Challenges in Answer Set Solving

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub*

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam

Abstract. Michael Gelfond's application of Answer Set Programming (ASP) in the context of NASA's space shuttle has opened the door of the ivory tower. His project has not only given our community self-confidence and served us as a reference for grant agencies and neighboring fields, but ultimately it helped freeing the penguins making them exclaim *"Yes, we can* [fly] *!"*. The community has taken up this wonderful assist to establish ASP as a prime tool for declarative problem solving in the area of Knowledge Representation and Reasoning. Despite this success, however, ASP has not yet attracted broad attention outside this area. This paper aims at identifying some current challenges that our field has to overcome in the mid-run to ultimately become a full-fledged technology in Informatics.

1 Introduction

A central goal of the field of Knowledge Representation and Reasoning is to furnish methods for automated yet declarative problem solving. Unlike programming, where the idea is to use programs to specify how a problem is to be solved, the idea is to view a program as a formal representation of the problem as such. Accordingly, the mere execution of a traditional program is replaced by an automated search through the solution space spanned by the problem's representation. Looking at chess, this amounts to specifying the rules of chess and searching in the resulting state space rather than writing a chess program. In the latter case, the intelligence lies with the programmer, while in the former it is the system that cares about finding a solution in a smart way. Also, the procedural approach is bound to playing chess, while the problem-solving oriented approach is free to play any other game that can be specified in the realm of the input language, as witnessed in the area of General Game Playing [1].

Answer Set Programming (ASP; [2]) is nowadays one of the most popular approaches to declarative problem solving. This is due to its appealing combination of a rich yet simple modeling language with high-performance solving capacities. ASP has its roots in Knowledge Representation and (Nonmonotonic) Reasoning, Logic Programming (with negation), Databases, and Boolean Constraint Solving. ASP allows for solving all search problems in *NP* (and NP^{NP}) in a uniform way, offering more succinct problem representations than propositional logic [3]. Meanwhile, ASP has been used in many application areas, among them, product configuration [4], decision support for NASA shuttle controllers [5], composition of Renaissance music [6], synthesis of multiprocessor systems [7], reasoning tools in systems biology [8, 9], team-building [10],

* Affiliated with the School of Computing Science at Simon Fraser University, Canada, and the Institute for Integrated and Intelligent Systems at Griffith University, Australia.

and many more.[1] The success story of ASP has its roots in the early availability of ASP solvers, beginning with the *smodels* system [11], followed by *dlv* [12], SAT-based ASP solvers, like *assat* [13] and *cmodels* [14], and the conflict-driven ASP solver *clasp*, demonstrating the performance and versatility of ASP solvers by winning first places at international competitions like ASP'09, PB'09, and SAT'09.

Despite this success, ASP has not yet become an out-of-the-box technology like, for instance Prolog, or even a natural subject of undergraduate teaching, like relational databases or functional programming. A major prerequisite for this ambitious goal is to enable the scalability of ASP, even when used by non-expert users. However, ASP's declarative methodology for problem posing and solving does not scale for free. Actually, the decent performance of ASP systems often conceals a lack of scalability and has so far impeded the community's awareness of this limitation. In what follows, we aim at sharpening the community's awareness of the challenges ahead of us and to motivate a joined effort in addressing them. Enjoy!

2 ASP Solving

As with traditional computer programming, the ASP solving process amounts to a closed loop. Its steps can be roughly classified into

1. Modeling,
2. Grounding,
3. Solving,
4. Visualizing, and
5. Software Engineering.

We have illustrated this process in Figure 1 by giving the associated components. It all

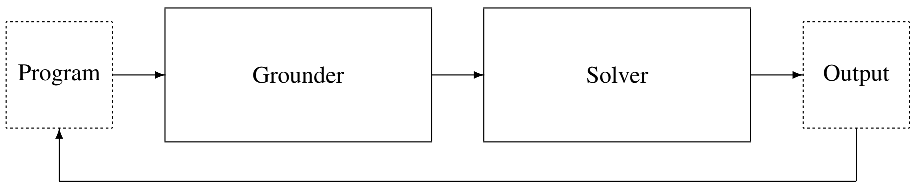

Fig. 1. ASP Solving Process

starts with a modeling phase, which results in a first throw at a representation of the given problem in terms of logic programming rules. The resulting program[2] is usually

[1] See `http://www.cs.uni-potsdam.de/~torsten/asp` for an extended listing of ASP applications.

[2] This is of course a misnomer but historically too well established to be dropped.

formulated by means of first-order variables, which are systematically replaced by elements of the Herbrand universe in a subsequent grounding phase. This yields a finite propositional program that is then fed into the actual ASP solver. The output of the solver varies depending on the respective reasoning mode. Often, it consists of a textual representation of a sequence of answer sets. Depending on the quality of the resulting answer, one then either refines the last version of the problem representation or not.

As pointed out in the introductory section, the strongholds of ASP are usually regarded to be its rich modeling language as well as its high-performance solving capacities. We revisit both topics, Modeling and Solving, in the following sections and close by commenting briefly on the remaining issues.

