Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning
Applications and Systems

ALASPO: An Adaptive Large-Neighbourhood ASP Optimiser

Thomas Eiter[1] , **Tobias Geibinger**[1] , **Nelson Higuera**[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,
[3]Bosch Center for AI, Robert Bosch Campus 1, 71272 Renningen, Germany
{eiter, oetsch, nhiguera}@kr.tuwien.ac.at, {tgeibing, musliu}@dbai.tuwien.ac.at,
daria.stepanova@de.bosch.com

Abstract

We present the system ALASPO which implements Adaptive Large-neighbourhood search for Answer Set Programming (ASP) Optimisation. Large-neighbourhood search (LNS) is a meta-heuristic where parts of a solution are destroyed and reconstructed in an attempt to improve an overall objective. ALASPO currently supports the ASP solver clingo, as well as its extensions clingo-dl and clingcon for difference and full integer constraints, and multi-shot solving for an efficient implementation of the LNS loop. Neighbourhoods can be defined in code or declaratively as part of the ASP encoding. While the method underlying ALASPO has been described in previous work, ALASPO also incorporates portfolios for the LNS operators along with self-adaptive selection strategies as a technical novelty. This improves usability considerably at no loss of solution quality, but on the contrary often yields benefits. To demonstrate this, we evaluate ALASPO on different optimisation benchmarks.

1 Introduction

Answer-set programming (ASP) (Brewka, Eiter, and Truszczyński 2011; Lifschitz 2019; Gebser et al. 2012) is a declarative KR paradigm that is increasingly used to solve challenging optimisation problems. We present the system ALASPO (Adaptive Large-neighbourhood search for ASP Optimisation) that leverages ASP optimisation with *large-neighbourhood search* (LNS) (Shaw 1998; Pisinger and Ropke 2010), a powerful meta-heuristic where parts of a solution are destroyed and reconstructed in an attempt to improve an overall objective.

LNS is commonly used for 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), but for ASP it has only recently been exploited (Eiter et al. 2022; Geibinger, Mischek, and Musliu 2021). The method starts by obtaining a first solution for an optimisation problem encoded in ASP (either with a construction heuristic or from an ASP solver). Then, a *neighbourhood*, i.e., a part of the solution, is *relaxed*, i.e., deleted from the answer set. Next, an ASP solver is used to reconstruct a solution with a better overall objective value. Relaxation and reconstruction steps proceed until a global time limit is reached.

The system currently supports the ASP solver clingo (Gebser et al. 2019; Gebser et al. 2016), as well as its extensions clingo-dl (Janhunen et al. 2017) and clingcon (Banbara et al. 2017) for difference and full integer constraints, respectively. For an efficient implementation of the LNS loop, it exploits multi-shot solving to preserve the solver state and avoid unnecessary grounding between solver calls. The system features pre-defined neighbourhood definitions that can be used out-of-the-box, but also supports custom neighbourhoods provided by the user. They can be given either in code or, more conveniently, declaratively as part of the ASP encoding.

The method underlying ALASPO is described and evaluated in recent work (Eiter et al. 2022). As a major technical novelty, ALASPO allows to bundle neighbourhoods and search configurations into portfolios. Switching between operators can be done on the fly during search, but the system also provides full self-adaptive modes where no user intervention is required. In previous work (Eiter et al. 2022), one could only use a single neighbourhood operator. With ALASPO, the user can select a set of operators, and the tool self-adapts to use effective ones through selection strategies. Therefore, the solver requires less tuning which improves usability We evaluate the adaptive modes introduced in this paper on different optimisation benchmarks.

