–(0, 372, 47)
`3-06`	(0, 286, 44)	(**0**, **222**, **43**)–(0, 312, 52)
`3-11`	(0, 821, 74)	(**0**, **713**, **65**)–(0, 725, 65)
`3-20`	(0, 1 006, 66)	(**0**, **946**, **68**)–(0, 963, 67)
`3-26`	(0, 1 061, 77)	(**0**, **1 037**, **78**)–(0, 1 078, 75)
`3-27`	(0, 393, 25)	(**0**, **376**, **24**)–(0, 393, 24)
`3-29`	(0, 509, 67)	(**0**, **465**, **59**)–(0, 470, 63)
`4-02`	(0, 466, 50)	(**0**, **388**, **39**)–(0, 401, 54)

Table 6
Average percentage of improvement and relative standard deviation over plain `clingo` for ASP with (adaptive) LNS for the Shift Design problem.

	Shift Design
single LNS operators	7.29 ± 2.22
uniform roulette	$\mathbf{12.14} \pm 1.66$
adaptive roulette $\alpha = 0.2$	11.47 ± 0.56
adaptive roulette $\alpha = 0.4$	11.22 ± 1.24
adaptive roulette $\alpha = 0.6$	10.95 ± 2.12
adaptive roulette $\alpha = 0.8$	10.61 ± 1.69
dynamic	6.61 ± 2.02

need to be connected to a limited number of other units such that all communication requirements are fulfilled. The objective is to minimise the number of control units that are used.

The problem served a benchmark for ASP solvers in the past.[11] Although others have demonstrated that ASP solvers can be very effective for this problem when combined with the right heuristics [53], our goal here is a different one. In the spirit of declarative problem solving, we use an encoding for PUP that is as intuitive and close to the problem statement as possible. We are interested in comparing the performance of `clingo` without heuristics on such a direct encoding against ALASPO, hence letting the solver take care of the optimisations instead of implemented more advanced heuristics.

In particular, we use a slightly modified version of the encoding introduced by Aschinger et al. [54], shown in Fig. 8. We omit an ordering constraint requiring communication units to be used sequentially that was added to improve performance.

For our experiments, we set the time limit to 300 seconds and used 112 problem instances of varying difficulty obtained from the 2011 ASP competition. From the 112 instances, we only consider the ones for which `clingo` could find a solution within the time limit, which left us with 78 instances. The `clingo` configuration that worked best was

```
--opt-strat=usc,3 --configuration=trendy
```

and is therefore the baseline for our further evaluations.

We compared `clingo` against ALASPO with different adaptive strategies for its default portfolio. The best results were obtained by using a custom-defined declarative neighbourhood, which is shown in Fig. 9. The idea behind this neighbourhood definition is that if we remove a unit from a solution, then all connection links from that unit to sensors, zones, and other units are removed as well.

The results are shown in Fig. 10 as box plots that visualise the relative difference to the best solutions (median, 5, 95 percentiles) for `clingo` and ALASPO. For ALASPO, the results are the averages over five runs. The improvement in the objective value relative to the time needed is depicted in Fig. 11. We obtain the best performance using ALASPO with the declarative neighbourhood, but ALASPO with its default portfolio works already very well and finds better solutions than plain `clingo` for 55 instances and solutions of the same quality for the rest. Overall, our experiments show the we can boost the performance of an ASP solver on a direct encoding by using LNS for optimisation. This helps to alleviate the burden on users of optimising ASP encodings manually.

[11] The official problem suites for the 2011 ASP competition can be found at https://www.mat.unical.it/aspcomp2011/OfficialProblemSuite.

```
zone(Z) :- zone2sensor(Z, _).
sensor(S) :- zone2sensor(_, S).

1 { unit2zone(U, Z): unit(U) } 1 :- zone(Z).
1 { unit2sensor(U, S): unit(U) } 1 :- sensor(S).

:- unit(U), unitCap(C), C + 1 { unit2zone(U, Z): zone(Z) }.
:- unit(U), unitCap(C),
   C + 1 { unit2sensor(U, S): sensor(S) }.

partnerunits(U, P) :- unit2zone(U, Z), zone2sensor(Z, S),
                      unit2sensor(P, S), U != P.
partnerunits(U, P) :- partnerunits(P, U), unit(U), unit(P).

:- unit(U), interUnitCap(C),
   C+1 { partnerunits(U, P): unit(P) }.

unitUsed(U) :- unit2zone(U, Z).
unitUsed(U) :- unit2sensor(U, S).
L { unitUsed(X): unit(X) } U :- lower(L), upper(U).
#minimize{ 1,X: unitUsed(X) }.
```

Fig. 8. Encoding of the Partner Units Problem.

```
_lns_select(U) :- unitUsed(U).
_lns_fix(unit2zone(U, Z), U) :- _lns_select(U),
                                zone(Z), unit2zone(U, Z).
_lns_fix(unit2sensor(U, S), U) :- _lns_select(U),
                                  sensor(S), unit2sensor(U, S).
_lns_fix(partnerunits(U, P), U) :- _lns_select(U), _lns_select(P),
                                   U != P, partnerunits(U, P).
_lns_fix(partnerunits(P, U), U) :- _lns_select(U), _lns_select(P),
                                   U != P, partnerunits(P, U).
```

Fig. 9. Declarative specification of a custom Partner Units Problem neighbourhood.

7.3. Parallel machine scheduling

As a more advanced application of LNS with `clingo-dl`, we deal with a parallel machine scheduling problem with sequence-dependent setup times, release dates, and machine capabilities from an industrial semiconductor production plant. In recent work [35], an ASP approach with difference logic has been introduced for this problem. Further and quite considerable improvements are possible with LNS and ASP. The ASP encoding we used is an improved version of the original.[12]

The objective is to assign jobs to machines such that the makespan, i.e., the total execution length, of the schedule is minimal. Solutions are represented via predicate `assigned/2`, which defines the machine assignment, and `next/3`, which defines a total order of jobs on the machines.

Relaxation on random visible atoms is not suitable here, since dependencies between atoms make it likely that removed atoms will be reconstructed. We consider two neighbourhoods for this problem:

- (*job*) select a number of jobs and fix any atoms over `assigned/2` and `next/3` that mention this job; and
- (*machine*) select a number of machines and relax all jobs on them.

For (*machine*), we ensure that the machine that determines the makespan in the current solution is always part of the selection, as otherwise improvements are impossible. Similarly, should the first neighbourhood select no job from the machine determining the makespan, we remove an arbitrary job from the selection and add a random job from that machine. At each LNS step, we choose either the *job* or the *machine* neighbourhood at random.

In principle, we could use `clingo-dl` to obtain an initial solution. However, it is beneficial to construct one using a simple greedy heuristic: starting from an empty schedule; while some job is unassigned, we pick one with minimal release date and put it on a machine such that the makespan of the partial schedule increases the least. This algorithm always produces a fairly good-quality feasible schedule.

[12] The encoding can be found at https://doi.org/10.5281/zenodo.11058964. We also included it in the appendix.

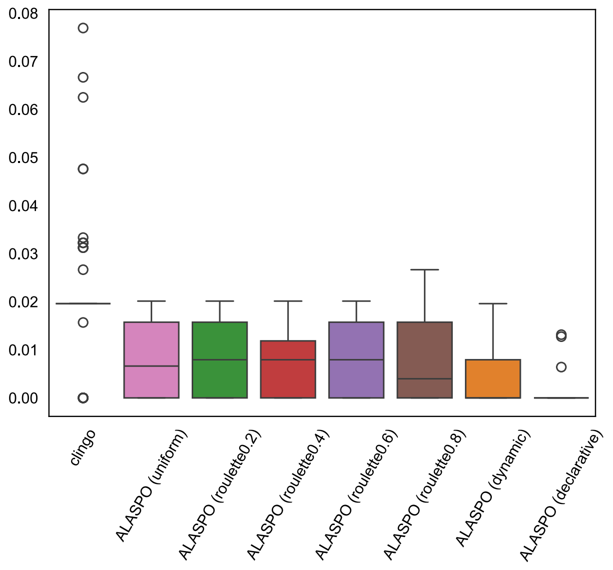


Fig. 10. Relative differences to best solution for `clingo` and ALASPO for the Partner Units Problem.

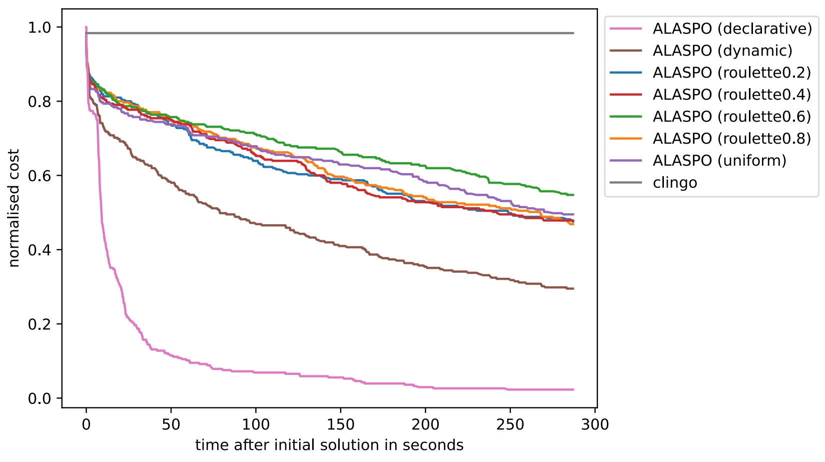


Fig. 11. Mean solution quality over time for `clingo` and ALASPO for the Partner Units Problem. The x-axis contains the time from the first found solution, whereas the y-axis gives the normalised mean cost over all runs and instances in the benchmark.