Grounding, or more generally propositionalization, is common to all approaches exploiting the computational advantages of propositional representations; besides ASP this includes, for instance, Satisfiability Checking (SAT;[15]) and Planning [16]. Unlike the latter, however, ASP is the only area having widely used, highly optimized grounding systems. Even so, some problems are as a matter of fact prone to a combinatorial blow-up in space and thus beyond the realm of any efficient grounding procedure. Given that such problems usually comprise large but flat domains, there is definitely a need to further investigate database techniques for grounding or finite-domain constraint processing techniques for hybrid solving, as done in [17–19].

Visualization becomes indispensable when problems and their resulting answer sets involve a high number of entities and relations among them. Although the relational nature of ASP suggests tables as an obvious means of visualization, at the end of the day, tables are just another domain-independent way of characterizing the actual objects of interest. But despite this application-specific nature of visualization, it remains a great challenge whether a highly declarative, general-purpose paradigm such as ASP can be lifted from a purely textual level towards a more appealing graphical stage. A first approach in this direction is described in [20].

Software-engineering becomes more and more important to ASP in view of its increasing range of applications. Among others, this involves effective development tools, including editors and debuggers, as well as the dissemination of (open-source) tools and libraries connecting ASP to other computing paradigms. For instance, classical debugging techniques do often not apply to ASP because of its high degree of declarativity, or in other words, its lack of a procedural semantics that could be subject to debugging and tracing. This "curse of declarativity" is well recognized withing the ASP community and addressed within a dedicated workshop series [21, 22]; first approaches can be found in [23–26].

3 Modeling

ASP Modeling is an art; it requires craft, experience, and knowledge. Although the resulting problem specifications are usually quite succinct and easy to understand, crafting such beautiful specification that also shows its best performance is not as obvious as it might seem. To illustrate this, we conduct a little case study in the next section.

All experiments were conducted with `gringo` (3.0.0[3]) and `clasp` (1.3.4).

[3] The release candidate was referred to as `bingo`.

3.1 A case-study

Let us consider the well-known n-Queens problem that consists of placing n queens on an $n \times n$-square board such that no queen may attacks another one.

Following the common *generate-and-test* methodology of ASP, this problem can be specified in four rules, the first providing a generator positioning n queens on the $n \times n$ board, and the three remaining ones excluding two queens on the same row, column, and diagonal, respectively. The first throw at a formalization of these constraints in ASP is given in Table 1.

```
% place n queens on the chess board
n { q(1..n,1..n) } n.

% at most one queen per row/column
:- q(X,Y1), q(X,Y2), Y1 != Y2.
:- q(X1,Y), q(X2,Y), X1 != X2.

% at most one queen per diagonal
:- q(X1,Y1), q(X2,Y2), X1 != X2, #abs(X1-X2) == #abs(Y1-Y2).
```

Table 1. n-Queens problem, first throw.

A first improvement is to eliminate symmetric ground rules, expressing the same constraint. For example, rule `:- q(X,Y1), q(X,Y2), Y1 != Y2.` gives rise to ground instances `:- q(3,1), q(3,2).` and `:- q(3,2), q(3,1).` both of which prohibit the same placements of queens. This redundancy can be removed by some simple symmetry breaking. In our example, it suffices to replace inequality `Y1 != Y2` by `Y1 < Y2`. Globally applying this simple way of symmetry breaking to the encoding in Table 1 yields the one in Table 2. The latter encoding strictly halves the number of ground instances obtained from the three integrity constraints. For instance, on the 10-Queens problem, the number of ground rules drops from 2941 to 1471. Despite this reduction, the improved encoding still scales poorly, as witnessed by the 1646701 rules obtained after 28.26s on the 100-Queens problem (cf. Table 5 at the end of this section).

Analyzing the encoding in Table 2 a bit further reveals that all three integrity constraints give rise to a cubic number of ground instances, that is, on the n-Queens problem they produce $O(n^3)$ ground rules. This can be drastically reduced by replacing the rule restricting placements in rows, viz. `:- q(X,Y1), q(X,Y2), Y1 < Y2.`, by[4]

```
:- X = 1..n, not 1 { q(X,Y) } 1.
```

[4] The construct `X = 1..n` can be read as $X \in \{1, \ldots, n\}$.

```
% place n queens on the chess board
n { q(1..n,1..n) } n.

% at most one queen per row/column
:- q(X,Y1), q(X,Y2), Y1 < Y2.
:- q(X1,Y), q(X2,Y), X1 < X2.

% at most one queen per diagonal
:- q(X1,Y1), q(X2,Y2), X1 < X2, #abs(X1-X2) == #abs(Y1-Y2).
```

Table 2. n-Queens problem, second throw.

asserting that there is exactly one queen in a row. One rule per row, results in $O(n)$ rules (each of size $O(n)$) rather than $O(n^3)$ as before. Clearly, the same can be done for columns, yielding `:- Y = 1..n, not 1 { q(X,Y) } 1.` Note that the new rules imply that there is *exactly one* queen per row and column, respectively. Hence, we may replace the cardinality constraint `n { q(1..n,1..n) } n.` by the unconstrained choice `{ q(1..n,1..n) }`. This is advantageous because it constitutes practically no constraint for `clasp`. Finally, what can we do about the integrity constraint controlling diagonal placements? It fact, the same aggregation can be done for the diagonals, once we have an enumeration scheme. The idea is to enumerate diagonals in two ways, once from the upper left corner to the lower right corner, and similarly from the upper right corner to the lower left corner. Let us illustrate this for $n = 4$:

