Completion of Disjunctive Logic Programs*

Mario Alviano and **Carmine Dodaro**
Department of Mathematics and Computer Science, University of Calabria, Italy
{alviano, dodaro}@mat.unical.it

Abstract

Clark's completion plays an important role in ASP computation: it discards unsupported models via unit resolution; hence, it improves the performance of ASP solvers, and at the same time it simplifies their implementation. In the disjunctive case, however, Clark's completion is usually preceded by another transformation known as shift, whose size is quadratic in general. A different approach is proposed in this paper: Clark's completion is extended to disjunctive programs without the need of intermediate program rewritings such as the shift. As in the non-disjunctive case, the new completion is linear in size, and discards unsupported models via unit resolution. Moreover, an ad-hoc propagator for supported model search is presented.

1 Introduction

Answer Set Programming (ASP) is a well established declarative language for knowledge representation and reasoning [Niemelä, 1999; Marek and Truszczyński, 1999; Lifschitz, 2002; Baral, 2003; Gelfond and Kahl, 2014]. In fact, ASP programs are interpreted according to stable model semantics, where each stable model provides a plausible scenario or solution to the encoded problem. The original definition of stable model, applicable to programs with atomic heads, was soon extended to handle programs with disjunctive heads [Przymusinski, 1991]. In the subsequent years, relevant properties regarding stable model semantics for disjunctive programs were discovered, and discussed in the literature. In particular, broad classes of programs with good computational properties were identified: stable model existence belongs to the first level of the polynomial hierarchy for *head cycle free* [Ben-Eliyahu and Dechter, 1994], one level below than the general case. Even more relevant, the notion of *tightness* was extended to disjunctive programs [Fages, 1994; Erdem and Lifschitz, 2003; Lee and Lifschitz, 2003], proving that stable models actually coincide with *supported* models for a huge class of programs.

*This work was partially supported by MIUR under PON project "Ba2Know (Business Analytics to Know) Service Innovation - LAB", No. PON03PE_00001_1, and by Gruppo Nazionale per il Calcolo Scientifico (GNCS-INdAM).

The notion of *support* is intuitive, and expected to be satisfied by any stable model of the input program: a model is supported if for each true atom there is a rule whose body is true, and whose head is satisfied only by the atom itself. Such a notion can be enforced by a transformation known as *Clark's completion* [Clark, 1977] possibly preceded by another transformation known as *shift* [Ben-Eliyahu and Dechter, 1994], which provide a concrete strategy for computing stable models of tight programs by means of efficient SAT solvers. Moreover, since stable models are expected to be supported models as well, shift, completion and SAT solving are usually embedded in modern ASP solvers in combination with other techniques such as unfounded set inference via source pointers [Simons *et al.*, 2002], and stability check via SAT oracle calls [Leone *et al.*, 1997].

This paper highlights a source of inefficiency in current ASP solvers: the shift of a rule is quadratic in the number of atoms occurring in its head. Actually, this quadratic blow up was underestimated so far because the size of disjunctive heads is often bounded by the input program, which means that the data complexity is not affected. However, recent progresses in the formalization of the input language of GRINGO [Gebser *et al.*, 2015a], a widely used ASP grounder, mentioned *conditional literals*, a construct that essentially allows to specify unbounded disjunctions. As a consequence, the quadratic blow up may occur for fixed non-ground programs.

Minimal Hitting Set computation is a prominent example of a problem that can be naturally encoded in ASP by means of unbounded disjunctions. Given a collection C of subsets of a set S, a hitting set for C is a subset S' of S such that S' contains at least one element from each subset in C. A hitting set S' for C is minimal if there is no $S'' \subset S'$ such that S'' is a hitting set for C. Hitting sets are widely used in artificial intelligence. For example, it is well-known that minimal hitting sets provide a duality relationship between *Minimal Unsatisfiable Subsets* (MUSs) and *Minimal Correction Subsets* (MCSs) [Liffiton and Sakallah, 2008; Ignatiev *et al.*, 2015]. A natural encoding for this problem in the input language of GRINGO is the following:

$$s'(X) : in(X, Y) :- c(Y). \quad (1)$$

where the database contains $c(y)$ if set y belongs to C, and $in(x, y)$ if $x \in S$ belongs to $y \in C$. The output of GRINGO is the propositional program comprising, for all $y \in C$, a rule of the following form: $\bigvee_{x \in y} s'(x) \leftarrow$.

Empirical evidence of the inefficiency of currently implemented algorithms for processing the above program, and any other program containing at least one rule whose head size is not negligible, motivates a more in depth analysis of the problem. In particular, new techniques to handle rules with unbounded heads are required in order to overcome the limits of the shift. As proved in Section 3, simple arguments such as bounding head sizes by means of auxiliary atoms would solve the quadratic blow up of the shift, but would also destroy the link between stable models of the original program and supported models of the rewritten program. Even worse, a single rule with three or more atoms in its head would be sufficient to destroy other desirable computational properties such as tightness and head cycle freeness.

The proposed solution is thus to rethink the completion in the disjunctive case. As in the original completion, the new technique consists in building a CNF whose models are one-to-one with the supported models of the input program. Moreover, and even more important from the computational point of view, all inferences commonly implemented by ASP solvers for supported model search are preserved by the proposed transformation. As an additional contribution, a specific propagator for handling support inferences in disjunctive rules is described and implemented. In the tested instances, the new completion and the propagator provide a sensible performance gain to the solver WASP [Alviano *et al.*, 2014].

