Influence of ASP Language Constructs on the Performance of State-of-the-art Solvers*

Richard Taupe[1,2] and Erich Teppan[1]

[1] Alpen-Adria-Universität Klagenfurt,
Universitätsstr. 65-67, 9020 Klagenfurt, Austria
erich.teppan@aau.at
[2] Siemens AG Österreich, Corporate Technology,
Siemensstr. 90, 1210 Vienna, Austria
richard.taupe@siemens.com

Abstract. Answer Set Programming (ASP) under the stable model semantics has evolved to an extremely powerful approach to represent and solve even industrial-sized combinatorial problems in real-life application domains. ASP supports various language constructs which can be used to express the same realities in syntactically different, but semantically equivalent ways. However, these equivalent programs may not perform equally well. This is because performance depends on the underlying solver implementations that may treat different language constructs differently. As performance is very important for the successful application of ASP in real-life domains, knowledge about the mutual interchangeability and performance of ASP language constructs is crucial for knowledge engineers. In this article, we present an investigation on how the usage of different language constructs affects the performance of state-of-the-art solvers and grounders on benchmark problems from the ASP competitions 2013 and 2014.

Keywords: Answer Set Programming · Logic Progamming · Stable Models · ASP Competition · Performance Tuning · Benchmarking

1 Introduction

In recent years, Answer Set Programming (ASP) under the stable model semantics [19] has evolved to an extremely powerful approach to represent and solve even industrial-sized combinatorial problems in real-life application domains [4, 16, 18]. There are mainly two reasons for these significant advancements: First, the performance of state-of-the-art ASP systems has tremendously improved, e.g. by the incorporation of conflict-driven search [17], portfolio solving [14], lazy grounding [21,28], or the inclusion of powerful heuristics like berkmin or vsids [23]. Second, the language itself has been extended by expressive

* This article summarizes the key findings of Richard Taupe's master's thesis [27]. Erich Teppan provided the idea for the topic and supervised the thesis. Both authors contributed equally to this paper.

constructs like disjunction, choice rules, aggregates, weak constraints or optimization statements [3, 13, 20, 22, 26].

As a consequence, various language features allow to express the same concept in different ways. This leads to problem encodings that are syntactically different, but semantically identical. Syntactical differences also affect solving performance. This depends on implementation details of the solvers, which may treat different language constructs differently.

As performance issues play a big role for the successful application of ASP in real-life domains, two questions are of special importance for knowledge engineers:

- Which language constructs are mutually interchangeable without changing program size or readability[3] significantly?
- What is the effect of the different language constructs on time and memory requirements of both grounding and solving?

In this article, we present an experimental evaluation which gives some answers to these questions. The experiments were conducted using the benchmark problems from the ASP competitions 2013 [1][4] and 2014 [7][5]. For each of these problems, several encodings were produced. It was ensured that the different encodings for a particular problem were equivalent, i.e. they produced the same solutions. In order to be able to attribute changes in performance to a certain language construct, the syntactical differences were kept small and modular. All encodings have been tested on a number of test instances randomly drawn from the ASP competitions. They were solved by numerous combinations of state-of-the-art grounders and solvers. Statistical analysis of the experimental data revealed interesting dependencies between the usage of certain language constructs, problems, grounders, and solvers in terms of memory and time consumption.

The remainder of this article is structured as follows. Section 2 refreshes some concepts of ASP that are important to understand this article. Introductions to ASP syntax and semantics can be found elsewhere, e.g. in [3, 5, 9, 13]. Section 3 describes the experimental setup. Section 4 discusses the most interesting results of our evaluations. Finally, Section 5 briefly concludes the article.

[3] For example, choice rules can be converted into a number of disjunctive rules linear in the number of choice elements [3], and for counting aggregates there exists a translation which is quadratic in space [13]. Both methods are not studied in this article since they decrease readability in the general case.

[4] `https://www.mat.unical.it/aspcomp2013/`

[5] `https://www.mat.unical.it/aspcomp2014/`

2 Background

We deal with answer-set programs that are sets of rules. There are several types of rules: we consider normal rules, facts, constraints, disjunctive rules, and choice rules. Rule bodies may contain aggregates. Typical aggregate functions are *min*, *max*, *count* and *sum*. For an introductory textbook, see [13]. Shorter introductions to ASP are provided by [5, 9]. The standardized input language of ASP systems is defined in [3].

