Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning
KR in the Wild

Large Neighborhood Prioritized Search for Combinatorial Optimization with Answer Set Programming

Irumi Sugimori[1], Katsumi Inoue[2], Hidetomo Nabeshima[3], Torsten Schaub[4], Takehide Soh[5], Naoyuki Tamura[5], Mutsunori Banbara[1]
[1]Nagoya University, Japan
[2]National Institute of Informatics, Japan
[3]University of Yamanashi, Japan
[4]Universität Potsdam, Germany
[5]Kobe University, Japan

Abstract

We propose *Large Neighborhood Prioritized Search* (LNPS) for solving combinatorial optimization problems in Answer Set Programming (ASP). LNPS is a metaheuristic that starts with an initial solution and then iteratively tries to find better solutions by alternately destroying and prioritized searching for a current solution. Due to the variability of neighborhoods, LNPS allows for flexible search without strongly depending on the destroy operators. We present an implementation of LNPS based on ASP. The resulting *heulingo* solver demonstrates that LNPS can significantly enhance the solving performance of ASP for optimization. Furthermore, we establish the competitiveness of our LNPS approach by empirically contrasting it to (adaptive) large neighborhood search.

1 Introduction

Systematic search and Stochastic Local Search (SLS) are two major methods for solving a wide range of combinatorial optimization problems. Each method has strengths and weaknesses. Systematic search can prove the optimality of solutions, but in general, it does not scale to large instances. SLS can find near-optimal solutions within a reasonable amount of time, but it cannot guarantee the optimality of solutions. Therefore, there has been an increasing interest in the development of *hybrids* between systematic search and SLS (Hoos and Stützle 2015).

Large Neighborhood Search (LNS) (Shaw 1998; Pisinger and Ropke 2019) is one of the most studied hybrids. LNS is an SLS-based metaheuristic that starts with an initial solution and then iteratively tries to find better solutions by alternately *destroy*ing and *repair*ing a current solution. Since the repair operators can be implemented with systematic solvers, the LNS heuristic has been shown to be highly compatible with mixed integer programming (Fischetti and Lodi 2003; Danna, Rothberg, and Pape 2005) and constraint programming (Shaw 1998; Dekker et al. 2018; Björdal et al. 2019; Björdal et al. 2020).

Answer Set Programming (ASP) (Lifschitz 2019) is a declarative programming paradigm for knowledge representation and reasoning. Due to remarkable improvements in the efficiency of ASP solvers, ASP has been successfully applied in diverse areas of artificial intelligence and computer science, such as robotics, computational biology, product configuration, decision support, scheduling, planning, constraint satisfaction, model checking, timetabling, and many others (Erdem, Gelfond, and Leone 2016; Alviano, Dodaro, and Maratea 2018; Ali, El-Kholany, and Gebser 2023; Banbara et al. 2013; Banbara et al. 2019).

The use of LNS with ASP has been recently explored (Eiter et al. 2022b), and soon afterward extended to *Adaptive Large Neighborhood Search* (Adaptive LNS) (Eiter et al. 2022a). The *ALASPO* solver, an ASP-based implementation of adaptive LNS, has demonstrated that LNS can boost the solving performance of ASP on hard optimization problems (Eiter et al. 2022a). However, LNS strongly depends on the destroy operators since the undestroyed part is fixed. In general, it cannot guarantee the optimality of solutions. It is therefore still particularly challenging to develop a universal algorithm for ASP which has the advantages of both systematic search and SLS.

In this paper, we propose *Large Neighborhood Prioritized Search* (LNPS) for solving combinatorial optimization problems in ASP. LNPS is a metaheuristic that starts with an initial solution and then iteratively tries to find better solutions by alternately destroying and *prioritized searching* for a current solution. We present the design and implementation of LNPS based on ASP. To evaluate the effectiveness of our approach, we conduct experiments on a benchmark set used in (Eiter et al. 2022a).

The main contributions and results of our paper are summarized as follows:

1. We propose Large Neighborhood Prioritized Search (LNPS). Since the undestroyed part is not fixed and can be prioritized (i.e., *variability*), the LNPS heuristic allows for flexible search without strongly depending on the destroy operators. Moreover, LNPS guarantees the optimality of solutions.

2. We present a design and implementation of LNPS based on ASP. In our approach, the LNPS algorithm can be compactly implemented by using multi-shot ASP solving and heuristic-driven ASP solving, in our case via *clingo*'s Python API (Gebser et al. 2019; Kaminski et al. 2023) and heuristic statements (Gebser et al. 2013).

3. The resulting *heulingo* solver is a tool for heuristically-driven answer set optimization. *heulingo* can handle any ASP encodings for optimization without any modification. All we have to do is to add an LNPS configuration in a declarative way. *heulingo* also supports the traditional LNS heuristic.
4. Our empirical analysis considers a challenging benchmark set used in (Eiter et al. 2022a). We succeeded in significantly enhancing the solving performance of *clingo* for optimization. Furthermore, *heulingo* demonstrated that the LNPS approach allows us to compete with ASP-based adaptive LNS (Eiter et al. 2022a).

Overall, the proposed LNPS can represent a significant contribution to the state-of-the-art of ASP solving for optimization as well as hybrids between systematic search and SLS.

2 Background

In this paper, ASP programs are written in the language of *clingo* (Gebser et al. 2015). ASP programs are finite sets of *rules*. Rules are of the form

$$\mathtt{a}_0\ \mathtt{:-}\ \mathtt{a}_1,\ldots,\mathtt{a}_m,\mathtt{not}\ \mathtt{a}_{m+1},\ldots,\mathtt{not}\ \mathtt{a}_n\mathtt{.}$$

Each $\mathtt{a}_i$ is a propositional *atom*. An atom `a` and its negation `not a` are called *literal*. The left of `:-` is a *head*, and the right is a *body*. The connectives `:-`, ‘`,`’, and `not` represent if, conjunction, and default negation, respectively. A period ‘`.`’ terminates each rule. Intuitively, the rule means that $\mathtt{a}_0$ must be assigned to true if $\mathtt{a}_1$, …, $\mathtt{a}_m$ are true and $\mathtt{a}_{m+1}$, …, $\mathtt{a}_n$ are false. A rule whose body is empty (i.e., $\mathtt{a}_0\mathtt{.}$) is called *fact*. Facts are always true. A rule whose head is empty is called *integrity constraint*:

$$\mathtt{:-}\ \mathtt{a}_1,\ldots,\mathtt{a}_m,\mathtt{not}\ \mathtt{a}_{m+1},\ldots,\mathtt{not}\ \mathtt{a}_n\mathtt{.}$$

An integrity constraint represents that the conjunction of literals in the body must be false. Semantically, an ASP program induces a collection of *answer sets*. Answer sets are distinguished models of the program based on stable model semantics (Gelfond and Lifschitz 1988). ASP has some convenient language constructs for modeling combinatorial (optimization) problems. A *conditional literal* is of the form $\ell_0\mathtt{:}\ell_1,\ldots,\ell_m$. Each ℓ_i is a literal, and ℓ_1, …, ℓ_m is called condition like in mathematical set notation. A *cardinality constraint* of the form $\{c_1\mathtt{;}\ldots\mathtt{;}c_n\}\ \mathtt{=}\ k$ represents that exactly k conditional literals among $\{c_1,\ldots,c_n\}$ must be satisfied. A weak constraint of the form $\mathtt{:\sim}\ \boldsymbol{L}\mathtt{.}\,[w\mathtt{,}\boldsymbol{t}]$ represents preferences in ASP, which is equivalent to $\mathtt{\#minimize}\ \{w\mathtt{,}\boldsymbol{t}\mathtt{:}\boldsymbol{L}\}\mathtt{.}$ Here, w is a weight, and $\boldsymbol{t}$ and $\boldsymbol{L}$ are tuples of terms and literals, respectively.