	1	2	3	4
1	1	2	3	4
2	2	3	4	5
3	3	4	5	6
4	4	5	6	7

	1	2	3	4
1	4	3	2	1
2	5	4	3	2
3	6	5	4	3
4	7	6	5	4

These two enumeration schemes can be captured by the equations $D = X + Y - 1$ and $D = X - Y + n$, respectively. For instance, the first equation tells us that diagonal 6 consists of positions $(4, 3)$ and $(3, 4)$. Given both equations, we may replace the rule restricting placements in diagonals by the two following rules:

```
:- D = 1..n*2-1, not { q(X,Y) : D==X-Y+n } 1.
:- D = 1..n*2-1, not { q(X,Y) : D==X+Y-1 } 1.
```

As above, we thus obtain one rule per diagonal, inducing $O(n)$ ground rules (each of size $O(n)$). The resulting encoding is given in Table 3.

For 10 and 100 queens, the encoding of Table 3 yields 55 and 595 ground rules, respectively, in contrast to the 1471 and 1646701 rules obtained with the encoding in Table 2. Despite the much smaller grounding size, however, the grounding time does not scale as expected. To see this, note that grounding the encoding in Table 3 for 100

```
% place n queens on the chess board
{ q(1..n,1..n) }.

% exactly one queen per row/column
:- X = 1..n, not 1 { q(X,Y) } 1.
:- Y = 1..n, not 1 { q(X,Y) } 1.

% at most one queen per diagonal
:- D = 1..n*2-1, not { q(X,Y) : D==X-Y+n } 1.
:- D = 1..n*2-1, not { q(X,Y) : D==X+Y-1 } 1.
```

Table 3. n-Queens problem, third throw.

queens takes less than second, while 500 queens require more than 100 seconds of grounding time (although only 2995 ground rules are produced).

Further investigations[5] reveal that the last two rules in Table 3 are the source of the problem. In fact, it turns out that during grounding the tests `D==X-Y+n` and `D==X-Y-1` are repeated over and over. This can be avoided by pre-calculating both conditions. To this end, we add the rules

```
d1(X,Y,X-Y+n) :- X = 1..n, Y = 1..n.
d2(X,Y,X+Y-1) :- X = 1..n, Y = 1..n.
```

and replace the two conditions `D==X-Y+n` and `D==X-Y-1` by `d1(X,Y,D)` and `d2(X,Y,D)`, respectively. The resulting encoding is given in Table 4. Although this encoding adds a quadratic number of facts, their computation is linear and exploits indexing techniques known from database systems.

3.2 Some Hints on (Manual) Modeling

Finally, let us give some hints on modeling based upon our experience.

1. Keep the grounding compact
   – If possible, use aggregates
   – Try to avoid combinatorial blow-up
   – Project out unused variables
   – But don't remove too many inferences!
2. Add additional constraints to prune the search space
   – Consider special cases
   – Break symmetries
   – ...
   – Test whether the additional constraints really help
3. Try different approaches to model the problem

[5] This can be done with `gringo`'s debug option `--verbose`.

```
% place n queens on the chess board
{ q(1..n,1..n) }.

% exactly one queen per row/column
:- X = 1..n, not 1 { q(X,Y) } 1.
:- Y = 1..n, not 1 { q(X,Y) } 1.

% pre-calculate the diagonals
d1(X,Y,X-Y+n) :- X = 1..n, Y = 1..n.
d2(X,Y,X+Y-1) :- X = 1..n, Y = 1..n.

% at most one queen per diagonal
:- D = 1..n*2-1, not { q(X,Y) : d1(X,Y,D) } 1.
:- D = 1..n*2-1, not { q(X,Y) : d2(X,Y,D) } 1.
```

Table 4. n-Queens problem, fourth throw.

n	Encoding 1		Encoding 2		Encoding 3		Encoding 4	
50	2.95	42.10	1.95	41.16	0.12	0.04	0.05	0.05
100	41.50	—	28.26	—	0.81	0.16	0.13	0.18
500	—	—	—	—	96.91	16.34	3.60	16.84
1000	—	—	—	—	767.70	166.80	20.98	168.75

Table 5. Experiments contrasting different encodings of the n-Queens problem. All runs conducted with `clasp --heuristic=vsids --quiet`.

  - Problems involving time steps might be parallelized
4. It (still) helps to know the systems
   - `gringo` offers options to trace the grounding process
   - `clasp` offers many options to configure the search[6]

3.3 Non-ground pre-processing

We have conducted a preliminary case-study illustrating the potential of non-ground pre-processing techniques. To this end, we explored two simple techniques.[7]

Concretion The idea of concretion is to replace overly general rules by their effectively used partial instantiations. In other words, concretion eliminates redundant rule instances from the program whenever their contribution is re-constructable from an answer set and not needed otherwise. Consider the following simple program.

```
q(X,Y) :- p(X), p(Y).
r(X)   :- q(X,X).
```

[6] `clasp` was run with `--heuristic=vsids` to solve the large n-Queens problem.

[7] We are grateful to Michael Grosshans and Arne König for accomplishing this case-study!