There are several ways to define the semantics of an answer-set program, i.e. to define the set of answer sets $AS(\Pi)$ of an answer-set program Π. An overview of the different semantics is provided by [24]. Probably the most popular semantics is based on the *Gelfond-Lifschitz reduct* [19]. A semantics that covers aggregates also is the so-called *FLP semantics*, named after its devisors Faber, Leone, and Pfeifer [12]. A variant that applies to choice rules also is presented in [7].

2.1 Grounding and Solving

Most ASP systems split the solving process into grounding and solving. The former part produces the grounding of a program, i.e. its variable-free equivalent. Thereby, the variables in each rule of the program are substituted by constants. The latter part then solves this propositional encoding. While our analysis includes a wide range of systems, we will highlight some results for the grounder *gringo* [15] and the solver *clasp* [17].

2.2 ASP Coding Practices

It is common practice in ASP to encode the generic problem specification and the instance data as two separate programs. We call them *encoding* and *instance*, respectively. While the encoding always stays the same, it can be run together with different instance programs to solve different problem instances [6, 13].

An instance typically contains only facts involving certain predicates. We call this set of predicates the program's *input signature*. On the other hand, solutions to the problem are also encoded by a specific set of predicates, which we call the program's *output signature*. Given an answer-set program Π, we denote by $in(\Pi)$ its input signature and by $out(\Pi)$ its output signature [6].

2.3 Equivalence

Various notions of program equivalence have been proposed for ASP. Two answer-set programs Π and Π' are *(ordinarily) equivalent* (written as $\Pi \equiv \Pi'$), if they have the same answer sets, i.e. $AS(\Pi) = AS(\Pi')$. *Uniform equivalence* is fulfilled w.r.t. Π and Π' if they are ordinarily equivalent when combined with any set of *facts*, i.e. for any set of facts F it holds that $\Pi \cup F \equiv \Pi' \cup F$ [8]. The stronger notion of *strong equivalence* holds if the two programs are ordinarily

equivalent when combined with any program Q, i.e. for all programs Q it holds that $\Pi \cup Q \equiv \Pi' \cup Q$ [25].

As the rule transformations discussed in this paper often change the program's predicates, the definition of equivalence used herein is based on the concept of *output-equivalence*[6] [6]:

For a set X of atoms and a set P of predicate symbols[7], let $X_{|P}$ be the subset of X that only contains the atoms whose predicate symbol is in P. Two *encodings* Π and Π' are output-equivalent, if and only if

1. their input and output signatures coincide, i.e. $in(\Pi) = in(\Pi')$ and $out(\Pi) = out(\Pi')$, and
2. for each *instance* I, i.e. a set of facts of predicates in $in(\Pi)$, it holds that:
   - For each answer set $X \in AS(\Pi \cup I)$ there exists an answer set $X' \in AS(\Pi' \cup I)$ such that $X_{|out(\Pi)} = X'_{|out(\Pi')}$ and vice versa.

In other words, two programs are output-equivalent if and only if for any input instance their answer sets, projected onto the programs' output predicates, are the same [6].

3 Experimental Setup

In order to test whether and how the inclusion or exclusion of certain language constructs influences the solving performance, we tested variations of problem encodings on a set of state-of-the-art grounders and solvers.

An overview on the experimental setup is given in Figure 1. The ASP competitions 2013 [1] and 2014 [7] served as source for problems, problem encodings and problem instances. Basic problem encodings obtained from the competitions' websites were supplemented by handmade variations to use different language constructs. Four instances also provided by the competitions were randomly chosen for each problem. The problems were then solved by a set of ASP systems. This process was controlled by a benchmarking tool, limiting and measuring the systems' resource consumption. The results produced were validated by checker programs. The data collected during this process comprises information on a) time and space used by grounders and solvers; b) problem, encoding, and language constructs involved; c) correctness of the results computed by the solvers; d) and the instance solved.

Let us emphasize that we are primarily interested in the impact of coding style rather than in the performance of the solvers themselves. Thus, solvers were treated as black boxes in our experiments.

[6] Output-equivalence is similar to the notion of uniform equivalence under *projected answer sets* [10], where projection to an arbitrary set of atoms is allowed.