2 Background

Let $\mathcal{A}$ be a fixed, countable set of (propositional) *atoms*, including $\bot$. A *literal* ℓ is either an atom p, or its negation $\neg p$. Let $\overline{\ell}$ denote the complement of ℓ, i.e., $\overline{p} := \neg p$, and $\overline{\neg p} := p$, for all $p \in \mathcal{A}$. For a set S of literals, $\overline{S} := \{\overline{\ell} \mid \ell \in S\}$, $S^+ := S \cap \mathcal{A}$, and $S^- := \overline{S} \cap \mathcal{A}$.

A (disjunctive) *rule* r has the following form:

$$p_1 \vee \cdots \vee p_m \leftarrow \ell_1 \wedge \cdots \wedge \ell_n \tag{2}$$

where $m \geq 1$, $n \geq 0$, $p_1, \ldots, p_m$ are distinct atoms, and $\ell_1, \ldots, \ell_n$ are distinct literals. Define $B(r) := \{\ell_1, \ldots, \ell_n\}$, and $H(r) := \{p_1, \ldots, p_m\}$, referred to as the *body* and the *head* of r, respectively. A *program* Π is a set of rules. Let $At(\Pi)$ denote the set of atoms occurring in Π, and $heads(\Pi, p)$ be the set of rules in Π whose head contains p.

A *CNF* Γ is a set of clauses, where each *clause* φ is a set of literals. For $n \geq 0$, and $\ell_0, \ldots, \ell_n$ being literals, formula

$$\ell_0 \leftrightarrow \ell_1 \wedge \cdots \wedge \ell_n \tag{3}$$

is a compact representation of the following clauses: $\{\ell_0\} \cup \{\overline{\ell_i} \mid i \in [1..n]\}$; $\{\overline{\ell_0}, \ell_i\}$, for all $i \in [1..n]$. Similarly,

$$\ell_0 \leftrightarrow \ell_1 \vee \cdots \vee \ell_n \tag{4}$$

is a compact representation of the following clauses: $\{\ell_0, \overline{\ell_i}\}$, for all $i \in [1..n]$; $\{\overline{\ell_0}\} \cup \{\ell_i \mid i \in [1..n]\}$. (For $n = 0$, the connective $\wedge$ or $\vee$ on the right of $\leftrightarrow$ will be specified in order to avoid ambiguities.)

For π being a set, a rule, a program, or a CNF, let $|\pi|$ denote the size of π, i.e., the number of atoms occurring in π, where each occurrence count 1. Note that formula (3) has size $(n + 1) + 2 \cdot n = 3 \cdot n + 1$; the same for (4).

A (partial) *interpretation* is a set I of literals containing $\neg\bot$; I is *consistent* if $I^+ \cap I^- = \emptyset$, and *total* for a program or CNF π if $I^+ \cup I^- = At(\pi)$. Let V be a set of atoms. The restriction of I to V is $I|_V := I \cap (V \cup \overline{V})$. Two sets $\mathbf{I}, \mathbf{J}$ of interpretations are *equivalent* wrt a set V of (visible) atoms, denoted $\mathbf{I} \equiv_V \mathbf{J}$, if both $|\mathbf{I}| = |\mathbf{J}|$, and $\{I|_V \mid I \in \mathbf{I}\} = \{J|_V \mid J \in \mathbf{J}\}$ (where the first condition is important for counting and enumeration of models [Janhunen, 2006]).

Relation $\models$ is defined as follows: for r of the form (2), $I \models r$ if $H(r) \cap I \neq \emptyset$ whenever $B(r) \subseteq I$; for a program Π, $I \models \Pi$ if $I \models r$ for all $r \in \Pi$; for a CNF Γ, $I \models \Gamma$ if $\varphi \cap I \neq \emptyset$ for all $\varphi \in \Gamma$. For π being a program or a CNF, a *model* of π is a total, consistent interpretation I such that $I \models \pi$. A model I is possibly referred to by the set I^+ of its positive literals. Let $Mod(\pi)$ denote the set of models of π. For a program Π, I is *supported* in Π if for all $p \in I$ there is a rule r such that $B(r) \subseteq I$, and $H(r) \cap I = \{p\}$. A *supported model* of Π is a model of Π that is supported in Π. Let $SupMod(\Pi)$ denote the set of supported models of Π.

Example 1. The models of the following program Π_1:

$$a \vee b \vee c \leftarrow \qquad b \leftarrow a \qquad c \leftarrow \neg a$$

are $\{c\}$, $\{a, b\}$, $\{c, b\}$, and $\{a, b, c\}$. The first is the only supported model of Π_1. Consider now the following CNF Γ_1:

$$\begin{array}{lll} a^1 \leftrightarrow \neg b \wedge \neg c & b^2 \leftrightarrow a & a \leftrightarrow a^1 \\ b^1 \leftrightarrow \neg a \wedge \neg c & c^3 \leftrightarrow \neg a & b \leftrightarrow b^1 \vee b^2 \\ c^1 \leftrightarrow \neg a \wedge \neg b & & c \leftrightarrow c^1 \vee c^3 \end{array}$$

Its only model is $\{c, c^1, c^3\}$, and therefore $Mod(\Gamma_1) \equiv_{At(\Pi)} SupMod(\Pi_1)$. ■

A model I of a program Π is *stable* if it satisfies the following *stability condition*: there is no $J^+ \subset I^+$ such that $J \models \Pi^I$, where Π^I is the *reduct* obtained from Π by first removing each rule r such that $B(r)^- \cap I \neq \emptyset$, and then removing all negative literals [Przymusinski, 1991]. Let $StMod(\Pi)$ be the set of stable models of Π. It is well known that $StMod(\Pi) \subseteq SupMod(\Pi)$ for any program Π, while $StMod(\Pi) = SupMod(\Pi)$ does not hold in general. Actually, the fact that $StMod(\Pi) \subseteq SupMod(\Pi)$ can be used in practice by ASP solvers: the stability condition has to be checked only on supported models; moreover, there is a relevant class of program, known as *tight* [Lee and Lifschitz, 2003] and defined below, for which $StMod(\Pi) = SupMod(\Pi)$ holds.