Given that the binary predicate q is only used with identical arguments, concretion replaces the first rule by

```
q(X,X) :- p(X).
```

Similarly, concretion replaces the first rule in

```
q(X,X) :- p(X).
r(X)   :- q(X,2).
```

by

```
q(2,2) :- p(2).
```

Note that concretion does not preserve answer sets. However, the original answer sets can be reconstructed from the resulting ones by means of the original program.

Projection aims at reducing the number of variables in a rule in order to scale down the number of its ground instances. To this end, one eliminates variables with singleton (or say "isolated") occurrences and replaces the encompassing literal(s) with a new literal only containing the remaining variables.
For illustration, consider the rule

```
q(X) :- z(X,W), v(X,Y,Z,0), u(Z,W).
```

In this rule, variable Y is irrelevant to the remainder of the rule and can thus be eliminated by projection. As a result, projection replaces the above rule by the two following ones.

```
q(X) :- z(X,W), v_new(X,Z), u(Z,W).
v_new(X,Z) :- v(X,Y,Z,0).
```

The predicate v_new yields the value combinations necessary for deriving instances of q. Note that this reduces the number of variables from four to three, which may significantly reduce the number of ground instances depending on the size of the respective domains.
Projection was first applied to ASP in [27], tracing back to well-known database techniques [28, p. 176].

Similar and often much more refined techniques can be found in the literature, however, frequently in different research areas, like (deductive) databases, traditional logic programming, automated planning, etc.

As a proof-of-concept, we have implemented both techniques in the prototypical grounder *pyngo*[8] [29] and conducted some preliminary experiments. First and foremost, it is worth mentioning that both techniques are useless for common benchmarks because most of them have been designed by experts in ASP. For instance, both techniques cannot really improve the encodings furnished during the last modeling-oriented ASP competition [30]. Hence, our experiment design rather aimed at use-cases covering non-expert usage of ASP.

[8] *pyngo* is a Python-based grounder, developed by Arne König for rapid prototyping of grounder features.

The two use-cases envisaged for concretion are the usage of library programs in more specific contexts. To this end, we took encodings necessitating complex sub-problems. The first benchmark set computes for each element of the residue class ring modulo n the multiplicative inverse, while the second one takes numbers n and a and conducts the Solovay-Strassen primality test. Note that the benchmarks involve computing the greatest common divisor, quadratic remainders, and potencies in the residue class rings.

Table 5(a) summarizes the results obtained on both benchmarks. We measured the run-time of *clingo* restricted to 120sec on a 3.6GHz PC running Linux. Comparing

(a) Multiplicative Inverse

n	Original	Transform
50	0.680	0.010
100	10.690	0.060
200	–	0.210
500	–	2.030
1000	–	15.490
1500	–	51.000
2000	–	114.240
2500	–	–

(b) Solovay-Streets Test

n	Original	Transform
100	0.130	0.000
500	12.650	0.050
1000	97.880	0.120
2000	–	0.390
5000	–	2.410
10000	–	9.590
20000	–	37.370
50000	–	–

Table 6. Experimental results applying concretion

the run-times on the original encoding with those obtained after applying concretion (indicated as 'Transform' in Table 5(a)), we observe a clear improvement after pre-processing. In this case, this betterment is due to the fact that all of the aforementioned sub-problems were subject to concretion.

Our second experiment deals with single-player games from the area of General Game Playing [1] and aims at illustrating the potential of projection. All benchmarks were obtained through automatic transformations from original specifications in the Game Description Language [31] into ASP. This provides us with a benchmark set not at all designed for ASP and rather comparable to Prolog programs (over a finite Herbrand base).

Table 7 summarizes our results. This time we distinguish between grounding and solving time, and provide as well the size of the ground program. This is interesting because the program modifications of projection are more substantial, and may thus have different effects. As before, all benchmarks were run on a 3.6GHz PC under Linux yet now with a timeout of 1200sec; the grounding size is given in MB. We observe in 22 out of 32 benchmarks an improvement that usually affected both grounding as well as solving time. A remarkable decrease was observed on *Sudoku*, where grounding time and size was reduced by two orders of magnitude and solving time dropped from over 20min, viz. the cut-off time, to a bit more than 4sec. But despite these ameliorations, projection can also lead to a deterioration of performance, as drastically shown on *God*, where projection increased the grounding by an order of magnitude and pushed solving time beyond the cut-off.