[7] To distinguish predicates with the same symbol but different arities, identifiers like p/n can be used, where p is the predicate's name and $n \in \mathbb{N}_0$ its arity.

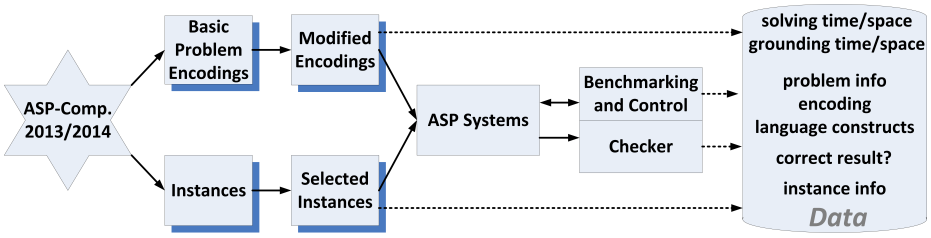

Fig. 1. Experimental setup

3.1 Problems and Encodings

All problems that appeared in the ASP competitions 2013 and 2014 were used for our experiments, except for *Chemical Classification*, *Reachability* and *Strategic Companies*. For those three, no alternative encodings could be produced[8].

At least one of the official encodings provided for each problem was used as a basis. From this basis, variations were produced by making small changes where special language constructs were used. For the 24 problems that were used in our experiments, 223 different problem encodings were formulated, i.e. 9.3 encodings per problem on average (not every language construct was applicable to every problem). Many of these 223 encodings had to be additionally adapted for grounders accepting different dialects of ASP[9].

In the following paragraphs, we present examples for the exchange of certain language constructs by others. When doing so, attention has to be paid to preserve output equivalence as described in Section 2.3.

Replacing classical negation. In many cases, classical negation can be replaced by introducing a new predicate. For example, rule (1) from an encoding of the *Bottle Filling* problem contains classical negation.

$$\mathit{filled}(X,Y); \neg\mathit{filled}(X,Y) \leftarrow \mathit{bottle}(B,X,Y). \tag{1}$$

If the negated version of the *filled* predicate is not used anywhere else in the program, the disjunction is only used to guess whether $\mathit{filled}(X,Y)$ is present or not. In this case, rule (2) without classical negation is equivalent to (1).

$$\mathit{filled}(X,Y); \mathit{unfilled}(X,Y) \leftarrow \mathit{bottle}(B,X,Y). \tag{2}$$

Replacing guessing constructs. By *guessing constructs*, we denote the ASP language features that are used to encode the non-deterministic guessing part of

[8] Official encodings for *Chemical Classification* and *Reachability* do not use any special language constructs that could be replaced. The one for *Strategic Companies* uses disjunction, which however cannot be converted using any of the techniques presented in this paper because it is not head-cycle free [22] in this case.

[9] Encodings can be downloaded at `http://isbi.aau.at/hint/misc`.

a program. These features are disjunction, shifted disjunction, and (bounded or unbounded) choice rules.

If the whole program is head-cycle free[10] w.r.t. a disjunctive rule, the disjunction can be *shifted* into the body [8] or replaced by a choice rule. For (2), shifting results in (3).

$$\begin{aligned} \mathit{filled}(X,Y) &\leftarrow \mathit{bottle}(B,X,Y), \mathit{not}\ \mathit{unfilled}(X,Y). \\ \mathit{unfilled}(X,Y) &\leftarrow \mathit{bottle}(B,X,Y), \mathit{not}\ \mathit{filled}(X,Y). \end{aligned} \tag{3}$$

The alternative choice rule is shown in (4).

$$1\{\mathit{filled}(X,Y); \mathit{unfilled}(X,Y)\}1 \leftarrow \mathit{bottle}(B,X,Y). \tag{4}$$

Furthermore, a choice rule with explicit bounds can often be expressed as an unbounded choice rule, which may require the addition of a small number of constraints. For example, the bounded choice rule (4) can as well be represented by the set of rules in (5).

$$\begin{aligned} \{\mathit{filled}(X,Y)\} &\leftarrow \mathit{bottle}(B,X,Y). \\ \mathit{unfilled}(X,Y) &\leftarrow \mathit{bottle}(B,X,Y), \mathit{not}\ \mathit{filled}(X,Y). \end{aligned} \tag{5}$$

If *unfilled* is not used anywhere else, the second rule in (5) can even be omitted.