The *(positive) dependency graph* of Π, denoted $\mathcal{G}_\Pi$, has vertices $At(\Pi)$, and arcs from any $p \in H(r)$ to any $q \in B(r)^+$, for all $r \in \Pi$. Π is tight if all (strongly connected) components of $\mathcal{G}_\Pi$ are singleton. A broader class is referred in the literature as *head cycle free (HCF)* programs [Ben-Eliyahu and Dechter, 1994], defined next, and particularly relevant for computational complexity: checking $StMod(\Pi) \neq \emptyset$ is known to be Σ_2^P-complete, but only NP-complete if Π is HCF. Π is HCF if there is no rule $r \in \Pi$ of the form (2) such that p_i, p_j, for some $1 \leq i < j \leq m$, belong to the same component of $\mathcal{G}_\Pi$.

Example 2. Program Π_1 from Example 1 is tight, and therefore $\{c\}$ is its only stable model. Indeed, $\emptyset$ is not a model of $\Pi_1^{\{c\}} = \{a \vee b \vee c \leftarrow, c \leftarrow\}$. ■

Model search usually takes advantage of inference rules that iteratively add *inferred literals* to an interpretation I. Given a CNF Γ, if there is a clause $\varphi \in \Gamma$ such that $\varphi \setminus \overline{I} = \{\ell\}$, then ℓ is inferred by *unit resolution*. Given a program Π, if there is a rule $r \in \Pi$ such that both $B(r) \subseteq I$, and $H(r) \setminus I^- = \{p\}$, then p is inferred by *forward chain*; if there is a rule $r \in \Pi$ such that both $H(r) \subseteq I^-$, and $B(r) \setminus I = \{\ell\}$, then $\overline{\ell}$ is inferred by *backward chain*.

Supported model search for a program Π can use more inference rules. Let $psup(\Pi, p, I)$ denote the set of *possible supports* for p, i.e.,

$$\{r \in heads(\Pi, p) \mid H(r) \setminus I \subseteq \{p\}, B(r) \cap \overline{I} = \emptyset\}. \quad (5)$$

If there is an atom $p \in At(\Pi)$ such that $psup(\Pi, p, I) = \emptyset$, then $\neg p$ is inferred by *lack of support*. If there is an atom $p \in I^+$ such that $psup(\Pi, p, I) = \{r\}$, then literals in $B(r) \cup \overline{H(r) \setminus \{p\}}$ are inferred by *last support*.

Let $unit(\Gamma, I)$ be the set of literals inferred by unit resolution on Γ and I; let $unit^*(\Gamma, I)$ be the limit of $I_0 := I$, $I_{i+1} := I_i \cup unit(\Gamma, I_i)$, for all $i \geq 0$. Let $sinf(\Pi, I)$ be the set of literals inferred by all inference rules on Π and I, i.e., forward and backward inference, and last and lack of support; let $sinf^*(\Pi, I)$ be the limit of $I_0 := I$, $I_{i+1} := I_i \cup sinf(\Pi, I_i)$, for all $i \geq 0$. A CNF Γ is *inference preserving* wrt a program Π, denoted $\Gamma \sqsupseteq^{inf} \Pi$, if for all consistent interpretations I such that $I \subseteq At(\Pi) \cup \overline{At(\Pi)}$ the following conditions are satisfied: Let $I' := sinf^*(\Pi, I)$, and $I'' := unit^*(\Gamma, I)$; (i) $I' \subseteq I''$; (ii) if $(I')^+ \cup (I')^- = At(\Pi)$, then $(I'')^+ \cup (I'')^- = At(\Gamma)$. Intuitively, (i) guarantees that all inferences on Π are covered by unit resolution on Γ, and (ii) ensures that truth values of atoms in Γ are implied by truth values of atoms in Π.

Example 3. Γ_1 and Π_1 from Example 1 satisfy $\Gamma_1 \sqsupseteq^{inf} \Pi_1$. For example, for $I = \{\neg a\}$, I' is $\{\neg a, c, \neg b\}$ (c is inferred by forward chain, and $\neg b$ by lack of support). In this case, I'' is $\{\neg a, \neg a^1, c^3, c, \neg b^1, \neg b^2, \neg b, c^1\}$, and condition (i) is satisfied. Also note that I' and I'' satisfy condition (ii): For $I = \{a\}$, I' is inconsistent (b is inferred by forward chain, and then $\neg a$ by lack of support); in this case, I'' is inconsistent as well (and contains b, and $\neg a$). ■

Stable model search is usually implemented by combining (supported) model search with other techniques widely acknowledged in ASP computation, i.e., unfounded set inference via source pointers [Simons *et al.*, 2002], and stability check via SAT oracle calls [Leone *et al.*, 1997]. These techniques are out of the scope of the paper. The next section is thus mainly focused on the computation of supported models.

3 Computation

Supported models of programs without disjunction can be computed by means of the so-called *Clark's completion* [Clark, 1977]. Let Π be a program without disjunction. The completion of Π, denoted $comp(\Pi)$, is the set of clauses

$$p_1^r \leftrightarrow \ell_1 \wedge \cdots \wedge \ell_n \quad (6)$$

for all $r \in \Pi$ of the form (2) with $m = 1$, where p_1^r is a fresh atom (true if and only if r is a support of p_1), together with

$$p \leftrightarrow \bigvee_{r \in heads(\Pi, p)} p^r \quad (7)$$

for all $p \in \mathcal{A}$. Moreover, two other strengths of completion are that inferences are preserved, and the construction is linear in size.

