On the Implementation of Weak Constraints in WASP (Preliminary report) [⋆]

Mario Alviano[1], Carmine Dodaro[1], Joao Marques-Silva[2], and Francesco Ricca[1]

[1] Department of Mathematics and Computer Science,
University of Calabria, 87036 Rende, Italy
{alviano,dodaro,ricca}@mat.unical.it
[2] CSI/CASL, University College Dublin
jpms@ucd.ie

Abstract. Optimization problems in Answer Set Programming (ASP) are usually modeled by means of programs with weak constraints. These programs can be handled by algorithms for solving Maximum Satisfiability (MaxSAT) problems, if properly ported to the ASP framework. This paper reports on the implementation of several of these algorithms in the ASP solver WASP, whose empirical analysis highlights pros and cons of different strategies for computing optimal answer sets.

1 Introduction

Answer Set Programming (ASP) [1] is a powerful language for knowledge representation and declarative problem-solving, which has been developed in the field of nonmonotonic reasoning and logic programming. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use a solver to find such a solution [2]. The core language of ASP, which features disjunction in rule heads and nonmonotonic negation in rule bodies, can express all problems in the second level of the polynomial hierarchy [3]. Nonetheless, several extensions to the original language were proposed over the years to further improve ASP modeling capabilities, such as aggregates [4] for concise modeling of properties over sets of data, and weak constraints [5] for modeling optimization problems.

Nowadays, ASP is considered a powerful tool for developing advanced applications because robust implementations are available [6]. ASP applications often demand good performance in hard-to-solve problems, thus the development of more effective and faster ASP systems is a crucial and challenging research topic.

[⋆] This research has been partly supported by the European Commission, European Social Fund of Regione Calabria, the Regione Calabria under project PIA KnowRex POR FESR 2007- 2013 BURC n. 49 s.s. n. 1 16/12/2010, and the Italian Ministry of University and Research under PON project “Ba2Know S.I.-LAB” n. PON03PE_0001.

The improvements obtained in this respect are witnessed by the results of the ASP Competition series (see e.g. [6, 7]).

As a matter of fact, the recent performance boost in ASP solving technologies [8, 9] was obtained by adapting and properly modifying several techniques that were originally introduced for SAT solving and Constraint Satisfaction, like the DPLL backtracking search algorithm [10], *clause learning* [11], *backjumping* [12], *restarts* [13], and *conflict-driven heuristics* [14]. Among the recently-introduced ASP solvers that are based on the above-mentioned techniques is WASP [8].

An aspect that has received less attention in WASP up to now is the implementation of weak constraints. Weak constraints can be handled by properly porting to the ASP framework a number of algorithms [15–17] that were proposed for solving Maximum Satisfiability (MaxSAT) problems.

In this paper we report on our recent experience in the implementation of several algorithms for evaluating weak constraints in WASP. We considered both the model-guided algorithm *mgd* [18], a technique inspired by *optsat* [17] that we call *opt*, and core-guided algorithms *bcd* [15] and *oll* [19].

The paper reports also the results of a preliminary experiment that was conceived to assess the behavior of the implementation of these algorithms in WASP. Instances were taken from both the fourth ASP Competition [7], and the 2008 MaxSAT competition. The empirical analysis highlights pros and cons of the different strategies for computing optimal answer sets. On one hand the better performance of core-guided algorithms in MaxSAT instances is confirmed, on the other hand the model-guided algorithms behave well in specific problems.

It is worth observing that the first attempt of porting core-guided algorithms to ASP was described in [19], in which the algorithm *oll* was proposed. Nonetheless, to the best of our knowledge, no previous attempt to porting *bcd*, *mgd* and *opt* algorithms to the ASP framework is reported in the literature.

2 Preliminaries

Syntax and semantics of propositional ASP programs are briefly introduced in this section. (For complementary introductory material on ASP see [1, 20, 21].)

Syntax. Let $\mathcal{A}$ be a fixed, countable set of propositional atoms. An aggregate atom is of the following form:

$$\{b_1 = w_1, \ldots, b_m = w_m, {\sim}b_{m+1} = w_{m+1}, \ldots, {\sim}b_n = w_n\} \geq lb \tag{1}$$

where $b_i \in \mathcal{A}$ $(i = 1, \ldots, n)$, lb and each w_i $(i = 1, \ldots, n)$ are positive integers, $\sim$ denotes *negation as failure*, and $n \geq m \geq 0$. Integer lb is called lower bound, while w_i $(i = 1, \ldots, n)$ is the weight associated with the i-th literal in the aggregate. An atom is either a propositional atom, or an aggregate atom. A literal is either an atom (a positive literal), or an atom preceded by one or more occurrences of $\sim$ (a negated literal).

A (disjunctive) rule is of the following form:

$$a_1 \vee \cdots \vee a_m \leftarrow \ell_1, \ldots, \ell_n \tag{2}$$

where $a_i \in \mathcal{A}$ $(i = 1, \ldots, m)$, ℓ_j $(j = 1, \ldots, n)$ is a literal, $m \geq 0$ and $n \geq 0$. For a rule r of the form (2), disjunction $a_1 \vee \cdots \vee a_m$ is called the *head* of r, denoted $H(r)$; conjunction $\ell_1, \ldots, \ell_n$ is named the *body* of r, denoted $B(r)$; positive and negated literals in $B(r)$ are denoted $B^+(r)$ and $B^-(r)$, respectively. A *constraint* is a rule of the form (2) such that $m = 0$. A constraint is possibly associated with a positive integer by the partial function *weight*. For a compact representation, the weight will be sometimes indicated near the implication arrow, e.g., $\leftarrow_3 a, {\sim}b$ is a constraint of weight 3.