Replacing aggregates. Rules with counting aggregates can sometimes be replaced by a small set of normal rules without aggregates. For example, (6) is a (simplified) integrity constraint from an encoding of *Knight Tour with Holes*.

$$\leftarrow \mathit{cell}(X,Y), \mathit{not}\ \#\mathit{count}\{X1,Y1 : \mathit{move}(X1,Y1,X,Y)\} = 1. \tag{6}$$

This constraint enforces that for each $\mathit{cell}(X,Y)$ there must be exactly one other cell from which it is visited. This can also be expressed by the introduction of two new predicates *e1* and *e2* reflecting whether there is exactly one or at least two moves entering from different cells and a constraint stating that the first one must hold, as shown in (7).

$$\begin{aligned} &\leftarrow \mathit{cell}(X,Y), \mathit{not}\ \mathit{e1}(X,Y). \\ \mathit{e1}(X,Y) &\leftarrow \mathit{move}(X1,Y1,X,Y),\ \mathit{not}\ \mathit{e2}(X,Y). \\ \mathit{e2}(X,Y) &\leftarrow \mathit{move}(X1,Y1,X,Y),\ \mathit{move}(X2,Y2,X,Y), X1 <> X2. \\ \mathit{e2}(X,Y) &\leftarrow \mathit{move}(X1,Y1,X,Y),\ \mathit{move}(X2,Y2,X,Y), Y1 <> Y2. \end{aligned} \tag{7}$$

Min aggregates can be replaced by rules which check whether there is a smaller element.

$$\mathit{next}(X,Y) \leftarrow \mathit{pair}(X,Y), Y = \#\mathit{min}\{Z : \mathit{pair}(X,Z)\}. \tag{8}$$

[10] Please find more information about head-cycles in [22].

Thereby, rule (8) from an encoding of *Graceful Graphs* can also be expressed by (9).

$$\begin{aligned} existsSmaller(X,Y) &\leftarrow pair(X,Y), pair(X,S), S < Y. \\ next(X,Y) &\leftarrow pair(X,Y), not\ existsSmaller(X,Y). \end{aligned} \tag{9}$$

Similar techniques can be used to eliminate *max* aggregates.

3.2 Benchmarking

We tested our encodings on different grounders and solvers in various configurations which already proved a certain stability in the ASP competitions or in our own evaulations. Tables 1 and 2 give an overview on the used grounders and solvers, which were used in all combinations (henceforth called *systems*) where the solver was compatible to the grounder's output format[11].

Table 1. Used grounders

Grounder	**Version**
gringo	3.0.5
gringo	4.4.0
dlvg[12]	BEN/Sep 29 2014

Table 2. Used solvers

Solver	**Version**	**Configurations**
clasp	3.1.0	auto, frumpy, jumpy, tweety, handy, crafty, trendy
claspfolio	2.0.0	*ASP Competition 2013 Package*
cmodels	3.85	minisat1, minisat2, simo, zchaff
dlv[13]	BEN/Dec 17 2012	
GnT	2.1	
lp2bv[14]		LP2BV-1, LP2BV-2, LP2BV2+BOOLECTOR
lp2sat[14]		lp2graph, lp2sat, lp2sat3+glucose, lp2sat3+lingeling, lp2maxsat+clasp
lp2normal2	1.7	lp2normal2+clasp
MinisatID	3.9.3	
smodels	2.34	
wasp	1.0	berkmin, berkmintwo, berkmintwolight, firstundefined, mixed, mixedprelookahead, berkminlimitedactivity
wasp	2.0	

[11] The output format of both *gringo* versions is understood by every solver except for *dlv*, and the output format of *dlvg* is compatible only to *wasp 1.0* [2].

[12] The authors would like to thank Francesco Calimeri for providing this unreleased version of the grounder to us.

Some of the mentioned solvers were additionally used in several configurations, e.g. *clasp* was used with varying heuristics. Details are given in the third column of Table 2.