The implementation in ALASPO is easy to extend by overloading predefined member functions for obtaining an initial solution and defining the neighbourhood. For quality control, solutions encountered are also verified in Python.

Again, we compare plain `clingo-dl` with ALASPO. The time limit is 15 minutes overall and 15 seconds for LNS steps. For the neighbourhoods, we fixed 80% of the jobs or all but two of the machines, respectively. In the baseline setting, we handpicked a relaxation size of 20% of the jobs. The other tested configurations use different sizes of the custom operator as specified in the portfolio.

The results for the 500 instances from the original paper are visualised in Fig. 12 as box plots and given numerically in Table 7. We show the median as well as 5 and 95 percentiles of the relative difference to the best solutions for `clingo-dl` and ALASPO with and without the construction heuristic. For both variants of ALASPO, all configurations were run 5 times for each instance and the average was taken as the result.

Without the construction heuristic, the tailored LNS approach and all the ALASPO strategies except for dynamic achieve very similar performance over the whole benchmark. Fig. 12 shows that the tailored LNS achieves a slightly better median relative difference, whereas Table 7 gives one of the adaptive roulette strategies as the one with the highest average improvement over `clingo`. However, the differences are marginal.

In the comparison of all approaches using the construction heuristic, ALASPO with the dynamic strategy clearly performs best followed by the handpicked LNS and the other strategies.

Fig. 13 shows an analysis of the solution quality over time for each of the ALASPO strategies, `clingo-dl`, and the LNS baseline. Without the greedy initialisation, the ALASPO approaches behave quite similarly and gradually improve the cost, whereas `clingo-`

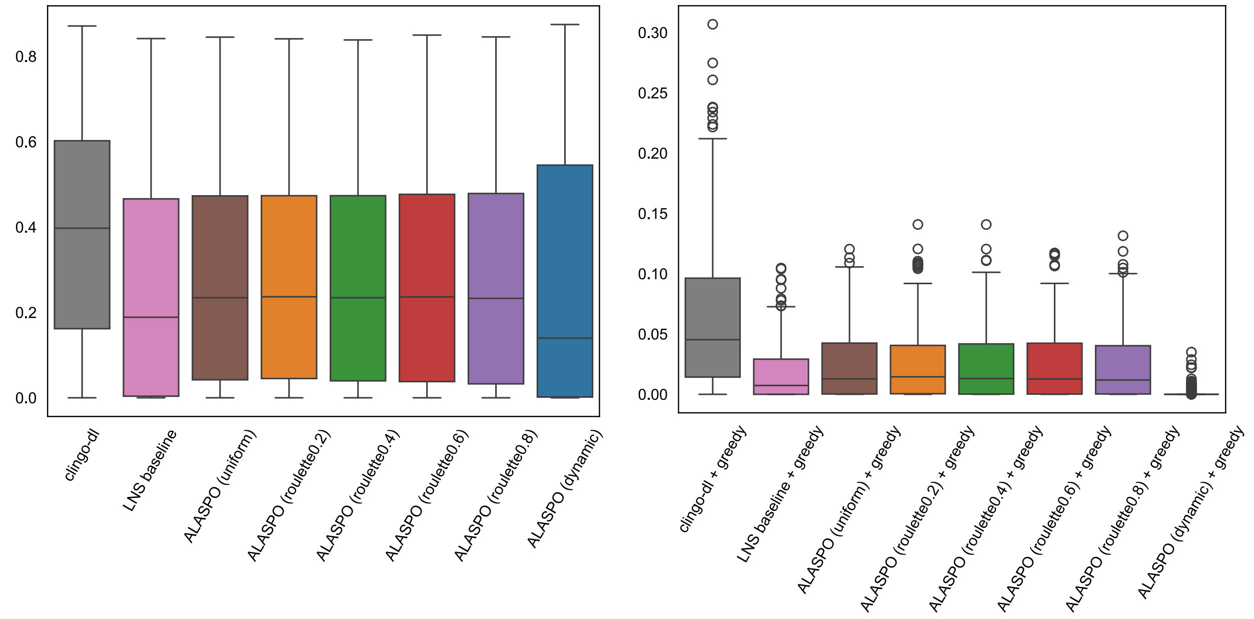


Fig. 12. Relative differences to best solution for `clingo-dl` and ALASPO for Parallel Machine Scheduling without (left) and with (right) greedy initialisation.

Table 7
Average percentage of improvement and relative standard deviation of ALASPO over `clingo` with construction heuristic (left column) and `clingo` without greedy construction heuristic (right column) for Parallel Machine Scheduling.

	no greedy init	with greedy init
single LNS operators	19.65 ± 0.77	3.45 ± 0.73
uniform roulette	21.16 ± 0.73	3.19 ± 0.73
adaptive roulette $\alpha = 0.2$	20.59 ± 0.72	2.88 ± 0.73
adaptive roulette $\alpha = 0.4$	**21.18** ± 0.72	2.88 ± 0.73
adaptive roulette $\alpha = 0.6$	20.96 ± 0.72	2.78 ± 0.73
adaptive roulette $\alpha = 0.8$	20.69 ± 0.73	2.82 ± 0.73
dynamic	11.45 ± 0.81	**5.66** ± 0.73

`dl` does not improve much after 200 seconds. With greedy initialisation, `clingo-dl` plateaus much earlier and fails to improve significantly over the heuristic initial solution. In difference, ALASPO keeps finding better solutions and especially the dynamic strategy quickly finds high quality solutions which are better than the ones found by the rest. Nonetheless, this strategy still gradually improves the cost until the time limit is met.

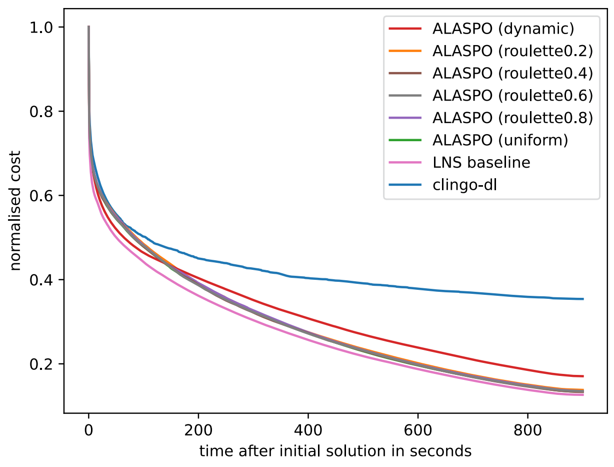
Lastly, in order to give a sense of the improvement ALASPO achieves upon `clingo-dl` for Parallel Machine Scheduling, Fig. 14 compares the best schedules, i.e., solutions, obtained by ALASPO with the greedy initialisation and the dynamic strategy and plain `clingo-dl` for a medium sized instance. The objective is to minimise the makespan which is the maximum completion time of all jobs. It should be easy to see that the schedule produced by ALASPO is much better in terms of makespan than `clingo-dl` giving an improvement of over 42%. In practice, this results in much higher job throughput and optimises efficiency.

7.4. Test laboratory scheduling

In this section, we consider the *Test Laboratory Scheduling Problem* (TLSP), more specifically, the TLSP-S problem, which is a variant of TLSP. A complete and formal definition of both problems was given by Mischek and Musliu [51,55]; an encoding of TLSP-S using constraint answer-set programming and `clingcon` can be found in previous work [19]. We confine here to an informal description of the problem.

Briefly, each instance of TLSP-S consists of a set of *projects*, each of which contains *jobs* to be scheduled. Each job has several properties, like a time window in which it needs to be scheduled, a set of possible execution modes, a duration, resource requirements, as well as linked and preceding jobs.

The goal of TLSP-S is to find an assignment of a mode, a time slot interval, and resources to each job such that all constraints are fulfilled and the objective function, which is the weighted sum of the violations of five soft constraints, is minimised.



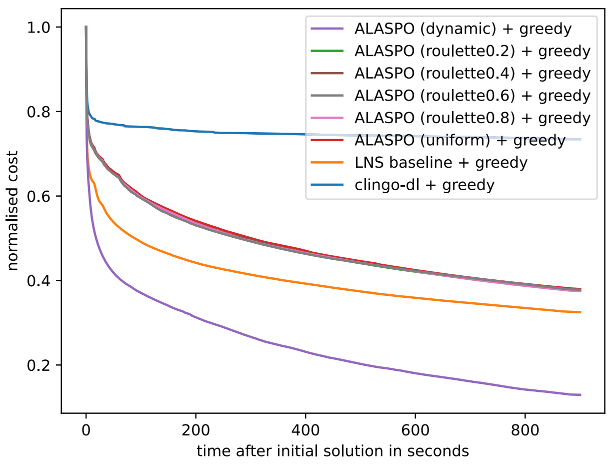


Fig. 13. Mean solution quality over time for `clingo-dl` and ALASPO for Parallel Machine Scheduling without (left) and with (right) greedy initialisation. The x-axis contains the time from the first found solution, whereas the y-axis gives the normalised mean cost over all runs and instances in the benchmark.

