On the Implementation of Weight Constraint Rules in Conflict-Driven ASP Solvers

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

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

Abstract. We present the first comprehensive approach to integrating cardinality and weight rules into conflict-driven ASP solving. We begin with a uniform, constraint-based characterization of answer sets in terms of nogoods. This provides the semantic underpinnings of our approach in fixing all necessary inferences that must be supported by an appropriate implementation. We then provide key algorithms detailing the salient features needed for implementing weight constraint rules. This involves a sophisticated unfounded set checker as well as an extended propagation algorithm along with the underlying data structures. We implemented our techniques within the ASP solver *clasp* and demonstrate their effectiveness by an experimental evaluation.

1 Introduction

One of the most appealing features of Answer Set Programming (ASP; [1]) is its rich declarative modeling language. Among the most popular language constructs are cardinality and weight constraints [2] being particular forms of count and sum aggregates.

Existing techniques for implementing such aggregates fall into two categories. Traditional backtracking-oriented ASP solvers like *smodels* [2] use counter-based algorithms based on [3]. On the other hand, SAT-based ASP solvers like *cmodels* [4] eliminate such aggregates by transforming them into normal (or nested) logic programming rules. While the former approach has proven its versatility, it does not carry over to modern ASP solving technology based on backjumping and conflict-driven learning [5, 6]. Although this is accomplishable by the transformational approach, it fails to scale due to a significant increase in space [7].

We address this problem and present the first comprehensive approach to integrating weight constraint rules into conflict-driven ASP solving. To this end, we begin with a uniform, constraint-based characterization of answer sets in terms of nogoods. This provides the semantic underpinnings of our approach in fixing all necessary inferences that must be supported by an appropriate implementation. We then provide key algorithms detailing the salient features needed for implementing weight constraint rules. This involves a sophisticated unfounded set checker as well as an extended propagation algorithm along with the underlying data structures. Our techniques are implemented within the ASP solver *clasp* [8]. We evaluate the performance of *clasp* relative to the two existing approaches and thus demonstrate its effectiveness.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

2 Background

Following [2], we consider *weight constraint programs* over an alphabet $\mathcal{A}$, consisting of *weight rules* of the form

$$v\,\{a_0 = 1\} \leftarrow w\,\{a_1 = w_1, \ldots, a_m = w_m, \sim a_{m+1} = w_{m+1}, \ldots, \sim a_n = w_n\} \quad (1)$$

where $v \in \{0,1\}$, w is a non-negative integer, w_i are positive integers for $1 \leq i \leq n$, and a_i are *atoms* in $\mathcal{A}$ for $0 \leq i \leq n$. Furthermore, we assume $a_j \neq a_k$ for $0 < j < k \leq m$ and $m < j < k \leq n$, respectively. The set of atoms occurring in a weight constraint program Π is denoted by $A(\Pi)$. A *weight literal* is of the form $a = w$ or $\sim a = w$; a and $\sim a$ are *regular literals*, where $\sim$ stands for default negation. For a set A of atoms, we let $\sim A = \{\sim a \mid a \in A\}$; for a set L of regular literals, let $L^+ = \{a \mid a \in L \cap \mathcal{A}\}$ and $L^- = \{a \mid a \in L \cap \sim\mathcal{A}\}$. We define $A(l = w) = a$ for the atom in a weight literal $l = a$ or $l = \sim a$, respectively, and $W(l = w) = w$ for its weight. Accordingly, for a set $\mathcal{L}$ of weight literals, $A(\mathcal{L}) = \{A(\ell) \mid \ell \in \mathcal{L}\}$ and $\Sigma[\mathcal{L}] = \sum_{\ell \in \mathcal{L}} W(\ell)$.

For a rule r as in (1), let $H(r) = v\,\{a_0 = 1\}$ be the *head* of r, $B(r) = w\,\{a_1 = w_1, \ldots, a_m = w_m, \sim a_{m+1} = w_{m+1}, \ldots, \sim a_n = w_n\}$ the *body* of r, and $lb(B(r)) = w$ the *lower bound* of $B(r)$. Such a body constitutes a *weight constraint*. We extend the above projections to weight constraints as follows. Given $\mathcal{W} = B(r)$ for r as in (1), we define $\mathcal{W}^+ = \{a_1 = w_1, \ldots, a_m = w_m\}$, $\mathcal{W}^- = \{\sim a_{m+1} = w_{m+1}, \ldots, \sim a_n = w_n\}$, and $A(\mathcal{W}) = A(\mathcal{W}^+ \cup \mathcal{W}^-)$. A set X of atoms satisfies $\mathcal{W}$, written $X \models \mathcal{W}$, if

$$\Sigma[\{p \in \mathcal{W}^+ \mid A(p) \in X\} \cup \{n \in \mathcal{W}^- \mid A(n) \notin X\}] \;\geq\; lb(\mathcal{W})\,.$$

That is, a weight constraint is satisfied if the sum of the weights of its satisfied literals does not fall below the lower bound given by w. Accordingly, rule r is satisfied by X, written $X \models r$, if $X \models B(r)$ implies $X \models H(r)$; and $X \models \Pi$ if $X \models r$ for all $r \in \Pi$.

For a rule r as in (1) and a set X of atoms, the reduct of $B(r)$ wrt X is defined as

$$B(r)^X = w'\,B(r)^+ \text{ where } w' = \max\left\{0\,,\, lb(B(r)) - \Sigma[\{n \in B(r)^- \mid A(n) \notin X\}]\right\}.$$

Given this, the *reduct* of a weight constraint program Π wrt X is

$$\Pi^X = \left\{\, 1\,\{a_0 = 1\} \leftarrow B(r)^X \mid r \in \Pi, A(H(r)) \cap X = \{a_0\}\,\right\}.$$

Finally, X is an *answer set* of Π if $X \models \Pi$ and $Y \not\models \Pi^X$ for all $Y \subset X$.

As detailed in [2], weight constraint rules are expressive enough to (linearly) capture normal rules, integrity constraints, cardinality rules, and general weight constraint rules of the form $\mathcal{W}_0 \leftarrow \mathcal{W}_1, \ldots, \mathcal{W}_n$, where $\mathcal{W}_i$ is a general weight constraint for $0 \leq i \leq n$.

3 Inferences from Weight Constraint Programs

This section provides the logical fundament of the computational techniques detailed in Section 4. To this end, we adapt the nogood-based characterization of answer sets from [8] to accommodate weight constraints. As a result, we obtain a clear semantic framework to specify (unit) propagation over weight rules.

