Large-Neighbourhood Search for Optimisation in Answer-Set Solving

Thomas Eiter[1], **Tobias Geibinger**[1], **Nelson Higuera Ruiz**[1], **Nysret Musliu**[1,2], **Johannes Oetsch**[1], **Daria Stepanova**[3]

[1] Institute for Logic and Computation, TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria,
[2] CD-Lab Artis, TU Wien, Favoritenstraße 9–11, 1040 Vienna, Austria,
[3] Bosch Center for AI, Robert Bosch Campus 1, 71272 Renningen, Germany
{eiter, nhiguera, oetsch}@kr.tuwien.ac.at, {tgeibing, musliu}@dbai.tuwien.ac.at, daria.stepanova@de.bosch.com

Abstract

While Answer-Set Programming (ASP) is a prominent approach to declarative problem solving, optimisation problems can still be a challenge for it. Large-Neighbourhood Search (LNS) is a metaheuristic for optimisation where parts of a solution are alternately destroyed and reconstructed that has high but untapped potential for ASP solving. We present a framework for LNS optimisation in answer-set solving in which neighbourhoods can be specified either declaratively as part of the ASP encoding or automatically generated by code. To effectively explore different neighbourhoods, we focus on multi-shot solving as it allows to avoid program regrounding. We illustrate the framework on different optimisation problems some of which are notoriously difficult, including shift planning and a parallel machine scheduling problem from semi-conductor production, which demonstrate the effectiveness of the LNS approach.

Introduction

Efficient solver technology and a simple modelling language have put Answer-Set Programming (ASP) (Lifschitz 2019) at the forefront of approaches to declarative problem solving with a growing number of applications in academia and industry. 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 (Hoos, Lindauer, and Schaub 2014), adding domain-specific heuristics (Dodaro et al. 2016; Gebser et al. 2013), or manual code rewriting for exploiting symmetries or achieving a smaller program grounding, these approaches might often need considerable time or expertise.

Large Neighbourhood Search (LNS) (Shaw 1998; Pisinger and Ropke 2010) is a metaheuristic 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., MIP (Danna, Rothberg, and Pape 2005; Rothberg 2007) and CP (Shaw 1998; Perron, Shaw, and Furnon 2004; Berthold et al. 2011; Björdal et al. 2020). For ASP however, to the best of our knowledge this potential is by and largely untapped. Recent work (Geibinger, Mischek, and Musliu 2021) 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 is unexplored. This is of particular interest, as ASP is offering problem-solving capacities beyond other solving paradigms such as MIP and CP (Dantsin et al. 2001; Leone et al. 2006).

Our main contribution is a framework for LNS optimisation for answer-set solving. To effectively explore different neighbourhoods, we build on the recent solver features of *multi-shot solving* and *solving under assumptions* (Gebser et al. 2019). 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. Learned heuristics and constraints can be kept between solver calls and repeated program grounding is effectively avoided. Solving under assumptions is a mechanism by which we can temporally fix parts of a solution between solver calls. While the underlying ideas are generic, we present our framework for the solvers `clingo` and its extension `clingo-dl` for difference logic, as well as `clingcon` for ASP with integer constraints from the Potassco family.[1]

We introduce two principled ways of using LNS with ASP.

- First, we present a system that can be used out of the box with all the supported ASP solvers. Different neighbourhoods can be seamlessly specified in a declarative way as part of the ASP encoding itself. To this end, dedicated predicates are used (no language extension is needed). If no neighbourhood is specified, an automatically generated random neighbourhood is the default. Already the latter turns out to be quite effective for many problems. We demonstrate this solver and its effectiveness for different optimisation problems. In particular, we use the well-known problems Social Golfer and Travelling Salesperson Problem, as well as generating smallest sets of clues for Sudoku. Furthermore, we consider an optimisation variant of the Strategic Companies problem and Shift Design (Abseher et al. 2016) as a real-world inspired benchmark. Throughout, LNS with ASP yields improved bounds compared to plain ASP with no or little extra effort.
- The second way to use LNS with ASP in our framework is by instantiating an abstract Python class that realises the