A propositional program Π_R is a set of rules. The set of constraints in Π_R is denoted $\mathit{constraints}(\Pi_\mathrm{R})$, while the remaining rules are denoted $\mathit{rules}(\Pi_\mathrm{R})$. A program with weak constraints Π is a pair $(\Pi_\mathrm{R}, \Pi_\mathrm{W})$, where Π_R is a propositional program and Π_W is a subset of $\mathit{constraints}(\Pi_\mathrm{R})$. Π_W is the set of *weak constraints*, while $\mathit{constraints}(\Pi_\mathrm{R}) \setminus \Pi_\mathrm{W}$ is the set of *hard constraints*. To simplify the presentation, any program $\Pi = (\Pi_\mathrm{R}, \Pi_\mathrm{W})$ is assumed to obey the following syntactic restriction, also known as *stratification of aggregates*: Let G^Π be a graph having a node for each propositional atom in $\mathcal{A}$ and an arc connecting a to b if there are a rule $r \in \Pi_\mathrm{R}$ and a literal $\ell \in B^+(r)$ such that $a \in H(r)$ and either $b = \ell$ or ℓ is an aggregate of the form (1) and $b = b_i$ for some $i \in [1..m]$. For each aggregate ℓ of the form (1) occurring in Π, there is no rule $r \in \Pi_\mathrm{R}$ such that b_i $(i \in [1..m])$ and some $a \in H(r)$ occurs in a cycle of G^Π.

Semantics. An interpretation I is a subset of $\mathcal{A}$. Relation $\models$ is defined as follows: For a propositional atom $a \in \mathcal{A}$, $I \models a$ if $a \in I$. For an aggregate atom a of the form (1), $I \models a$ if the following inequality is satisfied:

$$\sum_{i \in [1..m] : b_i \in I} w_i + \sum_{j \in [m+1..n] : b_j \notin I} w_j \geq lb.$$

For a negative literal ${\sim}a$, $I \models {\sim}a$ if $I \not\models a$. For disjunctions and conjunctions, $I \models a_1 \vee \cdots \vee a_n$ if $I \models a_i$ for some $i \in [1..n]$, and $I \models a_1, \ldots, a_n$ if $I \models a_i$ for each $i \in [1..n]$. For a rule r of the form (2), $I \models r$ if $I \models a_1 \vee \cdots \vee a_m$ whenever $I \models \ell_1, \ldots, \ell_n$. For a program $\Pi = (\Pi_\mathrm{R}, \Pi_\mathrm{W})$, $I \models \Pi$ if $I \models \Pi_\mathrm{R}$, where Π_R is seen as a conjunction.

The definition of stable models is based on a notion of program reduct [1]: Let Π_R be a propositional program, and I an interpretation. The reduct of Π_R w.r.t. I, denoted Π_R^I, is obtained from Π_R by deleting each rule r such that $I \not\models B^-(r)$, and removing negated literals in the remaining rules. An interpretation I is a stable model of Π_R if $I \models \Pi_\mathrm{R}^I$ and there is no $J \subset I$ such that $J \models \Pi_\mathrm{R}^I$. Let $SM(\Pi_\mathrm{R})$ denote the set of stable models of Π. Program Π is *coherent* if $SM(\Pi_\mathrm{R}) \neq \emptyset$, otherwise it is *incoherent*.

For a program with weak constraints $\Pi = (\Pi_\mathrm{R}, \Pi_\mathrm{W})$, each interpretation I is associated with a cost:

$$\mathit{cost}(\Pi_\mathrm{W}, I) := \sum_{r \in \Pi_\mathrm{W} : I \not\models r} \mathit{weight}(r).$$

A stable model I of $\Pi_\mathrm{R} \setminus \Pi_\mathrm{W}$ is optimal for Π if there is no $J \in SM(\Pi_\mathrm{R} \setminus \Pi_\mathrm{W})$ such that $\mathit{cost}(\Pi_\mathrm{W}, J) < \mathit{cost}(\Pi_\mathrm{W}, I)$.

Function RelaxWeakConstraint(r: weak constraint, **var** R: set)

```
1 begin
2     Let $aux$ be a fresh atom;
3     $R := R \cup \{aux = weight(r)\}$;
4     return $\leftarrow B(r), {\sim}aux$;
```

3 Algorithms

Modern algorithms for MaxSAT are based on iterative invocations of a SAT solver, some of which are adapted to ASP optimization problems in this section. Intuitively, the adapted algorithms operate by iteratively calling an ASP solver and by relaxing weak constraints. Roughly, a weak constraint is relaxed by adding a fresh atom aux to its body. (We also assume that aux is irrelevant for the coherence of the processed program, e.g., by introducing a rule of the form $aux \leftarrow {\sim}{\sim}aux$.) Relaxed weak constraints can be disabled during the stable model search if necessary. The algorithms considered in this section can be classified in two categories, namely *core-guided* and *model-guided* algorithms. Core-guided algorithms start by considering weak constraints as hard constraints and then selectively relax some of them until an optimum cost is found. Model-guided algorithms instead start by ignoring weak constraints and then enforce an improvement on the cost of the computed stable models. In the following if $\Pi = (\Pi_{\mathrm{R}}, \Pi_{\mathrm{W}})$ is an input program then $\Pi_{\mathrm{R}} \setminus \Pi_{\mathrm{W}}$ is assumed to be coherent.