An LNS approach for TLSP-S utilising ASP has been considered in previous work [19] already. However, the approach there is different from our framework, since it uses the ASP solver as a black-box. At each iteration of the LNS loop, a number of projects are relaxed and a sub-instance is generated reflecting this. This sub-instance is then grounded, solved, and optimised by the ASP solver until a given time limit is reached. Notably, the generation of this sub-instance requires quite a bit of coding, whereas we can use the same ASP encoding as is and utilise LNS with ALASPO rather easily.

Since minimisation of integer variables in `clingcon` works differently than minimisation in plain ASP, the optimisation strategies developed for `clingo` cannot be used directly on them. To be able to experiment with different `clingo` optimisation strategies, we extended the original encoding so that standard ASP minimisation directives can be used. This is done essentially by translating the integer variables `clingcon` back to `clingo` atoms. The complete encoding can be found in the appendix.

In our encodings, the solutions are represented as follows. We use the predicates `start/2`, `modeAssign/2`, `empAssign/2`, `wbAssign/2` and `equipAssign/2` to represent solutions. The first attribute is always the id of a job and the second the start time, mode, employee, workbench, and piece of equipment, respectively. Obviously, each job only gets assigned one start time and one mode, but for the resources, multiple assignments are possible.

Using the same benchmark set of 33 instances as Danzinger et al. [56,57], which includes two additional real-life instances that have not been used in previous work [19], we found that using standard ASP minimisation with options

```
--opt-strat=usc,3 --configuration=tweety
```

works best for the problem. Hence, we used it as a baseline for comparison and also for the solver setting in ALASPO.

Table 8 shows the results of the comparison between `clingcon` and ALASPO using different strategies. The time limit was 30 minutes for each instance for both `clingcon` and ALASPO. For ALASPO, the average of 5 runs is reported. The experiment shows that ALASPO improves the objective value for half of the instances, and it always yielded solutions at least as good as those found by `clingcon`; when there was no improvement, all strategies achieved the same objective value which incidentally is also the optimal solution. In fact, the optimal solution was also found for instances 17, 21, and 22. Improvements manifested in the case of higher objective values (with the exception of instance 11), resulting in significant gains of up to 33.9% (instance 29). Furthermore, it appears that the dynamic strategy usually works best, yielding the top objective values for 31 of the 33 instances and beaten only on two instances by different versions of the adaptive strategy.

We further investigated how a declarative neighbourhood vs. a neighbourhood written in Python perform on TLSP-S. The declarative neighbourhood for TLSP-S that we used is described in Fig. 15. Intuitively, this neighbourhood restricts the relaxation to projects. For each project that does not get selected, it fixes the assignments and the start time. Furthermore, we restricted the relaxation sizes by selecting between 1 and 5 projects. We employ the dynamic strategy and set the time limit for the reconstruction step to 30 seconds. In contrast, the neighbourhood operator written in Python selects a random but overlapping combination of projects of the current neighbourhood size, which is a number maintained by ALASPO and can be arbitrarily large. If there are no sets of projects of the desired size, the largest possible combination is selected, and if no projects overlap at all, a single random project is chosen.

Table 9 shows a comparison of the tailored applications of ALASPO on TLSP-S with the default configuration using the dynamic strategy. The values are the absolute and the relative improvements of the objective value compared to `clingcon`, respectively. It can be seen that the neighbourhood operator written in Python (PYT) achieves the best results with the declarative neighbourhood (DEC) following closely behind. While the default configuration of ALASPO (DEF) gives overall the smallest performance improvements, it is still quite competitive with the customised versions and slightly outperforms those approaches in several cases. On average, the improvement of each of DEC, PYT, and DEF over `clingcon` is about 9% with individual relative improvements of up to 42%. Overall, this shows that through some problem-dependent tinkering, either specifying a neighbourhood declaratively or in imperative code,

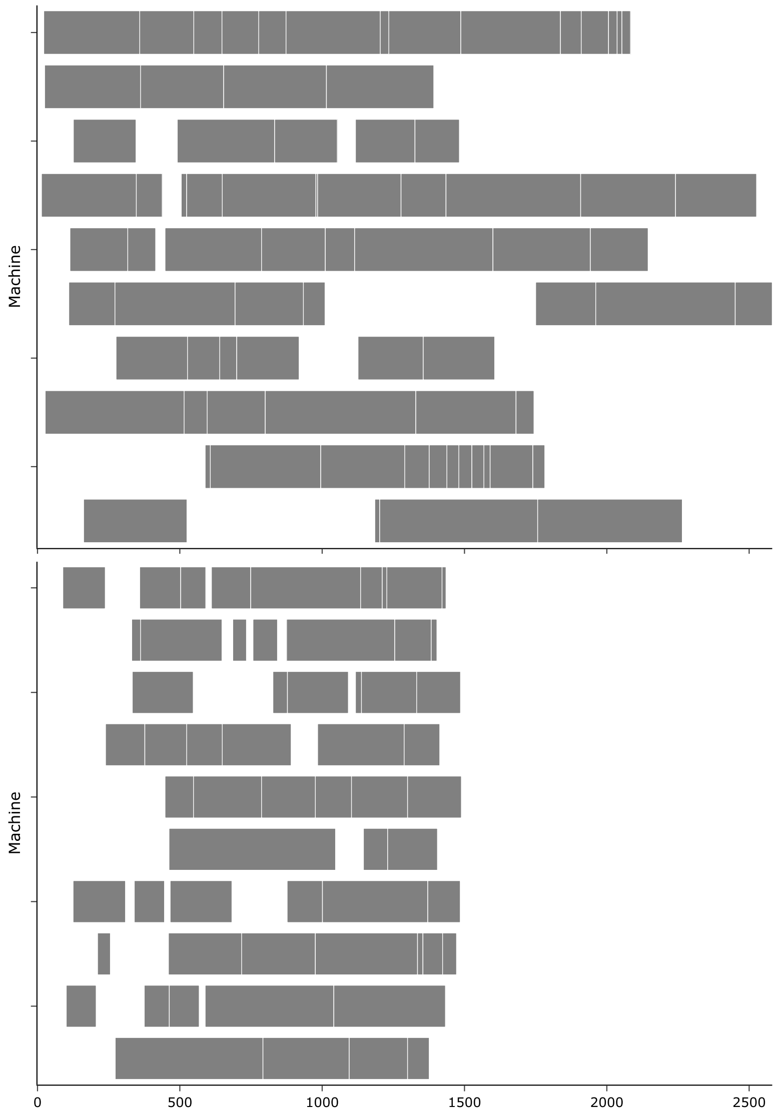


Fig. 14. Schedules obtained by ALASPO (bottom) with greedy initialisation and the dynamic strategy and plain `clingo-dl` (top) for an exemplary instance with 10 machines (`201_10_59_L`). The machines are on the y-axis, whereas the boxes represent the scheduled jobs with their respective start and completion times.

```
_lns_select(P) :- project(P).
_lns_fix(start(J, S), P) :- _lns_select(P),
                            projectAssignment(J, P),
                            start(J, S).
_lns_fix(workbenchAssign(J, W), P) :- _lns_select(P),
                                      projectAssignment(J, P),
                                      workbenchAssign(J, W).
_lns_fix(empAssign(J, E), P) :- _lns_select(P),
                                projectAssignment(J, P),
                                empAssign(J, E).
_lns_fix(equipAssign(J, E), P) :- _lns_select(P),
                                  projectAssignment(J, P),
                                  equipAssign(J, E).
_lns_fix(modeAssign(J, M), P) :- _lns_select(P),
                                 projectAssignment(J, P),
                                 modeAssign(J, M).
```

Fig. 15. Encoding of a custom TLSP-S neighbourhood.

Table 8
Objective values for `clingcon` vs. ALASPO for instances of TLSP-S. The results for ALASPO show the objective values averaged over five runs.

