Towards Inductive Learning of Domain-Specific Heuristics for ASP

Richard Comploi-Taupe[1]

[1]Siemens AG Österreich, Vienna, Austria

Abstract

Domain-specific heuristics are a crucial technique for the efficient solving of problems that are large or computationally hard. Answer Set Programming (ASP) systems support declarative specifications of domain-specific heuristics to improve solving performance. However, such heuristics must be invented manually so far. Inventing domain-specific heuristics for answer-set programs requires expertise with the domain under consideration and familiarity with ASP syntax, semantics, and solving technology. The process of inventing useful heuristics would highly profit from automatic support. This paper presents a first step in this direction. We use Inductive Logic Programming (ILP) to learn declarative domain-specific heuristics from examples stemming from (near-)optimal answer sets of small but representative problem instances. Our experimental results indicate that the learned heuristics improve solving performance and solution quality when solving larger, harder problem instances.

Keywords

Answer Set Programming, Domain-Specific Heuristics, Inductive Logic Programming

1. Introduction

Answer Set Programming (ASP) [1, 2, 3, 4] is a declarative problem-solving approach applied successfully in many industrial and scientific domains. For large and complex problems, however, domain-specific heuristics may be needed to achieve satisfactory performance [5, 6].

Therefore, state-of-the-art ASP systems offer ways to integrate domain-specific heuristics in the solving process. An extension for WASP [7] facilitates external *procedural* heuristics consulted at specific points during the solving process via an API [5]. *Declarative* specifications of domain-specific heuristics in the form of so-called *heuristic directives* are supported by CLINGO [8, 9, 10] and ALPHA [11, 12, 13, 14].

However, such heuristics must be invented manually so far. Human domain experts and ASP experts are needed to invent suitable domain-specific heuristics. This paper presents a first step toward the automatic learning of declarative heuristics.

Our core idea is to use Inductive Logic Programming (ILP) to learn declarative domain-specific heuristics from examples stemming from (near-)optimal answer sets of small but representative problem instances. These heuristics can then be used to improve solving performance and solution quality for larger, harder problem instances. Our experimental results are promising, indicating that this goal can be achieved.

1st International Workshop on HYbrid Models for Coupling Deductive and Inductive ReAsoning (HYDRA)

richard.taupe@siemens.com (R. Comploi-Taupe)

0000-0001-7639-1616 (R. Comploi-Taupe)

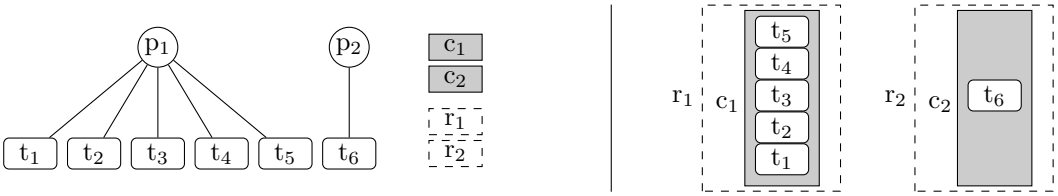

Figure 1: Sample HRP instance (left) and one of its solutions (right) [13]

After covering preliminaries in Section 2, we present our main contribution in Section 3. Section 4 presents experimental results, and Section 5 describes related work. Section 6 concludes the paper by giving an outlook on future work.

2. Preliminaries

In this section, we introduce a running example and cover preliminaries on domain-specific heuristics in ASP and inductive learning in ASP. We assume familiarity with ASP and refer to [1, 2, 3, 4] for detailed introductions.

2.1. Running Example: The House Reconfiguration Problem (HRP)

The House Reconfiguration Problem (HRP) [15] is an abstracted version of industrial (re)configuration problems, e.g., rack configuration. A complete description is available from the ASP Challenge 2019,[1] and an encoding is available in Anna Ryabokon's PhD thesis [16].

Formally, HRP is defined as a modification of the House Configuration Problem (HCP) [13].