Example 1. In the following we will use program $\Pi = (\Pi_{\mathrm{R}}, \Pi_{\mathrm{W}})$ to illustrate the differences between the algorithms, where Π_{R} consists of the following rules:

$$\begin{array}{lll} r_1 : a \vee b \leftarrow & r_3 : \leftarrow_1 a & r_5 : \leftarrow_1 b \\ r_2 : c \vee d \leftarrow & r_4 : \leftarrow_2 c & r_6 : \leftarrow_2 d \end{array}$$

and $\Pi_{\mathrm{W}} = \{r_3, r_4, r_5, r_6\}$.

3.1 Core-guided Algorithms

Core-guided algorithms are based on the concept of unsatisfiable core first introduced in the context of SAT solving [22]. According to the original definition, an unsatisfiable core of an unsatisfiable CNF ϕ is a subset of ϕ that is also unsatisfiable. The analogous notion in ASP can be stated as follows: An unsatisfiable core of an incoherent propositional program Π_{R} is a set $\Pi_{core} \subseteq constraints(\Pi_{\mathrm{R}})$ such that $rules(\Pi_{\mathrm{R}}) \cup \Pi_{core}$ is incoherent. For example, a core of Π_{R} in Example 1 is $\{r_3, r_5\}$. In this section we describe two core-guided algorithms, namely *bcd* [23] and *oll* [19]. The algorithms use function $ASPSolver(\Pi, PrefChoices)$, where Π is a program and $PrefChoices$ is a set of literals, whose output is a triple (res, Π_{core}, I), where res is a string, Π_{core} a set of rules and I an interpretation. Intuitively, the function searches for a stable model of Π. If one is

Algorithm 1: bcd

```
Input  : A program $\Pi = (\Pi_\mathrm{R}, \Pi_\mathrm{W})$
Output: The optimum cost for $\Pi$
1  begin
2      $(\Pi_\mathrm{S}, Cores, OPT) := (\Pi_\mathrm{W}, \emptyset, 0)$;
3      repeat
4          $\Pi_{aggr} := \emptyset$;
5          foreach $C \in Cores$ do
6              if $C.lb + 1 = C.ub$ then $C.mb := C.ub$; else $C.mb := \lfloor \frac{C.ub + C.lb}{2} \rfloor$;
7              $\Pi_{aggr} := \Pi_{aggr} \cup \{\leftarrow C.R \geq C.mb + 1\}$;
8          $(res, \Pi_{core}, I) := ASPSolver(\Pi_\mathrm{R} \cup \Pi_{aggr}, \emptyset)$;
9          if $res \neq INCO$ then
10             $OPT := cost(\Pi_\mathrm{W}, I)$;
11             foreach $C \in Cores$ do
12                 $C.ub := \sum_{aux=w \in C.R \wedge I \models aux} w$;
13         else
14             $SubCores := \{C \in Cores \mid C.core \cap \Pi_{core} \neq \emptyset\}$;
15             if $\Pi_{core} \cap \Pi_\mathrm{S} = \emptyset$ and $|SubCores| = 1$ then
16                 Let $SubCores = \{C\}$;
17                 $C.lb := C.mb$;
18             else
19                 Let $C$ be a new structure;
20                 $(C.core, C.R) := (\emptyset, \emptyset)$;
21                 foreach $r \in \Pi_{core} \cap \Pi_\mathrm{S}$ do
22                     $r' := RelaxWeakConstraint(r, C.R)$;
23                     $\Pi_\mathrm{S} := \Pi_\mathrm{S} \setminus \{r\}$;
24                     $\Pi_\mathrm{R} := (\Pi_\mathrm{R} \setminus \{r\}) \cup \{r'\}$;
25                     $C.core := C.core \cup \{r'\}$;
26                 $(C.lb, C.ub) := (0, 1 + \sum_{aux=w \in C.R} w)$;
27                 foreach $C' \in SubCores$ do
28                     $(C.core, C.R) := (C.core \cup C'.core, C.R \cup C'.R)$ ;
29                     $(C.lb, C.ub) := (C.lb + C'.lb, C.ub + C'.ub)$;
30                 $Cores := (Cores \setminus SubCores) \cup \{C\}$;
31     until $\forall C \in Cores\ C.lb + 1 \geq C.ub$;
32     return $OPT$;
```

found, say I, the function returns $(FOUND, \emptyset, I)$. Otherwise, the function returns $(INCO, \Pi_{core}, \emptyset)$, where Π_{core} is a core of Π. During the search, the first choices are those specified by the input parameter $PrefChoices$.