	`clingcon`	ALASPO					
		uniform roulette	adaptive roulette				dynamic
			$\alpha = 0.2$	$\alpha = 0.4$	$\alpha = 0.6$	$\alpha = 0.8$	
1	**98.0**	**98.0**	**98.0**	**98.0**	**98.0**	**98.0**	**98.0**
2	**149.0**	**149.0**	**149.0**	**149.0**	**149.0**	**149.0**	**149.0**
3	**73.0**	**73.0**	**73.0**	**73.0**	**73.0**	**73.0**	**73.0**
4	**105.0**	**105.0**	**105.0**	**105.0**	**105.0**	**105.0**	**105.0**
5	**283.0**	**283.0**	**283.0**	**283.0**	**283.0**	**283.0**	**283.0**
6	**307.0**	**307.0**	**307.0**	**307.0**	**307.0**	**307.0**	**307.0**
7	**162.0**	**162.0**	**162.0**	**162.0**	**162.0**	**162.0**	**162.0**
8	**310.0**	**310.0**	**310.0**	**310.0**	**310.0**	**310.0**	**310.0**
9	**501.0**	**501.0**	**501.0**	**501.0**	**501.0**	**501.0**	**501.0**
10	**856.0**	**856.0**	**856.0**	**856.0**	**856.0**	**856.0**	**856.0**
11	659.0	**564.0**	**564.0**	**564.0**	**564.0**	**564.0**	**564.0**
12	**656.0**	**656.0**	**656.0**	**656.0**	**656.0**	**656.0**	**656.0**
13	**340.0**	**340.0**	**340.0**	**340.0**	**340.0**	**340.0**	**340.0**
14	**420.0**	**420.0**	**420.0**	**420.0**	**420.0**	**420.0**	**420.0**
15	**1084.0**	**1084.0**	**1084.0**	**1084.0**	**1084.0**	**1084.0**	**1084.0**
16	1199.0	**1138.2**	1139.2	**1138.2**	1138.4	1138.6	**1138.2**
17	**1194.0**	**1194.0**	**1194.0**	**1194.0**	**1194.0**	**1194.0**	**1194.0**
18	1506.0	1359.6	1358.4	1358.2	1358.6	1358.2	**1358.0**
19	2571.0	1947.2	1969.6	1946.4	1947.8	1957.2	**1929.0**
20	2773.0	2130.8	2153.6	2140.6	2147.8	2150.6	**2110.4**
21	**679.0**	**679.0**	**679.0**	**679.0**	**679.0**	**679.0**	**679.0**
22	**765.0**	**765.0**	**765.0**	**765.0**	**765.0**	**765.0**	**765.0**
23	2586.0	2092.8	2114.4	2115.6	**2058.4**	2114.2	2163.4
24	1952.0	1778.4	1778.8	1780.2	1779.6	1780.4	**1776.8**
25	3150.0	2190.4	2267.0	2260.8	2199.0	2276.4	**2154.4**
26	3186.0	2708.4	2708.4	2710.6	2686.8	2691.0	**2671.8**
27	2755.0	2208.6	2221.4	2203.6	2227.0	2203.0	**2179.6**
28	2753.0	2332.2	2336.4	2335.4	2330.4	2336.4	**2328.8**
29	5851.0	4071.0	4085.4	4030.0	4139.8	4189.0	**3865.8**
30	6162.0	4881.4	4889.8	4856.8	4890.0	4879.6	**4714.4**
31	4450.0	3338.6	3285.4	3343.4	3349.0	3371.6	**3278.4**
32	3012.0	2534.6	2533.0	2556.8	2514.4	2524.4	**2510.2**
33	2665.0	2592.4	2594.8	2594.0	2593.6	**2590.8**	2591.4

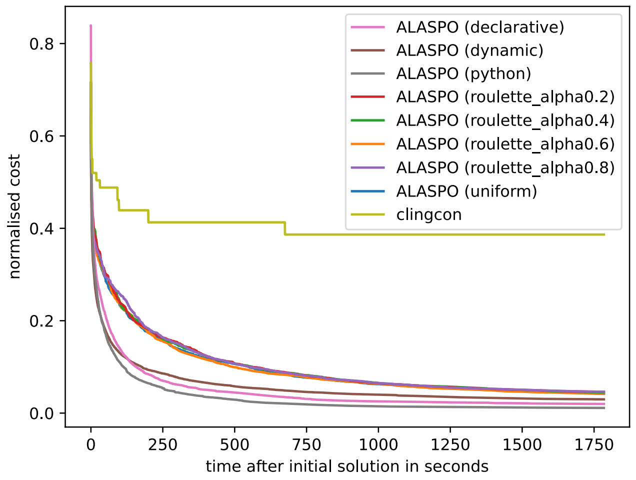


Fig. 16. Mean solution quality over time for `clingcon` and ALASPO for Test Laboratory Machine Scheduling. The x-axis contains the time from the first found solution, whereas the y-axis gives the normalised mean cost over all runs and instances in the benchmark.

the results of ALASPO can be improved. Notably, the results from the out-of-the-box solver – with the same `clingcon` parameters – are already a big improvement over `clingcon`, with average and maximal improvement of 8.8% and 33.9%, respectively.

Fig. 16 shows an analysis of the solution quality over time for each of the ALASPO strategies and `clingcon`. The plot shows that `clingcon` generally achieves its best result quite early in the search an then fails to find further improvements. ALASPO, on the

Table 9
Comparison of improvements over `clingcon` for TLSP-S using different neighbourhood operators in ALASPO (default = dynamic strategy). The entries show the average objective value of five runs minus the value achieved by `clingcon`, and the relative gain achieved.

	declarative (DEC)	Python (PYT)	default (DEF)	relative gain (DEC)	(PYT)	(DEF)
1	**0.0**	**0.0**	**0.0**	**0.0%**	**0.0%**	**0.0%**
⋮	⋮	⋮	⋮	⋮	⋮	⋮
9	**0.0**	**0.0**	**0.0**	**0.0%**	**0.0%**	**0.0%**
10	0.4	1.2	**0.0**	0.0%	0.1%	**0.0%**
11	**-95.0**	**-95.0**	**-95.0**	**-14.4%**	**-14.4%**	**-14.4%**
12	**0.0**	**0.0**	**0.0**	**0.0%**	**0.0%**	**0.0%**
13	**0.0**	**0.0**	**0.0**	**0.0%**	**0.0%**	**0.0%**
14	**0.0**	**0.0**	**0.0**	**0.0%**	**0.0%**	**0.0%**
15	0.8	**0.0**	**0.0**	0.1%	**0.0%**	**0.0%**
16	-60.4	-60.2	**-60.8**	-5.0%	-5.0%	**-5.1%**
17	0.6	**0.0**	**0.0**	0.1%	**0.0%**	**0.0%**
18	**-149.6**	-148.2	-148.0	**-9.9%**	-9.8%	-9.8%
19	-611.6	**-647.0**	-642.0	-23.8%	**-25.2%**	-25.0%
20	**-693.2**	-684.0	-662.6	**-25.0%**	-24.7%	-23.9%
21	**0.0**	**0.0**	**0.0**	**0.0%**	**0.0%**	**0.0%**
22	0.2	2.0	**0.0**	0.0%	0.3%	**0.0%**
23	-616.6	**-619.8**	-422.6	-23.8%	**-24.0%**	-16.3%
24	-155.2	**-177.0**	-175.2	-8.0%	**-9.1%**	-9.0%
25	-987.6	**-1066.8**	-995.6	-31.4%	**-33.9%**	-31.6%
26	**-604.2**	-547.4	-514.2	**-19.0%**	-17.2%	-16.1%
27	-595.0	**-603.2**	-575.4	-21.6%	**-21.9%**	-20.9%
28	-428.0	**-430.2**	-424.2	-15.5%	**-15.6%**	-15.4%
29	-2340.2	**-2460.0**	-1985.2	-40.0%	**-42.0%**	-33.9%
30	**-1680.8**	-1664.6	-1447.6	**-27.3%**	-27.0%	-23.5%
31	-1026.0	-1160.6	**-1171.6**	-23.1%	-26.1%	**-26.3%**
32	-459.4	-493.4	**-501.8**	-15.3%	-16.4%	**-16.7%**
33	-66.0	-71.2	**-73.6**	-2.5%	-2.7%	**-2.8%**

other hand, improves the solutions much more gradually, but very quickly becomes significantly better than `clingcon`. The ALASPO adaptive strategies all seem to behave quite similarly, whereas the dynamic, declarative, and custom Python approach achieve better solutions earlier in the search.

8. Related work

We next review related work on LNS for ASP optimisation and related paradigms.

8.1. Optimisation in ASP solving

ASP optimisation has seen a number of improvements in recent years [58]. In particular, unsatisfiable-core analysis [46,59] significantly boosted optimisation capabilities of ASP solvers. An unsatisfiable core is a subset of weak constraints that cannot be jointly satisfied and thus provides an underestimate of the cost of an optimal stable model. Such cores are repeatedly relaxed until an optimal solution is found. Our LNS method leverages on such techniques as they are employed during the reconstruction step. Note that then the solver will either find the optimal solution relative to the fixed neighbourhood in an LNS step, or timeout and report no solution at all. In this context, Saikko et al. [60] introduced a hybrid approach that uses an ASP solver for extracting unsatisfiable cores and integer programming to compute minimum-cost hitting sets that are used for determining upper cost bounds.

Recent work deals with modifying ASP optimisation statements using auxiliary atoms over additional rules that encode comparator networks to provide further branching points during search [61]. This has the potential to lead to exponentially smaller search spaces.

While we deal with more straight-forward optimisation statements that can be expressed with weak constraints, we also mention that there is related work on how to address more complex preferences and optimization requirements with meta-programming techniques and preference programs [62,63].

The idea of enumerating optimal answer sets has been generalised by Pajunen and Janhunen [64], who studied approaches to enumerate answer sets in the order of optimality. This allows to compute the k best answer sets and has interesting applications for, e.g., Bayesian sampling.

8.2. LNS for ASP, MIP, and CP solvers

For a general overview of combinations of exact algorithms and meta-heuristics to solve combinatorial optimisation problems, we refer to a respective survey article [65]. Within the classification introduced there, our work falls within the category of incorporating exact algorithms in meta-heuristics.

The only work that touches on LNS in the context of ASP prior to this work is the application of `clingcon` for Test Laboratory Scheduling [19]. There, `clingcon` was used as a black-box solver to find an assignment for a sub-problem within an LNS loop, without using multi-shot solving. In principle, a similar black-box approach for other ASP solvers like `wasp` [66,67] is possible, but an empowered multi-shot solving approach needs further efforts.