Multi-shot ASP solving introduces new language constructs: `#program` and `#external` statements. The former statement of the form `#program` $p(t)$`.` is used to separate an ASP program into several parameterizable subprograms. The predicate p is a subprogram name and the optional parameter t is a symbolic constant. `base` is a default subprogram with an empty parameter and includes rules that are not subject to any `#program` statements. The latter statement of the form `#external` a`.` represents that the atom a is an *external* atom whose truth value can be changed later on. By default, the initial truth value of external atoms is false. Heuristic-driven ASP solving allows for customizing the search heuristics of *clingo* from within ASP programs. Heuristic information is represented by `#heuristic` statements of the form `#heuristic` $a\mathtt{:}\boldsymbol{L}\mathtt{.}\,[w\mathtt{,}m]$ where m and w are terms representing a heuristic modifier and its value, respectively.

Listing 1: A traditional ASP encoding for TSP solving

```
{ 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]
```

Listing 2: *clingo* program activating or deactivating heuristic statements on demand for TSP solving

```
 1 #program heu.
 2 #heuristic cycle(X,Y): heu(cycle(X,Y),W,
     M). [W,M]
 3
 4 #script(python)
 5 from clingo import Number, Function
 6 def main(ctl):
 7     ctl.ground([("base",[])])
 8     ctl.solve()
 9     a = Function("heu",[Function("cycle
     ",[Number(1),Number(2)]),Number(1),
     Function("true")])
10     ctl.add("ext",[],f"#external {a}.")
11     ctl.ground([("ext",[])])
12     ctl.ground([("heu",[])])
13     ctl.assign_external(a,True)
14     ctl.solve()
15     ctl.release_external(a)
16     ctl.solve()
17 #end.
```

clingo provides a Python API for controlling ASP's grounding and solving process. For illustration, let us consider the well-known Traveling Salesperson Problem (TSP). A traditional ASP encoding for TSP solving (Eiter et al. 2022b) is shown in Listing 1. The atom `cycle(X,Y)` represents that a directed edge `X` $\rightarrow$ `Y` is in a Hamiltonian cycle. That is, it characterizes a solution. A *clingo* program activating or deactivating heuristic statements on demand for TSP solving is shown in Listing 2. This program consists of a subprogram `heu` and an embedded Python script.

Once *clingo* accepts the program (Listing 2) combined with a TSP instance of fact format and ASP encoding (Listing 1), a *clingo* object is created and bound to variable `ctl` (cf. Line 6). The instance and the encoding of Listing 1 are grounded by the `ground` function in Line 7. The `solve` function in Line 8 triggers computing stable models (i.e.,

Algorithm 1 Large Neighborhood Prioritized Search

```
Input: a feasible solution $x$
 1: $x^* \leftarrow x$
 2: while stop criterion is not met do
 3:     $x^t \leftarrow \mathit{prioritized\text{-}search}(\mathit{destroy}(x))$
 4:     if $\mathit{accept}(x^t, x)$ then
 5:         $x \leftarrow x^t$
 6:     end if
 7:     if $c(x^t) < c(x^*)$ then
 8:         $x^* \leftarrow x^t$
 9:     end if
10: end while
11: return $x^*$
```

solutions of TSP) for the ground program. That is, the first call of `solve` performs a plain TSP solving with *clingo*.

Next, the external statement of `heu(cycle(1,2),1, true)` is added to the subprogram `ext` by the `add` function in Line 10. This external atom is used to activate or deactivate the heuristic statement in Line 2. The external and heuristic statements are grounded in Lines 11 and 12, respectively. Since the truth value of the external atom is set to true by the `assign_external` function in Line 13, `#heuristic cycle(1,2). [1,true]` is activated in the second call of `solve` in Line 14. Intuitively, this heuristic statement means that the atom `cycle(1,2)` is set to true with higher priority during the search. More precisely, the solver decides first on `cycle(1,2)` of level `1` (0 by default for each atom) with a positive sign. Finally, the heuristic statement is deactivated in the third call of `solve` in Line 16, since the truth value of the external atom is permanently set to false by the `release_external` function in Line 15.

3 Large Neighborhood Prioritized Search

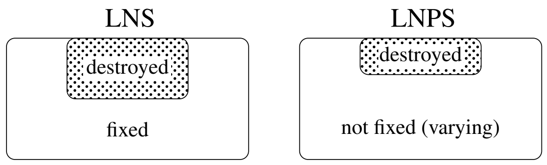
We consider that the Combinatorial Optimization Problem (COP) is a minimization problem. The task of COP is to find a solution x^* such that $c(x^*) \leq c(x)\ \forall x \in X$, where X is the finite set of feasible solutions, and $c : X \rightarrow \mathbb{R}$ is an objective function that maps from a solution to its cost.

We propose an SLS-based metaheuristic called Large Neighborhood Prioritized Search (LNPS) for solving COPs. LNPS starts with an initial solution and then iteratively tries to find better solutions by alternately destroying a current solution and reconstructing it with prioritized search. We define the *prioritized search* as a systematic search for which its branching heuristic can be configured (or customized) to meet the specific needs of users based on the priority of assignments to each variable.

The algorithm of LNPS is shown in Algorithm 1. Three variables are used in the algorithm. The variable x^* is the best solution obtained during the search. The variable x is the current solution. The variable x^t is a temporal solution which can be accepted as the current solution or discarded. The $\mathit{destroy}$ operator randomly destroys parts of x and returns the undestroyed part. The $\mathit{prioritized\text{-}search}$ operator returns a feasible solution, which is reconstructed by reassigning values of not only variables in the destroyed part but also ones in the undestroyed part by prioritized search.

The best solution x^* is initialized in Line 1. The loop in Lines 2–10 is repeated until a stop criterion is met. Typical choices for the stop criterion would include the optimality of x^*, a time-limit, or a limit on the number of iterations. The $\mathit{destroy}$ and $\mathit{prioritized\text{-}search}$ operators in Line 3 are alternately applied to find a new solution x^t. The new solution is evaluated in Line 4 whether or not it becomes the new current solution. In Line 5, the current solution x is updated if necessary. There are several ways to implement the *accept* function. The simplest way is to accept a strictly better solution than the current solution. And also, the new solution is checked in Line 7 whether or not it is better than the best known solution. In Line 8, the current best solution x^* is updated if necessary. Finally, the best solution is returned.

We discuss the main features of LNPS compared with traditional LNS. In the following figure, each outer rectangle represents a current solution, in which the dotted part represents the destroyed part.



In LNS (Pisinger and Ropke 2019), the destroy operator, particularly *the percentage of destruction*, plays an essential role since the undestroyed part is fixed. The percentage of destruction should be sufficiently large such that a neighborhood includes better solutions, and be sufficiently small such that the solver finds one of them. In addition, LNS cannot guarantee the optimality of solutions in general.

In contrast, LNPS can provide flexible search with weakened dependency on the destroy operators since the undestroyed part is not fixed (varying) and can be prioritized. Due to this *variability*, the percentage of destruction can be smaller in LNPS. LNPS can guarantee the optimality of obtained solutions by appropriately designing a stop criterion of prioritized search. The easiest way is gradually increasing the time-limit or the solve-limit on conflicts or restarts.