Algorithm bcd. Algorithm 1 is called core-guided binary search with disjoint cores, in short *bcd* and implements a binary search of the optimal solution. In a nutshell, all weak constraints are initially considered as hard constraints and a stable model is searched. If the processed program is incoherent then an unsatis-

Algorithm 2: oll

```
   Input  : A program $\Pi = (\Pi_{\mathrm{R}}, \Pi_{\mathrm{W}})$
   Output: The optimum cost for $\Pi$
1  begin
2      $(\Pi_{\mathrm{S}}, \Pi_{aggr}) := (\Pi_{\mathrm{W}}, \emptyset)$;
3      $(res, \Pi_{core}, I) := ASPSolver(\Pi_{\mathrm{R}} \cup \Pi_{aggr}, \emptyset)$;
4      if $res \neq INCO$ then return $cost(\Pi_{\mathrm{W}}, I)$;
5      foreach $\leftarrow R \geq lb \in \Pi_{aggr} \cap \Pi_{core}$ such that $|R| > lb$ do
6          $\Pi_{\mathrm{S}} := \Pi_{\mathrm{S}} \cup \{\leftarrow_1 R \geq lb\}$;
7          $\Pi_{aggr} := (\Pi_{aggr} \setminus \{\leftarrow R \geq lb\}) \cup \{\leftarrow R \geq lb+1\}$;
8      $R := \emptyset$;
9      foreach $r \in \Pi_{core} \cap \Pi_{\mathrm{S}}$ do
10         $\Pi_{\mathrm{S}} := \Pi_{\mathrm{S}} \setminus \{r\}$;
11         $\Pi_{\mathrm{R}} := (\Pi_{\mathrm{R}} \setminus \{r\}) \cup \{RelaxWeakConstraint(r, R)\}$;
12     $\Pi_{aggr} := \Pi_{aggr} \cup \{\leftarrow R \geq 2\}$;
13     goto 3;
```

fiable core is computed and stored in a set $Cores$, which is initially empty. Weak constraints in the computed core are relaxed and the new relaxing atoms are used to build a constraint comprising a single aggregate atom aimed at performing a binary search on the subsequent coherence tests. In fact, each unsatisfiable core is associated with a lower and an upper bound, which are updated during the computation. More in detail, whenever an incoherent program is processed, the new unsatisfiable core Π_{core} is merged with each element in $Cores$ intersecting Π_{core}. In this way $Cores$ is guaranteed to contain pairwise disjoint unsatisfiable cores, which actually represent disjoint subproblems. When no weak constraint needs to be relaxed, and Π_{core} intersects only one previously computed core C, the lower bound of C is increased because the subproblem associated with C has no solution of cost smaller than $(C.lb + C.ub)/2$. In fact, such a subproblem is represented by a constraint added at line 7. Hence, the new lower bound of C will force the algorithm to search for a solution of higher cost, actually resulting in a binary search of the optimum cost. When instead a stable model I is found, OPT as well as the upper bounds of the computed cores are updated according to I. This process is repeated until $Cores$ contains unsolved subproblems.

Example 2. Consider the program in Example 1. Initially, Π_{S} contains all weak constraints, i.e., r_3–r_6, while $Cores$ is empty (line 2). Program $\Pi_{\mathrm{R}} = \{r_1, \ldots, r_6\}$ is incoherent, and thus an unsatisfiable core Π_{core} is computed, say $\{r_3, r_5\}$. Weak constraints r_3 and r_5 are thus relaxed (line 22), i.e., they are replaced by the following constraints:

$$r_3' :\leftarrow a, {\sim}aux_3 \qquad r_5' :\leftarrow b, {\sim}aux_5$$

where aux_3 and aux_5 are fresh atoms. Rules r_3' and r_5' are stored in a new structure C_1 in $Cores$, whose lower and upper bounds are initially set to 0 and 3 (line 26). The subsequent coherence test must also satisfy a constraint obtained from C_1, that is, $\leftarrow \{aux_3 = 1, aux_5 = 1\} \geq 2$. However, the processed program is still incoherent and an unsatisfiable core $\{r_4, r_6\}$ is returned. Weak constraints r_4 and r_6 are thus relaxed, i.e., they are replaced by the following constraints:

$$r'_4 :\leftarrow a, \sim aux_4 \qquad r'_6 :\leftarrow b, \sim aux_6$$

where aux_4 and aux_6 are fresh atoms. Rules r'_4 and r'_6 are stored in a new structure C_2 in $Cores$, whose lower and upper bounds are initially set to 0 and 5, so that the next coherence check must also satisfy $\leftarrow \{aux_4 = 2, aux_6 = 2\} \geq 3$. Actually, a stable model is found, say $\{a, c, aux_3, aux_4\}$, which means that the current optimal solution has cost 3. Upper bounds of C_1 and C_2 are updated to 1 and 2, respectively. The next coherence check must thus satisfy $r_1, r_2, r'_3, \ldots, r'_6$, and the additional constraints $r_{C_1} : \leftarrow \{aux_3 = 1, aux_5 = 1\} \geq 2$ and $r_{C_2} : \leftarrow \{aux_4 = 2, aux_6 = 2\} \geq 2$. However, the unsatisfiable core $\{r'_4, r'_6, r_{C_2}\}$ is returned. In this case the lower bound of C_2 is set to 1 and the algorithm terminates returning 3, i.e., the optimal cost. In fact, the lower and the upper bounds of C_1 are 0 and 1, respectively, and the lower and the upper bounds of C_2 are 1 and 2, respectively. Hence, for each structure in $Cores$ the lower bound + 1 is greater than the upper bound (line 31).

