On Minimal Corrections in ASP

Mikoláš Janota[1] and Joao Marques-Silva[2]

[1] IST/INESC-ID, Lisbon, Portugal
[2] LaSIGE, Faculty of Science, University of Lisbon, Portugal

Abstract. As a programming paradigm, answer set programming (ASP) brings about the usual issue of the human error. Hence, it is desirable to provide automated techniques that could help the programmer to find the error. This paper studies the question of computing a subset-minimal correction of a contradictory ASP program. A contradictory ASP program is often undesirable and we wish to provide an automated way of fixing it. We consider a minimal correction set of a contradictory program to be an irreducible set of rules whose removal makes the program consistent. In contrast to propositional logic, corrections of ASP programs behave non-monotonically. Nevertheless, we show that a variety of algorithms for correction set computation in propositional logic can be ported to ASP. An experimental evaluation was carried showing that having a portfolio of such algorithms is indeed of benefit.

1 Introduction

Answer set programming (ASP) is a powerful paradigm for modeling and solving combinatorial and optimization problems in artificial intelligence. An inconsistent program is such a program that does not have a solution. This might be due to a bug in the program and for such cases it is desirable to provide the programmer with tools that would help him identify the issue.

The primary motivation for this paper is to focus on program's *input*. In theory, an input to an ASP program is really just another ASP program that is joined to the original one. In practice, however, the conceptual division between a program and its input plays an important role in the program's development. Indeed, a program, as a set of rules, expresses the semantics of the problem being solved. The input is a set of *facts* describing a particular instance of the problem. Such input is typically large.

This paper asks the question, how to resolve situations when there is an error in the given input? In particular, we consider scenarios when the input leads to a contradiction in the program. Consider the following simple program.

$$\leftarrow \mathit{not}\ \mathit{move}(a). \qquad \texttt{\% program} \qquad (1)$$

$$\mathit{move}(a) \leftarrow \mathit{stone}(b), \mathit{not}\ \mathit{stone}(c). \qquad \texttt{\% program} \qquad (2)$$

$$\mathit{stone}(c) \leftarrow . \qquad \texttt{\% input} \qquad (3)$$

By rule (1) the program requires $\mathit{move}(a)$ be true. To achieve that, however, $\mathit{stone}(b)$ must be true and $\mathit{stone}(c)$ must be false (by rule (2)). The input is

specified as the fact (3). Altogether, the program is *inconsistent* since rule (2) is not applicable. If we wish to modify the input so it becomes consistent, the fact $stone(c)$ must be removed and the fact $stone(b)$ must be added. It is the aim of this paper to compute such corrections to inconsistent inputs. Further, we aim at corrections that are *irreducible*, i.e. that do not perform unnecessary changes. Note that this program could also be made consistent by modifying the rules (1) and (2). Similarly, one could add the fact $move(a)$. This might be undesirable as these represent the rules of the modeled move (in a game, for instance). In the end, however, it is the programmer that must decide what to correct. The objective of the proposed tool-support is to pin-point the source of the inconsistency.

We show that the problem of inconsistency corrections is closely related to a problem of *maximal consistency*, which we define as identifying a subset-maximal set of atoms that can be added to the program as facts while preserving consistency. Maximal consistency lets us provide a solution to inconsistency correction but it is also an interesting problem to study on its own.

Maximal consistency is closely related to the concept of *maximally satisfiable sets (MSS)* in propositional logic. For a formula in conjunctive normal form, an MSS is a subset of the formula's clauses that is satisfiable and adding any clause to it makes it unsatisfiable. There is also a dual, minimal correction subset (MCS), which is a complement of an MSS [29].

An important difference between propositional logic and ASP is that ASP is *not monotone*. This means that an algorithm for calculating MSSes cannot be immediately used for ASP. We show, however, that it is possible to port existing MSS algorithms to ASP. This represents a great potential for calculating maximally consistent sets in ASP as a bevy of algorithms for MSS exist [4,31,32].