Furthermore, the undestroyed part can be configured (or customized) based on the priority of each variable. That is, LNPS allows for easy incorporation of domain-specific heuristics or domain-independent ones into the undestroyed part. For example, the most simple heuristic would be to keep an initial solution as much as possible. This can be achieved by setting the percentage of destruction to zero and giving high priority to full assignments of decision variables. Such a heuristic with zero destruction can be useful for quick *re-scheduling* in real-world applications (e.g., timetabling) rather than minimal perturbation with respect to an initial solution (Sakkout and Wallace 2000; Phillips et al. 2017; Zivan, Grubshtein, and Meisels 2011). Note that the zero destruction is completely useless for LNS since it turns out to be the same solution.

Finally, we discuss more details about the destroy and prioritized search operators. LNPS and LNS accept a single heuristic configuration. That is, they take a fixed percentage of destruction as input, which does not vary in different iterations throughout the search. Adaptive LNS (Ropke and Pisinger 2006), an extension of LNS, allows for multiple destroy and repair operators within the search.

In LNS, the destroy/repair operators can be interpreted as fix/optimize ones used in (Pisinger and Ropke 2019), respectively. The fix operator fixes part of the current solution, and the optimize operator tries to find better solutions by re-assigning values of variables in the non-fixed part. In general, the destroy operator may make a solution infeasible, and then the repair operator should reconstruct a feasible one, as discussed in (Pisinger and Ropke 2019). However, when the repair operator is implemented using systematic solvers, infeasible solutions trigger the solvers to decide unsatisfiability. In contrast, LNPS can always reconstruct feasible solutions due to the prioritized search.

4 *heulingo*: an ASP-based LNPS

We develop the *heulingo* solver which is an LNPS implementation based on ASP. The architecture of *heulingo* is shown in Figure 1. The *heulingo* solver accepts a COP instance and an LNPS configuration in ASP fact format. In turn, these facts are combined with an ASP encoding for COP solving, which are afterward solved by the LNPS algorithm powered by ASP solvers, in our case *clingo*. ASP facts of LNPS configurations specify the behavior of the LNPS heuristic, especially for the destroy and prioritized-search operators.

Implementation. The key idea is utilizing heuristic statements to implement the prioritized-search operator of LNPS. The input consists of a problem instance P, an ASP encoding E, and an LNPS configuration. The pseudo-code of our LNPS algorithm using *clingo*'s multi-shot ASP solving is shown in Algorithm 2. The variables sol, sol_best, and sol_tmp correspond to x, x^*, and x^t respectively in Algorithm 1. The variables $cost$, $cost\_best$, and $cost\_tmp$ represent the objective values $c(x)$, $c(x^*)$, and $c(x^t)$ respectively. The variable ctl is a solver object of the `Control` class in *clingo*'s Python API. The variable ret is used to store the solving result.

The outline of Algorithm 2 is as follows.

1. Ground $P \cup E$ (Line 1) and search for an initial solution (Line 2).
2. Generate heuristic statements (Lines 10–15), which can be activated or deactivated via external atoms later on.
3. Destroy parts of the current solution (Line 23).
4. Deactivate heuristic statements used in the previous iteration (Line 25).
5. Activate heuristic statements for the undestroyed part (Lines 33–35).
6. Prioritized search for a new solution (Line 37).
7. Update the current solution if necessary (Lines 41–43).
8. Update the best solution if necessary (Lines 44–46).
9. Go to Step 3 if the stop criterion is not met.
10. Return the best solution (Line 49).

First, in Line 1, the instance and the encoding in the `base` subprogram are grounded. The `solve` function in Line 2 triggers searching for an initial solution with a certain stop criterion, and then the current solution and its cost are initialized. The algorithm returns the current solution in Line 4 and terminates if it is optimal. Otherwise, the global best solution and its cost are initialized in Line 6.

The LNPS configuration (see below for details) is grounded and parsed in Lines 7–15. The heuristic statements of the form `#heuristic` $p(X_1,\ldots,X_n)$`:heuristic(`$p(X_1,\ldots,X_n)$`,W,M,t), W!=inf. [W,M]` are added to the `heuristic` subprogram in Line 16. The external atom `heuristic(`$p(X_1,\ldots,X_n)$`,W,M,t)` in the body is used to activate or deactivate the heuristic statements on demand. We refer to it as *heuristic atom*. In addition, two kinds of integrity constraints in Line 14 are added in a similar way to support traditional LNS.

Next, the loop in Lines 21–48 is repeated until a stop criterion is met. The variable $finished$, initialized to False in Line 17, is a flag for whether iterations should be finished or not. The $check\_variability$ function in Line 20 checks the variability of the LNPS configuration. That is, the variable $variability$ is set to False if the undestroyed part may be fixed for LNS, otherwise True.

In each iteration, the algorithm invokes the $destroy$ function in Line 23 to destroy parts of the current solution, according to the configuration. In turn, the $prioritize$ function in Line 24 is invoked to generate new heuristic atoms for the undestroyed part. The heuristic statements in the previous iteration are deactivated in Line 25. The external statements for new heuristic atoms are generated and added to the `external` subprogram in Lines 27–30.

We are now ready to try to find a better feasible solution. The external and heuristic statements are grounded in Lines 31 and 32, respectively. The heuristic statements are activated in Lines 33–35 by setting the truth value of their heuristic atoms to true. The `solve` function in Line 37 triggers heuristically searching for a new solution, and then the temporal solution and its cost are updated. Note that the stop criterion of `solve` in Lines 2 and 37 can be separately specified using *clingo*'s options: the time-limit in seconds (`--time-limit`) or the solve-limit on conflicts or restarts (`--solve-limit`). We adopt the latter solve-limit in the current implementation of *heulingo*.

In Line 38, the termination criterion is checked. The variable $finished$ is set to True in Line 39 if the new solution is optimal and the $variability$ flag is True. When the variability flag is False (i.e., LNS), the algorithm cannot terminate even if the new solution is optimal since the undestroyed part is fixed. The new solution is evaluated in Line 41 whether it can become the current solution or should be rejected. *heulingo* accepts only improving solution by default, but the acceptance criterion can be customized by a *heulingo*'s option. And also, the new solution is checked in Line 44 whether or not it is better than the best known solution. In Line 45, the current best solution is updated if necessary. At the end of the loop in Line 47, the value of *clingo*'s solve-limit is increased by the function $increase\_solve\_limit$ to guarantee the optimality of solutions. Finally, the algorithm