Algorithm oll. Algorithm 2 is called *oll* and is conceived for unweighted ASP optimization problems, i.e., for programs in which all weak constraints have the same weight. However, there are several possibilities for using the algorithm in case of weighted ASP optimization problems [19]. The one we consider replaces each weak constraint r by $weight(r)$ copies of r of weight 1. (Intuitively, a literal $\sim a_i$ is added the i-th copy of r, where a_i is a fresh propositional atom.) Roughly, the algorithm initially considers all weak constraints of the input program $\Pi = (\Pi_\mathrm{R}, \Pi_\mathrm{W})$ as hard constraints and searches for a stable model. If none is found then some weak constraints are relaxed, which means that they can possibly be violated during the search for a stable model. This process is iterated until a stable model is found. The algorithm uses a set Π_S for storing all weak constraints of Π that are not relaxed, so that any weak constraint is relaxed at most once. Initially, Π_S is equal to Π_W. The algorithm also uses a set Π_{aggr} of constraints created by the algorithm, which is initially empty and will store constraints consisting of a unique aggregate atom. If a stable model I for $\Pi_\mathrm{R} \cup \Pi_{aggr}$ is found then I is also an optimal solution of Π. Otherwise, an unsatisfiable core Π_{core} is computed and used for relaxing Π_R. More in detail, constraints in $\Pi_{aggr} \cap \Pi_{core}$ are moved into Π_S and replaced by copies with increased lower bounds, unless the copies are trivially satisfied. Constraints in $\Pi_\mathrm{S} \cap \Pi_{core}$ are then relaxed by procedure $RelaxWeakConstraint$. Finally, an aggregate containing the new relaxing atoms and unitary weights is added to Π_{aggr}.

Example 3. Consider the program in Example 1, where for simplicity we consider all weak constraint of weight 1. Initially, Π_S contains all weak constraints, i.e., r_3–r_6. Program $\Pi_\mathrm{R} = \{r_1, \ldots, r_6\}$ is incoherent, and thus an unsatisfiable core Π_{core}

Algorithm 3: mgd

```
Input  : A program $\Pi = (\Pi_{\mathrm{R}}, \Pi_{\mathrm{W}})$
Output: The optimum cost OPT for $\Pi$
1 begin
2     $(\Pi_{\mathrm{S}}, \Pi_{\mathrm{R}}, R, OPT) := (\Pi_{\mathrm{W}}, \Pi_{\mathrm{R}} \setminus \Pi_{\mathrm{W}}, \emptyset, 1 + \sum_{r \in \Pi_{\mathrm{W}}} weight(r))$;
3     $(res, \Pi_{core}, I) := ASPSolver(\Pi_{\mathrm{R}} \cup \{\leftarrow R \geq OPT\}, \emptyset)$;
4     if $res = INCO$ then return OPT;
5     $OPT := min(cost(\Pi_{\mathrm{W}}, I), OPT)$;
6     foreach $r \in \Pi_{\mathrm{S}}$ such that $I \not\models r$ do
7         $\Pi_{\mathrm{S}} := \Pi_{\mathrm{S}} \setminus \{r\}$;
8         $\Pi_{\mathrm{R}} := \Pi_{\mathrm{R}} \cup \{RelaxWeakConstraint(r, R)\}$;
9     goto 3;
```

is computed, say $\{r_3, r_4, r_5, r_6\}$. All weak constraints are thus relaxed, i.e., they are replaced by constraints $r'_3, \ldots, r'_6$ from Example 2. The subsequent coherence test must also satisfy a constraint r_{aggr} of the form $\leftarrow \{aux_3 = 1, aux_4 = 1, aux_5 = 1, aux_6 = 1\} \geq 2$. However, the processed program is still incoherent and an unsatisfiable core $\{r_{aggr}, r'_3, \ldots, r'_6\}$ is returned. Constraint r_{aggr} is thus added to Π_{S} in order to be relaxed, and its lower bound is increased by 1 in the subsequent coherence check. The relaxed version of r_{aggr} is $\leftarrow \{aux_3 = 1, aux_4 = 1, aux_5 = 1, aux_6 = 1\} \geq 2, {\sim}aux_{aggr}$, where aux_{aggr} is a fresh atom. (Actually, there is yet another trivial constraint, namely $\leftarrow \{aux_{aggr} = 1\} \geq 2$.) The processed program is now coherent and a stable model is computed, say $\{a, c, aux_3, aux_4\}$ of cost 2, which is also optimal. (Recall that we are considering the unweighted version of the program in Example 1.)

3.2 Model-guided Algorithms

Model-guided algorithms are aimed at producing solutions of improved cost, until an optimal solution is found. In this section we consider *mgd* and *opt*.

Algorithm mgd. Algorithm 3 is called *mgd*. In a nutshell, weak constraints are initially ignored and a stable model is found. Recall that if $\Pi = (\Pi_{\mathrm{R}}, \Pi_{\mathrm{W}})$ is an input program then $\Pi_{\mathrm{R}} \setminus \Pi_{\mathrm{W}}$ is assumed to be coherent. Violated weak constraints are relaxed and considered as hard constraints in the subsequent stable model searches. Moreover, the program is extended by a constraint of the form $\leftarrow R \geq OPT$, where R contains the relaxing atoms and the associated weights, and OPT is the current optimal cost. This process is iterated until the program becomes incoherent, which means that OPT is the optimal cost for the original program.