Definition 1 (HCP). *The input for the* House Configuration Problem (HCP) *is given by four sets of constants P, T, C, and R representing persons, things, cabinets, and rooms, respectively, and an ownership relation $PT \subseteq P \times T$ between persons and things.*

The task is to find an assignment of things to cabinets $TC \subseteq T \times C$ and cabinets to rooms $CR \subseteq C \times R$ such that: (1) each thing is stored in a cabinet; (2) a cabinet contains at most five things; (3) every cabinet is placed in a room; (4) a room contains at most four cabinets; and (5) a room may only contain cabinets storing things of one person.

Definition 2 (HRP). *The input for the* House Reconfiguration Problem (HRP) *is given by an HCP instance $H = \langle P, T, C, R, PT \rangle$, a legacy configuration $\langle TC', CR' \rangle$, and a set of things $T' \subseteq T$ that are defined as "long" (all other things are "short").*

The task is then to find an assignment of things to cabinets $TC \subseteq T \times C$ and cabinets to rooms $CR \subseteq C \times R$, that satisfies all requirements of HCP as well as the following ones: (1) a cabinet is either small or high; (2) a long thing can only be put into a high cabinet; (3) a small cabinet occupies 1 and a high cabinet 2 of 4 slots available in a room; (4) all legacy cabinets are small.

The sample HRP instance shown in Fig. 1 comprises two cabinets, two rooms, five things belonging to person p_1, and one thing belonging to person p_2. A legacy configuration is empty,

[1] https://sites.google.com/view/aspcomp2019/problem-domains

and all things are small. In a solution, the first person's things are placed in cabinet c_1 in the first room, and the thing of the second person is in cabinet c_2 in the second room. For this sample instance, a solution of HRP corresponds to a solution of HCP [13].

Instances consist of facts over the following predicates: `cabinetDomainNew/1` defines potential cabinets, and `roomDomainNew/1` defines potential rooms; `thingLong/1` defines which things are long; and `legacyConfig/1` defines all the other data in the legacy configuration, e.g., `legacyConfig(personTOthing(p1,t1))` defines that person `p1` owns thing `t1`, and `legacyConfig(roomTOcabinet(r1,c1))` specifies one tuple in the legacy assignment of cabinets to rooms.

The main two choice rules guessing the assignment of things to cabinets and the assignment of cabinets to rooms look as follows:

```
{ cabinetTOthing(C,T) } :- cabinetDomain(C), thing(T).
{ roomTOcabinet(R,C) }  :- roomDomain(R),  cabinet(C).
```

To define the domains of cabinets and rooms as the union of existing objects and newly available identifiers, the encoding also contains the following rules:

```
cabinetDomain(C) :- cabinetDomainNew(C).
roomDomain(R)    :- roomDomainNew(R).
```

Instances may optionally include atoms of various predicates to define *costs* for specific actions such as placing a thing in a cabinet, placing a cabinet in a room, reusing an existing placement of a thing in a cabinet, reusing an existing placement of a cabinet in a room, removing a thing from a cabinet, removing a cabinet from a room, etc. These costs are then determined based on the difference between solution $\langle TC, CR \rangle$ and legacy configuration $\langle TC', CR' \rangle$. A weak constraint in the encoding instructs the solver to minimise the costs.

Each available instance belongs to one of four instance classes [15, 16]: *Empty* ("ec", empty legacy configuration); *long* ("lt", some things are long); *new room* ("nr", some cabinets have to be reallocated to new rooms); and *swap* ("ss", only one person and a specific pattern of legacy configuration).

2.2. Domain-Specific Heuristics in ASP

To solve large instances of industrial problems, employing an ASP solver out-of-the-box may not be sufficient. Sophisticated encodings or solver tuning methods (such as portfolio solving) are common ways to deal with this issue.

Domain-specific heuristics are another way to speed-up answer set solving. They were even needed to achieve breakthroughs in solving industrial configuration problems with ASP [5].

Several approaches have been proposed to embed heuristic knowledge into the ASP solving process. HWASP [5] extends WASP [7] by facilitating external procedural heuristics consulted at specific points during the solving process via an API.