An *assignment* $\mathbf{A}$ over a *domain*, $dom(\mathbf{A})$, is a sequence $(\sigma_1, \ldots, \sigma_n)$ of (*signed*) *literals* σ_i of the form $\mathbf{T}v_i$ or $\mathbf{F}v_i$, where $v_i \in dom(\mathbf{A})$ for $1 \leq i \leq n$; $\mathbf{T}v_i$ expresses that v_i is *true* and $\mathbf{F}v_i$ that it is *false*. (We omit the attribute *signed* for literals whenever clear from the context.) The complement of a literal σ is denoted by $\overline{\sigma}$, that is, $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$. We sometimes abuse notation and identify an assignment with the set of its contained literals. Given this, we access the true and false variables in $\mathbf{A}$ via $\mathbf{A}^{\mathbf{T}} = \{v \mid \mathbf{T}v \in \mathbf{A}\}$ and $\mathbf{A}^{\mathbf{F}} = \{v \mid \mathbf{F}v \in \mathbf{A}\}$. For a canonical representation of (Boolean) constraints, we make use of nogoods [9]. In our setting, a *nogood* is a finite set $\{\sigma_1, \ldots, \sigma_m\}$ of literals, expressing a constraint violated by any assignment $\mathbf{A}$ containing $\sigma_1, \ldots, \sigma_m$. For a set Δ of nogoods, an assignment $\mathbf{A}$ is a *solution* for Δ if $\mathbf{A}^{\mathbf{T}} \cap \mathbf{A}^{\mathbf{F}} = \emptyset$, $\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}} = dom(\mathbf{A})$, and $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta$. Given a weight constraint program Π, we adopt the convention that $dom(\mathbf{A}) = A(\Pi) \cup \{H(r), B(r) \mid r \in \Pi\}$.

For a weight constraint $\mathcal{W}$, the following pair of sets of nogoods stipulates correspondence between the truth value of $\mathcal{W}$ and the sum of true literals' weights:

$$\omega(\mathcal{W}) = \big\{\{\mathbf{F}\mathcal{W}\} \cup \{\mathbf{T}A(p) \mid p \in P\} \cup \{\mathbf{F}A(n) \mid n \in N\} \mid P \subseteq \mathcal{W}^+, N \subseteq \mathcal{W}^-, \Sigma[P \cup N] \geq lb(\mathcal{W})\big\} \tag{2}$$

$$\varpi(\mathcal{W}) = \big\{\{\mathbf{T}\mathcal{W}\} \cup \{\mathbf{F}A(p) \mid p \in P\} \cup \{\mathbf{T}A(n) \mid n \in N\} \mid P \subseteq \mathcal{W}^+, N \subseteq \mathcal{W}^-, \Sigma[(\mathcal{W}^+ \setminus P) \cup (\mathcal{W}^- \setminus N)] < lb(\mathcal{W})\big\}\,. \tag{3}$$

Observe that the nogoods in $\omega(\mathcal{W})$ and $\varpi(\mathcal{W})$, respectively, capture the weakest conditions under which $\mathcal{W}$ evaluates to true or false, respectively. In general, the number of such weakest conditions is exponential in the number of literals in $\mathcal{W}$. Hence, it is impractical to explicitly construct $\omega(\mathcal{W})$ and $\varpi(\mathcal{W})$, and we below develop implementation techniques for unit propagation that work on $\mathcal{W}$ directly.

The correspondence between the truth of a weight constraint and its elements can be formalized as follows.

Proposition 1. *Let $\mathcal{W}$ be a weight constraint and $\mathbf{A}$ be an assignment such that $\mathbf{A}^{\mathbf{T}} \cap \mathbf{A}^{\mathbf{F}} = \emptyset$ and $\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}} = A(\mathcal{W})$. Then, the following statements hold:*

1. *$\delta \setminus \mathbf{A} = \{\mathbf{F}\mathcal{W}\}$ for some $\delta \in \omega(\mathcal{W})$ iff $\mathbf{A}^{\mathbf{T}} \models \mathcal{W}$;*
2. *$\delta \setminus \mathbf{A} = \{\mathbf{T}\mathcal{W}\}$ for some $\delta \in \varpi(\mathcal{W})$ iff $\mathbf{A}^{\mathbf{T}} \not\models \mathcal{W}$.*

Proposition 1 shows that the weight constraints in a weight constraint program are fully determined by their literals when collecting the nogoods for all heads and bodies:

$$\Omega(\Pi) = \bigcup\nolimits_{r \in \Pi} \big(\omega(H(r)) \cup \varpi(H(r)) \cup \omega(B(r)) \cup \varpi(B(r))\big)\,.$$

As an answer set X of a program Π is a minimal model of Π^X, we have that a corresponding total assignment $\mathbf{A}$, viz., $\mathbf{A}^{\mathbf{T}} \cap A(\Pi) = X$, must be a model of Π, and each atom in X needs to be supported by a rule r such that $B(r) \in \mathbf{A}^{\mathbf{T}}$. When combined with $\Omega(\Pi)$, the following set of nogoods formalizes these two requirements:

$$\Delta(\Pi) = \big\{\{\mathbf{F}H(r), \mathbf{T}B(r)\} \mid r \in \Pi\big\} \cup \big\{\{\mathbf{T}a, \mathbf{F}B(r) \mid r \in \Pi, A(H(r)) = \{a\}\} \mid a \in A(\Pi)\big\}\,.$$

Proposition 2. *Let Π be a weight constraint program and $X \subseteq A(\Pi)$. Then, $X \models \Pi$ such that, for every $a \in X$, there is some $r \in \Pi$ with $A(H(r)) = \{a\}$ and $X \models B(r)$ iff there is a (unique) solution $\mathbf{A}$ for $\Omega(\Pi) \cup \Delta(\Pi)$ such that $\mathbf{A^T} \cap A(\Pi) = X$.*

The nogoods associated with weight constraint programs allow us to identify propagation operations along with their reasons. We say that a nogood δ is *unit-resulting* wrt an assignment $\mathbf{A}$ if $\delta \setminus \mathbf{A} = \{\sigma\}$ and $\overline{\sigma} \notin \mathbf{A}$. In such a situation, $\overline{\sigma}$ is mandatory to avoid the inclusion of δ in $\mathbf{A}$; in other words, δ *implies* $\overline{\sigma}$ wrt $\mathbf{A}$. The process of iteratively adding implied literals to $\mathbf{A}$ until violating some nogood or reaching a fixpoint (without any further implied literals) is called *unit propagation*. The implementation within *clasp* of unit propagation on nogoods in $\Omega(\Pi)$ is detailed in Section 4. Note that $\Omega(\Pi)$ merely provides a logical specification, while *clasp* works on weight constraints directly and determines nogoods in $\Omega(\Pi)$ only if needed as reasons.

In order to also capture minimality of an answer set X as a model of Π^X, for a program Π and a (partial) assignment $\mathbf{A}$, we define a set $U \subseteq A(\Pi)$ as *unfounded* for Π wrt $\mathbf{A}$ if, for every rule $r \in \Pi$, some of the following conditions holds:

1. $A(H(r)) \cap U = \emptyset$,
2. $B(r) \in \mathbf{A^F}$, or
3. $\Sigma[\{p \in B(r)^+ \mid A(p) \notin \mathbf{A^F} \cup U\} \cup \{n \in B(r)^- \mid A(n) \notin \mathbf{A^T}\}] < lb(B(r))$.

If U is unfounded for Π wrt $\mathbf{A}$, it means that none of its atoms belongs to any answer set given by a total extension of $\mathbf{A}$. In fact, the first condition expresses that r cannot support U, while the second condition checks that r is not applicable under $\mathbf{A}$. Finally, the third condition detects cases where $lb(B(r))$ cannot be reached via (weight) literals not false under $\mathbf{A}$, thereby, disregarding positive literals that depend on U.