Example 4. Consider the program in Example 1. Initially, Π_{S} contains all weak constraints, i.e., r_3–r_6, which are instead removed from Π_{R}. The optimal value OPT is initially set to 7, and set R is empty. A stable model for $\Pi_{\mathrm{R}} = \{r_1, r_2\}$ is computed, say $\{a, c\}$, and stored in variable I. The cost of this solution is 3,

Algorithm 4: opt

```
Input  : A program $\Pi = (\Pi_{\mathrm{R}}, \Pi_{\mathrm{W}})$
Output: The optimum cost OPT for $\Pi$
1 begin
2     $(R, OPT) := (\emptyset, 1 + \sum_{r \in \Pi_{\mathrm{W}}} weight(r))$ ;
3     foreach $r \in \Pi_{\mathrm{W}}$ do
4         $\Pi_{\mathrm{R}} := (\Pi_{\mathrm{R}} \setminus \{r\}) \cup \{RelaxWeakConstraint(r, R)\}$;
5     $PrefChoices := \{{\sim}aux \mid aux = w \in R\}$;
6     $(res, \Pi_{core}, I) := ASPSolver(\Pi_{\mathrm{R}} \cup \{\leftarrow R \geq OPT\}, PrefChoices)$;
7     if $res = INCO$ then return OPT;
8     $OPT := cost(\Pi_{\mathrm{W}}, I)$;
9     goto 6;
```

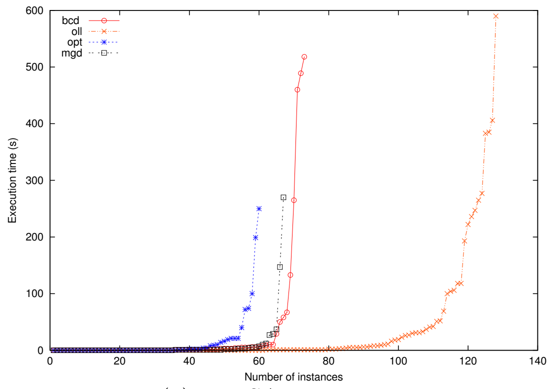
thus OPT is updated. Weak constraints r_3 and r_4 are then relaxed, i.e., they are removed from Π_{S}, and Π_{R} is extended with the following constraints:

$$r'_3 :\leftarrow a, {\sim}aux_3 \qquad r'_4 :\leftarrow c, {\sim}aux_4$$

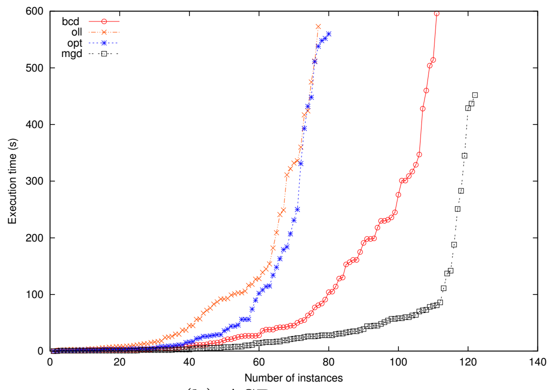
where aux_3 and aux_4 are fresh atoms. After this process, set R is $\{aux_3 = 1, aux_4 = 2\}$, and the subsequent coherence test must satisfy the constraint $\leftarrow R \geq 3$. Let us assume that the returned stable model I is $\{a, d, aux_3\}$, with cost 3. Weak constraint r_6 is then relaxed: Again, it is removed from Π_{S} and Π_{R} is extended with r'_6, i.e., $\leftarrow d, {\sim}aux_6$ where aux_6 is a fresh atom. R is extended with $aux_6 = 2$ and a new stable model is searched. Say that $I = \{b, c, aux_4\}$ is returned, again with cost 3. Weak constraint r_5 is relaxed: It is removed from Π_{S} and Π_{R} is extended with r'_5, i.e., $\leftarrow b, {\sim}aux_5$. R is extended with $aux_5 = 1$ and a new stable model is searched, but the program is now incoherent. The algorithm thus terminates by returning 3, i.e., the optimal cost for the original program.

Algorithm opt. Algorithm 4 is called *opt*. Concisely, all weak constraints are relaxed and a stable model is found. During the stable model search, the branching heuristic is forced to falsify relaxing atoms when possible. Moreover, as in mgd, a constraint with an aggregate atom is used to force an improvement of the solution within each stable model found.

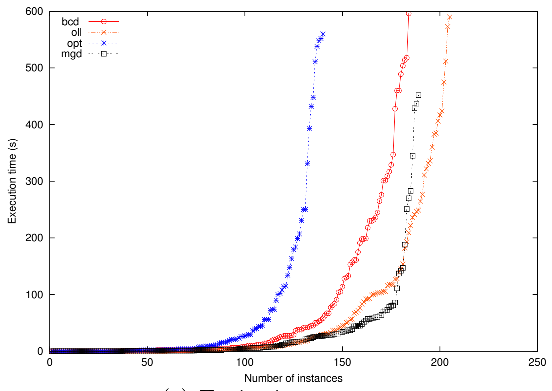
Example 5. Consider again the program in Example 1. Initially, OPT is 7, and Π_{R} contains r_1, r_2 and the relaxed constraints r'_3–r'_6 introduced in Example 4. Moreover, R is $\{aux_3 = 1, aux_4 = 2, aux_5 = 1, aux_6 = 2\}$ and $PrefChoices$ is $\{{\sim}aux_3, {\sim}aux_4, {\sim}aux_5, {\sim}aux_6\}$. Then, a stable model I for Π_{R} is found, say $\{a, c, aux_3, aux_4\}$ of cost 3. OPT is thus updated to 3 and a new stable model is searched, with the additional constraint $\leftarrow R \geq 3$. The resulting program is actually incoherent, and the algorithm terminates.