A declarative approach to formulating domain-specific heuristics in ASP is provided by CLINGO,[2] supporting `#heuristic` directives [9, 10]. Heuristic directives enable the declarative specification of weights determining atom and sign orders in a solver's internal decision heuristics. An atom's weight influences the order in which atoms are considered by the solver when

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

making a decision. A sign modifier instructs whether the selected atom must be assigned true or false. Atoms with a higher weight are assigned a value before atoms with a lower weight.

In the syntax for (non-ground) heuristic directives in CLINGO (1), ha is an atom, hB is a conjunction of literals representing the heuristic body, and w, p, and m are terms [10].

$$\#\texttt{heuristic}\ \ ha : hB. \qquad [w@p, m] \tag{1}$$

The optional term p gives a preference between heuristic values for the same atom (preferring those with higher p). The term m specifies the type of heuristic information and can take the following values: `sign`, `level`, `true`, `false`, `init` and `factor`. For instance, heuristics for m=`init` and m=`factor` allow modifying initial and actual atom scores evaluated by the solver's decision heuristics (e.g., VSIDS). The m=`sign` modifier forces the decision heuristics to assign an atom ha a specific sign, i.e., true or false, and m=`level` allows for the definition of an order in which the atoms are assigned—the larger the value of w, the earlier an atom must be assigned. Finally, m=`true` specifies that a should be assigned true with weight w if hB is satisfied, and m=`false` is the analogue heuristics that assigns a false.

A new approach implemented in the lazy-grounding ASP system ALPHA, based on the CLINGO approach, has introduced novel semantics for heuristic directives aimed at non-monotonic heuristics [12, 13, 14].

2.3. Inductive Learning in ASP

Inductive Logic Programming (ILP) is an approach to learning a program that explains a set of examples given some background knowledge. ILASP [17, 18] is a system capable of learning Answer Set Programs, including normal rules, choice rules, and hard and weak constraints.

ILASP operates on a *learning task*, which consists of three components [17]: The *background knowledge* B (an ASP program already known before learning), the *mode bias* M (that expresses which ASP programs can be learned), and the *examples* E (which specify properties the learned program must satisfy). When the properties specified by a particular example in E are satisfied, the example is said to be *covered*.

ILASP finds a program (often called a hypothesis) H such that $B \cup H$ covers every example in E (or, if the examples are considered *noisy*, such that the total penalty of non-covered examples is minimised) [17]. H is an element of the search space defined by M.

2.3.1. Mode Bias

The mode bias consists of a set of *mode declarations*. There are several types of mode declarations [19], two of which we will use in this paper: `#modeh` and `#modeb` specify what the heads and bodies of learned rules may look like, respectively. A *placeholder* is a term `var(t)` or `const(t)` for some constant term `t`. Such placeholders can be replaced by any variable or constant (respectively) of *type* `t`.

As a simple example, Listing 1 shows part of the mode bias for a learning task for the HRP.

Listing 1: Part of the mode bias for HRP

```
#modeh(cabinetTOthing(var(cabinet),var(thing))).
```

```
#modeb(cabinetDomainNew(var(cabinet))).
#modeb(thing(var(thing))).
```

The first mode declaration specifies that the binary predicate `cabinetTOthing` can be used in the head of rules and that its terms are of variable types `cabinet` and `thing` (in this order). The other two mode declarations specify which predicates can occur in the bodies of learned rules. Note that the same terms may be used in learned rules wherever the same term types are used.

Thus, the rule space defined by the mode bias given in Listing 1 consists of the following rule:

```
cabinetTOthing(V1,V2) :- cabinetDomainNew(V1), thing(V2).
```

2.3.2. Examples

A positive example is given by a `#pos` statement, and a negative example by a `#neg` statement [17, 19]. Each example consists of several components, the following of which are relevant for this paper:[3] A set of ground atoms called *inclusions*, a set of ground atoms called *exclusions*, and an optional set of rules (usually just facts) called *context*.