For each problem, we randomly selected four test instances, a number which is a compromise between good sample size and total computation time constraints. As there were 223 encodings, 73 systems in different configurations and 4 test instances for each problem, there were 65,116 different test cases. As it was known beforehand that the monolithic system *dlv* was not able to digest the encodings containing choice rules [11], the actual number of test cases was 63,884. Experiments were run on four virtual machines running Ubuntu 14.04.1 LTS trusty. Each machine had exclusive access to one processor with 2.53 GHz and 7.8 GB of RAM to preclude any side-effects between experiments running in parallel. The RAM available to ASP grounders and solvers was limited to 6 GB. Furthermore, there were time limits of 60 minutes for grounding and of 10 minutes for solving. When the timeout was reached, running systems were carefully terminated to start the next instance in a clean environment.

RunLim[15] was used to measure use of time and space during the execution of grounders and solvers. The correctness of the produced answer sets was validated by checkers obtained from the ASP competition 2014. For each test case, we recorded information on the systems involved, grounding and solving time and space, information about the problem and its encoding (including which language constructs were present), the problem instance, and the validation results of the checker.

Of the 63,884 test cases, as many as 25,913 produced a valid solution or proved to be unsatisfiable. In 16,774 cases the solver ran out of time and in 4695 cases it ran out of memory. In 1449 cases the grounder ran out of memory and in 552 cases it ran out of time. The remaining 14,501 cases had to be excluded from analysis, because they comprise the following cases: a) the solver had problems with some language constructs which could be identified later on, b) the solver produced an incorrect answer, or c) it aborted out of unknown reasons.

4 Results and Discussion

In this paper, only key aspects of the data collected in our benchmarking process can be presented. The full dataset is available on our website[16].

For reasons of readability we use short names in the tables within this section in order to refer to the different types of language constructs. The meaning should be clear from the context and the textual explanations.

Tables 3 and 4 list the median values for grounding and solving time and space for all systems and *gringo4/clasp* respectively w.r.t. classical negation as

[13] DLV is a monolithic system, comprising both grounder and solver.
[14] Packages that participated in the ASP Competitions 2013 and 2014 and incorporated lp2bv or lp2sat have been included in our analysis.
[15] `http://fmv.jku.at/runlim/`
[16] `http://isbi.aau.at/hint/misc`

well as *count*, *min* and *max* aggregates. Please note that the median of the total time (i.e. grounding + solving) is not the same as the median of grounding + the median of solving. For all constructs, only those problems were included in the analysis where the respective language construct could be varied, i.e. where there were encodings with and without the language construct. The number of these problems is given as *#probs*. The given number of *cases* indicates how often an encoding with (*yes*) or without (*no*) the respective language construct was used by a system.

Table 3. Medians for non-guessing constructs: space in MB, time in secs

all systems		**cases**	**grd-space**	**slv-space**	**grd-time**	**slv-time**	**total**
negation	no	29,513	8.00	150.80	0.29	87.33	182.13
(#probs=22)	yes	17,856	8.10	161.25	0.39	72.83	145.21
aggrcount	no	12,914	7.20	160.80	0.86	99.39	241.26
(#probs=9)	yes	12,358	6.70	103.85	0.18	128.32	285.22
aggrmin	no	1605	8.80	144.60	0.09	timeout	timeout
(#probs=1)	yes	1603	8.80	148.80	0.09	timeout	timeout
aggrmax	no	3796	0.00	25.35	0.00	16.63	16.83
(#probs=2)	yes	3796	0.00	25.50	0.00	16.56	16.95

Table 4. Medians for non-guessing constructs: space in MB, time in secs

gringo4/clasp		**cases**	**grd-space**	**slv-space**	**grd-time**	**slv-time**	**total**
negation	no	3629	7.20	45.30	0.09	43.03	49.78
(#probs=22)	yes	2306	7.20	48.90	0.09	25.42	27.51
aggrcount	no	1645	6.50	43.90	0.09	92.07	92.15
(#probs=9)	yes	1686	6.50	31.30	0.08	125.11	155.31
aggrmin	no	168	1.30	55.45	0.04	527.01	527.01
(#probs=1)	yes	168	0.00	54.05	0.00	339.56	339.56
aggrmax	no	476	0.00	5.70	0.00	1.46	1.46
(#probs=2)	yes	476	0.00	5.70	0.00	1.46	1.46

It appears that the inclusion of classical negation effected a slight increase in required space and grounding time, but also a decrease in solving time. This resulted in a lower total time. The differing case numbers are due to the fact that, in contrary to other guessing constructs, for unbounded choice rules no encodings including classical negation could be produced (in a natural and intuitive way). Taking into account that the slight positive influence of unbounded negation discovered by our experiments could not contribute to the positive influence of classical negation (since the two could not be used together), the latter can be seen as even stronger.