The use of LNS in MIP [14,15,68] and CP [16–18] is well explored. For declarative LNS neighbourhood definitions, the constraint modelling languages were extended to support solver-independent LNS [69–71]. These approaches share our goal of lifting LNS to the modelling level and also repeated solver calls are avoided. Our approach however merely requires dedicated predicates that can be defined by rules in the standard ASP modelling language without the need to extend the formalism. Also, it offers unlimited power for neighbourhood definitions by external plugins if needed.

Declarative LNS was also considered for Imperative-Declarative Programming [72], where LNS moves can be specified in a quite similar way as in our framework in predicate logic for the knowledge-base system IDP [73]. Our approach is specifically tailored to ASP and also includes strategies for adapting the LNS search during execution, which is not part of the IDP framework.

8.3. Construction and domain-specific heuristics for ASP

Another direction of improving ASP solver performance is to add domain-specific heuristics [9,10]. Dodaro et al. [9] extended the ASP solver WASP with a heuristic interface that allows to define heuristics that are applied during search. Gebser et al. [10] introduced an extension of the ASP modelling language for the Potassco ASP solvers where such search heuristics can be expressed as part of the ASP encoding. These heuristics affect which atoms are selected next during search and which truth value they get assigned and are thus quite different from the LNS meta-heuristic. However, they can be used within LNS to improve the performance of the reconstruction step and complement the LNS approach in this regard.

Gebser, Ryabokon, and Schenner [74] studied a combination of greedy algorithms with ASP. They used the greedy method to generate heuristics for accelerating an ASP solver but left the optimisation procedure unchanged. By our results, it would be of interest to see whether fruitful greedy heuristics for LNS with ASP could be (semi-)automatically constructed.

8.4. Machine learning for ASP

Other researchers have attempted to utilise machine learning in order to improve the robustness of ASP systems by selecting ASP algorithms based on classification on syntactic program features [75], or by finding the best solver configuration using static and dynamic features [8]. While these approaches can be readily utilised for ASP optimisation, in contrast to our meta-heuristic approach they use in the end complete search methods.

A direction related to automated solver selection is the idea to improve the performance of an ASP system by selecting an encoding that fits best [76], or by rewriting an encoding and selecting the rules that should be changed using machine learning techniques [77]. Naturally, optimising the encoding may also be added as a supplementary step to our approach; however, the core of our search method, on which we focus in this article, does not alter the form of the logic program. Investigating the selection and optimisation of the encoding, especially using machine learning, is however an interesting subject for future research.

9. Conclusion

We have introduced an optimisation framework for answer-set programming (ASP) that exploits large neighbourhood search (LNS) in order to iteratively improve solutions. Notably, different neighbourhoods for LNS can be seamlessly specified as part of the ASP encoding itself, by resorting to special predicates. In this way, the spirit of ASP as a declarative approach for rapid prototyping is maintained. Furthermore, since ASP is more expressive than related approaches such as mixed integer programming (MIP) or constraint programming (CP), our framework makes LNS viable even for search and optimisation problems whose decision variant is beyond NP.

Our version of LNS is adaptive so that different aspects of the meta-heuristic are adjusted on-the-fly using different strategies and feedback from the ASP solver. This considerably eases the burden on the user to find effective configurations by hand. We have presented the general adaptive LNS solver ALASPO that implements our optimisation framework and supports currently the solvers `clingo`, `clingo-dl`, and `clingcon` from the Potassco family. ALASPO relies on multi-shot solving for efficient search, supports portfolios to define different search and neighbourhood configurations, and can be used to set up LNS for ASP quickly under lean requirements. In order to make its usage more effective, we have also provided an in-depth discussion of how to use LNS optimisation for ASP. The methodological considerations cover the whole problem solving process from the beginning to the end, and they address, besides issues that are specific to ASP, also issues of a more general concern.

Finally, we have demonstrated that the meta-heuristic enrichment of adaptive LNS indeed boosts the problem solving capabilities of ASP for challenging optimisation benchmarks and real-world applications. Our experiments showed that our method of combining ASP with LNS is quite effective even in rather plain settings with a single neighbourhood operator and a fixed relaxation rate. However, better results are often achieved with more customised configurations. Furthermore, the experiments confirm that the adaptive strategies when used with reasonable portfolios that we have investigated achieve very good results most of the time. With some extra effort, the LNS solver can be customised for ASP applications by implementing problem-specific heuristics for, e.g., obtaining initial solutions. This can further boost performance as witnessed by a machine scheduling problem from the semiconductor industry.

9.1. Future work

The LNS framework that we have presented in this paper can be advanced in several directions. One such direction is to investigate and design new operators for defining neighbourhoods and for large neighbourhood search on them, respectively. For defining neighbourhoods, it appears to be suggestive to take dependencies between atoms more directly into account and to consider how frequently atoms are involved in conflicts; the latter allows us to steer search more effectively. Another direction is to advance adaptiveness. To this end, adaptive strategies for search may be constructed using reinforcement learning and solver statistics. Furthermore, learning techniques may be used to modify the underlying program, at least for the sake of efficient search, and to construct an initial solution using a heuristic [78]. As for usability, it would be desirable to add an explanation component to the framework, which offers the user information in order to support analysis of the search process and obtaining insight into it.

ALASPO is being actively developed and further improvements are planned. Besides extending the suite for predefined neighbourhood operators, further strategies for adaptiveness will be added over time, in alignment with the research directions mentioned above. Furthermore, we want to extend ALASPO to support further ASP systems like WASP [67].

CRediT authorship contribution statement

Thomas Eiter: Writing – review & editing, Writing – original draft, Supervision, Project administration, Methodology, Funding acquisition, Formal analysis, Conceptualization. **Tobias Geibinger:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Nelson Higuera Ruiz:** Writing – review & editing, Writing – original draft, Visualization, Software, Resources, Investigation, Data curation. **Nysret Musliu:** Writing – review & editing, Supervision, Methodology, Funding acquisition, Conceptualization. **Johannes Oetsch:** Writing – review & editing, Writing – original draft, Supervision, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Dave Pfliegler:** Writing – review & editing, Writing – original draft, Visualization, Software, Resources, Investigation, Data curation. **Daria Stepanova:** Writing – review & editing, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Johannes Oetsch reports financial support was provided by Bosch Center for Artificial Intelligence. Nelson Higuera Ruiz reports financial support was provided by Bosch Center for Artificial Intelligence. Tobias Geibinger reports financial support was provided by Austrian Academy of Sciences.

Data availability

URLs to access data and code are provided in the article.

Acknowledgements

This work was supported by funding from the Bosch Center for AI at Renningen, Germany and has benefited from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 101034440 (LogiCS@TUWien).

Tobias Geibinger is a recipient of a DOC Fellowship of the Austrian Academy of Sciences at the Institute of Logic and Computation at the TU Wien.

Furthermore, we would like to thank Michel Janus, Andrej Gisbrecht, and Sebastian Bayer for very helpful discussions on the scheduling problems at Bosch, Roland Kaminski who suggested an improvement for the Parallel Machine Scheduling encoding, as well as the reviewers for their insightful comments and suggestions.

Appendix A

A.1. ASP encoding for the parallel machine scheduling problem

```
{ assigned(J,M): capable(M,J) } = 1 :- job(J).
{ first(J,M): capable(M,J) } = 1 :- assigned(_,M).
{ last(J,M): capable(M,J) } = 1 :- assigned(_,M).
:- first(J,M), not assigned(J,M).
:- last(J,M), not assigned(J,M).

{next(J1,J2,M): capable(M,J1), J1 != J2} = 1 :- assigned(J2,M),
                                                not first(J2,M).
{next(J1,J2,M): capable(M,J2), J1 != J2} = 1 :- assigned(J1,M),
                                                not last(J1,M).
```

```
:- next(J1,J2,M), not assigned(J1,M).
:- next(J1,J2,M), not assigned(J2,M).
reach(J1,M) :- first(J1,M).
reach(J2,M) :- reach(J1,M), next(J1,J2,M).
:- assigned(J1,M), not reach(J1,M).

&diff{ 0 - compl(J2) } <= -(T+D+S) :- next(J1,J2,M),
            setup(J1,J2,M,S), duration(J2,M,D), release(J2,M,T).
&diff{ compl(J1) - compl(J2) } <= -(P+S) :- next(J1,J2,M),
                             setup(J1,J2,M,S), duration(J2,M,P).
&diff{ compl(J1) - makespan } <= 0 :- job(J1).
&diff{ compl(J1) - compl(J2) } <= -P :- next(J1,J2,M),
                                        duration(J2,M,P).
&diff{ 0 - compl(J1) } <= -(T+D) :- assigned(J1,M),
                              duration(J1,M,D), release(J1,M,T).

#show next/3.
#show assigned/2.
```