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

basic LNS loop for a solver. This can be the preferred way for more specialised ASP applications where neighbourhood definitions are easier to specify in an imperative language or when obtaining an initial solution from a construction heuristic instead of the ASP solver is beneficial. As an advanced showcase, we use a challenging parallel machine scheduling problem from industry (Eiter et al. 2021), where we can leverage the capabilities of `clingo-dl` and improve the state-of-the-art for this problem by using LNS with an efficient construction heuristic to start the search.

We proceed as follows. First, we present the background on ASP optimisation. Then, we describe our framework for LNS with ASP. Afterwards, we show how to tackle different optimisation problems with LNS and `clingo` and also present more advanced applications with `clingo` and `clingo-dl`. Before we conclude, we discuss related work in the penultimate section.

Background

Answer-Set Programming (ASP) (Lifschitz 2019; Gebser et al. 2012; Brewka, Eiter, and Truszczyński 2011) provides a declarative modelling language with rules of the form $Head$:- $Body$ (intuitively, $Head$ is true if $Body$ is true) that allows for a succinct representation of search and optimisation problems, for which solutions can be computed using dedicated ASP solvers. Problems are encoded in programs, i.e., finite sets of rules, whose *answer sets* (which are special models) yield the solutions of a problem. The latter can be computed using an answer-set solver, which commonly eliminates variables in rules in a preprocessing step called *grounding* (replacement by constant symbols) and then evaluates this ground (propositional) representation. We focus in this work on the multi-shot solver `clingo` and its extensions for theories (Gebser et al. 2019, 2016; Banbara et al. 2017; Janhunen et al. 2017). For a thorough introduction to the modelling language, we refer to the respective user guide.[2]

As an example for optimisation with `clingo`, consider the *Social Golfer Problem* (SGP): 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 Listing 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`.

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

Theory solving is a feature of `clingo` that allows extending the formalism by external theories like integer constraints in the style of SMT (Gebser et al. 2016). Using integer constraints can help immensely 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.

A number of recent ASP applications feature difference constraints for problems that involve timing constraints (Eiter et al. 2021; El-Kholany and Gebser 2020; Francescutto, Schekotihin, and El-Kholany 2021; Abels et al. 2019). For unrestricted integer constraints, `clingcon` (Banbara et al. 2017) or other constraint ASP systems (Balduccini and Lierler 2017; Lierler 2014) can be used.

The solver `clingo` supports hierarchical optimisation criteria and uses a range of model-guided methods (Gebser et al. 2011) as well as core-guided techniques (Andres et al. 2012) 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.

An LNS Framework for ASP

Large-Neighbourhood Search (LNS) (Shaw 1998; Pisinger and Ropke 2010) aims at gradually improving a solution by alternating a destroy and a recreate phase. The pseudo-code of a simple LNS procedure is given in Alg. 1. It starts with an initial solution. The operator $relax(\cdot)$ takes a solution and destroys parts of it by, for example, unassigning a specified percentage of all decision variables. The function $search(\cdot)$ takes a partial solution and tries to restore it to obtain an improved complete solution. This can be realised using any complete search method. The algorithm proceeds until a stop criterion, e.g. a global time limit, is met. LNS cannot show optimality of solutions in general, but this is often infeasible in practical optimisation settings anyway.

We use the ASP solvers `clingo`, `clingo-dl`, and `clingcon` to implement $search(\cdot)$. All of them support *multi-shot solving* (Gebser et al. 2019) which aids to implement the LNS heuristic 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 learned heuristics and constraints.

To realise the $relax(\cdot)$ operator, we use *solving under assumptions* (Gebser et al. 2019): assumptions temporarily fix truth values of atoms in a solver call. Between solver calls, we fix all atoms that are part of the solution that is not relaxed.