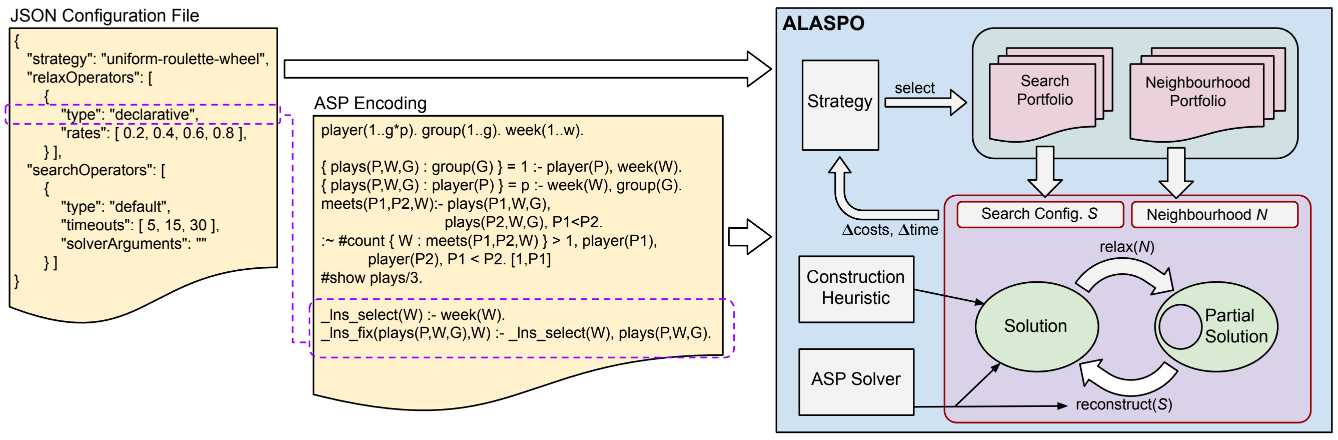
The system is publicly available under the MIT licence: https://gitlab.tuwien.ac.at/kbs/BAI/alaspo.

2 Architecture & Functionality

ALASPO is a system for ASP optimisation with support for different ASP solvers, search configurations, and neighbourhood definitions. Figure 1 gives on overview of how its components play together to realise the system's functionality.

The LNS loop. At the heart of ALASPO lies an LNS loop, where 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. Alternatively, it can be obtained by a custom procedure using a construction heuristic, which however is problem specific and must be provided by the user as an implementation of an abstract class in Python 3.


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 ],
        "solverArguments": ""
     } ]
}
```

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).
```

Figure 1: Adaptive LNS for ASP in the system ALASPO.

In each iteration of the LNS loop, the currently best solution I is relaxed using a *neighbourhood operator* N, which is a procedure to select a subset of the atoms in I. For instance, N could pick 20% of the visible atoms at random. Then, the resulting partial solution is reconstructed using the ASP solver with a constraint to obtain a better objective value than I. 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.

The optimisation problem is formulated in ASP and stored in one or multiple input files. The currently supported ASP solvers are clingo, clingo-dl, and clingcon from the Potassco family.[1] All solvers feature multi-shot solving and solving under assumptions (Gebser et al. 2019). The former is used to keep learned 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.

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. The portfolios contain some pre-defined neighbourhoods, but custom ones can be added.

The tool provides two problem independent pre-defined neighbourhood types: random-atoms and random-constants. *Random-atoms* relaxes a random sample of the visible atoms of an answer set. Although quite simple, this method is often surprisingly effective (Eiter et al. 2022). *Random-constants* selects a random sample from all constant symbols from the visible atoms, and relaxes all atoms that contain any constants from that selection. The idea is that often atoms are not independent and shared constants help to relax groups of related atoms together. Relaxation rates refer to the size of the random sample for both types.

Neighbourhood types can also be defined declaratively as part of the ASP encoding using dedicated predicates `_lns_select/1` and `_lns_fix/2`. An example appears in Fig. 1 for the Social Golfer problem; details are discussed in (Eiter et al. 2022). Roughly, `_lns_select` defines a set of terms from which a sample is selected. The size of that sample, as for the pre-defined types, is controlled using a respective relaxation rate. Predicate `_lns_fix` defines a mapping from terms of the sample to atoms that should not be relaxed between solver calls.

Neighbourhood operators can also be defined in code which can be useful for highly customised applications. An advanced example for a parallel machine scheduling problem is discussed by Eiter et al. (2022).

Strategies for adaptive LNS. While the system performs well with the right choice of search and neighbourhood operators, it is often better to make changes when the search gets stuck. ALASPO can be run in *interactive mode*, where the user can interrupt search when intervention seems necessary. Before search is resumed, a better fitting configuration from the portfolios can be selected. We also provide several *selection strategies* 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 the portfolio potentially more suitable LNS operators for the next iteration. A JSON file can be used to select a strategy and to specify neighbourhood operators with types and relaxation rates and search configurations with different time limits. This is then internally used to set up respective portfolios in ALASPO.