Algorithm 2 LNPS algorithm with *clingo*'s multi-shot ASP solving and heuristic statements

```
Input: $P$: problem instance, $E$: ASP encoding, $C$: LNPS configuration
 1: $ctl.ground$([("base", [])])      {ground the problem instance $P$ and the ASP encoding $E$}
 2: $(ret,\ sol,\ cost) \leftarrow ctl.solve()$      {search for an initial solution}
 3: if $ret =$ OPTIMUM FOUND then
 4:     return $sol$
 5: end if
 6: $sol\_best,\ cost\_best \leftarrow sol,\ cost$
 7: $ctl.ground$([("config", [])])      {ground the LNPS configuration $C$}
 8: $lnps\_config \leftarrow get\_lnps\_config()$
 9: $rules \leftarrow$ ""
10: for $c$ in $lnps\_config$ do      {generate heuristic statements to realize prioritized search}
11:     $p \leftarrow c.get\_predicate\_name()$
12:     $n \leftarrow c.get\_arity()$
13:     $atom \leftarrow$ "$p$(X$_1$,X$_2$, ...,X$_n$)"
14:     $rules \leftarrow rules +$ ":- not $atom$, heuristic($atom$,inf,true,t)."
            +":- $atom$, heuristic($atom$,inf,false,t)."
            +"#heuristic $atom$ :  heuristic($atom$,W,M,t), W!=inf.  [W,M]"
15: end for
16: $ctl.add$("heuristic", ["t"], $rules$)      {add the heuristic statements to the heuristic subprogram}
17: $finished \leftarrow$ False
18: $step \leftarrow 0$
19: $prev\_heu\_atoms \leftarrow$ []
20: $variability \leftarrow check\_variability(lnps\_config)$
21: while $finished =$ False do
22:     $step \leftarrow step + 1$
23:     $undestroyed \leftarrow destroy(sol,\ lnps\_config)$      {destroy parts of the current solution}
24:     $heu\_atoms \leftarrow prioritize(undestroyed,\ lnps\_config,\ step)$      {generate heuristic atoms for the undestroyed part}
25:     $ctl.release\_external(prev\_heu\_atoms)$      {deactivate the heuristic statements in the previous iteration}
26:     $statements \leftarrow$ ""
27:     for heuristic($a, w, m, t$) in $heu\_atoms$ do      {generate the external statements for heuristic atoms}
28:         $statements \leftarrow statements +$ "#external heuristic($a, w, m, t$)."
29:     end for
30:     $ctl.add$("external", [], $statements$)      {add the external statements to the external subprogram}
31:     $ctl.ground$([("external", [])])      {ground the external statements}
32:     $ctl.ground$([("heuristic", [$step$])])      {ground the heuristic statements}
33:     for heuristic($a, w, m, t$) in $heu\_atoms$ do      {activate the heuristic statements}
34:         $ctl.assign\_external$(heuristic($a, w, m, t$), True)
35:     end for
36:     $prev\_heu\_atoms \leftarrow heu\_atoms$
37:     $(ret,\ sol\_tmp,\ cost\_tmp) \leftarrow ctl.solve()$      {prioritized search for a new solution}
38:     if $ret =$ OPTIMUM FOUND and $variability =$ True then
39:         $finished \leftarrow$ True
40:     end if
41:     if $accept(cost\_tmp,\ cost) =$ True then      {evaluate whether the new solution can become the current solution}
42:         $sol,\ cost \leftarrow sol\_tmp,\ cost\_tmp$
43:     end if
44:     if $cost\_tmp < cost\_best$ then
45:         $sol\_best,\ cost\_best \leftarrow sol\_tmp,\ cost\_tmp$
46:     end if
47:     $increase\_solve\_limit()$
48: end while
49: return $sol\_best$
```

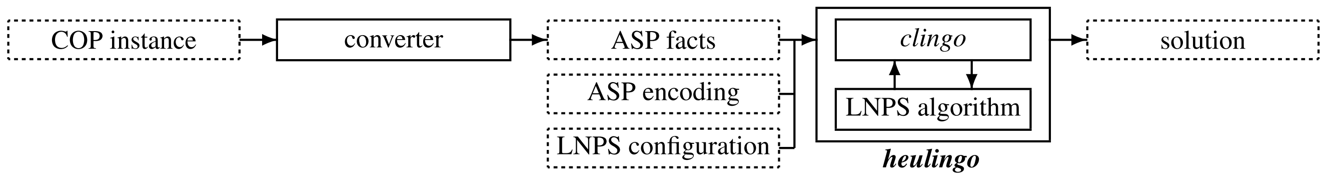


Figure 1: The architecture of *heulingo*

Listing 3: A simple LNPS heuristic for TSP solving

```
1  #program config.
2  _lnps_project(cycle,2).
3  _lnps_destroy(cycle,2,3,p(n)).
4  _lnps_prioritize(cycle,2,1,true).
```

returns the best solution in Line 49.

ASP fact format of LNPS configurations. We introduce three different kinds of predicates to specify configurations of the LNPS heuristic in the `config` subprogram. The predicate `_lnps_project/2` is used to define what subset of the atoms belonging to an answer set is subject to LNPS. We refer to the atoms via `_lnps_project/2` as *projected atoms*. In general, the projected atoms characterize a solution. The predicate `_lnps_destroy/4` is used to define what part of the projected atoms is destroyed and by what percentage. The predicate `_lnps_prioritize /4` is used to define how the projected atoms in the undestroyed part are prioritized (or fixed).

For illustration, let us consider a configuration of the LNPS heuristic for the TSP encoding in Listing 1. For this, the *random destruction* would be one of the most simple configurations, which is shown in Listing 3. Intuitively, the configuration represents a heuristic that randomly destroys a current solution, and the undestroyed part is not fixed but is kept as much as possible in each iteration. The atom `_lnps_project(cycle,2)` characterizes a solution as the atoms of `cycle/2` belonging to an answer set. The atom `_lnps_destroy(cycle,2,3, p(n))` means that the *destroy* function in Line 23 of Algorithm 2 randomly destroys `n`% of the projected atoms of `cycle/2`. Note that the third argument `3`=$(11)_2$ represents that all possible two arguments (`X`,`Y`) such that `cycle(X,Y)` holds are subject to destruction. The atom `_lnps_prioritize(cycle,2,1,true)` means that a statement `#heuristic cycle(X,Y): heuristic (cycle(X,Y),W,M,t), W != inf. [W,M]` is added to the `heuristic` subprogram in Line 16 of Algorithm 2. The traditional LNS heuristic of fixing the undestroyed part can be done by replacing a fact in Line 4 of Listing 3 with `_lnps_prioritize(cycle,2,inf ,true)`.

The main features of *heulingo*. The *heulingo* solver can be considered as a tool for heuristically-driven answer set optimization. In addition to the *variability* and *optimality* from LNPS, *heulingo* has the following features.

- *Expressiveness*: *heulingo* relies on ASP's expressive language that is well suited for modeling combinatorial optimization problems.
- *Implementation*: The LNPS algorithm can be compactly implemented using *clingo*'s multi-shot ASP solving and heuristic statements, as can be seen in Algorithm 2.
- *Domain heuristics*: *heulingo* allows for easy incorporation of domain heuristics in a declarative way, such as the random destruction in Listing 3. More sophisticated domain-specific heuristics can also be incorporated.
- *Usability and Compatibility*: *heulingo* can deal with any ASP encoding for optimization without any modification. All we have to do is to add an LNPS configuration. *heulingo* also supports the traditional LNS heuristic.

For *efficiency*, the question is whether the *heulingo* approach matches the performance of the (adaptive) LNS heuristic. We empirically address this question in the next section.

5 Experiments

To evaluate our approach, we carry out experiments on a challenging benchmark set and ASP encodings used in (Eiter et al. 2022a). The benchmark set consists of Traveling Salesperson Problem (TSP), Social Golfer Problem (SGP), Sudoku Puzzle Generation (SPG), Weighted Strategic Companies (WSC), and Shift Design (SD). The ASP encodings include an encoding for TSP solving in Listing 1.

We compare *heulingo* (LNPS) and *heulingo* (LNS) with *clingo* and *ALASPO*. Here, *heulingo* (X) indicates that *heulingo* uses heuristic X.

- We run *clingo*-5.6.2 [1] with the default configuration unless otherwise noted.
- We execute *heulingo* in 3 runs for each instance using the random destruction with different percentages.
- We execute *ALASPO* [2] in 3 runs for each instance with the best portfolio [3] presented in (Eiter et al. 2022a).

