Journal of Artificial Intelligence Research 1 (2011) 1-15 Submitted 3/11; published x/11

Weight and Cardinality Constraints in ASP

Stefania Costantini STEFANIA.COSTANTINI@UNIVAQ.IT
Dip. di Informatica, Università di L'Aquila
Via Vetoio Loc. Coppito, I-67010 L'Aquila, Italy

Andrea Formisano FORMIS@DMI.UNIPG.IT
Dip. di di Matematica e Informatica, Università di Perugia
via Vanvitelli, 1, I-06123 Perugia, Italy

1. ASP

In this section, we briefly recall the basics about Answer Set Programming (Baral, 2003; Lifschitz, 1999; Marek & Truszczyński, 1999). In this logical framework, a problem can be encoded —by using a function-free logic language— as a set of properties and constraints which describe the (candidate) solutions. More specifically, an *ASP-program* is a collection of *rules* of the form

$$H \leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_{m+n}.$$

where H is an atom $m \geqslant 0$, $n \geqslant 0$ and each L_i is an atom. The symbol *not* stands for what is called "negation-as-failure" or "default negation". Various extensions to the basic paradigm exist, that we do not consider here all of them as they are not essential in the present context. The left-hand side and the right-hand side of the clause are called *head* and *body*, respectively. A rule with empty head is a *constraint*. A rule with empty body is a *fact*. A constraint $\leftarrow L_1, \ldots, L_m, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_{m+n}$ can be seen as a shorthand for a rule of the form $p \leftarrow L_1, \ldots, L_m, \mathit{not}\ p, \mathit{not}\ L_{m+1}, \ldots, \mathit{not}\ L_{m+n}$ (where p is a fresh atom).

The semantics of ASP is expressed in terms of *answer sets* (or equivalently *stable models*, (Gelfond & Lifschitz, 1988)). Consider first the case of a ground[1] ASP-program P which does not involve negation-as-failure (i.e., $n = 0$). In this case, a set of atoms X is said to be an answer set for P if it is the (unique) least model of P. Such a definition is extended to any ground program P containing negation-as-failure by considering the Gelfond-Lifschitz *reduct* (GL-reduct) P^X (of P) w.r.t. a set of atoms X. P^X is defined as the set of rules of the form $H \leftarrow L_1, \ldots, L_m$ for all rules of P such that X does not contain any of the literals $L_{m+1}, \ldots, L_{m+n}$. Clearly, P^X does not involve negation-as-failure. The set X is an answer set for P if it is an answer set for P^X. The rationale behind the GL-reduct is that no atom which belongs to an answer set can (directly or indirectly) depend upon the negation of another atom that belongs to the same answer set. Therefore, the GL-reduct performed w.r.t. a set of atoms X which is a "candidate" answer set eliminates those rules involving some literal *not* A where $A \in X$. All negative literals can at this point be dropped from remaining rules as they trivially hold in X. Thus, X is actually an answer set if it models the resulting program. The meaning of a constraint then is that the literals in the body cannot be all true, otherwise the whole (resulting) rule could not be satisfied in any answer set.

1. As customary, a term (atom, literal, rule, ...) is ground if no variable occurs in it. A ground program is a program that does not contain variables.

Once a problem is described as an ASP-program P, its solutions (if any) are represented by the answer sets of P. Unlike other semantics, a logic program may have several answer sets, or may have no answer set, because conclusions are included in an answer set only if they can be justified. The following program has no answer set: $\{a \leftarrow \mathit{not}\ b.\ b \leftarrow \mathit{not}\ c.\ c \leftarrow \mathit{not}\ a.\}$. The reason is that in every minimal model of this program there is a true atom that depends (in the program) on the negation of another true atom. Whenever a program has no answer sets, we will say that the program is *inconsistent*. Correspondingly, checking for consistency means checking for the existence of answer sets.

Let us consider the program P consisting of the three rules

$$r \leftarrow p. \qquad p \leftarrow \mathit{not}\ q. \qquad q \leftarrow \mathit{not}\ p.$$

Such a program has two answer sets: $\{p, r\}$ and $\{q\}$. If we add the rule (actually, a constraint) $\leftarrow q.$ to P, then we rule-out the second of these answer sets, because it violates the new constraint. This simple example reveals the core of the usual approach followed in formalizing/solving a problem with ASP. Intuitively speaking, the programmer adopts a "generate-and-test" strategy: first (s)he provides a set of rules describing the collection of (all) potential solutions. Then, the addition of constraints rules-out all those answer sets that are not desired real solutions.