Proposition 1. *If Π is a program without disjunction, then:*

1. $Mod(comp(\Pi)) \equiv_{At(\Pi)} SupMod(\Pi)$*;*
2. $comp(\Pi) \sqsupseteq^{inf} \Pi$*;*
3. $|comp(\Pi)| \in O(|\Pi|)$.

Example 4. Let Π_2 be the following program:

$$\begin{array}{ll} a \leftarrow \neg b \wedge \neg c & b \leftarrow a \\ b \leftarrow \neg a \wedge \neg c & c \leftarrow \neg a \\ c \leftarrow \neg a \wedge \neg b & \end{array}$$

Its only supported model is $\{c\}$, and its completion $comp(\Pi_2)$ is Γ_1 from Example 1, whose only model is $\{c, c^1, c^3\}$. As observed in Example 3, $\Gamma_1 \sqsupseteq^{inf} \Pi_2$. ■

In order to apply completion to programs in general, a transformation known as *shift* is first applied to the input program Π, so to obtain a program $shift(\Pi)$ with the same supported models. Formally, let $shift(\Pi)$ be the program comprising rules

$$p_i \leftarrow l_1 \wedge \cdots \wedge l_n \wedge \bigwedge_{j \in [1..n], j \neq i} \neg p_j \qquad \forall i \in [1..m] \quad (8)$$

for all $r \in \Pi$ of the form (2). The strength of the shift is to preserve both supported models and inferences. On the other hand, the construction is not linear, but quadratic in size.

Proposition 2. *If Π is a program, then:*

1. $SupMod(shift(\Pi)) \equiv_{\mathcal{A}} SupMod(\Pi)$*, and therefore* $Mod(comp(shift(\Pi))) \equiv_{At(\Pi)} SupMod(\Pi)$*;*
2. $comp(shift(\Pi)) \sqsupseteq^{inf} \Pi$*;*
3. $|shift(\Pi)| \in O(|\Pi|^2)$*, so* $|comp(shift(\Pi))| \in O(|\Pi|^2)$*;*
4. $comp(\Pi) = comp(shift(\Pi))$ *if Π is disjunction-free.*

Example 5. Program Π_2 from Example 4 is $shift(\Pi_1)$. ■

Point 3 of Proposition 2 highlights a weakness of shift. Hence, an interesting question is whether a different strategy to avoid the quadratic blow up of the shift can be obtained when the computational task is to compute stable models, not just supported models. Actually, it is sufficient to add auxiliary atoms in order to rewrite the input program Π into a new program Π' whose head size is bound to 2. However, since such auxiliary atoms are essentially Tseitin's variables, implications in both directions have to be added to the program, thus introducing dependencies that are likely to deteriorate the performance of a solver. In more detail, auxiliary atoms $t_1^r, \ldots, t_m^r$ will be used to compactly represent subformulas in rules of the form (2). The *normalization* of Π, denoted $norm(\Pi)$, is the program comprising the following rules:

$$t_1^r \leftarrow \ell_1 \wedge \cdots \wedge \ell_n \quad (9)$$

$$t_i^r \leftarrow p_i \qquad \forall i \in [2..m] \quad (10)$$

$$t_i^r \leftarrow t_{i+1}^r \qquad \forall i \in [2..m-1] \quad (11)$$

$$p_i \vee t_{i+1}^r \leftarrow t_i^r \qquad \forall i \in [1..m-1] \quad (12)$$

$$p_m \leftarrow t_m^r \quad (13)$$

for all $r \in \Pi$ of the form (2), where $t_1^r, \ldots, t_m^r$ are fresh atoms.

Theorem 1. *If Π is a program, then:*

1. *$StMod(norm(\Pi)) \equiv_{At(\Pi)} StMod(\Pi)$;*
2. *$|norm(\Pi)| \in O(|\Pi|)$;*
3. *$SupMod(norm(\Pi)) \not\equiv_{At(\Pi)} SupMod(\Pi)$, and $comp(shift(norm(\Pi))) \not\sqsupseteq^{inf} \Pi$ are possible;*
4. *if Π contains some rule r such that $|H(r)| \geq 3$, then $norm(\Pi)$ is not HCF.*

Example 6. The normalization $norm(\Pi_1)$ of program Π_1 from Example 1 is the following program:

$$\begin{array}{lll} t_1^1 \leftarrow & a \vee t_2^1 \leftarrow t_1^1 & b \leftarrow a \\ t_2^1 \leftarrow b & b \vee t_3^1 \leftarrow t_2^1 & c \leftarrow \neg a \\ t_2^1 \leftarrow t_3^1 & c \leftarrow t_3^1 & \\ t_3^1 \leftarrow c & & \end{array}$$

Its only stable model is $\{c, t_1^1, t_1^3, t_1^2\}$, and therefore $StMod(norm(\Pi_1)) \equiv_{At(\Pi_1)} StMod(\Pi_1)$. On the other hand, $\{t_1^1, t_2^1, b\}$ is a supported model of $norm(\Pi_1)$, while $\{b\}$ is not a supported model of Π_1. Also note that $norm(\Pi_1)$ is not HCF, as b, t_2^1, t_3^1 belong to the same component of $\mathcal{G}_{norm(\Pi_1)}$. ■

Points 3–4 of Theorem 1 suggest that the weakness highlighted by point 3 of Proposition 2 may be better circumvented by directly extending completion to the disjunctive case. Hence, auxiliary atoms p_i^r will be used with the same meaning of the disjunction-free case, i.e., rule r of the form (2) supports atom p_i, for $i \in [1..m]$. However, since m may be greater than 1, other atoms occurring in the head of r have to be taken into account. Additional auxiliary atoms will be thus used, and in particular: s_i^r, true if and only if rule r may support p_i, for $i \in [1..m]$; d_i^r, true if and only if the disjunction $p_i \vee \cdots \vee p_m$ is true, for $i \in [2..m]$.