A positive example is covered iff there exists at least one answer set for $B \cup H$ that contains all of the inclusions and none of the exclusions. A negative example states that there must *not* exist an answer set that contains all of its inclusions and none of its exclusions [17, 19].

The *context* is the problem instance to which the inclusions and exclusions refer (considering the usual distinction between unvarying problem encoding and problem instances specified by facts) [17, 19].

Listing 2 shows a simplified example for HRP stating that `cabinetTOthing(1,2)` shall be true for the problem instance in which `cabinetDomainNew(1)` and `thing(2)` are true.

Listing 2: A simplified example in a learning task for HRP

```
#pos(
 { cabinetTOthing(1,2) },           % inclusions
 { },                               % exclusions
 { cabinetDomainNew(1). thing(2). } % context
).
```

3. Inductive Learning of Domain-Specific Heuristics

This section presents the main contribution of this paper—our approach to the inductive learning of domain-specific heuristics for ASP.

The basic idea is to solve a small but representative instance of a problem, use the resulting answer set as a positive example for inductive learning, learn a set of definite rules, and transform the learned rules into declarative heuristic directives in the form of Eq. (1) presented in Section 2.2. These heuristics can then be used to speed up solving larger/harder instances of the same problem.

The rule space for ILASP is defined as follows in our approach:

[3] So far, we do not use example identifiers, ordering examples, and noisy examples.

- All predicates appearing in the heads of choice rules[4] can be used in the heads of learned rules (`#modeh`).
- All (other) predicates appearing in the original program can be used in the bodies of learned rules (`#modeb`).
- The same variable type is used several times, wherever a variable denotes the same real-world concept.

The background knowledge is the original program without any instance.[5] Choice rules were not included, however, because we observed that ILASP only learns anything in our example domain when choice rules are not part of the background knowledge. We presume this is because we need to abstract away from the complete problem specification a bit to learn part of the missing information. Constraints were also not included because the rules we want to learn don't have to satisfy all constraints of the program. When used as heuristics, it suffices for them to give a general indication of what decisions might be useful during solving, even if some of these decisions will have to be backtracked.

As a positive example for learning, one answer set for a small but representative problem instance is used. In case the underlying problem is an optimisation problem (like the HRP described in Section 2.1), we propose to use a (near-)optimal answer set for this process. The (yet unproven) hypothesis is that learning from better answer sets yields better heuristics.

We use context-dependent examples; the context is given by the problem instance. The set of inclusions corresponds to the whole answer set, and the set of exclusions is empty.

3.1. Learning in the House Reconfiguration Problem

Let us re-consider the running example from Section 2.1, the House Reconfiguration Problem (HRP). The smallest instance is called "`ec-0001-house-t0100`"; it is an HCP instance (i.e., it does not contain a legacy configuration)[6] and contains 100 things distributed among 20 persons. A near-optimal[7] answer set for this instance was computed by CLINGO [8] and used as a single positive example.

We built a learning task for this simple instance as described at the beginning of Section 3. The entire mode bias is shown in Listing 3. Some predicates that we did not expect to be useful were not included in the mode bias to keep the search space in a feasible range and thus improve ILASP's solving performance. Some other predicates were removed since they led to learning rules that did not make sense in the domain context.

[4] More strictly speaking, the head of a choice rule contains a collection of choice elements, each of the form $a : l_1, \ldots, l_k$ [20]. The predicates used for a are the ones that can be used in the heads of learned rules.

[5] Rules involving aggregates had to be removed because they are not supported by ILASP. See https://doc.ilasp.com/specification/background.html for details on the syntax for the background knowledge.

[6] Since an HCP instance is not representative of HRP instances with non-empty legacy configuration, future work should address learning from small instances of non-HCP instance classes.

[7] CLINGO v. 5.4.0, which was used in its default configuration (i.e., with no parameters apart from file names) computed an answer set with optimisation value 120, but was not able to prove optimality within several days of search.