Apparently, aggregates with *count* function triggered a significant decrease in grounding size and time. However, solving was much harder compared to the

cases where this construct was not present. As the effort for grounding can be neglected in these cases, aggregates with *count* function had a negative overall effect. Since there were only very few problems which allowed encodings with and without *min* and *max* aggregates, the results are only representative for the problems where these constructs were varied. These were *Labyrinth* and *Weighted Sequence* for *max* aggregates and *Graceful Graphs* for *min* aggregates. Instances of *Graceful Graphs* were not hard to ground but very hard to solve such that the all-systems median of the solving time (and consequently of the total time) was a timeout. The combination of *gringo 4* and *clasp* performed considerably above average. *Clasp* showed significantly lower solving times in cases with *min* aggregates. The presence/absence of *max* aggregates effected only negligible differences over all systems and no differences at all for the combination of *gringo 4* and *clasp*.

Table 5. Medians for guessing constructs (#probs=19): space in MB, time in secs

all systems	**cases**	**grd-space**	**slv-space**	**grd-time**	**slv-time**	**total**
unbounded choice	6198	7.40	135.00	0.27	84.65	178.40
bounded choice	7589	7.20	98.90	0.09	147.72	260.95
disjunction	13,735	8.90	166.90	0.57	77.51	171.90
shifted disjunction	12,503	7.80	168.50	0.48	95.07	186.96

Table 6. Medians for guessing constructs (#probs=19): space in MB, time in secs

gringo4/clasp	**cases**	**grd-space**	**slv-space**	**grd-time**	**slv-time**	**total**
unbounded choice	861	7.00	35.30	0.09	76.70	91.45
bounded choice	1134	4.00	38.55	0.06	76.98	77.07
disjunction	1918	7.60	48.00	0.09	40.54	42.67
shifted disjunction	1742	7.00	40.10	0.09	58.63	73.11

Tables 5 and 6 show the measured median values w.r.t. the different guessing constructs for all systems and *gringo4/clasp*, respectively. Here, only those 19 problems were included for which there were encodings for all guessing constructs and each encoding included exactly one of them. Furthermore, only those systems were taken into account which could deal with all four guessing constructs. It can be seen that over all systems, bounded choice rules led to smaller space usage. The combination of *gringo 4* and *clasp* showed a similarly efficient space usage for all four constructs. Disjunction showed best solving and total times for all systems as well as for *gringo 4* and *clasp*.

Generally speaking, some tendencies can be witnessed throughout the set of experiments. However, it is also evident from the results that problem-level effects may contradict those tendencies. Table 7 shows some examples for that. Although disjunction showed the best performance over all problems, for the

Bottle Filling problem, unbounded choice rules (encoding 2) outperformed disjunction. On the other hand, for *Labyrinth* the median values for encodings without disjunction over all systems were timeouts. For *Labyrinth*, also *gringo*4 + *clasp* performed best on encodings with disjunction. For the *Partner Units* problem, *gringo*4 + *clasp* performed above average with encoding 6 that includes bounded choice rules, but below average with all other encodings.

Table 7. Problem-dependent medians: total time in secs

problem	enc.	constructs	all systems	gringo4/clasp
Bottle Filling	1	shifted disjunction	66.44	0.83
	2	unbounded choice	17.57	0.47
	3	disjunction	26.08	0.57
	4	disjunction, negation	26.83	1.06
	5	shifted disjunction, negation	54.70	0.89
	6	bounded choice	45.25	1.19
	7	bounded choice, neg.	43.14	1.26
Labyrinth	1	shifted disjunction	timeout	85.13
	2	bounded choice	timeout	98.47
	3	disjunction	261.68	42.43
	4	shifted disjunction, aggrmax	timeout	85.51
	5	bounded choice, aggrmax	timeout	99.48
	6	disjunction, aggrmax	261.61	42.46
Partner Units	1	disjunction, negation	149.09	580.67
	2	disjunction	148.08	540.89
	3	shifted disjunction, negation	156.76	572.88
	4	shifted disjunction	155.97	561.26
	5	unbounded choice	154.51	575.20
	6	bounded choice	595.23	405.73

5 Conclusions

Answer Set Programming (ASP) under the stable model semantics constitutes an extremely powerful approach to solve hard combinatorial problems. One reason for the success of ASP is the high performance of state-of-the-art solvers harnessing sophisticated conflict-driven search methods. Also, ASP provides superior problem encoding capabilities as it is declarative in nature and even provides language features beyond first order.