The main contributions of this paper are the following. (1) It devises a technique for adapting algorithms from MSS computation to maximal consistency in ASP. (2) Using the technique a handful of MSS algorithms is adapted to ASP. (3) It is shown how maximal consistency can be used to calculate minimal corrections to ASP inputs. (4) The proposed algorithms were implemented and evaluated on a number of benchmarks.

2 Background

We assume the reader's familiarity with standard ASP syntax and semantics, e.g. [5]. Here we briefly review the basic notation and concepts. In particular, a *(normal) logic program* is a finite set of *rules* of the following form.

$$a \leftarrow b_1, \ldots, b_m, not\, c_{m+1}, \ldots, not\, c_n,$$

where a, b_i, c_j are atoms. A *literal* is an atom or its default negation $not\, a$. A rule is called a *fact* if it has an empty body; in such case we don't write the symbol $\leftarrow$. For a rule r, we write $body(r)$ to denote the literals $b_1, \ldots, b_m, not\, c_{m+1}, \ldots, not\, c_n$ and we write $head(r)$ to denote the literal a. We write $B^+(r)$ for $b_1 \ldots, b_m$

and $B^-(r)$ for $c_{m+1}, \ldots, c_n$. Further, we allow *choice rules* of the form $n \leq \{a_1, \ldots, a_k\}$. (this is a special case of *weight constraint rules* [34, 43, 44]).

A program is called *ground* if it does not contain any variables. A *ground instance* of a program P, denoted as $grd(P)$, is a ground program obtained by substituting variables of P by all constants from its Herbrand universe.

The semantics of ASP programs can be defined via a *reduct* [23, 24]. Let I be a set of ground atoms. The set I is a *model* of a program P if $head(r) \in I$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$ for every $r \in grd(P)$. The reduct of a program P w.r.t. the set I is denoted as P^I and defined as follows.

$$P^I = \left\{ head(r) \leftarrow B^+(r) \mid r \in grd(P), I \cap B^-(r) = \emptyset \right\}$$

The set I is an *answer set* of P if I is a minimal model of P^I. Intuitively, this definition guarantees that an answer set contains only atoms that are justified by the rules of the program (cf. [28]). A choice rule $l \leq \{s_1, \ldots, s_k\}.$ in a program additionally guarantees that any answer set contains at least l atoms from $s_1, \ldots, s_k$, and, the rule provides a justification for any of those atoms. For precise semantics see [34, 44]. A program is *consistent* if it has at least one answer set, it is *inconsistent* otherwise.

3 Maximal Consistency in ASP

This section studies the problem of computing a maximal subset of given atoms whose addition to the program, as facts, yields a consistent program.

Definition 1 (maximal consistent subset). *Let P be a consistent ASP program and $\mathcal{S}$ be a set of atoms. A set $\mathcal{L} \subseteq \mathcal{S}$ is a* maximal consistent subset *of $\mathcal{S}$ w.r.t. P if the program $P \cup \{s. \mid s \in \mathcal{L}\}$ is consistent and for any $\mathcal{L}'$, such that $\mathcal{L} \subsetneq \mathcal{L}' \subseteq \mathcal{S}$, the program $P \cup \left\{s. \mid s \in \mathcal{L}'\right\}$ is inconsistent. If P is clear from the context, we simply say that $\mathcal{L}$ is an (in)consistent set of atoms.*

The definition of maximal consistent subset is syntactically similar to *maximal models* or MSSes in propositional logic. Semantically, however, there is an important difference due to nonmonotonicity of ASP. While in propositional logic the satisfiability of $\phi \wedge x \wedge y$ guarantees satisfiability of $\phi \wedge x$, in ASP it is not necessarily the case. Hence, algorithms for MSSes in propositional logic *cannot* be readily used for our maximal consistent subset. We will show, however, that it is possible to port these algorithms to ASP.