Listing 3: Full mode bias for HRP as used for our experiments

```
#modeh(cabinet(var(cabinet))).
#modeh(room(var(room))).
#modeh(cabinetTOthing(var(cabinet),var(thing))).
#modeh(roomTOcabinet(var(room),var(cabinet))).
#modeh(cabinetHigh(var(cabinet))).
#modeh(cabinetSmall(var(cabinet))).

#modeb(cabinetDomainNew(var(cabinet))).
#modeb(roomDomainNew(var(room))).
#modeb(cabinetDomain(var(cabinet))).
#modeb(roomDomain(var(room))).
#modeb(thing(var(thing))).
#modeb(personTOthing(var(person),var(thing))).
#modeb(legacyConfig(person(var(person)))).
#modeb(legacyConfig(thing(var(thing)))).
#modeb(legacyConfig(personTOthing(var(person),var(thing)))).
#modeb(thingShort(var(thing))).
#modeb(person(var(person))).
#modeb(personTOroom(var(person),var(room))).
```

ILASP was used with arguments `--version=2i --no-constraints --no-aggregates`, i.e., with version 2i of the learning algorithm and omitting constraints and aggregates (choice rules) from the search space. We chose version 2i instead of version 4 for performance reasons. ILASP 2i needed approximately 4 minutes to solve the full learning task on the author's personal computer, while ILASP 4 needed over 12 minutes and yielded a slightly different result.[8] As an alternative approach, ILASP 4 was able to solve the learning task within seconds when we cut down the instance by removing parts of the facts while still keeping the general structure of the instance.

The following rules form the learned hypothesis:[9]

```
cabinetSmall(V1)       :- cabinetDomainNew(V1).
cabinetTOthing(V1,V2) :- cabinetDomain(V1),
                         legacyConfig(personTOthing(V3,V2)).
roomTOcabinet(V1,V2)   :- cabinetDomain(V2), roomDomain(V1).
```

As a next step, we transformed these rules to heuristics in the form of Eq. (1) manually by using the head of each rule as ha and the body as hB. Since the heads of the heuristics shall eventually become true, we use the modifier `true`. And since we don't have any information on prioritising the heuristics at this point, all get the same weight 1. This is the result:

```
#heuristic cabinetSmall(V1) :
           cabinetDomainNew(V1).                                  [1,true]
#heuristic cabinetTOthing(V1,V2) :
           cabinetDomain(V1), legacyConfig(personTOthing(V3,V2)). [1,true]
#heuristic roomTOcabinet(V1,V2) :
           cabinetDomain(V2), roomDomain(V1).                     [1,true]
```

[8] When using ILASP 4 instead of ILASP 2i, the third rule in the learned hypothesis contains the atom `roomDomainNew(V1)` instead of `roomDomain(V1)`.

[9] Note that in ILASP's output, body literals are separated by semicolons instead of the usual commas, which is valid GRINGO syntax [10] but does not conform to ASP-Core-2 [20]. We replaced these semicolons with commas in our examples to avoid confusion.

The learned heuristics instruct the solver to try and create as many small cabinets as possible and to try all possible cabinet-to-thing and room-to-cabinet assignments.

3.2. Limitations and Future Work

Our construction of the mode bias was a bit ad hoc, as described above, aiming at a feasible size of the search space and learning reasonable rules. Finding a meaningful, consistent way to derive the mode bias is subject of future work.

We also want to explore if and how yet unused features of ILASP can be helpful in our approach. This includes learning non-normal rules and weak constraints, multiple examples, exclusions, noise, and using built-in predicates such as arithmetic comparisons in `#modeb`.

Potential extensions of our approach that should be addressed include assigning variable types automatically and determining different weights and priorities for the learned heuristic directives.

4. Experimental Results

To test the effects of the learned heuristics, we used CLINGO to solve all available HRP instances (94 in number) with and without the learned heuristics. The HRP instances stem from previous experiments [13, 14]. These instances were generated in the pattern of the original instances [15]. This pattern represents four different reconfiguration scenarios encountered in practice, and the instances are abstracted real-world instances. Our instances are considerably larger than the original ones, though (ranging up to 800 things, while the original instances used at most 280 things).