Given a rule γ in a language $\mathcal{L}$, the *grounding* of γ w.r.t. $\mathcal{L}$ is the set of all ground rules obtainable from γ through (ground) instantiation using the constant symbols of $\mathcal{L}$. Usually, given a program P and a rule $\gamma \in P$, we will consider the grounding of γ w.r.t. the language underlying P. The grounding of a set of rules is defined similarly. Given a (not necessarily ground) program P, a set of atoms is an answer set for P if it is an answer set for the grounding of P.

To find the solutions of an ASP-program, an ASP-solver is used. Several solvers have became available, see (Web references on ASP solvers), each of them being characterized by its own prominent valuable features. As it is well-known, ASP solvers produce the grounding of the given program as a first step, as they are able to find the answer sets of ground programs only.[2]

The expressive power of ASP, as well as, its computational complexity have been deeply investigated. The interested reader can refer, for instance, to (Dantsin, Eiter, Gottlob, & Voronkov, 2001). The reader can also see (Baral, 2003; Dovier, Formisano, & Pontelli, 2009), among others, for a presentation of ASP as a tool for declarative problem-solving.

2. Weight and Cardinality Constraints in ASP

Weight and Cardinality constraints were introduced in (Niemelä, Simons, & Soininen, 1999; Simons, Niemelä, & Soininen, 2002), where their semantics is also presented, as well as their implementation in the context of the smodels ASP solver. Though the computational complexity of ASP with weight constraints remains the same, the modeling power of the extended language is higher, as proved by the wide application of this construct (see, e.g., (Soininen, Niemelä, Tiihonen, & Sulonen, 2001)). A *Weight Constraint* is of the form (in the language of lparse (Syrjänen, 2000), i.e., in the language of the grounding module of smodels[3]):

$$l[a_1 = w_{a_1}, \ldots, a_n = w_{a_n}, \mathit{not}\ b_1 = w_{b_1}, \ldots, \mathit{not}\ b_m = w_{b_m}]u$$

2. Work is under way both theoretically and practically to overcome at least partially this limitation (cf., (Dal Palù, Dovier, Pontelli, & Rossi, 2009; Lefèvre & Nicolas, 2009), for instance). However, at present almost all ASP solvers perform the grounding.

3. This language has however been widely adopted. Most existing answer set solvers are compliant w.r.t. this language.

where the a_is and b_is are atoms. Each literal in a constraint has an associated weight, i.e., the weight of each a_i is w_{a_i} and the weight of each *not* b_j is w_{b_j} The numbers l and u give, respectively, the lower and upper bounds of the constraint. The weights and bounds are real numbers. Intuitively, a weight constraint is satisfied by a set of atoms S if the sum of weights of those literals occurring in the constraint that are satisfied by S is between l and u. Either of the bounds can also be omitted, in which case it is taken to be not relevant. The intended meaning is that an answer set is allowed to include a subset of the atoms occurring in the constraint so that the corresponding sum of weights results within $[l, u]$, where the weight of a negative literal *not* b_j is counted only if b_j is not in the answer set.

Plain literals can be seen as a special case of weight constraint, thus a program rule (called weight constraint rule) will have the form

$$C_0 \leftarrow C_1, \ldots, C_n.$$

where the C_is are weight constraints. As mentioned, the extra-expressivity is obtained without increasing complexity of ASP. As proved in (Niemelä et al., 1999; Simons et al., 2002), deciding whether a program composed of a set of ground weight constraint rules has an answer set is still NP-complete, and computing an answer set is still FNP-complete.