Listing 1: Encoding for the Social Golfer Problem.

```
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.
```

Algorithm 1: LNS optimisation for a minimisation problem

$s^* \leftarrow$ feasible solution
repeat
    $s' \leftarrow \mathit{search}(\mathit{relax}(s^*))$
    $\Delta c \leftarrow \mathit{cost}(s^*) - \mathit{cost}(s')$
    if $\Delta c > 0$ **then**
        $s^* \leftarrow s'$
    end if
until stop criterion met
return s^*

Defining the Neighbourhood

The LNS *neighbourhood* defines which parts of a solution are kept and which are destroyed in each iteration. Its structure 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, it 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 the previous section. 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 two and three. A potential neighbourhood could be to unassign random positions in the above schedule. Another one could be to destroy entire groups or even weeks.

Declarative neighbourhoods. To define a neighbourhood 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 used to define a mapping from S to atoms that should be fixed with assumptions between solver calls. The first argument is the atom to fix and the second is the corresponding term from S.

We illustrate this for different neighbourhood candidates for the Social Golfer Problem.

(*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.

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

Python plugins. 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. For example, assume we want to alternate between different neighbourhoods in the Social Golfer example and pick each with a specified probability. Our solver `clingo-lns`, which is described next, provides an easy way to plug in any neighbourhood definition.

The Solver clingo-lns

Our Python implementation of LNS with ASP in the loop, the solver `clingo-lns`, is publicly available.[3] Input files for ASP encodings and parameters are set via the command line, `--help` gives an overview. Solver options include the solver type (`clingo`, `clingo-dl`, or `clingcon`), a global time limit, a time limit for search within a particular neighbourhood, the size of the neighbourhood, and command line arguments to be passed to the solvers. Based on our experience,

[3] http://www.kr.tuwien.ac.at/research/projects/bai/aaai22.zip.

Algorithm 2: LNS with multi-shot solving and assumptions

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

```
1: c ← initialise clingo based solver
2: c.ground(P ∪ I)
3: s ← getInitialSolution(P ∪ I)
4: c.addBound(cost(s) − 1)
5: repeat
6:     s′ ← c.solve(t*, getMoveAssumptions(s, I))
7:     if SAT then
8:         s = s′
9:         c.addBound(cost(s) − 1)
10:    end if
11: until time passed > t
12: return s
```

the arguments that work well for an ASP solver carry gracefully over to the use within LNS. The solver supports minimisation and maximisation of hierarchical objective functions as well as minimisation of a single integer variable in `clingo-dl` mode.

Like we have already mentioned above, our implementation relies on multi-shot solving and solving under assumptions. The way those features are utilised to implement LNS is shown in Alg. 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* allows to 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. Now, after an initial solution was obtained, a bound is given to the interal solver telling it that the next solution has to have strictly better cost. At each iteration of the loop, the algorithm calls the internal solver with assumptions generated for this iteration and the given neighbourhood timeout. Intuitively, those assumptions specify which parts of the current solution are fixed in this iteration (or move). If the solver finds a solution, we update the incumbent and add a new bound, otherwise we do nothing and try again with different assumptions until we reach the timelimit.

While neighbourhoods can be specified as part of the ASP encoding, the solver will use random relaxation of the visible atoms specified via `#show` as the *default neighbourhood* if no other definition is found. For the Social Golfer example, this corresponds exactly to neighbourhood *pos*.

While the solver works already "out-of-the-box" with defaults for all search parameters as well as the neighbourhood, performance can often be improved by adjusting them. The size of the neighbourhood can be specified either as a ratio or as an absolute number of elements to fix or to relax from the terms specified via `_lns_select/1`; the default is to fix 80%. The size is too small if the ASP solver frequently reports unsatisfiability; it is too large (or the time limit for the solver calls too low) if the solver frequently times out.