Each of the machines used to run the experiments was equipped with two Intel® Xeon® E5-2650 v4 @ 2.20GHz CPUs with 12 cores. Furthermore, each machine had 251 GiB of memory and ran Ubuntu 16.04.1 LTS Linux. Scheduling of benchmarks was done with HTCondor™ together with the ABC Benchmarking System [21].[10] PYRUNLIM[11] was used to limit time consumption to 10 minutes per instance, memory to 40 GiB, and swapping to 0. Care was taken to avoid side effects between CPUs, e.g., by requesting exclusive access to an entire machine for each benchmark from HTCondor.

CLINGO was instructed to search for the optimal answer set in its default configuration, given an encoding including a weak constraint. Since the system operated only on a single encoding containing the learned heuristics, the parameter `--heu=Domain` was used to achieve one configuration in which CLINGO used the domain-specific heuristics (and one in which it did not). After 10 minutes per instance, search was aborted and the optimisation value of the best solution found so far was recorded.

Table 1 shows the achieved optimisation values and the relative improvement when using the learned heuristics for all 36 instances that could be solved with or without domain-specific heuristics. For the other 58 instances, no answer set could be found either way; therefore, they are not included in the table.

[10] http://research.cs.wisc.edu/htcondor, https://github.com/credl/abcbenchmarking
[11] https://alviano.com/software/pyrunlim/

Table 1
Experimental results: Achieved optimisation values without ("def") and with ("heu") learned heuristics, and relative improvement

Instance	def	heu	Improvement
ec-0001	159	120	25%
ec-0002	180	150	17%
ec-0003	216	183	15%
ec-0004	253	230	9%
ec-0005	250	271	-8%
ec-0006	285	327	-15%
ec-0007	363	1270	-250%
ec-0008	356	658	-85%
ec-0009	3313	879	73%
ec-0010	4187	1785	57%
ec-0011	3236	1970	39%
ec-0012	5515	∞	$-\infty$
lt-0001	2101	499	76%
lt-0002	4867	577	88%
lt-0003	6883	656	90%
lt-0004	5539	737	87%
lt-0005	5363	947	82%
lt-0006	7757	909	88%
lt-0007	∞	998	100%
lt-0008	7411	∞	$-\infty$
nr-0001	4116	909	78%
nr-0002	5708	622	89%
nr-0003	5526	683	88%
nr-0004	5846	758	87%
nr-0005	6879	818	88%
nr-0006	11399	895	92%
nr-0007	10326	1769	83%
nr-0008	8718	1035	88%
nr-0009	10841	∞	$-\infty$
ss-0001	132	545	-313%
ss-0002	1496	1040	30%
ss-0003	2555	1225	52%
ss-0004	3255	1402	57%
ss-0005	3883	2632	32%
ss-0006	∞	1814	100%
ss-0007	∞	3359	100%

The first column shows the instance identifier. The first two characters of each identifier refer to one of the four instance classes of the HRP (cf. Section 2.1). The numeric part of the identifier increases with increasing instance size.

The second and third columns contain the achieved optimisation values without and with learned heuristics, respectively. The symbol ∞ is used when no answer set could be found within the time limit of 10 minutes.

The fourth column shows the change in the optimisation value when using the learned heuristics relative to solving without domain-specific heuristics. A positive percentage signifies an improvement, and negative values indicate a deterioration. The value 100% is used in cases where an answer set could be found only when using heuristics[12] and the value $-\infty$ is used when an answer set could be found only without heuristics.

The learned heuristics seem to have positive effects even though they are (still) straightfor-

[12]The rationale for this is that the optimisation value of a minimisation problem can be considered infinitely high when no solution is found, and that dividing an infinitely high value by an infinitely high value approaches 1.

ward. The average cost improvement over all instances (excluding those where no answer set could be found when using heuristics) is 38%. However, improvement varies strongly between different instances. Furthermore, results are sensitive to the chosen time-out. For example, we observed stronger improvements (on fewer solved instances) when experimenting with a time-out of three minutes instead of ten.