Weight constraints have become in time a very important and widely used programming tool in ASP, also in the formulation where all weights are equal to one. Weight constraints in this special form are called *Cardinality Constraints*, for which the following shorthand form is provided:

$$l\,\{a_1, \ldots, a_n, \mathit{not}\ b_1, \ldots, \mathit{not}\ b_m\}\,u$$

In order to be able to compactly write down sets of literals for weight constraints, (Niemelä et al., 1999; Simons et al., 2002) introduce a notion of a conditional literal of the form $l : d$ where l is a literal and the conditional part d is a domain predicate, where the subset of given program defining *domain predicates* consists of *domain rules*, syntactically restricted so as to admit a unique answer set that should be relatively efficiently computable. All the other rules in the program are required by most answer set solvers to be *domain-restricted* in the sense that every variable in a rule must appear in a domain predicate which occurs positively in the body of the rule. A formal definition can be found in (Syrjänen, 2000), Section 4.4, or (Baral, 2003), Section 8.1.2.

A conditional literal corresponds to the sequence of all the instances of the literal l obtained by making a substitution to $l : d$ such that for the resulting $l' : d'$, d' is in the unique answer set of the domain part of the program. This allows programmers to write, in domain-restricted programs, weight and cardinality constraints involving variables. We remind the reader that, after parsing, current ASP solvers perform the *grounding* of given program where a module (that we will call "grounder") produces the ground instantiation of the program, by substituting all variables by means of constants in every possible way. The grounder also performs a number of useful checks and simplifications. For instance, it is able to get rid of domain predicates after computing their extension.

Definition of syntax and semantics of PWCs, basically reported from (Pichler, Rümmele, Szeider, & Woltran, 2010), is provided below. **Optional for Intelligent Agents students**

Definition 2.1 (Weight constraint programs (PWCs)) *A (ground) program with weight constraints (PWC) is a triple* $\Pi = (A, \mathcal{C}, \mathcal{R})$*, where* A *is a set of ground atoms,* $\mathcal{C}$ *is a set of weight constraints*

(WCs, or constraints for short), and $\mathcal{R}$ is a set of rules, defined as follows: each constraint $c \in \mathcal{C}$ is a triple (S, l, u) where S is a set of weight literals over A and $l \leq u$ are non-negative integers, the lower and upper bound. A weight literal over A is a pair (a, j) or $(\neg a, j)$ for $a \in A$ and $1 \leq j \leq u + 1$, the weight of the literal. Given constraint $c = (S, l, u) \in \mathcal{C}$, we indicate S with $Cl(c)$, l with $l(c)$ and u with $u(c)$. Moreover, we sometimes write $a \in Cl(c)$ and $\neg a \in Cl(c)$ as an abbreviation for $(a, j) \in Cl(c)$ and, respectively, $(\neg a, j) \in Cl(c)$ for an arbitrary j. A weight constraint where for every weight literal $(a, j) \in Cl(c)$ and $(\neg a, j) \in Cl(c)$ we have $j = 1$ is called a cardinality constraint*. A rule $r \in \mathcal{R}$ is a pair (h, b) where $h \in \mathcal{C}$ (the head) and $b \subseteq \mathcal{C}$ (the body). For any rule r, we indicate h with $(\mathrm{H}(r))$ and b with $\mathrm{B}(r)$. Given a constraint $c \in \mathcal{C}$ and a set of atoms $I \subseteq A$, we denote the weight of c in I by*

$$W(c, I) = \sum\nolimits_{(a,j) \in Cl(c), a \in I} j + \sum\nolimits_{(\neg a,j) \in Cl(c), a \notin I} j.$$

I is a model of c (denoted by $I \models c$) iff $l(c) \leq W(c, I) \leq u(c)$. For a set $C \subseteq \mathcal{C}$, $I \models C$ iff $I \models c$ for all $c \in C$. Moreover, C is a model of a rule $r \in \mathcal{R}$ (denoted by $C \models r$) iff $h(r) \in C$ or $b(r) \not\subseteq C$. For a set $R \subseteq \mathcal{R}$, $C \models R \Leftrightarrow C \models r$ for all $r \in R$. I is a model of Π (denoted by $I \models \Pi$) iff $\{c \in C : I \models c\} \models \mathcal{R}$. If the lower bound of a constraint $c \in \mathcal{C}$ is missing, we assume $l(c) = 0$. If the upper bound is missing, $I \models c$ iff $l(c) \leq W(c, I)$.