The completion of a (disjunctive) program Π, denoted $comp^{\vee}(\Pi)$, is the set of clauses

$$d_i^r \leftrightarrow p_i \vee d_{i+1}^r \qquad \forall i \in [2..m-1] \qquad (14)$$
$$d_m^r \leftrightarrow p_m \qquad \text{if } m \geq 2 \qquad (15)$$
$$s_1^r \leftrightarrow \ell_1 \wedge \cdots \wedge \ell_n \qquad (16)$$
$$s_i^r \leftrightarrow s_{i-1}^r \wedge \neg p_{i-1} \qquad \forall i \in [2..m] \qquad (17)$$
$$p_i^r \leftrightarrow s_i^r \wedge \neg d_{i+1}^r \qquad \forall i \in [1..m-1] \qquad (18)$$
$$p_m^r \leftrightarrow s_m^r \qquad (19)$$

for all $r \in \Pi$ of the form (2), together with (7) for all $p \in At(\Pi)$. Note that (15) defines d_m^r as an alias of p_m. Similarly, (19) defines s_m^r as an alias of p_m^r. It turns out that d_m^r and s_m^r could be simplified in the above construction, but they are left to ease the reading. Even more important, note that for $m = 1$ the above equations essentially give (6): only (16) and (19) are used in this case, and (16) is precisely (6) if s_1^r is replaced by its alias p_1^r.

Theorem 2. *If Π is a program, then:*

1. *$Mod(comp^{\vee}(\Pi)) \equiv_{At(\Pi)} SupMod(\Pi)$;*
2. *$comp^{\vee}(\Pi) \sqsupseteq^{inf} \Pi$;*
3. *$|comp^{\vee}(\Pi)| \in O(|\Pi|)$;*
4. *$comp(\Pi) = comp^{\vee}(\Pi)$ if Π is disjunction-free.*

Example 7. The completion of program Π_1 from Example 1 is the following CNF $comp^{\vee}(\Pi_1)$:

$$\begin{array}{lll} d_2^1 \leftrightarrow b \vee d_3^1 & d_3^1 \leftrightarrow c & \\ \{s_1^1\} & s_2^1 \leftrightarrow s_1^1 \wedge \neg a & s_3^1 \leftrightarrow s_2^1 \wedge \neg b \\ a^1 \leftrightarrow s_1^1 \wedge \neg d_2^1 & b^2 \leftrightarrow a & a \leftrightarrow a^1 \\ b^1 \leftrightarrow s_2^1 \wedge \neg d_3^1 & c^3 \leftrightarrow \neg a & b \leftrightarrow b^1 \vee b^2 \\ c^1 \leftrightarrow s_3^1 & & c \leftrightarrow c^1 \vee c^3 \end{array}$$

Its only model is $\{c, s_1^1, d_3^1, d_2^1, \neg a^1, \neg b^1, \neg a, s_2^1, \neg b^2, \neg b, s_3^1, c^1, c^3\}$, and therefore $Mod(comp^{\vee}(\Pi_1)) = SupMod(\Pi_1)$. As for inferences, for $I = \{\neg a\}$, $sinf^*(\Pi_1, I)$ is $\{\neg a, c, \neg b\}$, and $unit^*(comp^{\vee}(\Pi_1), I)$ is $\{\neg a, \neg a^1, d_2^1, c^3, c, d_3^1, \neg b^1, \neg b^2, \neg b, s_1^1, s_2^1, s_3^1, c^1\}$. Conditions (i)–(ii) of inference preserving are satisfied. Moreover, for $I = \{a\}$, $sinf^*(\Pi_1, I)$ and $unit^*(comp^{\vee}(\Pi_1), I)$ are inconsistent. ■

4 Implementation

The completion for disjunctive programs has been implemented in WASP, an open source ASP solver previously applying completion after shifting the input program. In the new version, the input program Π is possibly simplified after its parsing, and its completion $comp^{\vee}(\Pi)$ is computed. If necessary, other data structures are computed as it was done before. This is the case, for example, of non-tight components, for which source pointers are used, and of non-HCF components, for which stability checks are obtained via SAT oracle calls.

ASP solvers may also take advantage of other *propagators*, i.e., efficient data structures that essentially provide a compact representation of a set of clauses. In particular, in WASP, aggregates are handled by *pseudo-Boolean constraints* of the following form:

$$w_1 \cdot \ell_1 + \cdots + w_n \cdot \ell_n \geq k \qquad (20)$$

where $n, w_1, \ldots, w_n, k$ are positive integers, and $\ell_1, \ldots, \ell_n$ are literals. Intuitively, the above constraint compactly represents all clauses of the form $C \subseteq \{\ell_1, \ldots, \ell_n\}$ such that $\sum_{i \in [1..n], \ell_i \notin C} w_i < k$. Efficient data structures are thus employed to extend unit resolution: for all $i \in [1..n]$, ℓ_i is inferred from (20) if $\sum_{j \in [1..n], j \neq i, \overline{\ell_j} \notin I} w_j < k$, i.e., if the sum of the integers associated with true and undefined literals different from ℓ_i is smaller than k. Note that the propagator has to receive notifications only for literals ℓ_j ($j \in [1..n]$) whose complement is added to the current interpretation.