The following presentation uses the following auxiliary functions. The function $\texttt{choice}(\mathcal{S})$ yields the choice rule $0 \leq \{s_1, \ldots, s_k\}$ over the set $\mathcal{S} = \{s_1, \ldots, s_k\}$. The function $\texttt{atleast1}(\{s_1, \ldots, s_k\})$ yields the choice rule $1 \leq \{s_1, \ldots, s_k\}$. The function $\texttt{solve}(P)$ represents an ASP solver, which returns a pair (res, μ) where res is *true* and μ is an answer set of P if some exists. If P is inconsistent, res is *false* and μ is undefined.

A *brute force* approach to calculating a maximal consistent subset would be to enumerate all subsets of $\mathcal{S}$ and for each test whether it is still consistent. As there are $2^{|\mathcal{S}|}$ subsets of $\mathcal{S}$, this approach is clearly unfeasible.

Algorithm 1: Atleast-1 algorithm for maximal consistency.

```
1 ℒ ← ∅
2 while true do
3     P′ ← P ∪ {s. | s ∈ ℒ}
4     P′ ← P′ ∪ {atleast1(𝒮 ∖ ℒ).}
5     (res, μ) ← solve(P′)
6     if ¬res then return ℒ
7     ℒ ← μ ∩ 𝒮
```

A better approach is to *maximize* the number of $s \in \mathcal{S}$ that are set to true. Such ensures finding a maximal consistent subset with *maximum cardinality*. This can be done by iteratively calling an ASP solver while imposing increasing cardinality on the set $\mathcal{S}$. However, modern ASP solvers directly support minimization constraints through which maximization can be specified by minimizing negation of the atoms in $\mathcal{S}$. This approach was taken elsewhere [21, 45].

Finding a maximal correction subset with maximum cardinality might be computationally harder than finding *some* maximal correction subset. This is the purpose of the rest of the section.

We begin by an important observation that it is possible to check whether a set of atoms $\mathcal{L}$ can be extended into a consistent set of atoms by a single call to an ASP solver.

Observation 1 *Let P be an ASP program, $\mathcal{L} \subseteq \mathcal{S}$ be sets of atoms, and P' be defined as $P' = P \cup \{s. \mid s \in \mathcal{L}\} \cup \{\texttt{choice}(\mathcal{S} \setminus \mathcal{L}).\}$. There exists a consistent set $\mathcal{L}'$ s.t. $\mathcal{L} \subseteq \mathcal{L}' \subseteq \mathcal{S}$ iff P' has an answer set μ such that $\mathcal{L}' = \mathcal{S} \cap \mu$.*

Observation 1 enables us to check whether a set of atoms can be extended into a consistent set. This is done by letting the solver choose the additional atoms. Similar reasoning enables us to devise a test for checking that a consistent set $\mathcal{L}$ is already maximal. This is done by enforcing that $\mathcal{L}$ is extended by at least one element.

Observation 2 *Let P be an ASP program and $\mathcal{S}$ be a set of atoms, and P' be defined as $P' = P \cup \{s. \mid s \in \mathcal{L}\} \cup \{\texttt{atleast1}(\mathcal{S} \setminus \mathcal{L})\}$. A set $\mathcal{L} \subseteq \mathcal{S}$ is a maximal consistent subset of $\mathcal{S}$ iff $P \cup \{s. \mid s \in \mathcal{L}\}$ is consistent and P' is inconsistent.*

Combining Observation 1 and Observation 2 gives us Algorithm 1. The algorithm maintains a lower bound $\mathcal{L}$, which is initialized to the empty set. The set $\mathcal{L}$ grows incrementally until a maximal set is found. In each iteration it is tested whether $\mathcal{L}$ is already maximal in accord with Observation 2. The invariant of the loop is that the program $P \cup \{s. \mid s \in \mathcal{L}\}$ is consistent. The set $\mathcal{L}$ grows monotonically with the iterations because whenever a new answer set μ is obtained, it must contain all the atoms from the previous $\mathcal{L}$. Hence, the algorithm terminates in at most $|S|$ calls to an ASP solver. Note that the algorithm does not necessarily need all of the $|S|$ iterations since more than one element might be