The first strategy implemented in ALASPO is a *self-adaptive roulette-wheel strategy* from the literature (Laborie

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

and Godard 2007; Pisinger and Ropke 2007): each pair (S, N) of search and neighbourhood operators 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 defined as the ratio of cost improvement over time needed, and α is the *learning rate*. Then, a new pair of operators is selected with probability proportional to the weights.

The system implements also a simple non-adaptive version of the first strategy, *uniform roulette-wheel strategy*, where all pairs of operators have equal weight, and are thus selected with equal probability in each iteration.

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 neighborhoods 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 operators are randomly chosen.

3 Usage

ALASPO can be used out-of-the-box with little or no configuration. If needed, it can also be customised through either the command-line or an additional JSON configuration file.

Invocation. The tool is called in standard configuration for a problem stored in `program.lp`:

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

where it runs the system with its *default portfolio* which uses neighborhood 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, and the dynamic selection strategy. During execution, the solver will continuously report when new solutions are found.

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 (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 pre-defined ones (random-atoms or random constants), or specify that declaratively defined one from the ASP program should be used instead. A relaxation rate, an argument list that is passed to the ASP solver, and a time limit for reconstruction steps can be set in the command-line 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.

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.

A novel feature is that 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 previous work (Eiter et al. 2022), one of which is shown in Fig. 1.

Portfolios can be specified in an additional JSON file as depicted in Fig. 1. 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 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. Alternatively, the user can select the interactive mode from the command line and interrupt the solver during search to switch to other operators when search gets stuck.

Custom applications. We mention in passing that the system can be extended for custom applications by defining new neighbourhood types in code. 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 can be found in previous work (Eiter et al. 2022).

4 Experiments

An evaluation of ASP with LNS was given in the companion paper (Eiter et al. 2022). We revisit some of the benchmarks from there to evaluate how the new self-adaptive modes perform in comparison with hand-selected search and neighbourhood operators used previously.

All experiments were run on a cluster of 13 nodes with 2 Intel Xeon CPUs E5-2650 v4 (max. 2.90GHz, 12 physical cores, no hyperthreading) per node and a memory limit of 20GB for each process. Encodings, instances, logs, and random seeds are available at www.kr.tuwien.ac.at/research/projects/bai/kr22.zip.

	single LNS operators (Eiter et al. 2022)	uniform	portfolio with selection strategy				dynamic
			self-adaptive				
			$\alpha = 0.2$	$\alpha = 0.4$	$\alpha = 0.6$	$\alpha = 0.8$	
SG	15.91 ± 2.05	17.14 ± 2.45	18.36 ± 3.20	$\mathbf{20.00} \pm 2.79$	17.14 ± 1.73	18.36 ± 2.79	19.18 ± 2.19
TSP	$\mathbf{35.85} \pm 0.60$	19.65 ± 3.61	20.75 ± 3.13	20.54 ± 2.46	20.74 ± 2.72	21.17 ± 2.55	29.03 ± 2.32
SPG	$\mathbf{3.33} \pm 0.00$	0.47 ± 0.52	0.00 ± 0.00	2.38 ± 0.52	1.90 ± 0.80	-0.95 ± 0.80	$\mathbf{3.33} \pm 0.43$
WSC	39.70 ± 0.77	39.83 ± 0.52	40.01 ± 0.41	40.17 ± 1.25	39.96 ± 0.79	$\mathbf{40.25} \pm 1.27$	39.44 ± 1.71
SD	7.29 ± 2.22	$\mathbf{12.14} \pm 1.66$	11.47 ± 0.56	11.22 ± 1.24	10.95 ± 2.12	10.61 ± 1.69	6.61 ± 2.02

Table 1: Average percentual improvement and standard deviation over plain clingo for ASP with (adaptive) LNS.

Benchmark problems and setup. In our previous experiments, LNS was used with a single search configuration and single hand-selected neighbourhood for each benchmark. 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. The benchmark problems and settings are as follows:

Social Golfer (SG) is an NP-hard scheduling problem where golfers must be grouped in a weekly schedule such that individuals meet again as little as possible. The global time limit is $1800s$ per instance, and we use a declaratively specified neighbourhood (shown in Fig. 1 with the problem encoding) to relax entire weeks per iteration, a relaxation rate of 20%, and a time limit of $20s$ for the LNS reconstruction step.