Answer sets of a PWC can be obtained by means of an extension to the GL-reduct that, instead of removing rules where some *negative literal* in the body is not modeled in given set of atoms (candidate answer set) I, removes rules where some *weight constraint* in the body is not modeled. To make the check easier, upper bound of constraints is removed and lower bound is rearranged, so as to get rid of negative literals. All the remaining weight constraints are thus fulfilled (i.e., their bounds are respected). For each constraint occurring as the head of a rule, that rule is replicated, one copy for each of its positive literals which is in I. Thus, a positive PWC is obtained where the heads of rules are atoms. Finally, I is actually an answer set if it model this resulting program.

Definition 2.2 (PWC Semantics) *Consider PWC $\Pi = (A, \mathcal{C}, \mathcal{R})$ and an interpretation $I \subseteq A$. Following (Simons et al., 2002), the reduct c^I of a constraint $c \in \mathcal{C}$ w.r.t. I is obtained by removing all negative literals and the upper bound from c, and replacing the lower bound by*

$$l' = max(0, l(c) - \sum\nolimits_{(\neg a,j) \in Cl(c), a \notin I} j).$$

The reduct Π^I of program Π w.r.t. I can be obtained by first removing each rule $r \in \mathcal{R}$ which contains a constraint $c \in B(r)$ with $W(c, I) > u(c)$. Afterwards, each remaining rule r is replaced by the set of rules (h, b), where $h \in I \cap Cl(\mathrm{H}(r))$ and $b = \{c^I : c \in \mathrm{B}(r)\}$. Interpretation I is called an answer set (or stable model) of Π iff I is a model of Π^I and there exists no $J \subset I$ such that J is a model of Π^I.

3. Weight and Cardinality Constraints: a Case-Study

Our running example is freely inspired to the Italian Computer Science undergraduate Program, that we shortly describe here in its basic features.

In order to get a bachelor degree in Computer Science ("Informatica"), an Italian student is required to obtain 180 credits. Most of them must be obtained by attending courses and passing the

corresponding exams. The remaining ones can be obtained by means of stages and a short Thesis. There is a certain flexibility, so usually the number of credits that should be obtained from courses is allowed to vary within a range, say (just for example, as actual ranges vary among different Universities and tracks) between 153 and 171. There are different possible choices for the courses to attend, so students are required to present what is called a "plan of studies", that must be approved by a Committee.

Some courses must be taken at a certain year, for others there is some flexibility. For simplicity, we assume that the latter can be taken at any year and we neglect constraints related to the order in which certain courses should be taken. Basically, the above (as described up to now) might be summarized by a single ASP rule (precisely, a fact) consisting of the following weight constraint, that generates possible plans of studies (that we indicate with ps). There, the c_is are names of courses, and the w_{c_i}s are their values in credits. $in\_ps(c_i, j)$ means that course c_i is inserted into the plan of studies, at year j. Since some courses are assigned to a certain year and others are free, in the formulation below the $\mathcal{Y}$s can be either a constant (ranging from one to three) or a variable, say Y, in domain $course\_year$, again ranging from one to three:

$$153\ [in\_ps(c_1, \mathcal{Y}_1) = w_{c_1}, \ldots, in\_ps(c_n, \mathcal{Y}_n) = w_{c_n}]\ 171$$

A specific instance of this constraints might, e.g., look like:

$$153\ [in\_ps(algorithms\_with\_lab, 1) = 12, in\_ps(calculus\_basic, 1) = 6,\\ in\_ps(algorithms\_advanced,\ Y) : course\_year(Y) = 12, \ldots]\ 171$$

Below, for the convenience of the reader, we report a tiny instance of such a problem, in the form of real ASP code which can be run using smodels or any other solver compliant to the lparse language.

```
course_year(1..3).

32[in_ps(programming,1)=12,
in_ps(computer_architectures,V):course_year(V)=6,
in_ps(databases,V):course_year(V)=12,
in_ps(theoretical_cs,V):course_year(V)=6,
in_ps(algorithms,2)=12,
in_ps(calculus,3)=6,
in_ps(optimization,3)=6]52.
```

This reduced instance (where also the lower and upper bound are given just for an example) allows us to discuss some aspects of this formalization that might in our opinion be improved. To follow the discussion, the reader is required to understand basic ASP code. In fact, we will point out the pitfalls and possible improvements of the formalization. In the first place, we will propose standard ASP solutions. Then, we will propose an enhanced formulation which in our view encompasses the improvements in a more readable and also potentially more efficient way.