To describe unfounded set conditions in terms of nogoods, for a set U of atoms, we define the *external sets* of literals for U in a weight rule r, $ext_r(U)$, as:

$$\left\{A(P) \cup {\sim}A(N) \mid P \subseteq B(r)^+, N \subseteq B(r)^-, A(P) \cap U = \emptyset, \Sigma[P \cup N] \geq lb(B(r))\right\}.$$

Note that elements L of $ext_r(U)$ are exactly the sets of literals such that the third unfounded set condition does not apply to r as long as $(L^+ \cap \mathbf{A^F}) \cup (L^- \cap \mathbf{A^T}) = \emptyset$, that is, if no literal in L is falsified by $\mathbf{A}$. Furthermore, for $U \subseteq A(\Pi)$, we call a set $C \subseteq \bigcup_{r \in \Pi, L \in ext_r(U)} L$ of literals a *cover set* for U in Π, if $C \cap L \neq \emptyset$ for every $r \in \Pi$ and $L \in ext_r(U)$. Note that, for any $r \in \Pi$, a cover set C for U in Π satisfies $\Sigma[\{p \in B(r)^+ \mid A(p) \notin C \cup U\} \cup \{n \in B(r)^- \mid {\sim}A(n) \notin C\}] < lb(B(r))$; otherwise, we would have $L = \{l \mid (l = w) \in B(r), l \notin C \cup U\} \in ext_r(U)$ and $C \cap L = \emptyset$, so that C would not be a cover set for U in Π. Letting $cov_\Pi(U)$ denote the set of all cover sets for U in Π, for some $u \in U$, the *loop nogoods*, $\lambda(u, U)$, are:

$$\bigcup\nolimits_{\Lambda \subseteq \{r \in \Pi \mid A(H(r)) \cap U \neq \emptyset, ext_r(U) \neq \emptyset, A(B(r)^+) \cap U \neq \emptyset\}} \Big\{ \{\mathbf{F}a \mid a \in C^+\} \cup \{\mathbf{T}b \mid b \in C^-\} \cup \{\mathbf{T}u\} \cup \{\mathbf{F}B(r) \mid r \in \Pi \setminus \Lambda, A(H(r)) \cap U \neq \emptyset, ext_r(U) \neq \emptyset\} \mid C \in cov_\Lambda(U) \Big\}.$$

Note that, for all rules $r \in \Pi$ such that $A(H(r)) \cap U \neq \emptyset$ and $ext_r(U) \neq \emptyset$, nogoods in $\lambda(u, U)$ reflect the second ($B(r) \in \mathbf{A^F}$) and the third (via Λ) unfounded set condition. Given the correspondence of the truth value of $B(r)$ and those of its (weight)

literals stipulated via $\Omega(\Pi)$, the third condition needs to be checked separately only if $A(B(r)^+) \cap U \neq \emptyset$, which explains the choice of Λ. As with $\omega(\mathcal{W})$ and $\varpi(\mathcal{W})$ in (2) and (3), the size of $\lambda(u, U)$ is exponential in the number of literals in rule bodies, and on the implementation side, selected loop nogoods are determined on demand (see below).

For illustration, consider a program containing the following weight rules:

$$0\,\{a{=}1\} \leftarrow 2\,\{c{=}1, e{=}1, {\sim}b{=}1\} \tag{4}$$
$$1\,\{a{=}1\} \leftarrow 3\,\{b{=}2, {\sim}c{=}1, {\sim}d{=}1\} \tag{5}$$
$$1\{b{=}1\} \leftarrow 4\,\{a{=}3, c{=}2, {\sim}d{=}1, {\sim}e{=}3\}\,. \tag{6}$$

Taking $U = \{a, b\}$, we observe that the body of the rule in (4) does not positively depend on U, while the external sets for U in (5) are empty. For the rule r in (6), we get $ext_r(U) = \big\{\{c, {\sim}e\}, \{{\sim}d, {\sim}e\}, \{c, {\sim}d, {\sim}e\}\big\}$ and $cov_{\{r\}}(U) = \big\{\{{\sim}e\}, \{c, {\sim}d\}, \{c, {\sim}e\}, \{{\sim}d, {\sim}e\}, \{c, {\sim}d, {\sim}e\}\big\}$. Observe that $\{{\sim}e\}$ and $\{c, {\sim}d\}$ are the minimal cover sets for U in $\{r\}$, while the other three are subsumed by $\{{\sim}e\}$. We thus obtain the following non-redundant loop nogoods in $\lambda(u, U)$, where $u = a$ or $u = b$:

$$\big\{\mathbf{T}u, \mathbf{F}\,2\,\{c{=}1, e{=}1, {\sim}b{=}1\}, \mathbf{F}\,4\,\{a{=}3, c{=}2, {\sim}d{=}1, {\sim}e{=}3\}\big\}$$
$$\big\{\mathbf{T}u, \mathbf{F}\,2\,\{c{=}1, e{=}1, {\sim}b{=}1\}, \mathbf{T}e\big\}$$
$$\big\{\mathbf{T}u, \mathbf{F}\,2\,\{c{=}1, e{=}1, {\sim}b{=}1\}, \mathbf{F}c, \mathbf{T}d\big\}\,.$$

For a weight constraint program Π, we can now simply collect all loop nogoods:

$$\Lambda(\Pi) = \bigcup_{\emptyset \subset U \subseteq A(\Pi), u \in U} \lambda(u, U)\,.$$

These nogoods ultimately establish a one-to-one correspondence between answer sets and solutions.

Theorem 1. *Let Π be a weight constraint program and $X \subseteq A(\Pi)$. Then, X is an answer set of Π iff there is a (unique) solution $\mathbf{A}$ for $\Omega(\Pi) \cup \Delta(\Pi) \cup \Lambda(\Pi)$ such that $\mathbf{A^T} \cap A(\Pi) = X$.*

The basic *clasp* algorithm, relying on conflict-driven learning [5, 6], has been described in [8], and its global structure remains unaffected if the nogoods to work with are exchanged. However, the identification of unfounded sets, described in [10] for disjunctive offspring *claspD*, needs to be adapted to weight constraint programs. We thus provide the logics of a dedicated unfounded set checking algorithm in Algorithm 1; its implementation in *clasp* will be described in Section 4. Given a program Π and an assignment $\mathbf{A}$, we assume that there is a predefined set $Do \subseteq A(\Pi)$ of atoms to investigate. Furthermore, each atom $a \in \bigcup_{r \in \Pi} A(H(r))$ has a *source pointer* [2], denoted by $sp(a)$, to a weight constraint $B(r)$ such that $A(H(r)) = \{a\}$ for some $r \in \Pi$; a source pointer $sp(a)$ has an associated set $sp(a)^\#$ of atoms considered as not belonging to any unfounded set $U \subseteq A(\Pi) \setminus \mathbf{A^F}$. Finally, for every $a \in A(\Pi)$, number $c(a)$ denotes a strongly connected component C of the positive atom dependency graph of Π, defined by $(A(\Pi), \{(a, b) \mid r \in \Pi, A(H(r)) = \{a\}, b \in A(B(r)^+)\})$; atoms a of trivial strongly connected components (without edges) are identified by $c(a) = 0$. As pointed out in [2], unfounded set checking can be localized to non-trivial strongly

Algorithm 1: UNFOUNDEDSET

Input : A weight constraint program Π and an assignment $\mathbf{A}$.
Output : An unfounded set for Π wrt $\mathbf{A}$.