Example 8. Literal c^1 is inferred by the following constraint:

$$2 \cdot \neg s_1^1 + a + b + c + a^1 + b^1 + c^1 \geq 2$$

if the current interpretation is $\{s_1^1, \neg a, \neg a^1, c, \neg b, \neg b^1\}$. ■

A new propagator implementing all the inference rules for supported model search described in Section 2 has been implemented in WASP. The propagator actually combines ad-hoc data structures with a few clauses, and one pseudo-Boolean constraint. The general idea is to use auxiliary atoms $s_1^r, p_1^r, \ldots, p_m^r$ (as in $comp^{\vee}$) for each rule r of the form (2), and to compactly represent the following clauses:

$$\{s_1^r, \neg p_j^r\} \qquad \forall j \in [1..m] \qquad (21)$$
$$\{\neg p_i, \neg p_j^r\} \qquad \forall i \in [1..m], \forall j \in [1..m], i \neq j \qquad (22)$$
$$\{\neg s_1^r, p_i^r\} \cup \{p_j \mid j \in [1..m], j \neq i\} \qquad \forall i \in [1..m] \qquad (23)$$

together with (16). Moreover, clauses of the form (7) are added for each $p \in At(\Pi)$. In more detail, (21)–(22) are handled by storing two vectors, $v_0^r := [p_1, \ldots, p_m, \neg s_1^r]$ and $v_1^r := [p_1^r, \ldots, p_m^r]$; when the i-th literal of vector v_k^r ($k \in [0..1]$) is added to the current interpretation, the complement of each literal in position different from i of v_{1-k}^r is inferred. Clauses from (23), instead, are handled by adding the following pseudo-Boolean constraint:

$$2 \cdot \neg s_1^r + p_1 + \cdots + p_m + p_1^r + \cdots + p_m^r \geq 2. \qquad (24)$$

Indeed, since $\{p_i, \neg p_i^r\}$ belongs to (7), whenever $\neg p_i$ is added to the current interpretation, $\neg p_i^r$ is added as well. Thus, both (23) and (24) satisfy the following conditions: if $p_1, \ldots, p_m$ are false, then $\neg s_1^r$ is inferred; if $p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_m$, for some $i \in [1..m]$, and s_1^r is true, then p_i and p_i^r are inferred; no inference is done in the remaining cases.

Example 9. The disjunctive rule of program Π_1 from Example 1 (i.e., $a \vee b \vee c \leftarrow$) is handled by the following vectors:

$$v_0^1 = [a, b, c, \neg s_1^1] \qquad v_1^1 = [a^1, b^1, c^1]$$

by clause $\{s_1^1\}$, and by the pseudo-Boolean constraint reported in Example 8. The remaining rules are handled as in $comp(\Pi_1)$. Hence, the clauses defining a, b, c, b^2, and c^3 from Example 1 are also added. ■

Transformation $comp^\vee$ has been also implemented in an external tool producing DIMACS output in the style of LP2SAT [Janhunen and Niemelä, 2011]. Finally, the normalization $norm$ introduced in Section 3 has been implemented in an external tool whose input and output conform to the numeric format of GRINGO.

5 Experiment

The impact of the new techniques proposed in this paper was assessed empirically on three benchmarks: (i) instances from ASP Competitions containing choice rules easily replaceable by unbounded disjunctions; (ii) instances of *Minimal Hitting Set* downloaded from http://research.nii.ac.jp/~uno/dualization.html; (iii) single rule

$$p(X) : X = 1..n.$$

i.e., the symbolic version of program $\{\bigvee_{x \in [1..n]} p(x_i) \leftarrow\}$ for increasing values of constant $n \geq 1$. The experiment was run on an Intel Xeon CPU 2.4 GHz with 16 GB of RAM. Time and memory were limited to 600 seconds and 15 GB. The tested solvers are WASP, CLASP 3.1.3 [Gebser *et al.*, 2015b], and GLUCOSE 4.0 [Audemard and Simon, 2009].

Concerning the first benchmark, the tested encodings are *Sokoban*, *Solitaire*, *Visit All* and *Weighted Sequence Problem*. The number of instances solved by WASP rises from 74 to 79 when $shift + comp$ is replaced by $comp^\vee$. The performance achieved by using the propagator is similar, with 78 solved instances. These are good results if compared with CLASP: 80 instances are solved in the allotted time when the original encodings based on choice rules are used (74 when choice rules are replaced by disjunctions). On the other hand, on the tested instances already $norm$ provides a sensible performance gain to $shift + comp$, even not reaching the same performance of $comp^\vee$: WASP solves 77 instances, and CLASP 78.

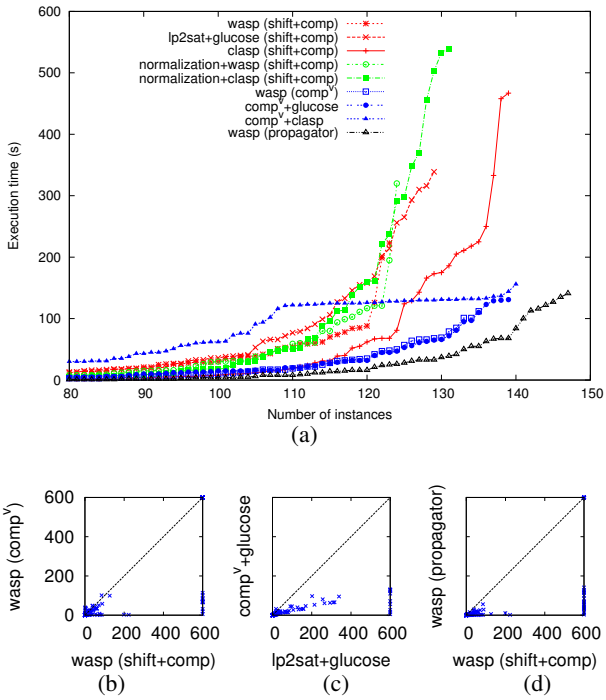

Figure 1: Performance on Minimal Hitting Set.