A.2. ASP encoding for the shift design problem

```
#const extra = 0. % change value to activate redundant
                  % constraints on shift starts
#const timeslots = days*timeslots_per_day.

day(0..days-1).
time(0..timeslots_per_day-1).
slot(0..timeslots-1).

next_time(T,(T+1) \ timeslots_per_day) :- time(T).
next_slot(S,(S+1) \ timeslots)         :- slot(S).

positive(S)   :- slot(S), required(S,R), 0 < R.
positive(S,R) :- positive(S), R = #max{ T : required(S,T) }.

excess_limit(S,M+R) :- limit(excess,M), positive(S,R).
excess_limit(S,M)   :- limit(excess,M), slot(S), not positive(S).
excess_limit        :- excess_limit(S,M).

shift_length(N,L-BL..L+AL) :- shift_length(N,L,AL,BL).
shift_start((T+AT) \ timeslots_per_day,L,BT+AT) :-
            shift_start(N,T,AT,BT), shift_length(N,L), 0 < L.
shift_start(T1,L,C-1) :- shift_start(T2,L,C),
                         next_time(T1,T2), 0 < C.
shift_start(T,L) :- shift_start(T,L,C).

shift_slot(timeslots_per_day*D+T,L) :- shift_start(T,L), day(D).

forward(S0,L,S0,L)   :- shift_slot(S0,L).
forward(S0,L,S2,C-1) :- forward(S0,L,S1,C), next_slot(S1,S2),
                        1<C.
forward(S1,C)        :- forward(S0,L,S1,C).

revisit(S,C,(C+timeslots-1)/timeslots) :- forward(S,C).

restrict(S2,1,M) :- forward(S2,1), excess_limit(S2,M).
restrict(S1,C,P) :- revisit(S1,C,F), excess_limit(S1,M),
  next_slot(S1,S2), restrict(S2,C-1,N), G=M/F, P=(G+N-|G-N|)/2.

demand(S,C,(R+F-1)/F) :- revisit(S,C,F), positive(S,R).

usage(S1,L,S2,C,M) :- forward(S1,L,S2,C), demand(S2,C,M),
                      not forward(S1,L,S2,C+timeslots).
usage(S1,L,M)      :- usage(S1,L,S2,C,M).
usage(S1,L)        :- usage(S1,L,M).
```

```

max_usage(S,L,N) :- usage(S,L), N = #max{ M: usage(S,L,M) }.
max_start(S,L,M) :- usage(S,L),
        M = #min{N: max_usage(S,L,N); P: restrict(S,L,P)}.

can_start(S,L,1..M) :- max_start(S,L,M).
can_start(S,L)      :- can_start(S,L,1).
can_start(S)        :- can_start(S,L).
max_start(S,M)      :- can_start(S),
                       M=#max{L: can_start(S,L)}.

can_run(S2,C) :- can_start(S1,L), forward(S1,L,S2,C).
can_run(S)    :- can_run(S,C).
max_run(S,M)  :- can_run(S), M = #max{ C: can_run(S,C) }.
min_run(S,N)  :- can_run(S), N = #min{ C: can_run(S,C) }.

dominate(S2,C) :- next_slot(S1,S2), can_run(S2,C),
                  max_run(S1,M), C < M.
dominate(S)    :- dominate(S,C).

initiate(S2,L)   :- next_slot(S1,S2), can_start(S2,L),
                    not can_run(S1,L+1).
initiate(S1,C+1) :- next_slot(S1,S2), initiate(S2,C),
                    dominate(S2,C).

ordering(S,C) :- initiate(S,C), not dominate(S,C).
ordering(S)   :- ordering(S,C).
continue(S,M) :- ordering(S), M = #min{ C: ordering(S,C) }.

order(S1,C1,C2) :- ordering(S1,C1), ordering(S1,C2), C1<C2,
                   C <= C1: ordering(S1,C), C < C2.
order(S1,C1,C2)     :- dominate(S1), continue(S1,C2),
                       C1 = #max{ C: dominate(S1,C) }.
order(S2,C1-1,C2-1) :- order(S1,C1,C2), next_slot(S1,S2),
                       1 < C1.

close(S2,C)      :- next_slot(S1,S2), can_run(S1,C+1),
                    max_start(S2,L), 0 < C, C < L.
close(S,C1,C2)   :- order(S,C1,C2), close(S,C2).
close(S,C1,C2)   :- order(S,C1,C2), min_run(S,C2).
close(S,C1,C2)   :- order(S,C1,C2), close(S,C2,C3).
close(S,C1,C2,I) :- order(S,C1,C2), can_start(S,C2,I),
                    not close(S,C1,C2).
close(S,C1,C2,I) :- order(S,C1,C2), close(S,C2,C3,I),
                    not close(S,C1,C2).

track(S1,C+1) :- next_slot(S1,S2), close(S2,C).
track(S1,C)   :- next_slot(S1,S2), can_run(S1,C),
                 can_start(S2,C-1).
track(S1,C)   :- next_slot(S1,S2), can_run(S1,C),
                 track(S2,C-1).

outer(S,C,P)    :- close(S,C), restrict(S,C,P).
outer(S,C,#sup) :- close(S,C), not excess_limit.

leave(S,L)    :- max_start(S,L,M), demand(S,L,N), M <= N,
                 extra != 0.
leave(S,L)    :- usage(S,L,S1,C,M), restrict(S1,C,P),
                 P <= M, extra != 0.
check(S,L)    :- can_start(S,L), extra != 0, not leave(S,L).
leave(S,L,C1) :- check(S,L), usage(S,L,S1,C1,M),
                 usage(S,L,S2,C2,N),
                 C1 < C2, M <= N.

admit(S,L,I,S1,C,M+1-I) :- check(S,L), can_start(S,L,I),
            usage(S,L,S1,C,M), I <= M, not leave(S,L,C).
```

```
admit(I,S1,C,N)      :- admit(S,L,I,S1,C,N).
trans(S,L,I)         :- admit(S,L,I,S1,C,N),
                        admit(S2,K,I,S1,C,N), L < K.
final(S,L,I)         :- check(S,L), can_start(S,L,I),
                        not trans(S,L,I).

%%%%%%%%%%%%%%%%%%
% GENERATE+TEST %
%%%%%%%%%%%%%%%%%%

{ cover(S,L,I) } :- can_start(S,L,I).
cover(S,C1,I)     :- cover(S,C2,I), close(S,C1,C2).
cover(S,C1,I)     :- cover(S,C2,I), close(S,C1,C2,I).
cover(S2,C-1,I)   :- cover(S1,C,I), next_slot(S1,S2),
                     min_run(S2,N), N < C.
cover(S2,C1,K)    :- cover(S2,C2,I), next_slot(S1,S2),
     order(S2,C1,C2), outer(S2,C1,P), run(S1,C1+1,J),
    K = I+J, K <= P.
:- cover(S2,C2,P+1-J), next_slot(S1,S2), order(S2,C1,C2),
   close(S2,C1), restrict(S2,C1,P), run(S1,C1+1,J).
:- cover(S,C,I), 1 < I, not cover(S,C,I-1).
:- positive(S,R), min_run(S,N), limit(shortage,M), M < R,
   not cover(S,N,R-M).

run(S,L,I,1)    :- can_start(S,L,I), cover(S,L,I),
                   not cover(S,C,I): order(S,L,C).
run(S,L,I,N+1) :- run(S,L,I+1,N), 0 < I,
                   not cover(S,C,I): order(S,L,C).

run(S,L,N)   :- run(S,L,I,N).
run(S2,C,N) :- run(S1,C+1,N), next_slot(S1,S2), track(S2,C).
:- run(S,C,N), 1 < N, not run(S,C,N-1).

start(S2,L,N) :- can_start(S2,L,N), next_slot(S1,S2),
                 run(S2,L,N), not run(S1,L+1,N).

clear(S2,C1,I)   :- admit(I,S2,C1,N),
                    not cover(S2,C2,N): order(S2,C1,C2).
clear(S1,C+1,I) :- next_slot(S1,S2), clear(S2,C,I),
                    not final(S2,C,I).
:- check(S,L), start(S,L,I), not clear(S,L,I).

%%%%%%%%%%%%
% OPTIMIZE %
%%%%%%%%%%%%

:~ cover(S,N,I), min_run(S,N),
not positive(S), optimize(excess,W,P). [ W@P,excess,S,I ]
:~ cover(S,N,I), min_run(S,N),
   positive(S,R), R < I,
                 optimize(excess,W,P). [ W@P,excess,S,I ]

:~ positive(S,R), min_run(S,N),
   K = #min{ R; M: limit(shortage,M) },
   I = R+1-K..R, not cover(S,N,I),
   optimize(shortage,W,P). [ W@P,shortage,S,I ]

:~ start(S,L,N),
   optimize(selected_shift,W,P).
        [ W@P,selected_shift,S \ timeslots_per_day,L ]

% Optimization criteria: Property, Weight, Level
optimize(selected_shift,1,1).
optimize(shortage,1,3).
optimize(excess,1,2).
```

```

%%%%%%%%%%%
% DISPLAY %
%%%%%%%%%%%

#show start/3.
```