As a consequence of the broad problem representation capabilities, there are many elegant ways to express the same issue differently. In particular, for most problems various encodings of similar readability including different language constructs can be created quite naturally. However, even if logically equivalent, the performance of different encodings may vary significantly, depending on the language constructs involved. The reason for that is clearly that different constructs are processed differently by the solver implementations.

The main goal of this work is to answer the question whether there is a relevant non-negligible impact of the used language constructs on runtime and space consumption and, if so, whether general tendencies can be identified. Our results suggest that the runtime and space consumption can depend heavily on the used constructs and is never to be neglected. Furthermore, some general tendencies were identified in our experiments. For example, normal disjunction had a positive overall effect on the solving speed compared to the other guessing constructs. A more fine-grained analysis revealed that, although there are general tendencies, the presence of positive or negative effects is highly dependent on the problem at hand and the ASP system used.

An important conclusion with respect to the implementation of an ASP solution for a real-life problem is that a small investment in producing various slightly differing problem encodings may pay off with remarkable performance gains.

Some minor issues are left for future work: Due to the problem landscape in the ASP competitions, *min* and *max* aggregates were used with very small sample sizes in our study, and *sum* aggregates were not studied at all. It would be interesting to compare those language constructs to other representations on a larger sample. Also, new solvers have emerged during and after our experiments, which could be included in future reiterations.

An obvious and important direction for future work is automatic code rewriting, similarly to query optimization for relational databases. Especially the usage within a portfolio solver seems to be promising in this context. Language construct replacements could be represented as additional parameters to be learned within the portfolio solver. However, for any application scenario a much more in-depth analysis of when certain rewritings are sound is needed.

6 Acknowledgements

The research for this paper was conducted in the scope of the project *Heuristic Intelligence (HINT)* funded by the Austrian research fund FFG under grant 840242.

References

1. Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L.K., Wallner, J.P., Xiao, G.: The Fourth Answer Set Programming Competition: Preliminary Report. In: Cabalar, P., Son, T.C. (eds.) Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Computer Science, vol. 8148, pp. 42–53. Springer, Berlin, Heidelberg (2013)
2. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A Native ASP Solver Based on Constraint Learning. In: Cabalar, P., Son, T.C. (eds.) Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Computer Science, vol. 8148, pp. 54–66. Springer, Berlin, Heidelberg (2013)