As for the second benchmark, the tested encoding is program (1). The execution time is plotted in Figure 1(a). It can be observed that the performance of both WASP and CLASP is affected negatively by the transformation $norm$. On the other hand, replacing $shift + comp$ by the new transformation $comp^\vee$ provides a sensible performance gain to WASP, which can be also observed for CLASP on hard instances; for WASP, an instance by instance comparison of $shift + comp$ and $comp^\vee$ is plotted in Figure 1(b), where the majority of the points are below the diagonal, indicating that $comp^\vee$ is faster than $shift + comp$ in the majority of the tested instances. The benefit of adopting $comp^\vee$ is also confirmed by the performance of GLUCOSE, which improves considerably when $comp^\vee$ replaces the $shift + comp$ transformation applied by LP2SAT; an instance by instance comparison is plotted in Figure 1(c), where almost all points are below the diagonal. Finally, the best performance is achieved by WASP with the propagator; Figure 1(d) highlights that the propagator has an advantage over $shift + comp$ in all tested instances.

The last benchmark in the experiment highlights how the quadratic blow up due to $shift$ may kill the computation of ASP solvers already before starting the actual stable model search procedure. Indeed, as it can be observed in Figure 2, the tested solvers cannot process disjunctions of size 30K in the allotted memory when $shift + comp$ is used. On the other hand, the amount of memory required by the other techniques discussed in this paper is linear, as expected.

6 Related Work

Supported model search is a significant part in the most frequent computational task of ASP solving, that is, computing

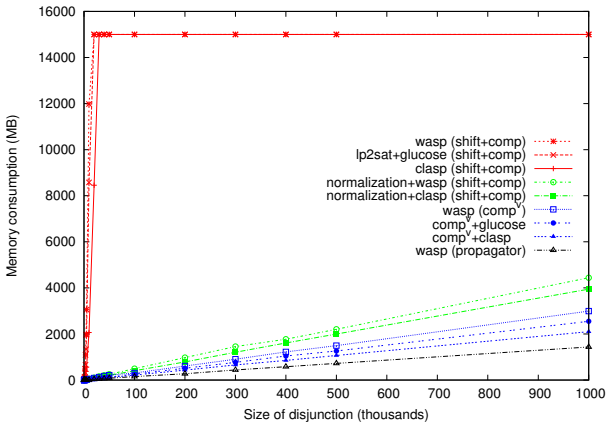

Figure 2: Memory consumption to handle long disjunctions.

stable models of an input program. When the input program contains disjunctive heads, many efficient ASP solvers implement all inference rules of supported model search via unit resolution on the CNF obtained by applying $shift{+}comp$. For example, CLASP 3 [Gebser *et al.*, 2015b], CMODELS 3 [Lierler, 2005], and WASP 2 [Alviano *et al.*, 2015a] implement this strategy. As a consequence, and since unbounded disjunctions were already used in the literature (for example in Encoding 3 of [Abseher *et al.*, 2015]), these excellent ASP solvers are affected by the quadratic blow up highlighted by the experiment reported in Section 5, and can be easily improved by replacing $shift{+}comp$ with the translation $comp^{\vee}$ introduced in this paper.

Technically, CMODELS 3 implements support inferences by means of an extension of Clark's completion that can be directly applied to disjunctive logic programs, and whose output is a set of propositional formulas [Lee and Lifschitz, 2003]. In more detail, given a program Π, CMODELS 3 builds an implication

$$\ell_1 \wedge \cdots \wedge \ell_n \rightarrow p_1 \vee \cdots \vee p_m \tag{25}$$

for each rule $r \in \Pi$ of the form (2), and an implication

$$p \rightarrow \bigvee_{r \in heads(\Pi,p)} \left(\bigwedge_{\ell \in B(r)} \ell \wedge \bigwedge_{q \in H(r) \setminus \{p\}} \neg q \right) \tag{26}$$

for each atom $p \in At(\Pi)$. As a first observation, the above construction is quadratic in size because of (26). Moreover, note that replacing the implication with an equivalence in (26), and rewriting it into an equi-satisfiable CNF by the Tseitin's transformation, would result into $comp(shift(\Pi))$. In the above transformation, instead, the implication is sufficient because of (25), which however has the side effect of introducing more Tseitin's variables during the computation of an equi-satisfiable CNF.

It is an interesting fact that the quadratic blow up in the implementation of support inferences for disjunctive logic programs does not affect DLV [Alviano *et al.*, 2010], and WASP 1 [Alviano *et al.*, 2013], which essentially take advantages of a propagator for this task. However, their propagators have fundamental differences with the propagator presented in Section 4. First of all, DLV does not implement conflict learning, which is instead supported by WASP and by its propagators: in case of conflict, clauses compactly represented by propagators are possibly provided to the learning procedure, which is therefore the same of CDCL-based SAT solvers [Silva and Sakallah, 1996]. Conflict learning is implemented by WASP 1 as well, but with a completely different approach: the learning procedure is properly adapted to handle all inference rules of stable model search, and is therefore difficult to maintain and optimize.