1 $Do \leftarrow Do \setminus \mathbf{A^F}$
2 $Add \leftarrow \{a \in A(\Pi) \setminus (\mathbf{A^F} \cup Do) \mid c(a) \neq 0, sp(a) \in \mathbf{A^F}\}$
3 **repeat**
4 $\quad Do \leftarrow Do \cup Add$
5 $\quad$ **foreach** $a \in A(\Pi)$ **such that** $sp(a)^{\#} \cap Add \neq \emptyset$ **do** $sp(a)^{\#} \leftarrow sp(a)^{\#} \setminus Add$
6 $\quad Add \leftarrow \{a \in A(\Pi) \setminus (\mathbf{A^F} \cup Do) \mid c(a) \neq 0, \Sigma[\{n \in sp(a)^- \mid A(n) \notin \mathbf{A^T}\} \cup \{p \in sp(a)^+ \mid A(p) \notin \mathbf{A^F}, c(A(p)) \neq c(a) \text{ or } A(p) \in sp(a)^{\#}\}] < lb(sp(a))\}$
7 **until** $Add = \emptyset$
8 **while** $Do \neq \emptyset$ **do let** $a \in Do$ **in**
9 $\quad U \leftarrow \{a\}$
10 $\quad$ **repeat**
11 $\quad\quad B \leftarrow \{B(r) \mid r \in \Pi, A(H(r)) \cap U \neq \emptyset, \Sigma[\{n \in B(r)^- \mid A(n) \notin \mathbf{A^T}\} \cup \{p \in B(r)^+ \mid A(p) \notin \mathbf{A^F} \cup U\}] \geq lb(B(r)), B(r) \notin \mathbf{A^F}\}$
12 $\quad\quad$ **if** $B = \emptyset$ **then return** U
13 $\quad\quad$ **else let** $\mathcal{W} \in B$ **in**
14 $\quad\quad\quad S \leftarrow \{s \in A(\mathcal{W}^+) \cap Do \mid c(s) = c(a)\}$
15 $\quad\quad\quad$ **if** $\Sigma[\{n \in \mathcal{W}^- \mid A(n) \notin \mathbf{A^T}\} \cup \{p \in \mathcal{W}^+ \mid A(p) \notin \mathbf{A^F} \cup S\}] \geq lb(\mathcal{W})$ **then**
16 $\quad\quad\quad\quad$ **if** $\{s \in A(\mathcal{W}^+) \mid c(s) = c(a)\} \neq \emptyset$ **then**
17 $\quad\quad\quad\quad\quad \mathcal{W}^{\#} \leftarrow \{s \in A(\mathcal{W}^+) \mid c(s) = c(a), s \notin \mathbf{A^F} \cup S\}$
18 $\quad\quad\quad\quad$ **foreach** $u \in U$ **such that** $\{r \in \Pi \mid A(H(r)) = \{u\}, B(r) = \mathcal{W}\} \neq \emptyset$ **do**
19 $\quad\quad\quad\quad\quad sp(u) \leftarrow \mathcal{W}$
20 $\quad\quad\quad\quad\quad U \leftarrow U \setminus \{u\}$
21 $\quad\quad\quad\quad\quad Do \leftarrow Do \setminus \{u\}$
22 $\quad\quad\quad$ **else** $U \leftarrow U \cup S$
23 $\quad$ **until** $U = \emptyset$
24 **return** $\emptyset$

connected components (SCCs) without sacrificing soundness. In turn, we require as an invariant that $(A(\Pi), \{(a, b) \mid a \in A(\Pi), c(a) \neq 0, b \in sp(a)^{\#}\})$ is an acyclic graph (viz., all of its SCCs must be trivial). We then skip unfounded set checks for a as long as $sp(a) \notin \mathbf{A^F}$ and $\Sigma[\{n \in sp(a)^- \mid A(n) \notin \mathbf{A^T}\} \cup \{p \in sp(a)^+ \mid A(p) \notin \mathbf{A^F}, c(A(p)) \neq c(a) \text{ or } A(p) \in sp(a)^{\#}\}] \geq lb(sp(a))$, which means that some acyclic justification exists for a so that it cannot be unfounded.

The main *clasp* algorithm [8] triggers propagation, which includes unfounded set checks, after every heuristic decision. We assume that Do is empty when a heuristic decision is made, and repeated calls to UNFOUNDEDSET may successively fill it with atoms to check for unfoundedness. As a matter of fact, atoms falsified by unit propagation can be excluded and are thus eliminated in Line 1 of Algorithm 1. Non-false atoms whose source pointers have been falsified are scheduled for an unfounded set check in

Line 2 and 4; note that such atoms a must belong to non-trivial SCCs of the positive atom dependency graph of Π ($c(a) \neq 0$). The purpose of Line 6 is to iteratively identify atoms a such that $sp(a) \notin \mathbf{A}^{\mathbf{F}}$, while the existence of an acyclic justification is still not guaranteed. Iteration is needed because, in Line 5, possible occurrences of atoms that got into the scope of unfounded checks are removed from $sp(a)^{\#}$, so that a previously known acyclic justification for a may be put into question.

Having collected all non-false atoms that possibly are unfounded, the loop in Line 8–23 tries to re-establish acyclic justifications for the atoms in Do, starting from one atom a at a time and filling a potential unfounded set U. In Line 11, we determine all non-false weight constraints whose lower bounds can be reached via non-false literals outside U and that thus may be usable to justify an atom in U. Conversely, if no such weight constraint exists, we have identified a nonempty unfounded set U and return it in Line 12. The propagation routine [8] of *clasp* will then take care of falsifying all atoms in U before the next call to UNFOUNDEDSET. If U is not (yet) unfounded, in Line 13, we pick an arbitrary weight constraint $\mathcal{W}$ whose bound can be reached without using U; and in Line 14, we determine all possibly unfounded atoms in $\mathcal{W}$ from the same (non-trivial) SCC as the initial atom a. The fact that only such atoms may be used to extend U in Line 22 exhibits the localization of unfounded set checks to SCCs. However, we only extend U if the addition makes the sum of non-false literals' weights from outside U drop below the lower bound of $\mathcal{W}$, as checked in Line 15. If the latter is not the case, we are sure that some atoms in U have an acyclic justification via $\mathcal{W}$, and such atoms cannot belong to an unfounded set. Furthermore, the atoms in $A(\mathcal{W}^{+})$ from the SCC of a that are already acyclicly justified can be memorized in $\mathcal{W}^{\#}$ (Line 16–17). As long as the justifications or the source pointers, respectively, of these atoms do not change, this helps to avoid further (unsuccessful) unfounded set checks (cf. the condition in Line 6) for the atoms in U justified via $\mathcal{W}$ in Line 19. Finally, the acyclicly justified atoms are removed from the unfounded set U to be computed as well as from the scope Do of unfounded set checks (Line 20–21).