ALASPO is an ASP-based implementation of adaptive LNS. *ALASPO* selects in each iteration a potentially more effective destroy operator from a pre-defined portfolio. The portfolio consists of a selection strategy, multiple destroy operators, and their percentages of destruction. *ALASPO* implements three selection strategies: self-adaptive roulette-wheel

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

[2]http://www.kr.tuwien.ac.at/research/projects/bai/kr22.zip

[3]We use dynamic.json for TSP and SPG, roulette_alpha0.4.json for SGP, roulette_alpha0.8.json for WSC, and uniform.json for SD.

Instance	*clingo*	*heulingo* (LNS)			*heulingo* (LNPS)			*ALASPO*		
		avg.	min.	max.	avg.	min.	max.	avg.	min.	max.
dom_rand_70_300_1155482584_3	591	438.7	427	454	**386.3**	383	390	424.3	397	444
rand_70_300_1155482584_0	552	371.7	351	393	**326.3**	320	333	367.7	349	384
rand_70_300_1155482584_11	606	447.0	436	454	**386.3**	381	392	447.0	433	466
rand_70_300_1155482584_12	540	386.3	364	406	**344.7**	341	349	380.7	371	386
rand_70_300_1155482584_14	567	393.7	388	404	**357.7**	355	359	397.0	382	409
rand_70_300_1155482584_3	575	444.7	428	458	**408.7**	398	419	450.0	445	459
rand_70_300_1155482584_4	649	476.7	464	483	**423.0**	419	428	475.7	470	479
rand_70_300_1155482584_5	601	420.0	397	449	**367.3**	361	374	396.3	393	401
rand_70_300_1155482584_7	604	435.0	429	446	**406.3**	405	407	442.0	428	462
rand_70_300_1155482584_8	553	441.7	426	461	**387.0**	385	389	427.0	412	441
rand_70_300_1155482584_9	546	414.3	391	427	**368.3**	365	372	403.3	402	405
rand_80_340_1159656267_0	714	464.7	446	492	**410.7**	410	411	479.0	476	484
rand_80_340_1159656267_10	654	494.0	480	503	**441.3**	438	445	499.7	495	507
rand_80_340_1159656267_11	731	528.7	509	539	**464.0**	458	475	520.7	497	534
rand_80_340_1159656267_13	686	467.7	437	487	**431.3**	426	440	471.3	466	477
rand_80_340_1159656267_15	720	492.7	484	499	**439.3**	435	446	478.0	471	488
rand_80_340_1159656267_16	667	546.7	525	559	**496.3**	492	499	558.7	551	571
rand_80_340_1159656267_17	737	501.3	492	509	**449.0**	443	457	472.3	461	479
rand_80_340_1159656267_18	674	484.7	466	510	**418.7**	417	420	488.0	477	506
rand_80_340_1159656267_4	590	471.7	442	511	**418.3**	413	421	462.3	460	466
Average rate	1.000	0.728			**0.649**			0.721		

Table 1: Comparison results on traveling salesperson problem

strategy, uniform roulette-wheel strategy, and dynamic strategy. *ALASPO* provides two destroy operators: random-atoms and random-constants. The random-atoms operator is the same as the random destruction as explained in Section 4. The random-constants operator randomly selects a sample from all constants of the atoms via #show statements, and destroys all atoms containing any selected constants.

Regarding time-limits, we basically used the same time-limits as (Eiter et al. 2022a). We ran *clingo* with the default for TSP and SGP, the multi-threaded for SPG, and the unsat-core for WSC and SD, which are basically the same as (Eiter et al. 2022a). We use Python 3.9.18 to run *heulingo* and *ALASPO*. We run TSP, SGP, and SPG on Mac OS Apple M1 Ultra (20-core CPU and 128GB memory), WSC on Mac OS Intel Xeon W (12-core CPU and 96GB memory), and SD on Mac OS Apple M1 (8-core CPU and 16GB memory).

Traveling salesperson problem is a well-known optimization problem. The task is to find a Hamiltonian cycle of minimum accumulated edge costs. The time-limit is 300s for each instance. The solve-limit of *heulingo* is set to 1,210,000 and 800,000 conflicts for finding an initial solution and each iteration, respectively. We use three different percentages of the random destruction: $\{1\%, 3\%, 5\%\}$ for *heulingo* (LNPS) and $\{28\%, 30\%, 32\%\}$ for *heulingo* (LNS).

Comparison results of obtained bounds are shown in Table 1. The column shows in order the instance names, the obtained bounds of *clingo*, and the average, minimum, and maximum of obtained bounds in 3 runs of *heulingo* (LNS), *heulingo* (LNPS), and *ALASPO*. The best average in each row is highlighted in bold. The bottom shows the average rates to the bounds obtained by *clingo*. *heulingo* (LNPS) is able to find the best bounds on average for all 20 instances. *heulingo* (LNPS) succeeds in improving the bounds of *clingo* by 35.1% on average.

Social golfer problem is a combinatorial optimization problem whose goal is to schedule g groups of p players for w weeks such that no two golfers play together more than once. We consider instances with $g = 8$, $p = 4$, and $8 \leq w \leq 12$. The time-limit is 1,800s for each w. The solve-limit of *heulingo* is set to 500,000 conflicts for finding an initial solution and each iteration. We use three different percentages of the random destruction: $\{55\%, 60\%, 65\%\}$ for *heulingo* (LNPS) and $\{60\%, 65\%, 70\%\}$ for *heulingo* (LNS). In addition, we use *heulingo*'s option to limit the deterioration of objective values during the search. More precisely, this option enforces that every time a current solution and its cost are updated, the cost minus one is set to the initial bound for the objective function in the next iteration.

Comparison results of obtained bounds are shown in Table 2. *heulingo* is able to find the best bounds on average for all $8 \leq w \leq 12$. *heulingo* (LNPS) provides the same or better bounds on minimum for all w than *heulingo* (LNS). *heulingo* (LNPS) succeeds in improving the bounds of *clingo* by 24.868% on average. *heulingo* performs slightly better on average than *ALASPO*.

Sudoku puzzle generation is an optimization problem whose goal is to find an $N \times N$ sudoku puzzle with a minimal number of hints. We consider two sizes of $N = 9$ and $N = 16$. A disjunctive ASP encoding (Eiter et al. 2022b) we used takes advantage of a *saturation* technique (Eiter and Gottlob 1995). The time-limit is 600s for each size N. We

#weeks (w)	*clingo*	*heulingo* (LNS)			*heulingo* (LNPS)			*ALASPO*		
		avg.	min.	max.	avg.	min.	max.	avg.	min.	max.
8	3	2.0	2	2	**1.7**	1	2	2.0	2	2
9	7	**4.3**	4	5	4.7	4	5	5.0	5	5
10	10	**7.0**	7	7	7.7	7	8	7.7	7	8
11	11	10.0	10	10	**9.7**	9	10	10.3	10	11
12	15	**13.0**	13	13	**13.0**	13	13	13.7	13	14
Average rate	1.00000	0.75134			**0.75132**			0.80013		

Table 2: Comparison results on social golfer problem

Size	*clingo*	*heulingo* (LNS)			*heulingo* (LNPS)			*ALASPO*		
		avg.	min.	max.	avg.	min.	max.	avg.	min.	max.
9×9	21	20.0	19	21	**19.3**	19	20	20.0	20	20
16×16	232	95.3	93	98	**93.0**	90	95	96.7	95	99
Average rate	1.000	0.682			**0.660**			0.685		

Table 3: Comparison results on sudoku puzzle generation

use *clingo*'s options `--configuration=many` and `-t4` as with (Eiter et al. 2022a) for all solvers. The solve-limit of *heulingo* is set to 300,000 and 6,000 conflicts for finding an initial solution and each iteration, respectively. We use three different percentages of the random destruction: $\{$12%, 14%, 16%$\}$ for *heulingo* (LNPS) and $\{$18%, 20%, 22%$\}$ for *heulingo* (LNS).