Algorithm 2: Iterative algorithm for maximal consistency.

```
1  ℒ ← ∅
2  while 𝒮 ≠ ∅ do
3      s_f ← pick an arbitrary element from 𝒮
4      𝒮 ← 𝒮 ∖ {s_f}
5      ℒ ← ℒ ∪ {s_f}
6      P′ ← P ∪ {choice(𝒮).}
7      P′ ← P′ ∪ {s. | s ∈ ℒ}
8      (res, μ) ← solve(P′)
9      if ¬res then ℒ ← ℒ ∖ {s_f}
10     else ℒ ← ℒ ∪ (μ ∩ 𝒮)
11 return ℒ
```

added to $\mathcal{L}$ in one iteration. Similar algorithms have been proposed in different contexts, including computing the backbone of a propositional formula [48] and computing a *minimal correction set (MCS)* of a propositional formula [31].

The second algorithm we propose also maintains a lower bound on a maximal consistent set and tests, one by one, the elements to be added. The pseudo-code can be found in Algorithm 2. In each iteration it tests whether an element s_f can be added to the current lower bound $\mathcal{L}$. If it is possible, the algorithm continues with a larger $\mathcal{L}$. If it is not, then s_f is removed from $\mathcal{L}$ and never inspected again.

In contrast to Algorithm 1, here it is not immediately clear that the returned set is indeed maximal. Indeed, how do we know that if it was impossible to add some element s_f to an earlier $\mathcal{L}_1$, that is it still impossible for the final $\mathcal{L}_f$? This follows from Observation 1 and from the fact that $\mathcal{L}$ grows monotonically throughout the algorithm. More precisely, it holds that $\mathcal{L}_1 \subseteq \mathcal{L}_f$ and our test checks that s_f cannot be added to any *superset* of $\mathcal{L}_1$, i.e. if it was *not* possible to extend $\mathcal{L}_1$ with s_f, then it is also impossible to extend $\mathcal{L}_f$ with it. As Algorithm 1, Algorithm 2 also performs at most $|\mathcal{S}|$ calls to an ASP solver. Similar algorithms have been proposed in a number of contexts, including computing an MCS [4,36].

The third algorithm we consider is inspired by the *progression algorithm* [32]. The algorithm is shown in Algorithm 3. It tries to add more than a single atom to the lower bound in one iteration—atoms are added in *chunks*. This is done progressively: in the very first iteration the chunk contains a single atom. The chunk size is doubled each time the current chunk is added with success. If it is not possible to extend the current lower bound $\mathcal{L}$ with the current chunk, the size is reset again to 1. Whenever a chunk of size 1 cannot be added to $\mathcal{L}$ the clause comprising the chunk is no longer inspected. This guarantees termination. Note that the algorithm aims at constructing $\mathcal{L}$ as quickly as possible by adding larger chunks of atoms at a time. The use of chunks finds other applications, including redundancy removal [6]

Algorithm 3: Progression-based algorithm for maximal consistency.

```
1  ℒ ← ∅                                          // consistency lower bound
2  K ← 1                                                          // chunk size
3  while 𝒮 ≠ ∅ do
4      𝒞 ← pick min(|𝒮|, K) arbitrary elements from 𝒮
5      𝒮 ← 𝒮 ∖ 𝒞
6      ℒ ← ℒ ∪ 𝒞
7      P′ ← P ∪ {choice(𝒮).}
8      P′ ← P′ ∪ {s. | s ∈ ℒ}
9      (res, μ) ← solve(P′)
10     if ¬res then
           // re-analyze chunk more finely
11         ℒ ← ℒ ∖ 𝒞
12         if K > 1 then 𝒮 ← 𝒮 ∪ 𝒞
13         K = 1                                          // reset chunk size
14     else
15         K ← 2K                                        // double chunk size
16         ℒ ← ℒ ∪ (μ ∩ 𝒮)
17 return ℒ
```