Notably, source pointers enable lazy unfounded set checking, performed only in reaction to changes in assignment $\mathbf{A}$. Beyond that, a second major benefit is backtrack-freeness. In fact, source pointers are still valid after backtracking, even though they might be set differently than in the state when $\mathbf{A}$ has previously been extended. However, only the existence of some acyclic justification for every non-false atom is important, while it is unnecessary to pick or reconstruct a specific one. The implementation of source pointers in *clasp*, described in the next section, follows this principle and does not reset source pointers upon backtracking (or backjumping, respectively).

4 Implementation of Weight Constraints in *clasp*

This section is dedicated to the implementation of weight constraints within the conflict-driven ASP solver *clasp*. Note that in *clasp* normal rules are *not* handled as described above. Instead, unit propagation on normal rules is applied by means of the more efficient, backtrack-free *Two-Watched-Literals* algorithm [11]. However, the dedicated treatment of weight constraints enables *clasp* to handle them natively, without relying on any transformation (cf. the comparison in Section 5).

Unit Propagation. The number of nogoods for a weight constraint $\mathcal{W}$ is in general exponential in the size of $A(\mathcal{W})$. Hence, it is impractical to explicitly construct $\omega(\mathcal{W})$ and $\varpi(\mathcal{W})$. Rather, our idea is to take advantage of Proposition 1 and to capture $\omega(\mathcal{W})$ and $\varpi(\mathcal{W})$ by two corresponding linear Pseudo-Boolean (PB) constraints (cf. [12]) that must be satisfied by any solution. In terms of weight constraint notation, we have

$$(PB_\omega) \quad w' \ \{\mathcal{W} = w', \sim A(p) = W(p), \ A(n) = W(n) \mid p \in \mathcal{W}^+, n \in \mathcal{W}^-\}$$
$$(PB_\varpi) \quad lb(\mathcal{W}) \ \{\sim\mathcal{W} = lb(\mathcal{W}), \ \ell \mid \ell \in \mathcal{W}^+ \cup \mathcal{W}^-\}$$

where $w' = (\Sigma[\mathcal{W}^+ \cup \mathcal{W}^-] - lb(\mathcal{W})) + 1$. The first PB constraint (PB_ω) is obtained from Proposition 1.1; it is satisfied by an assignment iff all nogoods in $\omega(\mathcal{W})$ are satisfied. The same holds for (PB_ϖ), obtained from Proposition 1.2, and nogoods in $\varpi(\mathcal{W})$. Note that $\mathcal{W}$ can be assigned to false, while (PB_ω) and (PB_ϖ) must always be satisfied.

For a PB constraint $\mathcal{W}$ (a true weight constraint) and an assignment $\mathbf{A}$, let $\mathcal{T}, \mathcal{U}, \mathcal{F}$ denote the literals of $\mathcal{W}^+ \cup \mathcal{W}^-$ being true, unassigned, and false in $\mathbf{A}$. Then, $\mathcal{W}$ is unit when $\Sigma[\mathcal{W}^+ \cup \mathcal{W}^-] - \Sigma[\mathcal{F}] < lb(\mathcal{W}) + W(\ell)$ for $W(\ell) = \max\{W(\ell') \mid \ell' \in \mathcal{U}\}$ and $\ell \in \mathcal{U}$. In this case, $\mathcal{W}$ implies ℓ, and the implying assignment is $\mathcal{F}$. Unit propagation for PB constraints can be implemented using the following procedure:

1. Initialize a counter $S_\mathcal{W}$ to $\Sigma[\mathcal{W}^+ \cup \mathcal{W}^-]$.
2. Whenever a literal ℓ in $\mathcal{W}^+ \cup \mathcal{W}^-$ becomes false, set $S_\mathcal{W}$ to $S_\mathcal{W} - W(\ell)$.
3. If $S_\mathcal{W} < lb(\mathcal{W}) + \max\{W(\ell') \mid \ell' \in \mathcal{U}\}$, set each literal $\ell \in \mathcal{U}$ to true whose weight $W(\ell)$ satisfies the condition $W(\ell) > S_\mathcal{W} - lb(\mathcal{W})$.

The *clasp* implementation allows for arbitrary *Boolean constraints* through an abstraction similar to the one in [13]. Each concrete constraint type must implement functions for propagation and calculation of reasons. Also, functions for simplifying the constraint and for updating the constraint on backtracking can be specified but are not mandatory. Another important abstraction used in *clasp* is that of a *watch list*. For each literal l, a list is maintained storing constraints that need to be updated when l becomes true. Each individual entry in a watch list stores (a reference to) a constraint and an integer. A constraint can use the integer, passed as an argument to its propagate function, to associate data with the watched literal, e.g., the literal's position in the constraint.

Based on these abstractions, we implemented a constraint type `WC`, combining unit propagation on (PB_ω) and (PB_ϖ) for a weight constraint $\mathcal{W}$. Observe that one of the two PB constraints is obsolete once $\mathcal{W}$ is assigned. Also, the literals of (PB_ω) and (PB_ϖ) differ only in their signs. Their weights are identical, as one can simply set the weight of $\mathcal{W}$ to $\max\{w', lb(\mathcal{W})\}$ in both constraints without affecting satisfiability.

In the following algorithms, we use symbols **true** and **false** to refer to assigned truth values. In addition to primitive types like `int`, we use the following abstract data types:

`Lit` The type of (signed Boolean) literals. A literal instance has three fields: a variable index, a sign flag, and a watched flag. The variable index stores the underlying variable of a literal. The sign flag indicates whether the variable is negated. The `operator ¬()` returns the complement of a literal l and, given an integer i, the expression $l * i$ returns $\neg l$ if $i < 0$ and l otherwise.

`Vec<T>` A dynamic array of type `T`. Given a `Vec<T> v` of size n, the element at position $1 \leq \mathtt{i} \leq n$ is accessed via `v[i]`.

Algorithm 2: `WC::propagate(Lit p, int wd, Solver s)`

```
   Input : A watched literal that became true, the data
           associated with the watch, and a solver object.
 1 int ac = sign(wd)                          /* get affected constraint and */
 2 Lit W = lits[1]*ac                         /* associated constraint literal */
 3 if s.isTrue(W) || active+ac == 0 then
 4     return NO CONFLICT                     /* constraint is satisfied or other is active */
 5 int idx = abs(wd)                                          /* index of ¬p */
 6 C(ac) = C(ac) - weight(idx)
 7 lits[idx].watched = false                                  /* mark as processed */
 8 trail.push(wd)                             /* remember for backtracking */
 9 while umax ≤ lits.size() && weight(umax) > C(ac) do
10     if lits[umax].watched then
11         active = ac                                      /* mark constraint as active */
12         trail.push(umax*ac)
13         Lit x = lits[umax]*ac
14         if not s.force(x,this) then
15             return CONFLICT
16     ++umax
17 return NO CONFLICT
```

The type `WC` has the following fields:

`lits` Stores the literals of (PB_ϖ) ordered by decreasing weight. The weight of $\neg\mathcal{W}$ is set to $\max\{w', lb(\mathcal{W})\}$, and hence `lits[1]` stores $\neg\mathcal{W}$.[1] The literals of (PB_ω) are accessed by multiplying the literals in `lits` with `-1`.