Comparison results of the obtained number of hints are shown in Table 3. *heulingo* (LNPS) is able to find the best bounds on average for both of the two sizes. *heulingo* (LNPS) succeeds in improving the bounds of *clingo* by 34% on average. *heulingo* provides a near-optimal solution of 19 hints for $N = 9$, compared with the known minimal hint of 17.

Weighted strategic companies (Eiter et al. 2022b) is an optimization variant of the Σ_2^P-hard strategic companies problem (Cadoli, Eiter, and Gottlob 1997). The task is to find strategic sets such that the sum of weights of strategic companies is minimized. The time-limit is 3,600s for each instance. We use *clingo*'s option `--opt-strategy =usc,15` for finding an initial solution in *heulingo* and *ALASPO*. [4] The solve-limit of *heulingo* is set to 30,000,000 and 60,000 conflicts for finding an initial solution and each iteration respectively on the first 9 instances in Table 4, and to 1,000,000 and 40,000 conflicts on the next 11 instances. We use three different percentages of the random destruction: $\{4\%, 7\%, 10\%\}$ for *heulingo* (LNPS) and $\{16\%, 19\%, 22\%\}$ for *heulingo* (LNS).

Comparison results of obtained bounds are shown in Table 4. *heulingo* (LNPS) is able to find the best bounds on average for 12 instances, compared with 1 of *clingo*, 3 of *heulingo* (LNS), and 4 of *ALASPO*. Although *clingo* performs well on WSC, *heulingo* (LNPS) succeeds in improving the bounds of *clingo* by 7.7% on average. *heulingo* performs slightly better on average than *ALASPO*.

Shift design is an employee scheduling problem. The task is to find staff schedules considering the minimization of the number of shifts and both over- and understaffing. The ASP encoding (Abseher et al. 2016) we used is based on lexicographic optimization of three objective functions. The time-limit is 3,600s for each instance. We use *clingo*'s options `--configuration=handy` and `--opt-strategy=usc,3` as with (Eiter et al. 2022a) for all solvers. The solve-limit of *heulingo* is set to 900,000 and 40,000 conflicts for finding an initial solution and each iteration, respectively. We use three different percentages of the random destruction: $\{$10%, 20%, 30%$\}$ for *heulingo* (LNPS) and $\{$25%, 35%, 45%$\}$ for *heulingo* (LNS). We use *heulingo*'s option to limit the deterioration of objective values during the search, as with SGP.

Comparison results of obtained bounds are shown in Table 5. *heulingo* (LNS) is able to find the best bounds for 5 among 8 instances. Although *heulingo* (LNPS) can find better bounds for all instances than *clingo*, it does not match the performance of *heulingo* (LNS) and *ALASPO*.

Summary and discussion. *heulingo* (LNPS) was able to find the best bounds on average for 37 among all 55 instances (67% in a total). *heulingo* (LNPS) succeeded in improving the bounds of *clingo* by 35.1% for TSP, 24.8% for SGP, 34.0% for SPG, and 7.7% for WSC. On the other hand, however, *heulingo* (LNPS) met the difficulty of decreasing the bounds sufficiently on shift design. To resolve this issue, by our results, it would be effective to extend *heulingo* for adaptive LNPS combining the LNS and LNPS heuristics.

In general, it can be a hard and time-consuming task to find the best configuration for LNPS. The configurations of *heulingo* were obtained in our preliminary experiments. We tested, for each benchmark problem, some percentages of destruction less than the ones used in (Eiter et al. 2022b), taking the variability of LNPS into account. And then, we

[4]We use this option because it provides better bounds than the default configuration in our preliminary experiments.

Instance	*clingo*	*heulingo* (LNS)			*heulingo* (LNPS)			*ALASPO*		
		avg.	min.	max.	avg.	min.	max.	avg.	min.	max.
wstratcomp_001	198988	**194247.3**	192783	195411	195288.3	194517	196179	204601.7	200494	208733
wstratcomp_006	81819	80058.0	79390	80392	78144.3	77420	78928	**75128.7**	74544	75654
wstratcomp_015	163906	**160918.0**	160718	161288	161968.3	161828	162150	198342.3	190565	212559
wstratcomp_018	129784	135377.3	135204	135724	131159.0	128835	133126	**118672.7**	114983	121040
wstratcomp_019	94978	96533.0	96533	96533	96194.7	95518	96533	**91169.7**	90602	91664
wstratcomp_030	182200	180349.7	179806	181285	**179982.0**	179535	180563	210109.7	208706	212666
wstratcomp_033	**193568**	196136.3	195765	196330	194886.3	193875	196009	220024.3	218604	222432
wstratcomp_042	133273	134648.0	134164	135235	130933.0	129125	132450	**117355.3**	115786	119145
wstratcomp_050	166498	**166067.0**	165159	166859	166249.0	165667	167011	190745.3	188446	193819
wstratcomp_051	69582	60815.7	60196	61432	**59637.0**	58882	60077	63122.0	62015	64043
wstratcomp_052	70868	62591.3	62508	62677	**62044.3**	61927	62188	63597.7	63370	63948
wstratcomp_053	72848	63239.0	62312	63749	**63175.0**	63082	63276	64441.7	64136	64955
wstratcomp_054	75352	63511.0	62482	64091	**62294.7**	62082	62492	63903.7	61825	66365
wstratcomp_055	77469	67033.7	66686	67277	**66889.3**	66823	67020	68190.7	67616	69190
wstratcomp_056	68919	61420.0	60967	62309	**61161.0**	60798	61616	66436.3	65371	67025
wstratcomp_057	67836	63465.7	63259	63806	**62630.7**	62519	62762	66218.3	66013	66570
wstratcomp_058	73174	63477.0	62912	64583	**61092.7**	61003	61256	65160.0	64094	66142
wstratcomp_059	71986	62733.0	62641	62828	**61061.3**	60615	61664	62271.0	61454	62885
wstratcomp_060	75302	66103.0	65476	66649	**64355.0**	64289	64399	68726.3	68383	69398
wstratcomp_061	70918	63846.7	63640	64116	**63805.3**	63561	64084	65452.7	65173	65842
Average rate	1.000	0.934			**0.923**			0.965		

Table 4: Comparison results on weighted strategic companies

Instance	*clingo*	*heulingo* (LNS)		*heulingo* (LNPS)		*ALASPO*	
		min.	max.	min.	max.	min.	max.
4_30m	(0, 427, 50)	(0, 310, 19)	(0, 311, 22)	(0, 353, 45)	(0, 375, 44)	**(0, 310, 14)**	(0, 310, 18)
6_15m	(0, 266, 46)	**(0, 175, 20)**	(0, 176, 24)	(0, 187, 33)	(0, 202, 35)	(0, 176, 31)	(0, 189, 38)
11_30m	(0, 821, 66)	**(0, 643, 46)**	(0, 694, 61)	(0, 709, 63)	(0, 719, 64)	(0, 645, 61)	(0, 685, 60)
20_30m	(0, 1035, 60)	**(0, 900, 49)**	(0, 951, 62)	(0, 944, 65)	(0, 952, 64)	(0, 906, 55)	(0, 926, 58)
26_30m	(0, 1062, 78)	**(0, 771, 41)**	(0, 771, 48)	(0, 991, 78)	(0, 1004, 82)	(0, 784, 68)	(0, 898, 78)
27_60m	(0, 393, 25)	(0, 362, 16)	(0, 362, 18)	(0, 387, 26)	(0, 393, 25)	**(0, 362, 15)**	(0, 362, 17)
29_30m	(0, 528, 59)	**(0, 447, 38)**	(0, 450, 45)	(0, 459, 57)	(0, 460, 59)	(0, 447, 42)	(0, 451, 48)
2_30m	(0, 456, 52)	(0, 388, 22)	(0, 388, 28)	(0, 394, 45)	(0, 398, 54)	**(0, 388, 14)**	(0, 388, 16)

Table 5: Comparison results on shift design

selected the best one and two values below and above it.