Heuristics and customised neighbourhoods. The methods of the solver that construct the neighbourhood in each step can easily be overloaded with customised versions to implement more complex behaviour than possible with the declarative option. In particular, this concerns the methods $getInitialSolution$ and $getMoveAssumptions$ as seen in Alg. 2. Overriding the former provides the ability to specify the initially used solution. 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. Furthermore, by overloading $getMoveAssumptions$, it is possible to declare neighborhoods which are more domain specific and are not based on random relaxation. On the technical side, overriding those methods is achieved by creating a Python class which derives the abstract implementation provided in our framework. We give examples for this later, which can serve as a blue-print for more customised applications.

Experiments on Benchmark Problems

We experimentally demonstrate the effectiveness of `clingo-lns` on different benchmark problems.[4] 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 compare `clingo-lns` against plain `clingo` as baseline with a time limit of 1800 seconds for each run. As instances, we consider problems with 8 groups of 4 golfers over 7 to 12 weeks. As stated above, the optimisation goal here is to minimise the number of times two players meet each other more than once. We use `clingo-lns` with the different neighbourhood definitions from the previous section. We report the best and worst solution found with `clingo-lns` in 5 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 `clingo-lns` 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.

[4]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 20GB. We used `clingo` v 5.5.1 and `clingo-dl` v 1.2.1. All encodings, instances, logs, and random seeds are available at http://www.kr.tuwien.ac.at/research/projects/bai/aaai22.zip.

w	clingo	clingo-lns			
		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

Table 1: `clingo` vs. `clingo-lns` for instances 8-4-w of the Social Golfer Problem with different neighbourhoods. For `clingo-lns`, we report the best and worst penalties over 5 runs.

	clingo	clingo-lns	
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)

Table 2: `clingo` vs. `clingo-lns` for 20 instances of the Travelling Salesperson Problem with average, best, and worst cost among 5 runs for `clingo-lns`.

Travelling Salesperson Problem. We next consider the well-known Travelling Salesperson Problem (TSP). The encoding in Listing 2 is an optimisation variant of the one from the Asparagus platform.[5] Instances were taken from Asparagus as well.

The overall time limit was set to 300 seconds, and we limited search within any neighbourhood to 5 seconds. We used `clingo-lns` out-of-the box with its default neighbourhood, i.e. random relaxation of the `cycle/2` atoms. We only increased the neighbourhood size from 20% relaxation rate to 30% as this helps with faster convergence for the considered instances. The results are given in Table 2, where we report the cost of the best round trip found by `clingo` as well as the best of worst costs found by `clingo-lns` 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 atoms `cycle/2` 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.

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

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 (Leone et al. 2006). In particular, checks in coNP are expressible (e.g., a TSP instance has no solution) with a *saturation technique* (Eiter and Gottlob 1995) that uses minimality of answer sets.

Suppose we want to compute Sudoku puzzles that give a smallest number of hints. Listing 3 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 `clingo-lns` with its default search parameters and options `--configuration=many` and `-t4` for `clingo`, which is the default portfolio for multi-threaded solving and four threads. While `clingo` finds a solution with 21 hints for the standard 9×9 grid within 10 minutes, we found puzzles with 19 hints using `clingo-lns`. 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 (Cadoli, Eiter, and Gottlob 1997): 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 Listing 4; the instances are those of the 3rd (Calimeri, Ianni, and Ricca 2014), 4th (Alviano et al. 2013) and 5th (Calimeri et al. 2016) ASP Competition with random weights from $[1, 1000]$ added.

We compare `clingo` (called via the Python API) against `clingo-lns`, where we use the default neighbourhood and relax 20% of the companies in each step. The global time limit was 1800 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.

Applications of LNS with ASP

We next turn to advanced use cases of ASP with LNS for problems with more direct real-world applications. In particular, we address the practically relevant *Shift Design* problem from the domain of work force scheduling as well as *Parallel Machine Scheduling* from semi-conductor production.