`active` An integer denoting whether both (PB_ω) and (PB_ϖ) are relevant (`0`), only (PB_ω) is relevant (`-1`), or only (PB_ϖ) is relevant (`1`) under the current assignment.

`C`$_\omega$ A counter initialized to $\Sigma[\texttt{lits}] - lb(PB_\omega)$.

`C`$_\varpi$ A counter initialized to $\Sigma[\texttt{lits}] - lb(PB_\varpi)$.

`umax` The index of the literal with the greatest weight not yet (known to be) assigned.

`trail` A queue of assigned literals used for backtracking and computing reasons.

Initially, `active` is `0`, `umax` is `1`, and the `trail` is empty. Also, we add watches $(\neg l_i, i)$ and $(l_i, -i)$ for all literals l_i in `lits` and set the watched flags of the literal instances to true. For example, consider $\mathcal{W} = 4\,\{a=3, c=2, \sim d=1, \sim e=3\}$. In this case, `lits` is $[\neg\mathcal{W}=6, a=3, \neg e=3, c=2, \neg d=1]$. Moreover, `C`$_\varpi$ is 11, `C`$_\omega$ is 9, and we add watches $(\mathcal{W}, 1), (\neg a, 2), \ldots, (d, 5)$ and $(\neg\mathcal{W}, -1), (a, -2), \ldots, (\neg d, -5)$.

Algorithm 2 shows the procedure for propagating a weight constraint, triggered when one of the watched literals becomes **true**. Staying with the example, assume that a is set to **true**. Then, Algorithm 2 is called with `p` $= a$ and `wd` $= -2$. From the sign of `wd`, we determine the affected PB constraint, i.e., (PB_ϖ) if `wd` > 0 and (PB_ω) if `wd` < 0. Since $\mathcal{W}$ is not yet assigned and `active` is `0`, (PB_ω) is relevant under

[1] We assume that for all literals ℓ in a weight constraint $\mathcal{W}$, we have $W(\ell) \leq lb(\mathcal{W})$. Weights greater than the lower bound are replaced with the bound in a preprocessing step.

Algorithm 3: `WC::reason(Lit p, Vec<Lit> out)`

Input : `A literal propagated by this constraint.`
Output: `A set of` **true** `literals implying p.`

```
1 foreach int d ∈ trail do
2     if sign(d) == active then
3         int idx = abs(d)
4         Lit x = lits[idx]*active
5         if not lits[idx].watched then out.push(¬x)
6         else if x == p then break
```

the current assignment, and so we decrease $\texttt{C}_\omega$ by 3 (the weight of a) in Line 6. We then set the watched flag of the literal instance to false to indicate that the respective counter was updated. Also, we push `wd` to the `trail` so that we can suitably increase $\texttt{C}_\omega$ again on backtracking and to compute reasons for assignments. Finally, given that $\texttt{lits[umax]} = \neg\mathcal{W}$ and $\texttt{weight(umax)} = 6 = \texttt{C}_\omega$, the while loop in Line 9–16 is skipped. Next, assume that c becomes **false**. Hence, $\texttt{wd} = 4$, and the affected constraint is (PB_ϖ). Since (PB_ϖ) is also not yet unit, no new assignments are derived. Finally, assume that d is assigned to **false**. From $\texttt{wd} = -5$, we again extract (PB_ω) as the affected PB constraint, and after decreasing $\texttt{C}_\omega$ to 5, we have $\texttt{weight(umax)} > \texttt{C}_\omega$. That is, the constraint is now unit so that the while loop is entered in Line 9. The loop considers only literal instances whose watched flags are true, while other literals were already processed. Since `lits[1]` has its watched flag set, the condition in Line 10 is satisfied, (PB_ω) is marked as active, and $\mathcal{W}$ is forced to **true**. Note that `lits[1]` is $\neg\mathcal{W}$, but after multiplying with `-1` (the active constraint), we correctly get $\mathcal{W}$. As mentioned before, `lits` is ordered by decreasing weight. Thus, after `umax` is increased, it points to a (the literal with the next greatest weight to consider), and as $3 \leq 5 = \texttt{C}_\omega$, propagation stops. When Algorithm 2 is then called for $\mathcal{W}$ and $\texttt{wd} = 1$, the condition in Line 3 is true (`active` is `-1` and `ac` is `1`), i.e., nothing needs to be done.

Conflict-Driven Nogood Learning. The CDNL algorithm [8] of *clasp* applies the common First-UIP scheme [5, 6] for resolving conflicts. The procedure starts with a violated nogood δ and resolves literals out of δ until only one literal assigned at the current decision level remains. For this to work, each concrete constraint type must implement a procedure, which, given a literal `p` implied by a constraint of that type, returns a set of (**true**) literals implying `p`. Algorithm 3 shows this procedure for weight constraints.

The idea is to dynamically extract a nogood from either $\omega(\mathcal{W})$ or $\varpi(\mathcal{W})$, depending on the currently active PB constraint. Reconsider the previous example and assume that Algorithm 3 is called with $\texttt{p} = \mathcal{W}$. The constraint's `trail` is $[-2, 4, -5, -1]$, and the active constraint (`-1`) is (PB_ω). Then, element 4 is skipped because it was not added by the active PB constraint (cf. Line 2). For the other elements, we check whether the corresponding literal instances still have their watched flags set. If not, the element corresponds to a literal that is **false** in the active constraint and thus belongs to the implying assignment. Otherwise, the literal is **true** and was forced by the active constraint. We also push such implied literals to the `trail` (cf. Line 12 in Algorithm 2)

because a weight constraint can become unit multiple times, and in that case only the **false** literals assigned earlier are part of the implying assignment. For the example, we add a and $\neg d$ to `out`, but not $\mathcal{W}$, because the watched flag of `lits[1]` is still set. Furthermore, since $\mathtt{p} = \mathcal{W}$, the condition in Line 6 is true, and the extracted nogood is $\{\mathbf{F}\mathcal{W}, \mathbf{T}a, \mathbf{F}d\} \in \omega(\mathcal{W})$. Accordingly, the fact that a is **true** and d is **false** provides a reason for $\mathcal{W}$ being **true**.

When a conflict is resolved and one or more decision levels are removed, constraint types implementing an undo function are notified. The corresponding procedure for weight constraints pops entries corresponding to unassigned literals from the `trail`, again using the sign of a stored integer to determine the affected PB constraint and the watched flag to distinguish a processed from an implied literal. Counters are only increased for the former, and the watched flag is then set back to true to indicate that the corresponding literal contributes again to the respective counter value. If the literal with the greatest weight, viz., $\neg\mathcal{W}$, is unassigned, the constraint can no longer be unit, and hence `active` is set back to `0`. Otherwise, `active` is left unchanged, meaning that the previously active PB constraint stays in effect. Finally, `umax` is set back to the index of the unassigned literal with the greatest weight.