It is well known that the quality of initial solutions plays an important role in the performance of SLS-based metaheuristics. The solve-limit (or the time-limit) should be large enough to obtain a good initial solution. On the other hand, it should be as small as possible in order to obtain sufficient improvements in the iteration phase. We observed that *clingo* quickly falls into saturated solutions for many instances, in the sense that it has the difficulty of decreasing the bounds sufficiently. Thus, we roughly estimated the number of conflicts on which *clingo* is stuck at saturated solutions, and then used it for the solve-limit of *heulingo*. The current implementation of *heulingo* adopts the solve-limit, rather than the time-limit, to reduce its non-deterministic behavior as much as possible.

We plan to extend the functionality of *clingo*'s API to measure whether or not *clingo* gets stuck during the search, and it will be helpful to develop not only adaptive LNPS but also other metaheuristics in ASP.

6 Related Work

New techniques for optimization in ASP have been continually developed, such as core-guided optimization (Alviano and Dodaro 2019; Alviano and Dodaro 2017; Andres et al. 2012) and complex preference handling (Brewka 2004; Brewka et al. 2015; Gebser, Kaminski, and Schaub 2011). Multi-shot ASP solving (Gebser et al. 2019; Kaminski et al. 2023) and heuristic-driven ASP solving (Dodaro et al. 2016; Gebser et al. 2013; Gebser, Ryabokon, and Schenner 2015) can be quite useful for implementing widely different SLS-based metaheuristics in ASP.

Besides work on LNS in ASP (Eiter et al. 2022b; Eiter et al. 2022a), LNS has been used in combination with MaxSAT (Demirovic and Musliu 2017; Hickey and Bacchus 2022), mixed integer programming (Fischetti and Lodi 2003; Danna, Rothberg, and Pape 2005), and constraint programming (Shaw 1998; Dekker et al. 2018; Björdal et al. 2019; Björdal et al. 2020). The use of SLS for SAT, considering variable dependencies (Kautz and Selman 2007), was studied in (Belov, Järvisalo, and Stachniak 2011).

On the traditional LNS heuristic, starting with (Shaw 1998), a great deal of research has been done (Pisinger and Ropke 2019). LNS and its extensions have been so far successfully applied in the areas of routing and scheduling problems, including vehicle routing problems (Shaw 1998), pickup and delivery problem with time windows (Ropke and Pisinger 2006), cumulative job shop scheduling problem (Godard, Laborie, and Nuijten 2005), and technician and task scheduling problem (Cordeau et al. 2010).

More recently, multi-agent path finding (Li et al. 2021; Phan et al. 2024; Tan et al. 2024), timetabling (Kiefer, Hartl, and Schnell 2017; Demirovic and Musliu 2017), bus driver scheduling (Mazzoli et al. 2024), and test laboratory scheduling (Geibinger, Mischek, and Musliu 2021) are well explored. By our results, it would be interesting to explore whether the variability feature of LNPS could be effective for these problems.

7 Conclusion

We proposed Large Neighborhood Prioritized Search (LNPS) for solving combinatorial optimization problems. We presented an implementation design of LNPS based on Answer Set Programming (ASP). The resulting *heulingo* solver is a tool for heuristically-driven answer set optimization. All source code including *heulingo* and benchmark problems is available from the web.[5]

The LNPS heuristic can be further extended to *adaptive LNPS* in which a potentially more effective combination of destruction and prioritization is selected in each iteration. The *heulingo* approach can be applied to a wide range of optimization problems. In particular, ASP-based LNPS for timetabling can be promising since ASP has been shown to be highly effective for curriculum-based course timetabling (Banbara et al. 2013; Banbara et al. 2019). For this, in our preliminary experiments, *heulingo* succeeded in finding improved bounds for 8 instances in the most difficult UD5 formulation, compared with the best known bounds obtained by more dedicated metaheuristics. From a broader perspective, integrating LNPS into MaxSAT and PB optimization would be interesting. We will investigate these possibilities, and our results will be applied to solving real-world applications.

Acknowledgments

This work was partially supported by ROIS NII Open Collaborative Research 2024-(24FP04), JSPS KAKENHI Grant Numbers (JP21H04905, JP22K11973, JP23K11047), JST CREST Grant Number JPMJCR22D3, and DFG grant SCHA 550/15.

[5]https://github.com/banbaralab/kr2024

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.

Ali, R.; El-Kholany, M.; and Gebser, M. 2023. Flexible job-shop scheduling for semiconductor manufacturing with hybrid answer set programming (application paper). In Hanus, M., and Inclezan, D., eds., *Proceedings of the Twenty-fifth International Symposium on Practical Aspects of Declarative Languages (PADL'23)*, volume 13880 of *Lecture Notes in Computer Science*, 1–11. Springer-Verlag.

Alviano, M., and Dodaro, C. 2017. Unsatisfiable core shrinking for anytime answer set optimization. In Sierra, C., ed., *Proceedings of the Twenty-sixth International Joint Conference on Artificial Intelligence (IJCAI'17)*, 4781–4785. IJCAI/AAAI Press.

Alviano, M., and Dodaro, C. 2019. Anytime answer set optimization via unsatisfiable core shrinking. *Theory and Practice of Logic Programming* 19:533–551.

Alviano, M.; Dodaro, C.; and Maratea, M. 2018. Nurse (re)scheduling via answer set programming. *Intelligenza Artificiale* 12(2):109–124.

Andres, B.; Kaufmann, B.; Matheis, O.; and Schaub, T. 2012. Unsatisfiability-based optimization in clasp. In Dovier, A., and Santos Costa, V., eds., *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12)*, volume 17, 212–221. Leibniz International Proceedings in Informatics (LIPIcs).

Banbara, M.; Soh, T.; Tamura, N.; Inoue, K.; and Schaub, T. 2013. Answer set programming as a modeling language for course timetabling. *Theory and Practice of Logic Programming* 13(4-5):783–798.

Banbara, M.; Inoue, K.; Kaufmann, B.; Okimoto, T.; Schaub, T.; Soh, T.; Tamura, N.; and Wanko, P. 2019. teaspoon: Solving the curriculum-based course timetabling problems with answer set programming. *Annals of Operations Research* 275(1):3–37.

Belov, A.; Järvisalo, M.; and Stachniak, Z. 2011. Depth-driven circuit-level stochastic local search for SAT. In Walsh, T., ed., *Proceedings of the Twenty-second International Joint Conference on Artificial Intelligence (IJCAI'11)*, 504–509. IJCAI/AAAI.

Björdal, G.; Flener, P.; Pearson, J.; and Stuckey, P. J. 2019. Exploring declarative local-search neighbourhoods with constraint programming. In Schiex, T., and de Givry, S., eds., *Proceedings of the Twenty-fifth International Conference on Principles and Practice of Constraint Programming (CP'19)*, volume 11802 of *Lecture Notes in Computer Science*, 37–53. Springer.