4 From Maximal Consistency to Minimal Corrections

In this section we see how maximal consistency is useful for calculating corrections to an inconsistent program. Hence, the objective is to calculate an irreducible correction to a given inconsistent program so it becomes consistent.

The concept of *(minimal) correction sets* commonly appears in propositional logic [4]. Correction sets, however, are meant to be removed from the formula in order to make it consistent. In ASP, however, corrections might consist of removal or *addition*. Indeed, unlike in propositional logic, an ASP program may become consistent after a fact (or rule) is added.

Enabling corrections by addition brings about a substantial difficulty as the universe of rules that can be potentially added to the program can be easily unwieldy or even infinite. Hence, we assume that the user (the programmer) provides us with a set of rules $\mathcal{R}$ that can be removed from the program and a set of rules $\mathcal{A}$ that can be added to the program.

Definition 2 ($(\mathcal{A},\mathcal{R})$-correction). *Let $\mathcal{A}$ and $\mathcal{R}$ be sets of rules and P be an inconsistent logic program. An* $(\mathcal{A},\mathcal{R})$-correction *of P is a pair (M_r, M_a) such that $M_r \subseteq \mathcal{R}$ and $M_a \subseteq \mathcal{A}$ and the program $(P \setminus M_r) \cup M_a$ is consistent.*

An $(\mathcal{A},\mathcal{R})$-correction (M_r, M_a) is minimal *if for any $(\mathcal{A},\mathcal{R})$-correction (M'_r, M'_a) such that $M'_r \subseteq M_r$ and $M'_a \subseteq M_a$, it holds that $M_a = M'_a$ and $M_r = M'_r$.*

We refer to the problem of calculating a minimal correction as MinCorrect; it accepts as input sets $\mathcal{A}$, $\mathcal{R}$, and a program P and outputs a pair (M_r, M_a) in accord with Definition 2.

We show how to translate MinCorrect to maximal consistent subset calculation. Construct a program P' from P as follows[3].

1. Introduce fresh atoms s^{r}_r for each $r \in \mathcal{R}$ and s^{a}_r for each $r \in \mathcal{A}$.
2. Replace each rule $r \in \mathcal{R}$ with $\mathit{head}(r) \leftarrow s^{\mathrm{r}}_r,\ \mathit{body}(r)$
3. Replace each rule $r \in \mathcal{A}$ with $\mathit{head}(r) \leftarrow \mathit{not}\ s^{\mathrm{a}}_r,\ \mathit{body}(r)$.

Let $\mathcal{S} = \{s^{\mathrm{r}}_r \mid r \in \mathcal{R}\} \cup \{s^{\mathrm{a}}_r \mid r \in \mathcal{A}\}$. Then any maximal consistent subset $\mathcal{L}$ of $\mathcal{S}$ w.r.t. P' gives us a solution to MinCorrect by setting $M_r = \{r \mid s^{\mathrm{r}}_r \notin \mathcal{L}\}$ and $M_a = \{r \mid s^{\mathrm{a}}_r \notin \mathcal{L}\}$. It is easy to see why that is the case. From the definition of maximal consistent subset, there must be an answer set μ of the program $P' \cup \{s. \mid s \in \mathcal{L}\}$. If $s^{\mathrm{r}}_r \in \mu$ for $r \in \mathcal{R}$, then μ also satisfies the original rule r, hence it is not necessary to remove r to achieve consistency. Similarly, if $s^{\mathrm{a}}_r \in \mu$, then the body of $\mathit{head}(r) \leftarrow \mathit{not}\ s^{\mathrm{a}}_r,\ \mathit{body}(r)$ is false, i.e. the rule is ineffective, and therefore it is not necessary to add the rule r to the original program to achieve consistency. Conversely, adding any further s^{r}_r or s^{a}_r to μ leads to inconsistency due to the definition of maximal consistent subset. Hence, M_r and M_a defined as above are irreducible.

