Large Neighborhood Prioritized Search for Combinatorial Optimization with Answer Set Programming

Irumi Sugimori¹, Katsumi Inoue², Hidetomo Nabeshima³, Torsten Schaub⁴, Takehide Soh⁵, Naoyuki Tamura⁵, Mutsunori Banbara¹
¹Nagoya University, Japan
²National Institute of Informatics, Japan
³University of Yamanashi, Japan
⁴Universität Potsdam, Germany
⁵Kobe 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 heduino 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 destroying and repairing 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; Björkäld et al. 2019; Björkäld 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
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

\[
a_0 :- a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n.
\]

Each \(a_i\) 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 ‘.’ represent if, conjunction, and default negation, respectively. A period ‘.’ terminates each rule. Intuitively, the rule means that \(a_0\) must be assigned to true if \(a_1, \ldots, a_m\) are true and \(a_{m+1}, \ldots, a_n\) are false. A rule whose body is empty (i.e., \(a_0\)) is called fact. Facts are always true. A rule whose head is empty is called integrity constraint:

\[
:- a_1, \ldots, a_m, \text{not } a_{m+1}, \ldots, \text{not } a_n.
\]

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 \(\ell_0: \ell_1, \ldots, \ell_m\). Each \(\ell_i\) is a literal, and \(\ell_1, \ldots, \ell_m\) is called condition like in mathematical set notation. A *cardinality constraint* of the form \(\{c_1; \ldots; c_n\} = k\) represents that exactly \(k\) conditional literals among \(\{c_1; \ldots; c_n\}\) must be satisfied. A weak constraint of the form \(\sim L; \{w, t\}\) represents preferences in ASP, which is equivalent to \#minimize \(\{w, t: L\}\). Here, \(w\) is a weight, and \(t\) and \(L\) 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 \(a: L; [w, m]\) where \(m\) and \(w\) are terms representing a heuristic modifier and its value, respectively.

*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 \(\text{cycle}(X, Y)\) 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 \(\text{heu}\) and an embedded Python script.

Once *clingo* accepts the program (Listing 2) combined

### Listing 1: A traditional ASP encoding for TSP solving

```plaintext
#program heu.
#heuristic cycle(X,Y): heu(cycle(X,Y),w,M). [W,M].
#end.

ctl.ground(["base",[]])
ctl.solve()

a = Function("heu",[Function("cycle ",[Number(1),Number(2)]),Number(1), Function("true")])
ctl.add("ext",[],f"#external {a}.")
ctl.ground(["ext",[]])
ctl.ground(["heu",[]])
ctl.assign_external(a,True)
ctl.solve()
ctl.release_external(a)
ctl.solve()
```

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

```plaintext
#external
#program
#heuristic

#program heu.
#heuristic cycle(X,Y): heu(cycle(X,Y),w,M). [W,M].
#end.

def main(ctl):
    ctl.ground(["base",[]])
    ctl.solve()
    a = Function("heu",[Function("cycle ",[Number(1),Number(2)]),Number(1), Function("true")])
    ctl.add("ext",[],f"#external {a}.")
    ctl.ground(["ext",[]])
    ctl.ground(["heu",[]])
    ctl.assign_external(a,True)
    ctl.solve()
    ctl.release_external(a)
    ctl.solve()
```

```python
from clingo import Number, Function
def main(ctl):
    ctl.ground(["base",[]])
    ctl.solve()
    a = Function("heu",[Function("cycle ",[Number(1),Number(2)]),Number(1), Function("true")])
    ctl.add("ext",[],f"#external {a}.")
    ctl.ground(["ext",[]])
    ctl.ground(["heu",[]])
    ctl.assign_external(a,True)
    ctl.solve()
    ctl.release_external(a)
    ctl.solve()
```
Algorithm 1 Large Neighborhood Prioritized Search

**Input:** a feasible solution \( x \)

1. \( x^* \leftarrow x \)
2. **while** stop criterion is not met **do**
3. \( x^t \leftarrow \text{prioritized-search}(\text{destroy}(x)) \)
4. **if** \( \text{accept}(x^t, x) \) **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** \( x^* \)

with a TSP instance of fact format and ASP encoding (Listing 1), a clingo object is created and bound to variable \( \text{ctl} \) (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., 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 \( \text{heu}(\text{cycle}(1,2), 1, \text{true}) \) is added to the subprogram \( \text{ext} \) 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 \( \text{cycle}(1,2) \). \( [1,\text{true}] \) is activated in the second call of solve in Line 14. Intuitively, this heuristic statement means that the atom \( \text{cycle}(1,2) \) is set to true with higher priority during the search. More precisely, the solver decides first on \( \text{cycle}(1,2) \) 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 destroy operator randomly destroys parts of \( x \) and returns the undestroyed part. The prioritized-search operator returns a feasible solution, which is reconstructed from 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 destroy and prioritized-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 LNPS.
since it turns out to be the same solution.

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 pseudo-code of our LNPS algorithm using *clingo*'s multi-shot ASP solving is shown in Algorithm 2. The key idea is utilizing the `#heuristic` statement to implement the prioritized-search operator of LNPS. The input consists of a problem instance, an ASP encoding, and an LNPS configuration. The variables `sol`, `sol_best`, and `sol_tmp` correspond to `x`, `x^*`, and `x^` respectively in Algorithm 1. The variables `cost`, `cost_best`, and `cost_tmp` represent the objective values `c(x)`, `c(x^*)`, and `c(x^)` 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.