Besides our experiments with CLINGO, we also experimented with the lazy-grounding ASP system ALPHA [11]. This system accepts heuristic directives in a slightly different syntax [13, 14]. Without domain-specific heuristics, ALPHA could solve none of the HRP instances under consideration. The heuristics learned by our approach enabled ALPHA to solve three instances (without optimisation, which is not yet supported by ALPHA).[13]

5. Related Work

Balduccini [22] has also presented an approach to learning domain-specific heuristics offline from representative instances. The basic idea, which is very different to our approach, is to record which choices are made in the path of a search tree that led to a solution and to use this information to compute probabilities for decisions on ground atoms. These probabilities are then used while solving other problem instances to reduce the likelihood of backtracks. The approach is restricted to DPLL-style solvers like SMODELS [23], and extending it to CDCL-based systems like CLINGO is mentioned as future work.

A similar approach is aimed at configuration problems encoded as constraint satisfaction problems (CSPs) [24].

6. Conclusions and Future Work

We have proposed a novel approach to inductively learning declarative specifications of domain-specific heuristics for ASP from answer sets of small but representative instances. Our approach employs the inductive learning system ILASP. Utilising an example representing a significant real-world configuration problem, we have demonstrated that simple heuristics can easily be learned.

Experimental results are promising: Some instances could be solved only using the learned heuristics, and solution quality improved considerably on average.

The fact that so far, we have only learned very simple heuristics and those already led to significant improvements is encouraging. Future work will show whether our method can be extended to learn more complex heuristics that can improve solving performance and solution quality even further.

Acknowledgments

The author is grateful to Mark Law for helping with using ILASP, and to the TU Wien for providing access to computational resources for running the experiments.

[13] Human-made heuristics, however, enable ALPHA to solve up to 59 of these instances [13, 14].

References

[1] M. Gelfond, Y. Kahl, Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach, Cambridge University Press, New York, NY, USA, 2014.

[2] V. Lifschitz, Answer Set Programming, Springer, 2019. doi:10.1007/978-3-030-24658-7.

[3] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Answer Set Solving in Practice, Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, 2012.

[4] C. Baral, Knowledge Representation, Reasoning and Declarative Problem Solving, Cambridge University Press, 2003.

[5] C. Dodaro, P. Gasteiger, N. Leone, B. Musitsch, F. Ricca, K. Schekotihin, Combining answer set programming and domain heuristics for solving hard industrial problems (application paper), Theory Pract. Log. Program. 16 (2016) 653–669. doi:10.1017/S1471068416000284.

[6] A. A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E. C. Teppan, Industrial applications of answer set programming, Künstliche Intell. 32 (2018) 165–176. doi:10.1007/s13218-018-0548-6.

[7] M. Alviano, G. Amendola, C. Dodaro, N. Leone, M. Maratea, F. Ricca, Evaluation of disjunctive programs in WASP, in: M. Balduccini, Y. Lierler, S. Woltran (Eds.), Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings, volume 11481 of *Lecture Notes in Computer Science*, Springer, 2019, pp. 241–255. doi:10.1007/978-3-030-20528-7_18.

[8] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with clingo, Theory Pract. Log. Program. 19 (2019) 27–82. doi:10.1017/S1471068418000054.

[9] M. Gebser, B. Kaufmann, J. Romero, R. Otero, T. Schaub, P. Wanko, Domain-specific heuristics in answer set programming, in: M. desJardins, M. L. Littman (Eds.), Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA, AAAI Press, 2013, pp. 350–356. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI13/paper/view/6278.

[10] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, P. Wanko, Potassco guide version 2.2.0, 2019. URL: https://github.com/potassco/guide/releases/tag/v2.2.0.

[11] A. Weinzierl, Blending lazy-grounding and CDNL search for answer-set solving, in: M. Balduccini, T. Janhunen (Eds.), Logic Programming and Nonmonotonic Reasoning - 14th International Conference, LPNMR 2017, Espoo, Finland, July 3-6, 2017, Proceedings, volume 10377 of *Lecture Notes in Computer Science*, Springer, 2017, pp. 191–204. doi:10.1007/978-3-319-61660-5_17.