Listing 2: Encoding for the Travelling Salesperson Problem.

```
{ 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.
```

Listing 3: Encoding for Sudoku Puzzle Generation

```
{ 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.
```

Shift Design. The goal is to align shifts so that over- and understaffing is avoided. We refer to Abseher et al. (2016) for a detailed problem description as well as the ASP encoding and the instances. 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 again use the solver `clingo` as baseline, but this time with the options `--opt-strat=usc,3` and `--configuration=handy` which runs `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 1 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 5 runs for the LNS approach. For 8 out of 33 instances from both data sets, neither approach could find any solution. For 17 instances `clingo` could find the optimal value and `clingo-lns` reported the same value as `clingo`. Results for the 7 remaining instances are given in Table 4, where we get indeed considerable improvements.

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 semi-conductor production plant. In recent work (Eiter et al. 2021), an ASP approach with difference logic has been introduced for this problem. Further improvements are possible with LNS and ASP.

The ASP encoding that we use is an improved version of the original one.[6] 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

[6] The encoding can be found at http://www.kr.tuwien.ac.at/research/projects/bai/aaai22.zip.

Listing 4: Encoding for Weighted Strategic Companies.

```
strategic(X1)|strategic(X2)|strategic(X3)|strategic(X4) :- produced_by(X,X1,X2,X3,X4).
strategic(W) :- controlled_by(W,X1,X2,X3,X4), strategic(X1), strategic(X2), strategic(X3),
      strategic(X4).
:~ strategic(C), weight(C,W). [W,C]
#show strategic/1.
```

	clingo	clingo-lns	
001	231092	209414.6	(207374–212612)
006	91221	94782.2	(88925–99116)
015	224472	210205.6	(207467–212338)
018	134757	129645.0	(124313–131251)
019	105481	103482.0	(98796–107197)
030	226653	213393.8	(203136–217266)
033	230732	219070.6	(217381–219493)
042	138809	125909.0	(124494–128524)
050	210771	190303.6	(186975–192591)
051	170227	76929.8	(69945–82626)
052	207188	90402.6	(84859–98724)
053	161343	74111.4	(69348–81988)
054	224058	76589.4	(72038–85074)
055	205034	95254.6	(82870–105613)
056	204921	84050.4	(78299–91111)
057	219262	79053.2	(74010–86627)
058	175945	73224.0	(70936–77177)
059	200575	73383.8	(70640–77592)
060	201830	87292.4	(82552–91980)
061	216207	74421.4	(71772–76291)

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

	clingo	clingo-lns
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	$(0, 1006, 66)$	$(0, 946, 68)$–$(0, 963, 67)$
3-26	$(0, 1061, 77)$	$(0, 1037, 78)$–$(0, 1078, 75)$
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 4: clingo vs. clingo-lns for Shift Design instances from DataSet3 and DataSet4 with best and worst objective value among 5 runs for clingo-lns. The values respectively correspond to shortage of staff, excess of staff and number of shifts.

next/3, which defines a total order of jobs on the machines.

The default random neighbourhood 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; (*machine*) select a number of machines and relax all jobs on them. We ensure that the machine determining 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 feasible schedule of fairly good quality.

The implementation in clingo-lns 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.

We compare plain clingo-dl with our LNS approach. The time limit is 15 minutes overall and 15 seconds for LNS steps. For the neighbourhoods, we fix 80% of the jobs or all but 2 of the machines, respectively. Furthermore, to avoid selections that are too large, we limit the number of selected jobs to 20% for both neighbourhoods.

The results for the 500 instances from the original paper are visualised in Fig. 1 as box plots. We show the median as well as 5 and 95 percentiles of the relative difference to the best solutions for clingo-dl with and without LNS respectively the construction heuristic. Both versions of LNS were run 5 times for each instance and the average was taken as the result. Plain LNS improved the performance of clingo-dl already, but the best results were obtained by using LNS with clingo-dl and the construction heuristic in combination. They significantly improve the published solutions for this problem.

Related Work

ASP solvers have seen a number of improvements for optimisation in recent years (Alviano et al. 2020) which makes them also attractive for LNS. Especially recent advances like using comparator networks (Bomanson and Janhunen 2020) and combining integer programming with ASP (Saikko et al. 2018) can be helpful in this context.