(a) MaxSAT instances



(b) ASP instances



(c) Entire instance set

Fig. 1. Cactus plots

4 Experiments

In this section we report the results of an experiment carried out to analyze the behavior of the four algorithms for computing optimal costs described in Section 3.

Hardware Setting. The experiment was run on a four core Intel Xeon CPU X3430 2.4 GHz, with 4 GB of physical RAM and running Linux Debian Lenny (32bit). Only one core was enabled, and time and memory limits were set to 600 seconds and 3 GB, respectively. Execution times and memory consumption were measured by the Benchmark Tool Run (http://fmv.jku.at/run/).

Implementation. The algorithms described in Section 3 were implemented in an experimental variant of the ASP solver WASP. The version of WASP that was used as base for the implementation is the one that participated in the 4th ASP Competition. The main algorithms were modified in order to allow several invocations of the main model search procedure from an external control procedure.

Benchmark setting. Tested instances were taken from the 4th ASP Competition [7] and from MaxSAT Competition 2008 (http://www.maxsat.udl.cat/08). MaxSAT instances are randomly selected among industrial instances. These are further divided into three sets, named MaxSAT, Partial-MaxSAT (some of the clauses must be satisfied), and Weighted-Partial-MaxSAT (presence of weighted clauses). ASP instances are from problems with weak constraints in the System Track of the 4th ASP Competition. We considered four problems corresponding to the following sets of instances: Abstract Dialectical Framework, Crossing Minimization, Maximal Clique, and Still Life. Non ground input programs are first processed by a version of the DLV grounder properly adapted to work with WASP. (We did not include the ValvesLocation which is too resource demanding already for the grounding phase.)

Results. The results are summarized in Table 1, Table 2 and Figure 1. Table 1 reports the number of solved instances and the percentage of solved instances

Table 1. Solved instances and percentage of solved instances

Problem	**bcd**	**oll**	**opt**	**mgd**	**#inst**
AbstractDialectical	109 (90.8%)	64 (53.3%)	73 (60.8%)	120 (100.0%)	120
CrossingMinim.	0 (0.0%)	10 (11.8%)	0 (0.0%)	0 (0.0%)	85
MaximalClique	0 (0.0%)	0 (0.0%)	3 (6.0%)	0 (0.0%)	50
StillLife	2 (7.7%)	3 (11.6%)	4 (15.4%)	2 (7.7%)	26
MaxSAT	6 (33.3%)	8 (44.4%)	4 (22.2%)	0 (0.0%)	18
Partial MaxSAT	66 (39.8%)	119 (71.7%)	55 (33.1%)	66 (39.8%)	166
Wei. Part. MaxSAT	1 (8.3%)	1 (8.3%)	1 (8.3%)	1 (8.3%)	12
Total ASP	**111 (39.5%)**	**77 (27.4%)**	**80 (28.5%)**	**122 (43.4%)**	**281**
Total MAXSAT	**73 (37.2%)**	**128 (65.3%)**	**60 (30.6%)**	**67 (34.2%)**	**196**
Total	**184 (38.6%)**	**205 (43.0%)**	**140 (29.3%)**	**189 (39.6%)**	**477**

within parenthesis for each considered algorithm. The last column reports the total number of instances for each set. Table 2 reports the number of wins of an algorithm, in terms of the number of instances in which an algorithm is the fastest, and the number of instances solved uniquely by one algorithm.

We observe that core-guided algorithms perform better in MaxSAT, where *oll* solves 128 instances and *bcd* 73. On the other hand *mgd* is very effective in the Abstract Dialectical Framework domain (100% of solved instances), and *opt* is the best in Maximal Clique (where other algorithms cannot solve any instance) and Still Life (where *opt* is always the fastest algorithm). The algorithm that solves more instances overall is *oll* (205) followed by *mgd* (189).

Figure 1 reports three cactus plots respectively analyzing the behavior of the algorithms in MaxSAT (a), ASP (b) and in the entire instance set (c). Recall that in a cactus plot the x-axis reports the number of instances that are solved within the time reported on the y-axis. By looking at the cactus plots we note that *oll* gives advantages (and it is very fast) only in MaxSAT instances, whereas is the worst algorithm in term of runtime performance and number of solved instances in ASP. In contrast, *mgd* seems to be the best option for ASP instances both in terms of number of solved instances and runtime performance. Nonetheless, it is important to note that *mgd* is very effective in Abstract Dialectical Frameworks only. Algorithm *opt* is effective in the remaining ASP domains but the number of solved instances is small, thus, this advantage is not visible in the cactus plots. In general the algorithm having the most uniform behavior overall is *bcd*. In fact, even if it is third overall in terms of number of solved instances (184), it performs similarly in ASP and MaxSAT and features several wins in both Partial MaxSAT (6) and Abstract Dialectical Framwework (15). As a matter of fact no algorithm outperforms all the others, or can be considered the best solution overall. Instead, each algorithm has either a specific domain where it is successful, or performs steadily in all the considered domains but not leading in any.