[12] R. Taupe, K. Schekotihin, P. Schüller, A. Weinzierl, G. Friedrich, Exploiting partial knowledge in declarative domain-specific heuristics for ASP, in: B. Bogaerts, E. Erdem, P. Fodor, A. Formisano, G. Ianni, D. Inclezan, G. Vidal, A. Villanueva, M. D. Vos, F. Yang (Eds.), Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25,

2019, volume 306 of *EPTCS*, 2019, pp. 22–35. doi:10.4204/EPTCS.306.9.

[13] R. Taupe, G. Friedrich, K. Schekotihin, A. Weinzierl, Solving configuration problems with ASP and declarative domain-specific heuristics, in: M. Aldanondo, A. A. Falkner, A. Felfernig, M. Stettinger (Eds.), Proceedings of the 23rd International Configuration Workshop (CWS/ConfWS 2021), Vienna, Austria, 16-17 September, 2021, volume 2945 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021, pp. 13–20. URL: http://ceur-ws.org/Vol-2945/21-RT-ConfWS21_paper_4.pdf.

[14] R. Comploi-Taupe, Speeding Up Lazy-Grounding Answer Set Solving, Ph.D. thesis, Alpen-Adria-Universität Klagenfurt, 2021. URL: https://digital.obvsg.at/urn/urn:nbn:at:at-ubk:1-41351.

[15] G. Friedrich, A. Ryabokon, A. A. Falkner, A. Haselböck, G. Schenner, H. Schreiner, (Re)configuration using answer set programming, in: K. M. Shchekotykhin, D. Jannach, M. Zanker (Eds.), Proceedings of the IJCAI 2011 Workshop on Configuration, Barcelona, Spain, July 16, 2011, volume 755 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2011, pp. 17–24. URL: http://ceur-ws.org/Vol-755/paper03.pdf.

[16] A. Ryabokon, Knowledge-based (Re)configuration of Complex Products and Services, Ph.D. thesis, Alpen-Adria-Universität Klagenfurt, 2015. URL: http://netlibrary.aau.at/urn:nbn:at:at-ubk:1-26431.

[17] M. Law, A. Russo, K. Broda, The ILASP system for inductive learning of answer set programs, CoRR abs/2005.00904 (2020). arXiv:2005.00904.

[18] M. Law, Conflict-driven inductive logic programming, Theory and Practice of Logic Programming (2022). doi:10.1017/S1471068422000011.

[19] ILASP Limited, The ILASP manual, 2022. URL: https://doc.ilasp.com/.

[20] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-Core-2 input language format, Theory Pract. Log. Program. 20 (2020) 294–309. doi:10.1017/S1471068419000450.

[21] C. Redl, Automated benchmarking of KR-systems, in: S. Bistarelli, A. Formisano, M. Maratea (Eds.), Proceedings of the 23rd RCRA International Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion 2016 (RCRA 2016), Genova, Italy, November 28, 2016, volume 1745 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2016, pp. 45–56. URL: http://ceur-ws.org/Vol-1745/paper4.pdf.

[22] M. Balduccini, Learning and using domain-specific heuristics in ASP solvers, AI Commun. 24 (2011) 147–164. doi:10.3233/AIC-2011-0493.

[23] T. Syrjänen, I. Niemelä, The smodels system, in: T. Eiter, W. Faber, M. Truszczynski (Eds.), Logic Programming and Nonmonotonic Reasoning, 6th International Conference, LPNMR 2001, Vienna, Austria, September 17-19, 2001, Proceedings, volume 2173 of *Lecture Notes in Computer Science*, Springer, 2001, pp. 434–438. doi:10.1007/3-540-45402-0_38.

[24] D. Jannach, Toward automatically learned search heuristics for CSP-encoded configuration problems – results from an initial experimental analysis, in: Configuration Workshop, 2013, pp. 9–13.