The use of LNS in MIP (Danna, Rothberg, and Pape 2005; Rothberg 2007; Ghosh 2007) and CP (Perron, Shaw, and Furnon 2004; Berthold et al. 2011; Björdal et al. 2020) is well explored. For declarative LNS neighbourhood definitions, the constraint modelling languages were extended to support solver-independent LNS search (Dekker et al. 2018; Björdal et al. 2018; Rendl et al. 2015). Our approach merely requires dedicated predicates that can be defined by rules,

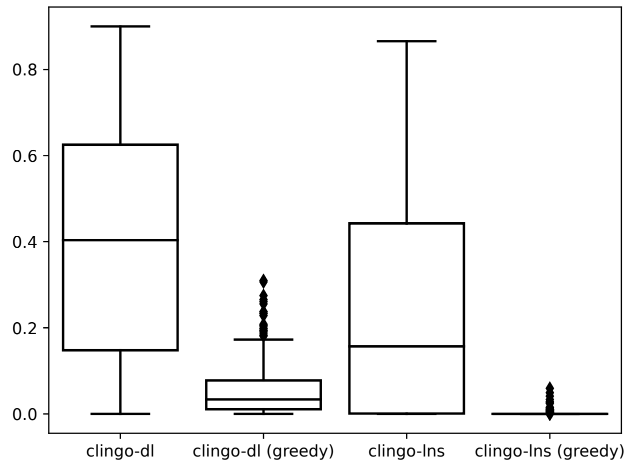

Figure 1: Relative differences to best solution for `clingo-dl` with LNS for Parallel Machine Scheduling.

and it offers unlimited power for neighbourhood definition by external plugins. Declarative LNS was also considered for Imperative-Declarative Programming, where LNS moves can be specified in predicate logic (Pham, Devriendt, and Causmaecker 2019).

The only work that touches on LNS in the context of ASP is the recent application of the `clingcon` for Test Laboratory Scheduling (Geibinger, Mischek, and Musliu 2021). 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` (Dodaro and Ricca 2020; Alviano et al. 2015) is possible, but an empowered multi-shot solving approach needs further efforts.

Gebser, Ryabokon, and Schenner (2015) 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 whether fruitful greedy heuristics for LNS with ASP could be (semi-)automatically constructed.

Conclusion

We have introduced an optimisation framework for ASP that exploits LNS and multi-shot solving, and we have demonstrated that this approach indeed boosts the capabilities of ASP for challenging optimisation problems. Notably, ASP makes LNS viable even for problems whose decision variant is beyond NP. We presented a general LNS solver that can be used to quickly set up LNS for ASP and to experiment with different neighbourhoods that can be specified as part of the ASP encoding itself. Thus, the spirit of ASP as a declarative approach for rapid prototyping is retained. With some extra effort, the LNS solver can be customised for ASP applications by implementing problem specific heuristics; this can further boost performance as witnessed by a machine scheduling problem from the industry.

For future work, we plan as a next step to make LNS self-adaptive so that parameters of the LNS search are adjusted on the fly during search.

Acknowledgments

This work was supported by funding from the Bosch Center for Artificial Intelligence. Furthermore, we would like to thank Michel Janus, Andrej Gisbrecht, and Sebastian Bayer for very helpful discussions on the scheduling problems at Bosch, as well as Roland Kaminski who suggested an improvement for the Parallel Machine Scheduling encoding.

References

Abels, D.; Jordi, J.; Ostrowski, M.; Schaub, T.; Toletti, A.; and Wanko, P. 2019. Train Scheduling with Hybrid ASP. In *Proceedings of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2019*), volume 11481 of *LNCS*, 3–17. Springer.

Abseher, M.; Gebser, M.; Musliu, N.; Schaub, T.; and Woltran, S. 2016. Shift design with answer set programming. *Fundamenta Informaticae*, 147(1): 1–25.