	Original			Transform		
Game	Grounding	Size	Solving	Grounding	Size	Solving
8puzzle	1.59	9.8	74.37	0.19	1.4	6.90
aipsrovers	0.23	1.9	1.16	0.20	1.9	1.08
asteroids	0.07	0.5	1.15	0.12	0.8	1.73
asteroidsparallel	0.17	1.2	53.38	0.21	1.6	39.02
asteroidsserial	0.40	2.8	14.62	0.49	4.0	4.81
blocksworldparallel	0.11	0.8	0.08	0.04	0.3	0.01
brainteaser	0.03	0.1	0.02	0.07	0.2	0.23
chinesecheckers	12.35	96.9	21.71	11.36	92.9	12.23
circlesolitaire	0.06	0.4	0.07	0.05	0.3	0.04
coins	0.04	0.2	0.02	0.03	0.2	0.03
firefighter	0.09	0.7	0.04	0.05	0.4	0.02
god	38.76	354.8	349.8	1151.91	426.5	–
hanoi6	1.29	9.6	-	1.36	11.0	–
hanoi7(1)	7.83	44.0	-	8.04	52.0	–
hanoi7(2)	7.84	44.0	4.83	8.14	52.0	7.42
hanoi	0.20	1.8	-	0.26	2.2	16.03
incredible	0.31	2.2	2.28	0.08	0.6	0.12
knightmove	0.24	1.7	-	0.25	1.8	1031.80
lightsout	0.04	0.1	2.16	0.02	0.2	15.73
maxknights	1.10	7.6	0.79	0.58	3.7	0.42
pancakes6	39.99	325.7	737.33	40.11	325.8	631.70
pancakes	43.72	325.7	556.22	39.28	325.8	465.92
peg(1)	64.42	509.2	-	3.43	30.7	–
peg(2)	66.68	509.2	-	3.24	30.7	–
queens	2.36	13.3	36.58	5.81	34.3	29.26
slidingpieces	3.93	24.6	2.79	3.53	32.5	4.66
snake2008	0.59	4.2	27.18	0.66	4.8	5.50
snake2009	0.94	6.5	542.57	0.85	6.3	–
sudoku	221.94	1643.8	-	3.64	34.1	4.30
tpeg	65.84	509.1	-	3.10	31.5	–
troublemaker	0.03	0.1	0.04	0.01	0.1	0.00
twistypassage	0.93	5.9	1.27	0.83	6.8	0.66

Table 7. Experimental results applying projection

All in all, our preliminary case-study demonstrates the great potential of automatic non-ground pre-processing techniques for improving ASP code. Moreover, it revealed significant research challenges in identifying not only more such pre-processing techniques but furthermore in gearing them towards true improvements.

4 Solving

Advanced Boolean Constraint Solving is sensitive to parameter tuning. Clearly, this carries over to modern ASP Solving. In fact, an ASP Solver like `clasp` offers an arsenal of parameters for controlling the search for answer sets. Choosing the right parameters often makes the difference between being able to solve a problem or not.

4.1 Another case-study

Let us analyze the performance of `clasp` in the context of the NP problems used at the 2009 ASP Solver Competition [30]. To this end, we begin with contrasting the default configuration of `clasp` with a slightly changed configuration denoted by `clasp`$^+$. The latter invokes `clasp` with options `--sat-prepro` and `--trans-ext=dynamic`. For comparison, we also give results for ASP solvers `cmodels` [32] and `smodels` [11].[9] All experiments were run on an Intel Quad-Core Xeon E5520, possessing 2.27GHz processors, under Linux. Each benchmark instance was run three times with every solver, each individual run restricted to 600 seconds and 2GB RAM. Our experiments are summarized in Table 8, giving average runtimes in seconds (and numbers of timed-out runs in parentheses) for every solver on each benchmark class, with timeouts taken as 600 seconds. The table gives in the column headed by # the number of instances per benchmark class. In addition, Table 8 provides the respective partition into satisfiable and unsatisfiable instances in parentheses. The rows marked with $\varnothing(\Sigma)$ provide the average runtimes and number of timeouts wrt the considered collection of benchmark classes.

We see that the performance of `clasp`'s default configuration is quite inferior to that of `clasp`$^+$ on this set of benchmarks. In fact, almost all benchmark classes contain extended rules. However, not all of them are substantial enough to warrant a dedicated treatment. This situation is accounted for by the configuration of `clasp`$^+$, using a hybrid treatment of extended rules. The option `--trans-ext=dynamic` excludes "small" extended rules from an intrinsic treatment (cf. [33]) and rather transforms them into normal ones. This results in a higher number of Boolean constraints, which is counterbalanced by invoking option `--sat-prepro` that enables Resolution-based pre-processing [34]. This greatly reduces the number of timeouts, namely, from 144 benchmarks unsolved by the default configuration to 105 unsolved ones by `clasp`$^+$.

Let us get a closer look at three classes that were difficult for `clasp`. To this end, it is important to get a good idea about the features of the considered benchmark class. This involves static properties of the benchmark as such as well as dynamic features reflecting its solving process.

[9] Version information is given below Table 8.