Unfounded Set Checking. A second set of data structures is used for representing the atoms and rule bodies that need to be considered during unfounded set checking (and extraction of loop nogoods). This is motivated by the fact that only the non-trivial SCCs of a program's positive atom dependency graph are relevant during unfounded set checks. For a program Π, *clasp*'s unfounded set checker stores the set $\{a \in A(\Pi) \mid c(a) \neq 0\}$ as `Atom` instances. For an atom a in that set, the corresponding `Atom` instance contains:

`scc` the atom's component number $c(a)$,
`p`$_s$ its set of possible sources $\{B(r) \mid r \in \Pi, A(H(r)) = \{a\}\}$,
`pos` the set of rule bodies $\{B(r) \mid r \in \Pi, a \in B(r)^+, A(H(r)) = \{a'\}, c(a') = c(a)\}$,
`source` (a pointer to) its current source $sp(a) \in \mathtt{p}_s$, and
`vs` a flag indicating whether `source` is currently valid. Initially, `vs` is set to false and a is added to Do (cf. Algorithm 1).

The set of (distinct) weight constraints $\{B(r) \mid r \in \Pi, A(H(r)) = \{a\}, c(a) \neq 0\}$ is represented by instances of type `Body`. For a weight constraint $\mathcal{W}$ in that set, the corresponding `Body` instance stores:

`scc` the body's component number, $c(\mathcal{W})$, that is set to $c(a)$ if there is some $r \in \Pi$ such that $A(H(r)) = \{a\}$, $B(r) = \mathcal{W}$, and $\{b \in A(\mathcal{W}^+) \mid c(b) = c(a) \neq 0\} \neq \emptyset$, or to 0 otherwise.[2]
`extern` its "external" literals $\{p \in \mathcal{W}^+, n \in \mathcal{W}^- \mid c(A(p)) = 0 \text{ or } c(A(p)) \neq c(\mathcal{W})\}$,
`intern` its "internal" literals $\{p \in \mathcal{W}^+ \mid c(A(p)) = c(\mathcal{W}) \neq 0\}$,
`heads` its "heads" $\{a \mid r \in \Pi, A(H(r)) = \{a\}, B(r) = \mathcal{W}, c(a) \neq 0\}$, and
`C` a counter initialized to $lb(\mathcal{W}) - \Sigma[\mathtt{extern}]$.

[2] Note that $c(\mathcal{W})$ is unique. If $r_1, r_2 \in \Pi$ with $B(r_1) = B(r_2) = \mathcal{W}$, $A(H(r_1)) = \{a_1\}$, $A(H(r_2)) = \{a_2\}$, $c(a_1) = c(b_1) \neq 0$, $c(a_2) = c(b_2) \neq 0$ for $b_1, b_2 \in \mathcal{W}^+$, then $c(a_1) = c(a_2)$.

Algorithm 4: `findSource(Atom a)`

```
1  Set<Atom> T = {a}, U = ∅
2  while T\U != ∅ do let Atom a ∈ T\U in
3      U = U ∪ {a}
4      foreach Body B ∈ a.p_s do
5          if B ∈ A^F then continue
6          else if B.scc != a.scc || B.C ≤ 0 || B.update() then
7              a.source = B
8              Set<Atom> S = {a}
9              while S != ∅ do let Atom a ∈ S in
10                 S = S\{a}, T = T\{a}, U = U\{a}, Do = Do\{a}
11                 foreach Body B ∈ a.pos do B.atomSourced(a,S)
12             break
13         else B.addUnsourced(T)
14 return U
```

Again, we use the watched flags of the literal instances in `B.extern` and `B.intern`, for an instance `B` of `Body`, to distinguish the literals that currently contribute to the value of `B.C` from the rest. That is, initially all literals in `B.extern` have their watched flags set to true, while those in `B.intern` have them set to false. Logically, the literals in `B.intern` whose watched flags are true correspond to the atoms in $\mathtt{B}^{\#}$. Furthermore, `B` is a valid source for an atom `a` in `B.heads` if `B` is not **false** and $\mathtt{B.C} \leq 0$ or $\mathtt{B.scc} \neq \mathtt{a.scc}$. In order to efficiently detect when one of the first two conditions is violated, we use watches for `B` as well as literals in `B.extern` and `B.intern`. During unit propagation, if a literal l in `B.extern` or `B.intern` whose watched flag is set becomes **false**, `B.C` is increased by `weight(`l`)`, and l's watched flag is set to false. In addition, invalidated sources are accumulated. Note that `B.C` is not updated during backtracking, but only during unfounded set propagation (see below).

Once unfounded set propagation begins, invalidated sources are used to initialize Add (cf. Algorithm 1). That is, we add all (non-**false**) atoms to Add whose sources are invalid. If `a.vs` is true for an atom `a` included in Add, we set it to false and propagate the removal of the source pointer by notifying all bodies in `a.pos`. Each affected body `B` then checks whether `a` currently belongs to $\mathtt{B}^{\#}$, i.e., whether `a` in `B.intern` has its watched flag set. In this case, the watched flag is set to false, and `B.C` is increased accordingly. Since this may invalidate `B`, the whole process is repeated until no more atoms are added to Add (and Do).

Following the idea of Algorithm 1, we then try to re-establish acyclic justifications for the atoms in Do, where Line 9–23 of Algorithm 1 are implemented as in Algorithm 4. The abstract data type `Set<Atom>` refers to the mathematical concept of a set (of `Atom` instances) along with operations on them. The atoms in `T` are considered in turn. That is, in each iteration of the loop starting in Line 2, one atom `a` is selected and added to a set `U`. Then, all non-**false** bodies `B` in $\mathtt{a.p}_s$ are inspected. At this point, `B` is updated only if it is currently not a valid source for `a`. That is, if $\mathtt{B.scc} = \mathtt{a.scc}$ and $\mathtt{B.C} > 0$, `B.update()` checks for literals in `B.extern` and `B.intern` that are

neither **false** nor have their watched flags set. If such a literal is found in `B.extern`, its watch flag is set and `B.C` is decreased by the literal's weight. For such a literal in `B.intern`, the same is done only if the corresponding atom currently has a valid source pointer. If even after updating `B` is not a valid source, `T` is extended with non-**false** atoms in `B.intern` lacking a valid source (Line 13). This is similar to Line 22 of Algorithm 1. In particular, since at this point `B.scc` is always equal to `a.scc`, the same localization to SCCs is achieved. On the other hand, if `B` is a valid source, it is used as new source for `a`, and the new source pointer is propagated by notifying all bodies in `a.pos`. Each affected body `B` that is currently not a valid source checks whether adding `a` to $\texttt{B}^{\#}$ turns it into a valid source. If so, non-**false** atoms in `B.heads` currently lacking a source are added to `S`. Thus, source pointer propagation is iterated until `S` is finally empty. Furthermore, atoms for which a new source pointer has been set are removed from `T`, `U`, and Do. Note that both during updates and source pointer propagation, $\texttt{B}^{\#}$ is extended only if `B` is not (yet) a valid source. This way, it is guaranteed that atoms added to $\texttt{B}^{\#}$ are acyclicly justified independently of `B`. Finally, once $\texttt{T} \setminus \texttt{U}$ is empty, all potential sources were inspected. Any remaining atoms in `U` are unfounded wrt the current assignment and are returned in Line 14.

Note that, for bodies of normal rules and weight constraints $\mathcal{W}$ with $c(\mathcal{W}) = 0$, the set of "external" literals is not relevant during unfounded set checking, and only the truth values of such bodies or weight constraints, respectively, are considered. Also, for bodies of normal rules, no (additional) watches are needed for body literals because unit propagation already falsifies such a body whenever one of its literals becomes **false**.