A.3. ASP encoding for the test laboratory scheduling problem

```
&dom{R..D} = start(J) :- job(J),
                         release(J, R), deadline(J, D).
&dom{R..D} = end(J) :- job(J),
                       release(J, R), deadline(J, D).

1 {modeAssign(J, M) : modeAvailable(J, M)} 1 :- job(J).

duration(J, T) :- job(J), modeAssign(J, M),
                  durationInMode(J, M, T).

&sum{end(J); -start(J)} = T :- job(J), duration(J, T).

&sum{start(J)} >= end(K) :- job(J), job(K), precedence(J, K).

&sum{start(J)} = 0 :- job(J), started(J).

1 {workbenchAssign(J,W): workbenchAvailable(J,W)} 1 :- job(J),
                                         workbenchRequired(J).

R {empAssign(J, E): employeeAvailable(J, E)} R :- job(J),
                                        modeAssign(J, M),
                                 requiredEmployees(M, R).

R {equipAssign(J,E): equipmentAvailable(J,E), group(E,G)} R :-
                       job(J),
                       group(_, G),
                       requiredEquipment(J,G, R).

:- job(J), job(K), linked(J, K), empAssign(J, E),
   not empAssign(K, E).

precedence(J, K), precedence(K, J) :- job(J), job(K),
                                      workbenchAssign(J, W),
                                      workbenchAssign(K, W),
                                      J < K.
precedence(J, K), precedence(K, J) :- job(J), job(K),
                                      empAssign(J, E),
                                      empAssign(K, E), J < K.
precedence(J, K), precedence(K, J) :- job(J), job(K),
                                      equipAssign(J, E),
                                      equipAssign(K, E),
                                      J < K.

start(J,S) :- job(J), &sum{start(J)} = S, S = R..D,
              deadline(J,D),
              release(J, R).
&sum{start(J)} = S :- start(J, S).

#minimize{ 1,E,J,s2 : job(J), empAssign(J, E),
                                not employeePreferred(J, E) }.

#minimize{ 1,E,P,s3 : project(P), empAssign(J, E),
                                projectAssignment(J, P) }.

&sum{delay(J); T} = end(J)  :- job(J), due(J, T),
                             &sum{end(J); -T} > 0.
```

```
&sum{delay(J)}=0 :- job(J), due(J, T), &sum{end(J); -T} <= 0.

delay(J,T) :-  job(J), &sum{delay(J)} = T, T = 0..M,
                         M = #max{D: deadline(J,D)}.
#minimize{ T,J,s4 : delay(J,T), job(J)}.

&dom{0..H} = projectStart(P) :- project(P), horizon(H).
&dom{0..H} = projectEnd(P)    :- project(P), horizon(H).
1 {firstJob(J) : job(J), projectAssignment(J, P)} 1  :-
                                                  project(P).
&sum{projectStart(P)} = start(J)   :- firstJob(J),
                                     projectAssignment(J, P).
&sum{projectStart(P)} <= start(J) :- job(J),
                                     projectAssignment(J, P).
1 {lastJob(J): job(J), projectAssignment(J,P)} 1 :- project(P).
&sum{projectEnd(P)} = end(J)   :- lastJob(J),
                                 projectAssignment(J, P).
&sum{projectEnd(P)} >= end(J) :- job(J),
                                 projectAssignment(J, P).
&sum{projectEnd(P)-projectStart(P): project(P)} = projectDelay.
projectDelay(D) :-
        &sum{projectDelay} = D, D = 0..H*C, horizon(H),
        C = #count{ P: project(P) }.
#minimize{D,s5 : projectDelay(D)}.

#show modeAssign/2.
#show workbenchAssign/2.
#show empAssign/2.
#show equipAssign/2.
#show start/2.
```

References

[1] T. Eiter, T. Geibinger, N. Higuera Ruiz, N. Musliu, J. Oetsch, D. Stepanova, Large-neighbourhood search for optimisation in answer-set solving, in: Proceedings of the 36th AAAI Conference on Artificial Intelligence (AAAI 2022), Proc. AAAI Conf. Artif. Intell. 36 (5) (2022) 5616–5625.

[2] T. Eiter, T. Geibinger, N. Higuera Ruiz, N. Musliu, J. Oetsch, D. Stepanova, ALASPO: an adaptive large-neighbourhood ASP optimiser, in: Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning (KR 2022), vol. 19(1), 2022, pp. 565–569.

[3] G. Brewka, T. Eiter, M. Truszczyński, Answer set programming at a glance, Commun. ACM 54 (12) (2011) 92–103.

[4] V. Lifschitz, Answer Set Programming, Springer, 2019.

[5] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer set solving in practice, Synth. Lect. Artif. Intell. Mach. Learn. 6 (3) (2012) 1–238.

[6] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Mag. 37 (3) (2016) 53–68.

[7] A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E.C. Teppan, Industrial applications of answer set programming, Künstl. Intell. 32 (2–3) (2018) 165–176.

[8] H. Hoos, M. Lindauer, T. Schaub, claspfolio 2: advances in algorithm selection for answer set programming, Theory Pract. Log. Program. 14 (4–5) (2014) 569–585.

[9] C. Dodaro, P. Gasteiger, N. Leone, B. Musitsch, F. Ricca, K. Shchekotykhin, Combining answer set programming and domain heuristics for solving hard industrial problems (application paper), Theory Pract. Log. Program. 16 (5–6) (2016) 653–669.

[10] M. Gebser, B. Kaufmann, J. Romero, R. Otero, T. Schaub, P. Wanko, Domain-specific heuristics in answer set programming, in: Proceedings of the 27th AAAI Conference on Artificial Intelligence (AAAI 2013), AAAI Press, 2013, pp. 350–356.

[11] Y. Dimopoulos, A. Sideris, Towards local search for answer sets, in: Proceedings of the 18th International Conference on Logic Programming (ICLP 2002), in: Lecture Notes in Computer Science, vol. 2401, Springer, 2002, pp. 363–377.

[12] P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, in: Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP 1998), in: LNCS, vol. 1520, Springer, 1998, pp. 417–431.

[13] D. Pisinger, S. Ropke, Large neighborhood search, in: Handbook of Metaheuristics, Springer, 2010, pp. 399–419.

[14] E. Danna, E. Rothberg, C.L. Pape, Exploring relaxation induced neighborhoods to improve MIP solutions, Math. Program. 102 (1) (2005) 71–90.

[15] E. Rothberg, An evolutionary algorithm for polishing mixed integer programming solutions, INFORMS J. Comput. 19 (4) (2007) 534–541.

[16] L. Perron, P. Shaw, V. Furnon, Propagation guided large neighborhood search, in: Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP 2004), in: LNCS, vol. 3258, Springer, 2004, pp. 468–481.

[17] T. Berthold, S. Heinz, M.E. Pfetsch, S. Vigerske, Large neighborhood search beyond MIP, in: Proceedings of the 9th Metaheuristics International Conference (MIC 2011), 2011, pp. 51–60.

[18] G. Björdal, P. Flener, J. Pearson, P.J. Stuckey, G. Tack, Solving satisfaction problems using large-neighbourhood search, in: Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming (CP 2020), in: LNCS, vol. 12333, Springer, 2020, pp. 55–71.

[19] T. Geibinger, F. Mischek, N. Musliu, Constraint logic programming for real-world test laboratory scheduling, in: Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI 21), AAAI Press, 2021, pp. 6358–6366.

[20] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, ACM Comput. Surv. 33 (3) (2001) 374–425.

[21] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, F. Scarcello, The DLV system for knowledge representation and reasoning, ACM Trans. Comput. Log. 7 (3) (2006) 499–562.

[22] S. Ropke, D. Pisinger, An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows, Transp. Sci. 40 (4) (2006) 455–472.

[23] P. Laborie, D. Godard, Self-adapting large neighborhood search: application to single-mode scheduling problems, in: Proceedings of the 3rd Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA 2007), 2007, pp. 276–284.