Alviano, M.; Calimeri, F.; Charwat, G.; Dao-Tran, M.; Dodaro, C.; Ianni, G.; Krennwallner, T.; Kronegger, M.; Oetsch, J.; Pfandler, A.; Pührer, J.; Redl, C.; Ricca, F.; Schneider, P.; Schwengerer, M.; Spendier, L. K.; Wallner, J. P.; and Xiao, G. 2013. The Fourth Answer Set Programming Competition: Preliminary Report. In *Proceedings of the 12th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2013*), volume 8148 of *LNCS*, 42–53. Springer.

Alviano, M.; Dodaro, C.; Leone, N.; and Ricca, F. 2015. Advances in WASP. In *Proceeding of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning* (*ICLP 2015*), volume 9345 of *LNCS*, 40–54. Springer.

Alviano, M.; Dodaro, C.; Marques-Silva, J.; and Ricca, F. 2020. Optimum stable model search: algorithms and implementation. *Journal of Logic and Computation*, 30(4): 863–897.

Andres, B.; Kaufmann, B.; Matheis, O.; and Schaub, T. 2012. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming* (*ICLP 2012*), volume 17 of *LIPIcs*, 211–221. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Balduccini, M.; and Lierler, Y. 2017. Constraint answer set solver EZCSP and why integration schemas matter. *Theory Practice of Logic Programming*, 17(4): 462–515.

Banbara, M.; Kaufmann, B.; Ostrowski, M.; and Schaub, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming*, 17(4): 408–461.

Berthold, T.; Heinz, S.; Pfetsch, M. E.; and Vigerske, S. 2011. Large Neighborhood Search beyond MIP. In *Proceedings of the 9th Metaheuristics International Conference* (*MIC 2011*), 51–60.

Björdal, G.; Flener, P.; Pearson, J.; Stuckey, P. J.; and Tack, G. 2018. Declarative Local-Search Neighbourhoods in MiniZinc. In *Proceedings of the 30th IEEE International Conference on Tools with Artificial Intelligence* (*ICTAI 2018*), 98–105. IEEE.

Björdal, G.; Flener, P.; Pearson, J.; Stuckey, P. J.; and Tack, G. 2020. Solving Satisfaction Problems Using Large-Neighbourhood Search. In *Proceedings of the 26th International Conference on Principles and Practice of Constraint Programming* (*CP 2020*), volume 12333 of *LNCS*, 55–71. Springer.

Bomanson, J.; and Janhunen, T. 2020. Boosting Answer Set Optimization with Weighted Comparator Networks. *Theory and Practice of Logic Programming*, 20(4): 512–551.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer Set Programming at a Glance. *Communications of the ACM*, 54(12): 92–103.

Cadoli, M.; Eiter, T.; and Gottlob, G. 1997. Default logic as a query language. *IEEE Transactions on Knowledge and Data Engineering*, 9(3): 448–463.

Calimeri, F.; Gebser, M.; Maratea, M.; and Ricca, F. 2016. Design and results of the Fifth Answer Set Programming Competition. *Artificial Intelligence*, 231: 151–181.

Calimeri, F.; Ianni, G.; and Ricca, F. 2014. The third open answer set programming competition. *Theory Practice of Logic Programming*, 14(1): 117–135.

Danna, E.; Rothberg, E.; and Pape, C. L. 2005. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming*, 102(1): 71–90.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys* (*CSUR*), 33(3): 374–425.

Dekker, J. J.; De La Banda, M. G.; Schutt, A.; Stuckey, P. J.; and Tack, G. 2018. Solver-independent large neighbourhood search. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming* (*CP 2018*), volume 11008 of *LNCS*, 81–98. Springer.

Dodaro, C.; Gasteiger, P.; Leone, N.; Musitsch, B.; Ricca, F.; and Shchekotykhin, K. 2016. Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). *Theory and Practice of Logic Programming*, 16(5-6): 653–669.