5 Experiments

We implemented our approach within the ASP solver *clasp* (1.2.0). Our experiments consider *clasp* (in its default configuration) using three different ways of treating weight constraints: (a) standard setting, using the described approach; (b) with (quadratic) transformation of weight constraints (cf. [7]); (c) with selective transformation of weight constraints. Variant (c) applies strategy (b) to weight constraints with lower bound 1 and whenever the number of resulting nogoods is smaller than 16, otherwise it applies strategy (a). We also consider *smodels* (2.33 with option `-restart`[3]) and *cmodels* (3.78) because of their distinct treatment of weight constraints. The full experiments, additionally including *pbmodels*, *smodelscc*, as well as *smodels* without lookahead, are given at [14] (see also below). We conducted experiments on a variety of benchmarks taken from the *SLparse* category of the first ASP system competition.[4] Among them, *BlockedNQueens*, *BoundedSpanningTree*, and *SocialGolfer* comprise choice and cardinality rules, while *TravelingSalesperson*, *WeightedLatinSquare*, and *WeightedSpanningTree* contain also weight rules. In addition, we consider a hand-crafted benchmark, *ExtHamPath*, possessing non-trivial unfounded sets due to recursive cardinality constraints. Each of the benchmark sets consists of five instances.[5]

[3] This variant of *smodels* performed best on our benchmarks.

[4] `http://asparagus.cs.uni-potsdam.de/contest`

[5] All benchmarks are available at [14].

Benchmark	*clasp* (a)	*clasp* (b)	*clasp* (c)	*cmodels*	*smodels*
BlockedNQueens	34.01(0)	70.11(0)	49.70(0)	570.20(0)	363.42(1)
BoundedSpanningTree	8.12(0)	9.94(0)	8.16(0)	19.53(0)	1381.98(4)
SocialGolfer	601.79(3)	606.13(3)	604.23(3)	1035.77(5)	1802.65(9)
TravelingSalesperson	2.17(0)	140.02(0)	1.40(0)	2195.75(6)	21.63(0)
WeightedLatinSquare	0.06(0)	0.59(0)	0.10(0)	1.19(0)	1700.90(2)
WeightedSpanningTree	5.08(0)	5.93(0)	5.67(0)	11.40(0)	953.97(2)
ExtHamPath	10.64(0)	84.72(0)	18.25(0)	28.67(0)	2223.28(6)
$\Sigma(\Sigma)$	661.87(3)	917.44(3)	687.51(3)	3862.51(11)	8649.47(24)

Table 1. Benchmark results on a 3.4GHz PC under Linux, each run restricted to 600s and 1GB.

Table 1 summarizes our results by giving the sum of runtimes obtained on the five instances in each benchmark set; each instance is measured by taking the average over three shuffles obtained with ASP tools from TU Helsinki.[6] A timeout is accounted for by the maximum time of 600s, and timeouts are also indicated in parentheses. We mention that *pbmodels* with *minisat+* and *satzoo* yields 6925.55(32) and 9954.02(36) in total, respectively; *smodels* without lookahead takes 13141.25(61).

Looking at *BlockedNQueens* and *TravelingSalesperson*, we observe a drastic effect through a dedicated treatment of cardinality and weight constraints. While instances of the former contain many relatively large cardinality constraints, instances of the latter contain a single weight constraint with 600 literals. In *clasp* (b), this leads to an extension of programs by over 600000 normal rules and more than 300000 auxiliary atoms. As a consequence, both transformation-based approaches, *clasp* (b) and *cmodels*, are outperformed by orders of magnitude by *clasp* (a/c) and *smodels*. Unlike this, *BoundedSpanningTree* and *SocialGolfer* include only a few small to midsize cardinality rules and so produce almost no overhead on transformation-based approaches. The same applies to *WeightedLatinSquare* and *WeightedSpanningTree* as regards the transformation of weight constraints. In contrast to the benchmarks from the *SLparse* category, *ExtHamPath* contains many small yet recursive cardinality rules inducing a large positive dependency graph and many non-trivial unfounded sets. We attribute *smodels*' poor performance on this benchmark to exhaustive lookahead operations. Given that the small size of cardinality constraints puts the remaining approaches on equal footing, the customized unfounded set algorithm in *clasp* (a) shows a decent performance.

Our experiments demonstrate that the combination of conflict-driven learning with a dedicated treatment of weight constraints has an edge over either singular approach. Although the overhead of a dedicated treatment seems disadvantageous on small weight constraints, the hybrid approach of *clasp* (c) does not improve on the overall performance of the fully dedicated one, viz., *clasp* (a).

6 Discussion

We presented a comprehensive approach to integrating weight (and cardinality) rules into conflict-driven ASP solving, utilizing a nogood-based characterization of answer

[6] `http://www.tcs.hut.fi/Software/asptools`

sets to specify (unit) propagation over weight rules. To be precise, we established a one-to-one correspondence between the answer sets of a weight constraint program and the solutions for the nogoods induced by the program. In view of the exponential number of loop nogoods, we developed a dedicated, source-pointer-based unfounded set checking algorithm that computes loop nogoods only on demand, while aiming at lazy unfounded set checking and backtrack-freeness. Similarly, we are faced with an exponential number of nogoods stemming from weight constraints, although language-extending, quadratic representations exist. Unlike this, we advocate a dedicated treatment of weight constraints, akin to the one used in *smodels* yet extended to conflict-driven learning and backjumping. We developed our computational approach from the semantic foundations laid in Section 3. Our design is guided by two Pseudo-Boolean constraints that must be satisfied by any solution. In view of this, Section 4 provided a rather detailed account of the key features of the weight constraint implementation in *clasp*. Our experiments show that our dedicated approach to handling weight constraints is competitive and does not seem to produce significant overhead on benchmarks with only small constraints, putatively favoring transformation techniques.

Acknowledgments. This work was supported by DFG under grant SCHA 550/8-1.

References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
3. Dowling, W., Gallier, J.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. Journal of Logic Programming **1** (1984) 267–284
4. Giunchiglia, E., Lierler, Y., Maratea, M.: Answer set programming based on propositional satisfiability. Journal of Automated Reasoning **36**(4) (2006) 345–377
5. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5) (1999) 506–521
6. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In ICCAD'01, (2001) 279–285
7. Ferraris, P., Lifschitz, V.: Weight constraints as nested expressions. Theory and Practice of Logic Programming **5**(1-2) (2005) 45–74
8. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In IJCAI'07, AAAI Press (2007) 386–392
9. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
10. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-driven disjunctive answer set solving. In KR'08, AAAI Press (2008) 422–432
11. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In DAC'01, ACM Press (2001) 530–535
12. Sheini, H., Sakallah, K.: Pueblo: A hybrid Pseudo-Boolean SAT solver. Journal on Satisfiability, Boolean Modeling and Computation **2** (2006) 165–189
13. Eén, N., Sörensson, N.: An extensible SAT-solver. In SAT'03, (2003) 502–518
14. `http://www.cs.uni-potsdam.de/clasp`