[24] C. Thomas, P. Schaus, Revisiting the self-adaptive large neighborhood search, in: Proceedings of the 15th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2018), in: LNCS, vol. 10848, Springer, 2018, pp. 557–566.
[25] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory Pract. Log. Program. 19 (1) (2019) 27–82.
[26] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, P. Wanko, Theory solving made easy with clingo 5, in: Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016), in: OASIcs, vol. 52, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2016, 2.
[27] T. Janhunen, R. Kaminski, M. Ostrowski, S. Schellhorn, P. Wanko, T. Schaub, Clingo goes linear constraints over reals and integers, Theory Pract. Log. Program. 17 (5–6) (2017) 872–888.
[28] M. Banbara, B. Kaufmann, M. Ostrowski, T. Schaub, Clingcon: the next generation, Theory Pract. Log. Program. 17 (4) (2017) 408–461.
[29] W. Harvey, CSPLib problem 010: social golfers problem, http://www.csplib.org/Problems/prob010, 2002.
[30] E.L. Lawler, The traveling salesman problem: a guided tour of combinatorial optimization, Wiley-Intersci. Ser. Discrete Math. Optim. (1985).
[31] H. Simonis, Sudoku as a constraint problem, in: CP Workshop on Modeling and Reformulating Constraint Satisfaction Problems, vol. 12, 2005, pp. 13–27.
[32] M. Cadoli, T. Eiter, G. Gottlob, Default logic as a query language, IEEE Trans. Knowl. Data Eng. 9 (3) (1997) 448–463.
[33] M. Abseher, M. Gebser, N. Musliu, T. Schaub, S. Woltran, Shift design with answer set programming, Fundam. Inform. 147 (1) (2016) 1–25.
[34] A.A. Falkner, A. Haselböck, G. Schenner, H. Schreiner, Modeling and solving technical product configuration problems, Artif. Intell. Eng. Des. Anal. Manuf. 25 (2) (2011) 115–129.
[35] T. Eiter, T. Geibinger, N. Musliu, J. Oetsch, P. Skocovsky, D. Stepanova, Answer-set programming for lexicographical makespan optimisation in parallel machine scheduling, in: Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning (KR 2021), 2021, pp. 280–290.
[36] D. Pisinger, S. Ropke, A general heuristic for vehicle routing problems, Comput. Oper. Res. 34 (8) (2007) 2403–2435.
[37] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-core-2 input language format, Theory Pract. Log. Program. 20 (2) (2020) 294–309.
[38] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Gener. Comput. 9 (3–4) (1991) 365–385.
[39] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, 2022.
[40] M.M. El-Kholany, M. Gebser, K. Schekotihin, Problem decomposition and multi-shot ASP solving for job-shop scheduling, Theory Pract. Log. Program. 22 (4) (2022) 623–639.
[41] G. Francescutto, K. Schekotihin, M.M. El-Kholany, Solving a multi-resource partial-ordering flexible variant of the job-shop scheduling problem with hybrid ASP, in: Proceedings of the 17th European Conference on Logics in Artificial Intelligence (JELIA 2021), in: LNCS, vol. 12678, Springer, 2021, pp. 313–328.
[42] D. Abels, J. Jordi, M. Ostrowski, T. Schaub, A. Toletti, P. Wanko, Train scheduling with hybrid ASP, in: Proceedings of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019), in: LNCS, vol. 11481, Springer, 2019, pp. 3–17.
[43] M. Balduccini, Y. Lierler, Constraint answer set solver EZCSP and why integration schemas matter, Theory Pract. Log. Program. 17 (4) (2017) 462–515.
[44] Y. Lierler, Relating constraint answer set programming languages and algorithms, Artif. Intell. 207 (2014) 1–22.
[45] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-criteria optimization in answer set programming, in: Technical Communications of the 27th International Conference on Logic Programming (ICLP 2011), in: LIPIcs, vol. 11, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2011, pp. 1–10.
[46] B. Andres, B. Kaufmann, O. Matheis, T. Schaub, Unsatisfiability-based optimization in clasp, in: Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012), in: LIPIcs, vol. 17, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2012, pp. 211–221.
[47] F. Calimeri, G. Ianni, F. Ricca, The third open answer set programming competition, Theory Pract. Log. Program. 14 (1) (2014) 117–135.
[48] M. Alviano, F. Calimeri, G. Charwat, M. Dao-Tran, C. Dodaro, G. Ianni, T. Krennwallner, M. Kronegger, J. Oetsch, A. Pfandler, J. Pührer, C. Redl, F. Ricca, P. Schneider, M. Schwengerer, L.K. Spendier, J.P. Wallner, G. Xiao, The fourth answer set programming competition: preliminary report, in: Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013), in: LNCS, vol. 8148, Springer, 2013, pp. 42–53.
[49] F. Calimeri, M. Gebser, M. Maratea, F. Ricca, Design and results of the fifth answer set programming competition, Artif. Intell. 231 (2016) 151–181.
[50] T. Eiter, G. Gottlob, On the computational cost of disjunctive logic programming: propositional case, Ann. Math. Artif. Intell. 15 (3–4) (1995) 289–323.
[51] F. Mischek, N. Musliu, A local search framework for industrial test laboratory scheduling, Ann. Oper. Res. 302 (2) (2021) 533–562, https://doi.org/10.1007/S10479-021-04007-1.
[52] E.C. Teppan, On the complexity of the partner units decision problem, Artif. Intell. 248 (2017) 112–122.
[53] C. Dodaro, P. Gasteiger, N. Leone, B. Musitsch, F. Ricca, K. Schekotihin, Combining answer set programming and domain heuristics for solving hard industrial problems (application paper), Theory Pract. Log. Program. 16 (5–6) (2016) 653–669.
[54] M. Aschinger, C. Drescher, G. Friedrich, G. Gottlob, P. Jeavons, A. Ryabokon, E. Thorstensen, Optimization methods for the partner units problem, in: Proceedings of the 8th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2011), in: Lecture Notes in Computer Science, vol. 6697, Springer, 2011, pp. 4–19.
[55] F. Mischek, N. Musliu, The test laboratory scheduling problem, Technical report, Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, TU Wien, 2018, CD-TR 2018/1.
[56] P. Danzinger, T. Geibinger, F. Mischek, N. Musliu, Solving the test laboratory scheduling problem with variable task grouping, in: Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS 2020), AAAI Press, 2020, pp. 357–365.
[57] P. Danzinger, T. Geibinger, D. Janneau, F. Mischek, N. Musliu, C. Poschalko, A system for automated industrial test laboratory scheduling, ACM Trans. Intell. Syst. Technol. 14 (1) (2023) 3.
[58] M. Alviano, C. Dodaro, J. Marques-Silva, F. Ricca, Optimum stable model search: algorithms and implementation, J. Log. Comput. 30 (4) (2020) 863–897.
[59] M. Alviano, C. Dodaro, F. Ricca, A MaxSAT algorithm using cardinality constraints of bounded size, in: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI 2015), AAAI Press, 2015, pp. 2677–2683.
[60] P. Saikko, C. Dodaro, M. Alviano, M. Järvisalo, A hybrid approach to optimization in answer set programming, in: Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR 2018), AAAI Press, 2018, pp. 32–41.
[61] J. Bomanson, T. Janhunen, Boosting answer set optimization with weighted comparator networks, Theory Pract. Log. Program. 20 (4) (2020) 512–551.
[62] M. Gebser, R. Kaminski, T. Schaub, Complex optimization in answer set programming, Theory Pract. Log. Program. 11 (4–5) (2011) 821–839.
[63] G. Brewka, J.P. Delgrande, J. Romero, T. Schaub, asprin: customizing answer set preferences without a headache, in: Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI 2015), AAAI Press, 2015, pp. 1467–1474.
[64] J. Pajunen, T. Janhunen, Solution enumeration by optimality in answer set programming, Theory Pract. Log. Program. 21 (6) (2021) 750–767.
[65] J. Puchinger, G.R. Raidl, Combining metaheuristics and exact algorithms in combinatorial optimization: a survey and classification, in: Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach: Proceedings of the 1st International Work-Conference on the Interplay Between Natural and Artificial Computation (IWINAC 2005), Part II, in: LNCS, vol. 3562, Springer, 2005, pp. 41–53.
[66] C. Dodaro, F. Ricca, The external interface for extending WASP, Theory Pract. Log. Program. 20 (2) (2020) 225–248.
[67] M. Alviano, C. Dodaro, N. Leone, F. Ricca, Advances in WASP, in: Proceeding of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (ICLP 2015), LNCS, vol. 9345, Springer, 2015, pp. 40–54.
[68] S. Ghosh, DINS, a MIP improvement heuristic, in: Proceedings of the 12th International Conference on Integer Programming and Combinatorial Optimization (IPCO 2007), in: LNCS, vol. 4513, Springer, 2007, pp. 310–323.
[69] J.J. Dekker, M.G. De La Banda, A. Schutt, P.J. Stuckey, G. Tack, Solver-independent large neighbourhood search, in: Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP 2018), in: LNCS, vol. 11008, Springer, 2018, pp. 81–98.

[70] G. Björdal, P. Flener, J. Pearson, P.J. Stuckey, G. Tack, Declarative local-search neighbourhoods in MiniZinc, in: Proceedings of the 30th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2018), IEEE, 2018, pp. 98–105.
[71] A. Rendl, T. Guns, P.J. Stuckey, G. Tack, Minisearch: a solver-independent meta-search language for MiniZinc, in: Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP 2015), in: LNCS, vol. 9255, Springer, 2015, pp. 376–392.
[72] B.D. Cat, B. Bogaerts, M. Bruynooghe, G. Janssens, M. Denecker, Predicate logic as a modeling language: the IDP system, in: Declarative Logic Programming: Theory, Systems, and Applications, ACM / Morgan & Claypool, 2018, pp. 279–323.
[73] T. Pham, J. Devriendt, P.D. Causmaecker, Declarative local search for predicate logic, in: Proceedings of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (ICLP 2019), in: LNCS, vol. 11481, Springer, 2019, pp. 340–346.
[74] M. Gebser, A. Ryabokon, G. Schenner, Combining heuristics for configuration problems using answer set programming, in: Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2015), in: LNCS, vol. 9345, Springer, 2015, pp. 384–397.
[75] M. Maratea, L. Pulina, F. Ricca, A multi-engine approach to answer-set programming, Theory Pract. Log. Program. 14 (6) (2014) 841–868.
[76] L. Liu, M. Truszczynski, Y. Lierler, A machine learning system to improve the performance of ASP solving based on encoding selection, in: Proceedings of the 16th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2022), in: LNCS, vol. 13416, Springer, 2022, pp. 415–428.
[77] E. Mastria, J. Zangari, S. Perri, F. Calimeri, A machine learning guided rewriting approach for ASP logic programs, in: Technical Communications of the 36th International Conference on Logic Programming (ICLP 2020), in: EPTCS, vol. 325, 2020, pp. 261–267.
[78] P. Tassel, B. Kovács, M. Gebser, K. Schekotihin, W. Kohlenbrein, P. Schrott-Kostwein, Reinforcement learning of dispatching strategies for large-scale industrial scheduling, in: Proceedings of the 32nd International Conference on Automated Planning and Scheduling (ICAPS 2022), AAAI Press, 2022, pp. 638–646.