Dodaro, C.; and Ricca, F. 2020. The external interface for extending WASP. *Theory and Practice of Logic Programming*, 20(2): 225–248.

Eiter, T.; Geibinger, T.; Musliu, N.; Oetsch, J.; Skočovskỳ, P.; and Stepanova, D. 2021. 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*), 280–290.

Eiter, T.; and Gottlob, G. 1995. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Ann. Math. Artif. Intell.*, 15(3-4): 289–323.

El-Kholany, M.; and Gebser, M. 2020. Job Shop Scheduling with Multi-shot ASP. In *Proceedings of the 4th Workshop on Trends and Applications of Answer Set Programming* (*TAASP 2020*).

Francescutto, G.; Schekotihin, K.; and El-Kholany, M. M. 2021. 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*), volume 12678 of *LNCS*, 313–328. Springer.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Wanko, P. 2016. Theory Solving Made Easy with Clingo 5. In *Technical Communications of the 32nd International Conference on Logic Programming* (*ICLP 2016*), volume 52 of *OASIcs*, 2:1–2:15. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2011. Multi-criteria optimization in answer set programming. In *Technical Communications of the 27th International Conference on Logic Programming* (*ICLP 2011*), volume 11 of *LIPIcs*, 1–10. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. Answer Set Solving in Practice. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(3): 1–238.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP Solving with clingo. *Theory and Practice of Logic Programming*, 19(1): 27–82.

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

Gebser, M.; Ryabokon, A.; and Schenner, G. 2015. Combining heuristics for configuration problems using answer set programming. In *Proceedings of the 13th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2015*), volume 9345 of *LNCS*, 384–397. Springer.

Geibinger, T.; Mischek, F.; and Musliu, N. 2021. Constraint Logic Programming for Real-World Test Laboratory Scheduling. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence* (*AAAI 2020*), 6358–6366. AAAI Press.

Ghosh, S. 2007. DINS, a MIP Improvement Heuristic. In *Proceedings of the 12th International Conference on Integer Programming and Combinatorial Optimization* (*IPCO 2007*), volume 4513 of *LNCS*, 310–323. Springer.

Hoos, H.; Lindauer, M.; and Schaub, T. 2014. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming*, 14(4-5): 569–585.

Janhunen, T.; Kaminski, R.; Ostrowski, M.; Schellhorn, S.; Wanko, P.; and Schaub, T. 2017. clingo goes Linear Constraints over Reals and Integers. *Theory and Practice of Logic Programming*, 17(5-6): 872–888.

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

Lierler, Y. 2014. Relating Constraint Answer Set Programming Languages and Algorithms. *Artificial Intelligence*, 207: 1–22.

Lifschitz, V. 2019. *Answer Set Programming*. Springer.

Perron, L.; Shaw, P.; and Furnon, V. 2004. Propagation Guided Large Neighborhood Search. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming* (*CP 2004*), volume 3258 of *LNCS*, 468–481. Springer.

Pham, T.; Devriendt, J.; and Causmaecker, P. D. 2019. Declarative Local Search for Predicate Logic. In *Proceedings of the 15th International Conference on Logic Programming and Nonmonotonic Reasoning* (*ICLP 2019*), volume 11481 of *LNCS*, 340–346. Springer.

Pisinger, D.; and Ropke, S. 2010. Large neighborhood search. In *Handbook of Metaheuristics*, 399–419. Springer.

Rendl, A.; Guns, T.; Stuckey, P. J.; and Tack, G. 2015. 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*), volume 9255 of *LNCS*, 376–392. Springer.

Rothberg, E. 2007. An Evolutionary Algorithm for Polishing Mixed Integer Programming Solutions. *INFORMS Journal on Computing*, 19(4): 534–541.

Saikko, P.; Dodaro, C.; Alviano, M.; and Järvisalo, M. 2018. 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*), 32–41. AAAI Press.

Shaw, P. 1998. 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*), volume 1520 of *LNCS*, 417–431. Springer.