First, in this setting the predicate in_ps should be defined only in the context of the constraint, in the sense that its definition should not possibly be altered by statements that lay outside. This might be obtained by means of a declaration, and might be easily checked during the grounding process. This aspect is impossible to obtain in standard ASP. Second, we would like to be sure that

constants which occur inside the constraint represent existing course (namely those defined below) and existing course years, as defined above.

```
course(programming).
course(computer_architectures).
course(databases).
course(algorithms).
course(theoretical_cs).
course(calculus).
course(optimization).
```

This kind of "domain checking" is very difficult to obtain in ASP. One way is reported below. One has to define, in order to respect domain restrictedness, one fact for each constant c occurring in the program, say $val(c)$. Then one may specify when each argument is mistaken (that is, it does not belong to the required domain). Finally, constraints should be added to state that such errors must not occur.

```
val(1).
val(2).
val(3).
val(programming).
val(computer_architectures).
val(databases).
val(algorithms).
val(theoretical_cs).
val(calculus).
val(optimization).

% domain restriction for arguments of in_ps
in_ps1err(K):- val(K),not course(K).
in_ps2err(Z):- val(Z),not course_year(Z).

% domain check for arguments of in_ps
:- val(C),val(Z),in_ps(C,Z),in_ps1err(C).
:- val(C),val(Z),in_ps(C,Z),in_ps2err(Z).
```

The above formulation has the problem that whenever one adds or drops new constants from the program (by adding/deleting rules), and we mean whatsoever constant, not only those related to the piece of code that we are considering, one has to update the val facts accordingly. Notice that the type checking is performed outside the weight constraint, with no evidence of the fact that it refers to it.

The above-listed weight constraint is not yet fully satisfactory with respect to all aspects of the problem at hand. An aspect which is not represented is that repetitions are not allowed: obviously, you cannot build a plan of study where you take for instance algorithms several times, e.g., both at the first and third year. This can be obtained by adding the following constraint:

```
:- in_ps(C,Y),course(C),course_year(Y),
in_ps(C,Y1),course_year(Y1),Y!=Y1.
```

Equivalently, one may adopt the cardinality constraint:

```
0{in_ps(C,Y):course_year(Y)}1:-course(C).
```

We remind the reader of how the grounding process treats such a constraint: first one rule for each course is generated, then the constraint is instantiated, for course c, to include the atoms $in\_ps(c,1)$, $in\_ps(c,2)$, $in\_ps(c,3)$ among which at most one can be chosen. The problem here is that, being this constraint located outside the overall weight constraint, there is no evidence of their being related.

In our case-study, it is always the case that some courses are mandatory. Moreover, some are not only mandatory, but also they must be situated at a certain course year. To establish that a course is mandatory, one might specify, for each such course c:

```
1{in_ps(c,Y):course_year(Y)}1.
```

For courses which are mandatory at a certain course year, one might specify, for each such course c that should be located at year y:

```
1{in_ps(c,y)}1.
```

Notice that this is not a repetition with respect to the main weight constraint: in fact, the weight constraint represents the plan of study that a student tries to construct, while the above constraints represent what should be done according to the regulations. It would seem advisable to join the two aspects: i.e., to introduce a form of weight constraints that allows one to define what (s)he would like, while enforcing respect of contextual restrictions.

A more explicit (though more lengthy) version of the above constraints (in the sense that from the code it is more easy to understand its purpose) would be the following.

```
mandatory(programming,1).
mandatory(computer_architectures,1).
mandatory(algorithms,2).
mandatory(theoretical_cs,3).
mandatory_course(databases).

% Mandatory courses always included
in_ps_course(C):- in_ps(C,Y),course(C),course_year(Y).
:- course(C),mandatory_course(C),not in_ps_course(C).
%in mandatory course year
:- in_ps(C,Y),course(C),course_year(Y),mandatory(C,Y1), Y!=Y1.
```