For non-tight programs, support inferences are not sufficient to compute stable models. For these programs, unfounded set inference is often implemented by means of source pointers [Simons *et al.*, 2002], which essentially enforce acyclicity in supporting rules. The implementation of source pointers requires notifications for variations of the possible supports of atoms in non-tight components, which are easy to obtain by observing the truth values of atoms of the form p^r. Moreover, for non-HCF programs, stability of models is checked by means of calls to coNP-oracles [Leone *et al.*, 1997]. In particular, CLASP 3 builds a CNF for checking the absence of unfounded sets in the model [Gebser *et al.*, 2013], which was observed to be quadratic in size and motivated the introduction in WASP 2 of a CNF of linear size for checking minimality of the model for the program reduct [Alviano *et al.*, 2015b]. The completion for disjunctive programs can be combined with these two techniques.

Finally, CLASP 3 implements the so-called *component-wise shift* in case of non-HCF programs [Drescher *et al.*, 2008]: head atoms belonging to the same component are not shifted, and the completion of the resulting program is computed by introducing additional Tseitin's variables for non-atomic heads. When all disjunctive rules contain atoms from the same component, the component-wise shift is not affected by the quadratic blow up of the shift. On the other hand, support inferences are not preserved in general, which is a weakness in comparison to the completion for disjunctive programs proposed in this paper.

7 Conclusion

Support inferences play an important role in stable model search. ASP solvers take advantage of SAT solving to handle this computational task. The CNF built for this purpose is thus a key component to achieve efficiency. Despite that, a quadratic CNF is currently adopted by many ASP solvers. The problem cannot be further underestimated because of a recently formalized construct of GRINGO [Gebser *et al.*, 2015a] that was already used in the literature [Abseher *et al.*, 2015]. Actually, two solutions to this problem are proposed in this paper, one using a CNF of linear size, and another using propagators for an even more efficient memory management.

References

[Abseher *et al.*, 2015] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, and Stefan Woltran. Computing secure sets in graphs using answer set programming. *Journal of Logic and Computation*, 2015.

[Alviano *et al.*, 2010] Mario Alviano, Wolfgang Faber, Nicola Leone, Simona Perri, Gerald Pfeifer, and Giorgio

Terracina. The disjunctive datalog system DLV. In *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*, pages 282–301. Springer, 2010.

[Alviano *et al.*, 2013] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native ASP solver based on constraint learning. In *LPNMR 2013*, volume 8148 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2013.

[Alviano *et al.*, 2014] Mario Alviano, Carmine Dodaro, and Francesco Ricca. Anytime computation of cautious consequences in answer set programming. *TPLP*, 14(4-5):755–770, 2014.

[Alviano *et al.*, 2015a] Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In *LPNMR 2015*, volume 9345 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2015.

[Alviano *et al.*, 2015b] Mario Alviano, Carmine Dodaro, and Francesco Ricca. Reduct-based stability check using literal assumptions. In *ASPOCP 2015*, 2015.

[Audemard and Simon, 2009] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *IJCAI 2009*, pages 399–404, 2009.

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

[Ben-Eliyahu and Dechter, 1994] Rachel Ben-Eliyahu and Rina Dechter. Propositional semantics for disjunctive logic programs. *Ann. Math. Artif. Intell.*, 12(1-2):53–87, 1994.

[Clark, 1977] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322, 1977.

[Drescher *et al.*, 2008] Christian Drescher, Martin Gebser, Torsten Grote, Benjamin Kaufmann, Arne König, Max Ostrowski, and Torsten Schaub. Conflict-driven disjunctive answer set solving. In *KR 2008*, pages 422–432. AAAI Press, 2008.

[Erdem and Lifschitz, 2003] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *TPLP*, 3(4-5):499–518, 2003.

[Fages, 1994] François Fages. Consistency of clark's completion and existence of stable models. *Meth. of Logic in CS*, 1(1):51–60, 1994.

[Gebser *et al.*, 2013] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Advanced conflict-driven disjunctive answer set solving. In *IJCAI*. IJCAI/AAAI, 2013.

[Gebser *et al.*, 2015a] Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz, and Torsten Schaub. Abstract gringo. *TPLP*, 15(4-5):449–463, 2015.

[Gebser *et al.*, 2015b] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Javier Romero, and Torsten Schaub. Progress in clasp series 3. In *LPNMR 2015*, volume 9345 of *Lecture Notes in Computer Science*, pages 368–383. Springer, 2015.

[Gelfond and Kahl, 2014] Michael Gelfond and Yulia Kahl. *Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach*. Cambridge University Press, 2014.

[Ignatiev *et al.*, 2015] Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, and Joao Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. In *CP 2015*, volume 9255 of *Lecture Notes in Computer Science*, pages 173–182. Springer, 2015.

[Janhunen and Niemelä, 2011] Tomi Janhunen and Ilkka Niemelä. Compact translations of non-disjunctive answer set programs to propositional clauses. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*, volume 6565 of *Lecture Notes in Computer Science*, pages 111–130. Springer, 2011.

[Janhunen, 2006] Tomi Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16(1-2):35–86, 2006.

[Lee and Lifschitz, 2003] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In *ICLP 2003*, volume 2916 of *Lecture Notes in Computer Science*, pages 451–465. Springer, 2003.

[Leone *et al.*, 1997] Nicola Leone, Pasquale Rullo, and Francesco Scarcello. Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. *Inf. Comput.*, 135(2):69–112, 1997.

[Lierler, 2005] Yuliya Lierler. Cmodels - SAT-based disjunctive answer set solver. In *LPNMR 2005*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer, 2005.

[Liffiton and Sakallah, 2008] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.

[Lifschitz, 2002] Vladimir Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138(1-2):39–54, 2002.

[Marek and Truszczyński, 1999] Victor W. Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm – A 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

[Niemelä, 1999] Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. Math. Artif. Intell.*, 25(3-4):241–273, 1999.

[Przymusinski, 1991] Teodor C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Comput.*, 9(3/4):401–424, 1991.

[Silva and Sakallah, 1996] João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.