Benchmark	#		`clasp`		`clasp`$^+$		`cmodels[m]`		`smodels`	
15Puzzle	16	(16/0)	33.01	(0)	20.18	(0)	31.36	(0)	600.00	(48)
BlockedNQueens	29	(15/14)	5.09	(0)	4.91	(0)	9.04	(0)	29.37	(0)
ChannelRouting	10	(6/4)	120.13	(6)	120.14	(6)	120.58	(6)	120.90	(6)
EdgeMatching	29	(29/0)	0.23	(0)	0.41	(0)	59.32	(0)	60.32	(0)
Fastfood	29	(10/19)	1.17	(0)	0.90	(0)	29.22	(0)	83.93	(3)
GraphColouring	29	(9/20)	421.55	(60)	357.88	(39)	422.66	(57)	453.77	(63)
Hanoi	15	(15/0)	11.76	(0)	3.97	(0)	2.92	(0)	523.77	(39)
HierarchicalClustering	12	(8/4)	0.16	(0)	0.17	(0)	0.76	(0)	1.56	(0)
SchurNumbers	29	(13/16)	17.44	(0)	49.60	(0)	75.70	(0)	504.17	(72)
Solitaire	27	(22/5)	204.78	(27)	162.82	(21)	175.69	(21)	316.96	(36)
Sudoku	10	(10/0)	0.15	(0)	0.16	(0)	2.55	(0)	0.25	(0)
WeightBoundedDomSet	29	(29/0)	123.13	(15)	102.18	(12)	300.26	(36)	400.84	(51)
$\varnothing(\Sigma)$ *(tight)*	264	(182/82)	*78.22*	*(108)*	*68.61*	*(78)*	102.50	(120)	257.99	(318)
ConnectedDomSet	21	(10/11)	40.42	(3)	36.11	(3)	7.46	(0)	183.76	(15)
GeneralizedSlitherlink	29	(29/0)	0.10	(0)	0.22	(0)	1.92	(0)	0.16	(0)
GraphPartitioning	13	(6/7)	9.27	(0)	7.98	(0)	20.19	(0)	92.10	(3)
HamiltonianPath	29	(29/0)	0.07	(0)	0.06	(0)	0.21	(0)	2.22	(0)
KnightTour	10	(10/0)	124.29	(6)	91.80	(3)	242.48	(12)	150.55	(3)
Labyrinth	29	(29/0)	123.82	(12)	82.92	(6)	142.24	(6)	594.10	(81)
MazeGeneration	29	(10/19)	91.17	(12)	89.89	(12)	90.41	(12)	293.62	(42)
Sokoban	29	(9/20)	0.73	(0)	0.80	(0)	3.39	(0)	176.01	(15)
TravellingSalesperson	29	(29/0)	0.05	(0)	0.06	(0)	317.82	(7)	0.22	(0)
WireRouting	23	(12/11)	42.81	(3)	36.36	(3)	175.73	(12)	448.32	(45)
$\varnothing(\Sigma)$ *(nontight)*	241	(173/68)	*43.27*	*(36)*	*34.62*	*(27)*	100.19	(49)	194.11	(204)
$\varnothing(\Sigma)$	505	(355/150)	*62.33*	*(144)*	*53.16*	*(105)*	101.45	(169)	228.95	(522)

`clasp` (1.3.1)
`clasp`$^+$ = `clasp --sat-prepro --trans-ext=dynamic`
`cmodels[m]` (3.79 with `minisat` 2.0)
`smodels` (2.34 with option `-restart`)

Table 8. Solving the 2009 ASP Competition (NP problems)

The *WeightBoundedDomSet* benchmark class consists of tight, rather small, unstructured logic programs having many solutions. The two latter features often suggest a more aggressive restart strategy, making the solver explore an increased number of different locations in the search space rather than performing fewer yet more exhaustive explorations.

Indeed, invoking `clasp` (1.3.1) with `--restarts=256`, indicated by `clasp`* below, yields an average Time of 4.64s (versus 123.13s) and makes the number of timeouts drop from 15 to zero.

Benchmark	#	`clasp`	`clasp`$^+$	`clasp`*	`cmodels[m]`	`smodels`
WBDS	29 (29/0)	123.13(15)	102.18(12)	4.64 (0)	300.26 (36)	400.84(51)

The *ConnectedDomSet* benchmark class consists of non-tight, rather small logic programs containing a single large integrity cardinality constraint. The difficulty in solving this class lies in the latter integrity constraint. In fact, the default configuration of `clasp` treats extended rules as special Boolean constraints rather then initially compiling them into normal rules. However, on this benchmark the conflict learning scheme of `clasp` spends a lot of time extracting all implicit conflicts comprised in this constraint. Unlike `clasp` (and `smodels`), `cmodels` unwraps these conflicts when compiling them into normal rules, so that its solving process needs not spend any time in recovering them.

This behavior is nicely reflected by invoking `clasp` (1.3.1) with Option `--trans-ext=weight`[10], indicated by `clasp`* below, yielding an average time of 4.19s (versus 40.42s) and no timeouts (versus 3).

Benchmark	#	`clasp`	`clasp`$^+$	`clasp`*	`cmodels[m]`	`smodels`
ConnectedDomSet	21 (10/11)	40.42(3)	36.11 (3)	4.19 (0)	7.46 (0)	183.76(15)

The *KnightTour* benchmark class consists of non-tight logic programs containing many large cardinality constraints and exhibiting many large back-jumps during solving. The latter runtime feature normally calls for progress saving [35] enforcing the same truth assignment to atoms chosen on subsequent descents into the search space. Also, it is known from the SAT literature that this works best in combination with an aggressive restart strategy.

In fact, invoking `clasp` (1.3.1) with `--restarts=256` and `--save-progress`, indicated by `clasp`* below, reduces the average time to 1.47s (versus 124.29s) and leaves us with no timeouts (versus 6).

Benchmark	#	`clasp`	`clasp`$^+$	`clasp`*	`cmodels[m]`	`smodels`
KnightTour	10 (10/0)	124.29(6)	91.80 (3)	1.47 (0)	242.48 (12)	150.55 (3)

4.2 Some Hints on (Manual) Solving

The question then arises how to deal with this vast "configuration space" and how to conciliate it with the idea of declarative problem solving. Currently, there seems to be no true alternative to manual fine-tuning when addressing highly demanding application problems.