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, ..., X_n):heuristic(p(X_1, ..., 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, ..., 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 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...
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", []])` {grounding $P \cup E$}
2. `(ret, sol, cost) ← ctl.solve()`
3. **if** `ret = OPTIMUM FOUND` **then**
   4. **return** `sol`
5. **end if**
6. `sol_best, cost_best ← sol, cost`
7. `ctl.ground(["config", []])` {grounding $C$}
8. `lnps_config ← get_lbps_config()`
9. `rules ← ""`
10. **for** $c$ **in** `lnps_config` **do**
11.   `p ← c.get_predicate_name()`
12.   `n ← c.get arity()`
13.   `atom ← "p(X_1, X_2, \ldots, X_n)"`
14.   `rules ← 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)`
17. `finished ← False`
18. `step ← 0`
19. `prev_heu_atoms ← []`
20. `variability ← check variability(lnps_config)`
21. **while** `finished = False` **do**
22.   `step ← step + 1`
23.   `undestroyed ← destroy(sol, lnps_config)`
24.   `heu_atoms ← prioritize(undestroyed, lnps_config, step)`
25.   `ctl.release_external(prev_heu_atoms)`
26.   `statements ← ""`
27.   **for** `heuristic(a, w, m, t)` **in** `heu_atoms` **do**
28.     `statements ← statements + "\#external heuristic(a,w,m,t)."
29.   **end for**
30. `ctl.add("external", [], statements)`
31. `ctl.ground(["external", []])`
32. `ctl.ground(["heuristic", [step]])`
33. **for** `heuristic(a, w, m, t)` **in** `heu_atoms` **do**
34.   `ctl.assign_external(heuristic(a, w, m, t), True)`
35. **end for**
36. `prev_heu_atoms ← heu_atoms`
37. `(ret, sol_tmp, cost_tmp) ← ctl.solve()`
38. **if** `ret = OPTIMUM FOUND and variability = True` **then**
39.   `finished ← True`
40. **end if**
41. **if** `accept(cost_tmp, cost) = True` **then**
42.   `sol, cost ← sol_tmp, cost_tmp`
43. **end if**
44. **if** `cost_tmp < cost_best` **then**
45.   `sol_best, cost_best ← sol_tmp, cost_tmp`
46. **end if**
47. `increase_solve_limit()`
48. **end while**
49. **return** `sol_best`
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 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.

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 \(g = 8, p = 4, 4\).

¹https://potassco.org/lingo/

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

³We use dynamic.json for TSP and SPG, roulette_alpha0.4.json for SGP, roulette_alpha0.8.json for WSC, and uniform.json for SD.
8 ≤ w ≤ 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 ≤ w ≤ 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 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 $\Sigma^p_2$-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 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

---

⁴We use this option because it provides better bounds than the default configuration in our preliminary experiments.
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* (LNS) 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 selected the best one and two values below and above it.

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*. 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), mixed integer programming (Fischetti and Lodi 2003; Danna, Rothberg, and Pape 2005), and constraint programming (Shaw 1998; Dekker et al. 2018; Björndal et al. 2019; Björndal 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), timetabling (Kiefer, Hartl, and Schnell 2017; Demirovic and Musliu 2017), 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 https://github.com/banbaralab/kr2024.

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

---

<table>
<thead>
<tr>
<th>Instance</th>
<th>clingo</th>
<th>heulingo (LNS)</th>
<th>heulingo (LNPS)</th>
<th>ALASPO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>min.</td>
<td>max.</td>
<td>min.</td>
<td>max.</td>
</tr>
<tr>
<td>4,30m</td>
<td>(0, 427, 50)</td>
<td>(0, 310, 19)</td>
<td>(0, 353, 45)</td>
<td>(0, 310, 14)</td>
</tr>
<tr>
<td>6,15m</td>
<td>(0, 266, 46)</td>
<td>(0, 175, 20)</td>
<td>(0, 187, 33)</td>
<td>(0, 176, 31)</td>
</tr>
<tr>
<td>11,30m</td>
<td>(0, 821, 66)</td>
<td>(0, 643, 46)</td>
<td>(0, 709, 63)</td>
<td>(0, 645, 61)</td>
</tr>
<tr>
<td>20,30m</td>
<td>(0, 1035, 60)</td>
<td>(0, 900, 49)</td>
<td>(0, 944, 65)</td>
<td>(0, 906, 55)</td>
</tr>
<tr>
<td>26,30m</td>
<td>(0, 1062, 78)</td>
<td>(0, 771, 41)</td>
<td>(0, 991, 78)</td>
<td>(0, 784, 68)</td>
</tr>
<tr>
<td>27,60m</td>
<td>(0, 393, 25)</td>
<td>(0, 362, 16)</td>
<td>(0, 387, 26)</td>
<td>(0, 362, 15)</td>
</tr>
<tr>
<td>29,30m</td>
<td>(0, 528, 59)</td>
<td>(0, 447, 38)</td>
<td>(0, 459, 57)</td>
<td>(0, 447, 42)</td>
</tr>
<tr>
<td>2,30m</td>
<td>(0, 456, 52)</td>
<td>(0, 388, 22)</td>
<td>(0, 394, 45)</td>
<td>(0, 388, 14)</td>
</tr>
</tbody>
</table>
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.

References