Travelling Salesman Problem (TSP) is the problem of finding a round trip with minimal costs in a graph. The global time limit is $300s$, and we use random-atoms with a time limit of $5s$ and a relaxation rate of 30% for LNS.

Sudoku Puzzle Generation (SPG) requests to generate a Sudoku puzzle with a minimal number of hints such that the solution is unique. The global time limit is $600s$, and we use "`--many -t4`" for clingo (the default portfolio for multi-treading with 4 threads). For LNS, we use random-atoms with a rate of 20% and time limit of $20s$.

Weighted Strategic Companies (WSC) is a version of the Σ_2^P-hard Strategic Companies Problem (Cadoli, Eiter, and Gottlob 1997) with additional weights for companies and the objective to find strategic sets with minimal total weight. Here, the global time limit is $1800s$, and we use random-atoms with a rate of 20% and a time limit of $30s$ for LNS.

Shift Design (SD) requires to align shifts such that over- and understaffing is avoided (Abseher et al. 2016). The global time limit is $3600s$ and the search configuration is "`--opt-strategy=usc,3 --configuration=handy`" which selects unsat-core based optimisation and defaults for large problems. For LNS, we use random-atoms with a rate of 70% and a time limit of $30s$. Also, pre-optimisation for 50 minutes is used to initialise search.

For our comparisons with the adaptive modes of ALASPO with portfolios, we used the solver settings described above and time limits from the default portfolio (cf. Section 3). 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 previous work together in a portfolio.

Results. The results of our experiments are summarised in Table 1. For all problems, we report the relative improvement in objective value for ASP with LNS in comparison with plain clingo averaged over 5 runs. The first column gives the results for LNS with single hand-selected LNS operators that fit the problem best as described above. The remaining columns 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 suboptimal operators become preferred ones. The setting with single hand-selected LNS operators is the sole winner in performance gain for the Travelling Salesperson Problem only, while it is outperformed by the portfolio approaches in three out of the four remaining problems. Notably, putting LNS operators into a portfolio is less effort for a user than determining manually which ones work best. The self-adaptive roulette-wheel strategy performs better than the uniform one in most of the cases, at least with the right learning rate, except for Shift Design, where uniform roulette works best overall. The dynamic strategy gives most consistently excellent improvements though not in all cases the best ones.

5 Discussion

The experiments show that our method of combining ASP with LNS is quite effective even with a single neighbourhood operator and a fixed relaxation rate. Often better results are achieved however with more customised configurations. Portfolios, as implemented in ALASPO in contrast to previous work where only single LNS operators were supported (Eiter et al. 2022), ease the burden on the user to find such configurations. Our experiments confirm that the investigated adaptive strategies with reasonable portfolios achieve very good results most of the time. Adaptive LNS has thus indeed the potential to enhance ASP optimisation in many applications; for some, this has already been demonstrated, others are planned.

For future work, we want to extend ALASPO to support further ASP systems like WASP (Alviano et al. 2015), add more pre-defined neighbourhoods, and to continue research on new self-adaptive strategies.

Acknowledgements

This work was supported by funding from the Bosch Center for AI at Renningen, Germany.

References

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.; 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.

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. 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.

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.

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

Eiter, T.; Geibinger, T.; Ruiz, N. H.; Musliu, N.; Oetsch, J.; and Stepanova, D. 2022. Large-neighbourhood search for optimisation in answer-set solving. In *36th AAAI Conference on Artificial Intelligence* (*AAAI-22*). http://www.kr.tuwien.ac.at/research/projects/bai/aaai22.pdf.

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.; 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. 2019. Multi-shot ASP Solving with clingo. *Theory and Practice of Logic Programming* 19(1):27–82.

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 21*), 6358–6366. AAAI Press.

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.

Laborie, P., and Godard, D. 2007. 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*).

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.

Pisinger, D., and Ropke, S. 2007. A general heuristic for vehicle routing problems. *Computers & Operations Research* 34(8):2403–2435.

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

Rothberg, E. 2007. An evolutionary algorithm for polishing mixed integer programming solutions. *INFORMS Journal on Computing* 19(4):534–541.

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.