Finally, to avoid a student giving too many exams, there is a ministry statement that enforces at least a minimum number of courses of the first two years to weigh 12 credits. Also, courses are allowed to belong to certain scientific areas, namely Computer Science, Mathematics, Physics and other different though related topics (within a list). However, there are directions stating that

every subject should contribute to the plan of studies for a quota ranging between a minimum and a maximum number of credits. These aspects can be coped in ASP, again with poor elaboration-tolerance. In fact, one should separately provide a description of courses, say e.g. the following, that is liable to easily become inconsistent with the overall weight constraint after modifications (course addition/deletion, changes in the number of credits).

```
course_description(programming,computer_science,12).
course_description(computer_architectures,computer_science,6).
course_description(databases,computer_science,12).
course_description(algorithms,computer_science,12).
course_description(theoretical_cs,computer_science,6).
course_description(calculus,mathematics,6).
course_description(optimization,mathematics,6).
```

Then, one has to use *aggregates* to count the number of occurrences of in_ps for the above aspects, i.e., courses of 12 credits in the first two years, and courses of each kind of required subject. After having computed aggregate values, suitable constraints should ensure that required intervals are respected. We do not even attempt here to produce the code, that would be very long and exceedingly involved. Just consider that the definition of an aggregate for counting on a single parameter is shown in (Baral, 2003), Section 2.1.14, and requires in practical cases several pages of code. It is relevant to notice that the DLV answer set solver (Leone, Pfeifer, Faber, Eiter, Gottlob, Perri, & Scarcello, 2006) provides pre-defined aggregates that can be used for coping in a concise way with this aspect.

References

Baral, C. (2003). *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press.

Dal Palù, A., Dovier, A., Pontelli, E., & Rossi, G. (2009). GASP: Answer set programming with lazy grounding. *Fundamenta Informaticae*, *96*, 297–322.

Dantsin, E., Eiter, T., Gottlob, G., & Voronkov, A. (2001). Complexity and expressive power of logic programming. *ACM Computing Surveys*, *33*(3), 374–425.

Dovier, A., Formisano, A., & Pontelli, E. (2009). An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *Journal of Experimental and Theoretical Artificial Intelligence*, *21*(2), 79–121.

Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming. In Kowalski, R., & Bowen, K. (Eds.), *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP/SLP'88)*, pp. 1070–1080. The MIT Press.

Lefèvre, C., & Nicolas, P. (2009). A first order forward chaining approach for answer set computing. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, No. 5753 in Lecture Notes in Computer Science, pp. 196–208. Springer.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., & Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational*

Logic, *7*(3), 499–562. User manual : http://www.dlvsystem.com/dlvsystem/html/DLV_User_Manual.html.

Lifschitz, V. (1999). Answer set planning. In De Schreye, D. (Ed.), *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pp. 23–37. The MIT Press.

Marek, V. W., & Truszczyński, M. (1999). Stable logic programming - An alternative logic programming paradigm. In *25 years of Logic Programming Paradigm*, pp. 375–398. Springer.

Niemelä, I., Simons, P., & Soininen, T. (1999). Stable model semantics of weight constraint rules. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, No. 1730 in Lecture Notes in Computer Science, pp. 317–331. Springer.

Pichler, R., Rümmele, S., Szeider, S., & Woltran, S. (2010). Tractable answer-set programming with weight constraints: Bounded treewidth is not enough. In *Proc. of the Twelfth International Conference on the Principles of Knowledge Representation and Reasoning (KR 2010)*. AAAI Press, Menlo Park.

Simons, P., Niemelä, I., & Soininen, T. (2002). Extending and implementing the stable model semantics. *Artificial Intelligence*, *138*(1-2), 181–234.

Soininen, T., Niemelä, I., Tiihonen, J., & Sulonen, R. (2001). Representing configuration knowledge with weight constraint rules. In *Proceedings of the AAAI Spring 2001 Symposium on Answer Set Programming (ASP'01): Towards Efficient and Scalable Knowledge*. AAAI Press, Menlo Park. Technical report SS-01-01.

Syrjänen, T. (2000). Lparse 1.0 user's manual.. Available at http://www.tcs.hut.fi/Software/smodels.

Web references on ASP solvers Clasp: http://http://potassco.sourceforge.net; Cmodels: http://www.cs.utexas.edu/users/tag/cmodels; DLV: http://www.dbai.tuwien.ac.at/proj/dlv; Smodels: http://www.tcs.hut.fi/Software/smodels.