3. ASP Standardization Working Group: ASP-Core-2 Input Language Format (2012-12-13), https://www.mat.unical.it/aspcomp2013/files/ASP-CORE-2.03b.pdf
4. Balduccini, M.: Industrial-Size Scheduling with ASP+CP. In: Delgrande, J.P., Faber, W. (eds.) Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Computer Science, vol. 6645, pp. 284–296. Springer, Berlin, Heidelberg (2011)
5. Brewka, G., Eiter, T., Truszczyński, M.: Answer set programming at a glance. Communications of the ACM 54(12), 92–103 (2011)
6. Buddenhagen, M., Lierler, Y.: Performance Tuning in Answer Set Programming. In: Calimeri, F., Ianni, G., Truszczyński, M. (eds.) Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Artificial Intelligence, vol. 9345, pp. 186–198. Springer International Publishing, Cham (2015)
7. Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: Design and results of the Fifth Answer Set Programming Competition. Artificial Intelligence 231, 151–181 (2016)
8. Eiter, T., Fink, M., Pührer, J., Tompits, H., Woltran, S.: Model-based recasting in answer-set programming. Journal of Applied Non-Classical Logics 23(1-2), 75–104 (2013)
9. Eiter, T., Ianni, G., Krennwallner, T.: Answer Set Programming: A Primer. In: Tessaris, S., Franconi, E., Eiter, T., Gutierrez, C., Handschuh, S., Rousset, M.C., Schmidt, R.A. (eds.) Reasoning Web. Semantic Technologies for Information Systems, Lecture Notes in Computer Science, vol. 5689, pp. 40–110. Springer (2009)
10. Eiter, T., Tompits, H., Woltran, S.: On Solution Correspondences in Answer-Set Programming. In: Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence. pp. 97–102 (2005)
11. Faber, W., Leone, N., Perri, S.: The Intelligent Grounder of DLV. In: Erdem, E., Lee, J., Lierler, Y., Pearce, D. (eds.) Correct Reasoning, Lecture Notes in Computer Science, vol. 7265, pp. 247–264. Springer, Berlin, Heidelberg (2012)
12. Faber, W., Pfeifer, G., Leone, N.: Semantics and complexity of recursive aggregates in answer set programming. Artificial Intelligence 175(1), 278–298 (2011)
13. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers (2012)
14. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M.T., Ziller, S.: A Portfolio Solver for Answer Set Programming: Preliminary Report. In: Delgrande, J.P., Faber, W. (eds.) Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Computer Science, vol. 6645, pp. 352–357. Springer, Berlin, Heidelberg (2011)
15. Gebser, M., Kaminski, R., König, A., Schaub, T.: Advances in gringo Series 3. In: Delgrande, J.P., Faber, W. (eds.) Logic Programming and Nonmonotonic Reasoning. Lecture Notes in Computer Science, vol. 6645, pp. 345–351. Springer, Berlin, Heidelberg (2011)
16. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in answer set programming. Theory and Practice of Logic Programming 11(Special Issue 4-5), 821–839 (2011)
17. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven Answer Set Solving: From Theory to Practice. Artificial Intelligence 187-188, 52–89 (2012)
18. Gebser, M., Schaub, T., Thiele, S., Usadel, B., Veber, P.: Detecting Inconsistencies in Large Biological Networks with Answer Set Programming. In: Garcia de la Banda, Maria, Pontelli, E. (eds.) Logic Programming, Lecture Notes in Computer Science, vol. 5366, pp. 130–144. Springer (2008)

19. Gelfond, M., Lifschitz, V.: The Stable Model Semantics For Logic Programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of the Fifth International Conference and Symposium of Logic Programming. pp. 1070–1080. MIT Press (1988)
20. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing 9(3-4), 365–385 (1991)
21. Lefèvre, C., Béatrix, C., Stéphan, I., Garcia, L.: ASPeRiX, a First Order Forward Chaining Approach for Answer Set Computing. CoRR abs/1503.07717 (2015)
22. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transaction on Computational Logic 7(3), 499–562 (2006)
23. Lewis, M.D.T., Schubert, T., Becker, B.W.: Speedup Techniques Utilized in Modern SAT Solvers. In: Bacchus, F., Walsh, T. (eds.) Theory and Applications of Satisfiability Testing. Lecture Notes in Computer Science, vol. 3569, pp. 437–443. Springer, Berlin, Heidelberg (2005)
24. Lifschitz, V.: Thirteen Definitions of a Stable Model. In: Blass, A., Dershowitz, N., Reisig, W. (eds.) Fields of Logic and Computation, Lecture Notes in Computer Science, vol. 6300, pp. 488–503. Springer, Berlin, Heidelberg (2010)
25. Lifschitz, V., Pearce, D., Valverde, A.: Strongly Equivalent Logic Programs. ACM Transaction on Computational Logic 2(4), 526–541 (2001)
26. Syrjänen, T.: Logic programming and cardinality constraints: theory and practice. Ph.D. thesis, Helsinki University of Technology (2009), `https://aaltodoc.aalto.fi/handle/123456789/4595`
27. Taupe, R.: Einfluss von Sprachkonstrukten auf die Lösbarkeit von Answer-Set-Programmen: Eine empirische Untersuchung aktueller ASP-Systeme. Masterarbeit, Alpen-Adria-Universität, Klagenfurt (2015), `http://ubdocs.aau.at/open/hssvoll/AC12297983.pdf`
28. Weinzierl, A.: Learning Non-Ground Rules for Answer-Set Solving. In: Pearce, D., Tasharrofi, S., Ternovska, E., Vidal, C. (eds.) 2nd Workshop on Grounding and Transformations for Theories With Variables (2013)