As rules of thumb, we usually start by investigating the following options:

- `--heuristic`: Try *vsids* instead of `clasp`'s default *berkmin*-style heuristic.
- `--trans-ext`: Applicable if a program contains extended rules, that is, rules including cardinality and weight constraints. Try at least the *dynamic* transformation.
- `--sat-prepro`: Resolution-based preprocessing works best on tight programs with few cardinality and weight constraints. It should (almost) always be used if extended rules are transformed into normal ones (via `--trans-ext`).
- `--restarts`: Try aggressive restart policies, like *Luby-256* or the *nested policy*, or try disabling restarts whenever a problem is deemed to be unsatisfiable.
- `--save-progress`: Progress saving typically works nicely if the average back-jump length (or the #choices/#conflicts ratio) is high ($\geq$10). It usually performs best if combined with aggressive restarts.

[10] `--trans-ext=integrity`, if using `clasp` 1.3.3 or later versions.

4.3 Portfolio-based solving

A first step to overcome the sensitivity of modern ASP solvers to parameter settings and thus to regain a certain degree of declarativity in solving is to use a portfolio-based approach to ASP solving. The general idea is to identify a set of different solving approaches, either different systems, configurations, or both, and to harness existing machine learning techniques to build a classifier, mapping benchmark instances to the putatively best solver configurations. Such approaches have already shown their versatility in neighboring areas such as Constraint and SAT solving [36, 37].

This idea resulted in the portfolio-based ASP-solver `claspfolio` [29], winning the first place in the category "Single-System Teams" at the Second ASP competition [30].[11] Unlike other heterogeneous approaches using distinct systems in their portfolio, `claspfolio` takes advantage of `clasp`'s manifold search gearing options to identify a portfolio of different configurations. This has originally resulted in 30 different `clasp` configurations that were run on an extensive set of benchmarks. These results are then used to select a reduced portfolio of "interesting" settings by eliminating configurations whose exclusion does not significantly decrease the quality of the overall selection. This resulted in 12 configurations on which a support-vector machine is trained to predict the potentially fastest configuration for an arbitrary benchmark instance. To this end, we extract from each training instance 140 static and dynamic features by appeal to `claspre`. While the static features, like number of rule types or tightness, are obtained while processing the input benchmark, the dynamic ones are obtained through a limited run of `clasp`, observing features like number of learned nogoods or average length of back-jumps. Once the support-vector machines are established, a typical run of `claspfolio` starts by launching `claspre` for feature extraction upon which the support-vector machines select the most promising solver configuration to launch `clasp`. The option `--mode=su` also allows for a two-step classification by first predicting whether the instance is SAT, UNSAT, or unknown and then selecting the best configuration among specific SAT-, UNSAT-, and remaining portfolios, respectively.

The computational impact of this approach can be seen by contrasting the performance of `claspfolio` (0.8) with that of `clasp`'s default (1.3.4), its best but fixed configuration, a random selection among the portfolio, and the virtually best `clasp` version obtained my taking for each benchmark the minimum run-time among all solvers in the portfolio. Considering all systems on a set of 2771 benchmark instances restricted to 1200s, we observed an average run-time of 87.95s for `clasp`'s default configuration (and 127 timeouts); `clasp`'s best configuration took 70,42s (79 timeouts), while the virtually best solver among all 30 configurations spent 20,46s and that among the portfolio 24.61s on average (both trivially without timeouts).[12] While a random selection among the portfolio configurations run 97.89s on average (145 timeouts), the trained approach of `claspfolio` used 38.85s of which it spent 37.38s on solving only (with 22 timeouts). `claspfolio` has thus a clear edge over `clasp`'s best but rigid configuration.

[11] `claspfolio` was developed by Stefan Ziller.

[12] Learning is restricted to benchmarks solvable by at least one configuration.

All in all, `claspfolio` takes some burden of parameter tuning away from us and lets us concentrate more on problem posing. Nonetheless real applications still need manual interference. In this respect, `claspfolio` can be used as a first guide indicating which search parameters are most promising for attacking an application at hand.[13] The true research challenge however lies in getting a solid understanding in the link between problem features and search parameters.

5 Conclusion

ASP has come a long way. Having its roots in Nonmonotonic Reasoning [38], we can be proud of having taught Tweety how to fly. We have build impressive systems by drawing on solid formal foundations. And although there is still a long way to go to establish ASP among the standard technologies in Informatics, the future is bright and conceals many interesting research challenges.

Thank you Michael (and Vladimir) for putting us on the right track!

Acknowledgments. This work was supported by the German Science Foundation (DFG) under grants SCHA 550/8-1 and -2.

References

1. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine **26**(2) (2005) 62–72
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
3. Lifschitz, V., Razborov, A.: Why are there so many loop formulas? ACM Transactions on Computational Logic **7**(2) (2006) 261–268
4. Soininen, T., Niemelä, I.: Developing a declarative rule language for applications in product configuration. In Gupta, G., ed.: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99). Volume 1551 of Lecture Notes in Computer Science., Springer-Verlag (1999) 305–319
5. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-prolog decision support system for the space shuttle. In Ramakrishnan, I., ed.: Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01). Volume 1990 of Lecture Notes in Computer Science., Springer-Verlag (2001) 169–183
6. Boenn, G., Brain, M., de Vos, M., Fitch, J.: Automatic composition of melodic and harmonic music by answer set programming. [39] 160–174
7. Ishebabi, H., Mahr, P., Bobda, C., Gebser, M., Schaub, T.: Answer set vs integer linear programming for automatic synthesis of multiprocessor systems from real-time parallel programs. Journal of Reconfigurable Computing (2009).
8. Erdem, E., Türe, F.: Efficient haplotype inference with answer set programming. [40] 436–441
9. Gebser, M., Schaub, T., Thiele, S., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. Theory and Practice of Logic Programming **11**(2) (2011) 1–38

[13] For instance, `claspfolio --dir=<dir> --skip-solving --fstats` provides a ranking of the best `clasp` configuration on the benchmark instances in directory `<dir>`.