Björdal, G.; Flener, P.; Pearson, J.; Stuckey, P. J.; and Tack, G. 2020. Solving satisfaction problems using large-neighbourhood search. In Simonis, H., ed., *Proceedings of the Twenty-sixth International Conference on Principles and Practice of Constraint Programming (CP'20)*, volume 12333 of *Lecture Notes in Computer Science*, 55–71. Springer.

Brewka, G.; Delgrande, J.; Romero, J.; and Schaub, T. 2015. asprin: Customizing answer set preferences without a headache. In Bonet, B., and Koenig, S., eds., *Proceedings of the Twenty-ninth National Conference on Artificial Intelligence (AAAI'15)*, 1467–1474. AAAI Press.

Brewka, G. 2004. Complex preferences for answer set optimization. In Dubois, D.; Welty, C.; and Williams, M., eds., *Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning (KR'04)*, 213–223. AAAI Press.

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.

Cordeau, J.; Laporte, G.; Pasin, F.; and Ropke, S. 2010. Scheduling technicians and tasks in a telecommunications company. *Journal of Scheduling* 13(4):393–409.

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

Dekker, J. J.; de la Banda, M. G.; Schutt, A.; Stuckey, P. J.; and Tack, G. 2018. Solver-independent large neighbourhood search. In Hooker, J. N., ed., *Proceedings of the Twenty-fourth International Conference on Principles and Practice of Constraint Programming (CP'18)*, volume 11008 of *Lecture Notes in Computer Science*, 81–98. Springer.

Demirovic, E., and Musliu, N. 2017. MaxSAT-based large neighborhood search for high school timetabling. *Computers & Operations Research* 78:172–180.

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

Eiter, T., and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence* 15(3-4):289–323.

Eiter, T.; Geibinger, T.; Higuera, N.; Musliu, N.; Oetsch, J.; and Stepanova, D. 2022a. ALASPO: an adaptive large-neighbourhood ASP optimiser. In Kern-Isberner, G.; Lakemeyer, G.; and Meyer, T., eds., *Proceedings of the Nineteenth International Conference on Principles of Knowledge Representation and Reasoning (KR'22)*.

Eiter, T.; Geibinger, T.; Higuera, N. R.; Musliu, N.; Oetsch, J.; and Stepanova, D. 2022b. Large-neighbourhood search for optimisation in answer-set solving. In *Proceedings of the Thirty-sixth AAAI Conference on Artificial Intelligence (AAAI'22)*, 5616–5625. AAAI Press.

Erdem, E.; Gelfond, M.; and Leone, N. 2016. Applications of ASP. *AI Magazine* 37(3):53–68.

Fischetti, M., and Lodi, A. 2003. Local branching. *Mathematical Programming* 98(1-3):23–47.

Gebser, M.; Kaufmann, B.; Otero, R.; Romero, J.; Schaub, T.; and Wanko, P. 2013. Domain-specific heuristics in answer set programming. In desJardins, M., and Littman, M., eds., *Proceedings of the Twenty-seventh National Conference on Artificial Intelligence (AAAI'13)*, 350–356. AAAI Press.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Lindauer, M.; Ostrowski, M.; Romero, J.; Schaub, T.; and Thiele, S. 2015. *Potassco User Guide*. University of Potsdam, 2 edition.

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.; Kaminski, R.; and Schaub, T. 2011. Complex optimization in answer set programming. *Theory and Practice of Logic Programming* 11(4-5):821–839.

Gebser, M.; Ryabokon, A.; and Schenner, G. 2015. Combining heuristics for configuration problems using answer set programming. In Calimeri, F.; Ianni, G.; and Truszczyński, M., eds., *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, volume 9345 of *Lecture Notes in Artificial Intelligence*, 384–397. Springer-Verlag.

Geibinger, T.; Mischek, F.; and Musliu, N. 2021. Constraint logic programming for real-world test laboratory scheduling. In *Proceedings of the Thirty-fifth AAAI Conference on Artificial Intelligence (AAAI'21)*, 6358–6366. AAAI Press.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R., and Bowen, K., eds., *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, 1070–1080. MIT Press.

Godard, D.; Laborie, P.; and Nuijten, W. 2005. Randomized large neighborhood search for cumulative scheduling. In Biundo, S.; Myers, K. L.; and Rajan, K., eds., *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS'05)*, 81–89. AAAI.

Hickey, R., and Bacchus, F. 2022. Large neighbourhood search for anytime MaxSAT solving. In Raedt, L. D., ed., *Proceedings of the Thirty-first International Joint Conference on Artificial Intelligence (IJCAI'22)*, 1818–1824. ijcai.org.

Hoos, H. H., and Stützle, T. 2015. Stochastic local search algorithms: An overview. In Kacprzyk, J., and Pedrycz, W., eds., *Handbook of Computational Intelligence*, Springer Handbooks. Springer. 1085–1105.

Kaminski, R.; Romero, J.; Schaub, T.; and Wanko, P. 2023. How to build your own asp-based system?! *Theory and Practice of Logic Programming* 23(1):299–361.

Kautz, H. A., and Selman, B. 2007. The state of SAT. *Discrete Applied Mathematics* 155(12):1514–1524.

Kiefer, A.; Hartl, R. F.; and Schnell, A. 2017. Adaptive large neighborhood search for the curriculum-based course timetabling problem. *Annals of Operations Research* 252(2):255–282.

Li, J.; Chen, Z.; Harabor, D.; Stuckey, P. J.; and Koenig, S. 2021. Anytime multi-agent path finding via large neighborhood search. In Zhou, Z., ed., *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI'21)*, 4127–4135. ijcai.org.

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

Mazzoli, T. M.; Kletzander, L.; Hentenryck, P. V.; and Musliu, N. 2024. Investigating large neighbourhood search for bus driver scheduling. In Bernardini, S., and Muise, C., eds., *Proceedings of the Thirty-fourth International Conference on Automated Planning and Scheduling (ICAPS'24)*, 360–368. AAAI Press.

Phan, T.; Huang, T.; Dilkina, B.; and Koenig, S. 2024. Adaptive anytime multi-agent path finding using bandit-based large neighborhood search. In Wooldridge, M. J.; Dy, J. G.; and Natarajan, S., eds., *Proceedings of the Thirty-eighth AAAI Conference on Artificial Intelligence (AAAI'24)*, 17514–17522. AAAI Press.

Phillips, A. E.; Walker, C. G.; Ehrgott, M.; and Ryan, D. M. 2017. Integer programming for minimal perturbation problems in university course timetabling. *Annals of Operations Research* 252(2):283–304.

Pisinger, D., and Ropke, S. 2019. Large neighborhood search. In Gendreau, M., and Potvin, J.-Y., eds., *Handbook of Metaheuristics*. Springer International Publishing. 99–127.

Ropke, S., and Pisinger, D. 2006. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science* 40(4):455–472.

Sakkout, H. E., and Wallace, M. 2000. Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints* 4(5):359–388.

Shaw, P. 1998. Using constraint programming and local search methods to solve vehicle routing problems. In Maher, M. J., and Puget, J., eds., *Proceedings of the Fourth International Conference on Principles and Practice of Constraint Programming (CP'98)*, volume 1520 of *Lecture Notes in Computer Science*, 417–431. Springer.

Tan, J.; Luo, Y.; Li, J.; and Ma, H. 2024. Benchmarking large neighborhood search for multi-agent path finding. *CoRR* abs/2407.09451.

Zivan, R.; Grubshtein, A.; and Meisels, A. 2011. Hybrid search for minimal perturbation in dynamic CSPs. *Constraints* 16(3):228–249.