It was expected to see core-guided algorithms to be more effective in MaxSAT, since these algorithms were proposed in this area. We have to report the efficacy of model-guided algorithms in ASP. One possible discriminator can be identi-

Table 2. Number of WINS and number of uniquely solved instances

Problem	**bcd**	**oll**	**opt**	**mgd**
AbstractDialectical	15 (0)	0 (0)	5 (0)	100 (10)
CrossingMinimi.	0 (0)	10 (10)	0 (0)	0 (0)
MaximalClique	0 (0)	0 (0)	3 (3)	0 (0)
StillLife	0 (0)	0 (0)	4 (1)	0 (0)
MaxSAT	0 (0)	5 (1)	3 (0)	0 (0)
Partial MaxSAT	6 (0)	85 (51)	18 (0)	10 (0)
Wei. Part. MaxSAT	0 (0)	0 (0)	0 (0)	1 (0)
Total ASP	**15 (0)**	**10 (10)**	**12 (4)**	**100 (10)**
Total MAXSAT	**6 (0)**	**90 (52)**	**21 (0)**	**11 (0)**
Total	**21 (0)**	**100 (62)**	**33 (4)**	**111 (10)**

fied by observing that ASP and MaxSAT instances have different densities of (ground) weak constraints, and in particular ASP instances usually have more hard constraints and rules than weak constraints. In fact, instances with relatively few weak constraints, such as ASP and to some extent Partial MaxSAT benchmarks, seem to be better handled by model-guided algorithms, whereas industrial instances, which are dense of weak constraints, seem to be better approached by core-guided algorithms.

5 Conclusion

This paper reports on the implementation of several algorithms for evaluating weak constraints in the ASP solver WASP. These algorithms are obtained by porting to ASP several techniques for solving Maximum Satisfiability (MaxSAT) problems. In particular, WASP implements the following four algorithms: *mgd* [18], a variant of *optsat* [17], *bcd* [15] and *oll* [19].

WASP was run on instances from both ASP and MaxSAT competitions. The results of this experiment showed pros and cons of the different strategies for computing optimal answer sets. On the one hand the better performance of core-guided algorithms in MaxSAT instances is confirmed, on the other hand the remaining algorithms behave well in ASP problems. No algorithm outperforms the others in all the considered domains, and specific algorithms may be more effective than others in specific domains. This paves the way for future work, which includes the implementation of a new approach that combines the advantages of the different algorithms, and a parallel implementation that runs several algorithms concurrently.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
2. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: Proc. of ICLP, The MIT Press (1999) 23–37
3. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Transactions on Database Systems **22** (1997) 364–418
4. Faber, W., Leone, N., Pfeifer, G.: Semantics and Complexity of Recursive Aggregates in Answer Set Programming. Artificial Intelligence **175** (2011) 278–298 Special Issue: John McCarthy's Legacy.
5. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE Transactions on Knowledge and Data Engineering **12** (2000) 845–860
6. Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M.C., Sirianni, M., Terracina, G., Veltri, P.: The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In: Proc. of LPNMR, LNCS Springer (2011) 388–403
7. Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C.,

Ricca, F., Schneider, P., Schwengerer, M., Spendier, L.K., Wallner, J.P., Xiao, G.: The Fourth Answer Set Programming Competition: Preliminary report. In: Proc. of LPNMR (2013) 42–53

8. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A Native ASP Solver Based on Constraint Learning. In: Proc. of LPNMR. Volume 8148 of LNCS. (2013) 54–66
9. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven Answer Set Solving. In: Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07), Morgan Kaufmann Publishers (2007) 386–392
10. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. Communications of the ACM **5** (1962) 394–397
11. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In: Proc. of ICCAD. (2001) 279–285
12. Gaschnig, J.: Performance measurement and analysis of certain search algorithms. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1979) Technical Report CMU-CS-79-124.
13. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting Combinatorial Search Through Randomization. In: Proc. of AAAI/IAAI, AAAI Press (1998) 431–437
14. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proc. of the DAC, ACM (2001) 530–535
15. Morgado, A., Heras, F., Liffiton, M.H., Planes, J., Marques-Silva, J.: Iterative and Core-guided MaxSAT Solving: A Survey and Assessment. Constraints **18** (2013) 478–534
16. Marques-Silva, J., Planes, J.: Algorithms for Maximum Satisfiability using Unsatisfiable Cores. In: DATE. (2008) 408–413
17. Rosa, E.D., Giunchiglia, E., Maratea, M.: Solving Satisfiability Problems with Preferences. Constraints **15** (2010) 485–515
18. Morgado, A., Heras, F., Marques-Silva, J.: Model-guided Approaches for MaxSAT Solving. In: ICTAI. (2013) 931–938
19. Andres, B., Kaufmann, B., Matheis, O., Schaub, T.: Unsatisfiability-based Optimization in Clasp. In ICLP (Technical Communications). (2012) 211–221
20. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
21. Gelfond, M., Kahl, Y.: Knowledge Representation, Reasoning, and the Design of Intelligent Agents: The Answer-Set Programming Approach. Cambridge University Press (2014)
22. Zhang, L., Malik, S.: Validating SAT Solvers Using an Independent Resolution-based Checker: Practical implementations and other applications. In: DATE, IEEE Computer Society (2003) 10880–10885
23. Heras, F., Morgado, A., Marques-Silva, J.: Core-guided Binary Search Algorithms for Maximum Satisfiability. In AAAI. (2011)