10. Grasso, G., Iiritano, S., Leone, N., Lio, V., Ricca, F., Scalise, F.: An ASP-based system for team-building in the Gioia-Tauro seaport. In Carro, M., Peña, R., eds.: Proceedings of the Twelfth International Symposium on Practical Aspects of Declarative Languages (PADL'10). Volume 5937 of Lecture Notes in Computer Science., Springer-Verlag (2010) 40–42
11. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic **7**(3) (2006) 499–562
13. Lin, F., Zhao, Y.: ASSAT: computing answer sets of a logic program by SAT solvers. Artificial Intelligence **157**(1-2) (2004) 115–137
14. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based answer sets solver enhanced to non-tight programs. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'04). Volume 2923 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2004) 346–350
15. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009)
16. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers (2004)
17. Mellarkod, V., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence **53**(1-4) (2008) 251–287
18. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. [41] 235–249
19. Drescher, C., Walsh, T.: A translational approach to constraint answer set solving. In: Theory and Practice of Logic Programming. Twenty-sixth International Conference on Logic Programming (ICLP'10) Special Issue. Volume 10(4-6)., Cambridge University Press (2010) 465–480
20. Cliffe, O., de Vos, M., Brain, M., Padget, J.: ASPVIZ: Declarative visualisation and animation using answer set programming. [39] 724–728
21. de Vos, M., Schaub, T., eds.: Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07). Number CSBU-2007-05 in Department of Computer Science, University of Bath, Technical Report Series (2007) ISSN 1740-9497.
22. de Vos, M., Schaub, T., eds.: Proceedings of the Second Workshop on Software Engineering for Answer Set Programming (SEA'09). Department of Computer Science, University of Bath, Technical Report Series (2009)
23. Brain, M., de Vos, M.: Debugging logic programs under the answer set semantics. In de Vos, M., Provetti, A., eds.: Proceedings of the Third International Workshop on Answer Set Programming (ASP'05). Volume 142., CEUR Workshop Proceedings (CEUR-WS.org) (2005) 141–152
24. Pontelli, E., Son, T.: Justifications for logic programs under answer set semantics. In Etalle, S., Truszczyński, M., eds.: Proceedings of the Twenty-second International Conference on Logic Programming (ICLP'06). Volume 4079 of Lecture Notes in Computer Science., Springer-Verlag (2006)
25. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In Baral, C., Brewka, G., Schlipf, J., eds.: Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07). Volume 4483 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2007) 31–43
26. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. [40] 448–453

27. Faber, W., Leone, N., Mateis, C., , Pfeifer, G.: Using database optimization techniques for nonmonotonic reasoning. In: Proceedings of the Seventh International Workshop on Deductive Databases and Logic Programming (DDLP'99). (1999) 135–139
28. Ullman, J.: Principles of Database and Knowledge-Base Systems. Computer Science Press (1988)
29. `http://potassco.sourceforge.net/`
30. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In Erdem, E., Lin, F., Schaub, T., eds.: Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LP-NMR'09). Volume 5753 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2009) 637–654
31. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General game playing: Game description language specification. Technical Report LG-2006-01, Stanford University (March 2008)
32. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning **36**(4) (2006) 345–377
33. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: On the implementation of weight constraint rules in conflict-driven ASP solvers. [41] 250–264
34. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In Bacchus, F., Walsh, T., eds.: Proceedings of the Eigth International Conference on Theory and Applications of Satisfiability Testing (SAT'05). Volume 3569 of Lecture Notes in Computer Science., Springer-Verlag (2005) 61–75
35. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In Marques-Silva, J., Sakallah, K., eds.: Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07). Volume 4501 of Lecture Notes in Computer Science., Springer-Verlag (2007) 294–299
36. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In Bridge, D., Brown, K., O'Sullivan, B., Sorensen, H., eds.: Proceedings of the Nineteenth Irish Conference on Artificial Intelligence and Cognitive Science (AICS'08). (2008)
37. Xu, L., Hutter, F., Hoos, H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. Journal of Artificial Intelligence Research **32** (2008) 565–606
38. Ginsberg, M., ed.: Readings in Nonmonotonic Reasoning. Morgan Kaufman, Los Altos (1987)
39. Garcia de la Banda, M., Pontelli, E., eds.: Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08). Volume 5366 of Lecture Notes in Computer Science., Springer-Verlag (2008)
40. Fox, D., Gomes, C., eds.: Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08). AAAI Press (2008)
41. Hill, P., Warren, D., eds.: Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09). Volume 5649 of Lecture Notes in Computer Science., Springer-Verlag (2009)