4.1 Minimal Corrections for Program Inputs

Here we come back to the problem that we used to motivate the paper in the introduction, which is to calculate corrections of program's data so that the program becomes consistent. Hence, we assume that there is a program P_e—representing the encoding of the problem and a program P_d—representing data to the encoding, consisting only of facts. Assuming that $P_e \cup P_d$ is inconsistent, we wish to identify correction to P_d that would make it consistent.

For such, we use the concept of minimal corrections as described above. What is needed is to specify the sets $\mathcal{R}$ and $\mathcal{A}$. To specify the set $\mathcal{R}$, the user just needs to identify facts in the given input that can be removed. This can for instance be done by giving predicate names. Specifying the set $\mathcal{A}$ is more challenging because it may comprise facts that do *not* appear in the program. Hence, we let the user give expressions E of the form $p(A_1) : t(A_2)$ where p and t are predicate symbols and A_1,A_2 appropriate arguments. The set $\mathcal{A}$ is obtained by an additional call to an ASP solver. This call constructs a program P' just as above and adds the rule `atleast1` $(\{s^{\mathrm{r}}_r \mid r \in \mathcal{R}\} \cup \{E_1, \ldots, E_u\})$. If this call is successful, the obtained answer set contains facts of the form $p(A_1)$ that constitute the set $\mathcal{A}$.

5 Experimental Evaluation

The proposed algorithms were implemented using the state-of-the-art ASP solver clingo [19], version 4.2.1. Algorithms 1–3 were implemented as presented, i.e. the

[3] Similar construction appears elsewhere [45].

ASP solver is invoked repeatedly from a python script. Computing a cardinality-maximal consistent subset was done by a single call to clingo, using the minimization rule syntax (:∼). For readability we refer to the individual approaches by single letters, Algorithm 1–letter **a** (**a**tleast-1 constraint); Algorithm 2–letter **u** (**u**nit addition); Algorithm 3–letter **p** (**p**rogression); the ma**x**imization approach is denoted by **x**. The evaluation was carried out on machines with Intel Xeon 5160 3GHz and 4GB of memory. The time limit was set to 1000 seconds and the memory limit to 2GB. The experimental evaluation considers several problems from the 2013 ASP Competition[4]. The following classes were considered.

Solitaire. In the game a single player moves and removes stones from the board according to the rule that a stone can be moved by two squares if that move jumps over another stone; the stone being jumped over is removed (similarly to *Checkers*). The problem to be solved in the context of this game is to perform T steps given T and an initial configuration. For instance, the problem does *not* have a solution if the board contains only one stone and at least one move is required to be made.

The size of the board is 7x7 with 2x2-size corners removed, thus comprising 33 squares. The initial configuration of the board is given by the predicates $\mathit{empty}(L)$ and $\mathit{full}(L)$, where L is a square of the board. The set $\mathcal{R}$ was specified as containing all the input's facts using *empty* and *full*. The set $\mathcal{A}$ was specified by the expressions $\mathit{empty}(L) : \mathit{location}(L)$ and $\mathit{full}(L) : \mathit{location}(L)$.

To generate unsatisfiable instances of the game, initial board configurations were generated randomly and the parameter T (number of steps) were fixed. We considered $T = 12$ and $T = 16$; 100 instances were generated for each T and only the inconsistent ones were considered. This process was repeated for other benchmarks.

Knight tour with holes. The input is a chessboard of size $N \times N$ with H holes. Following the standard chess rules, a knight chess piece must visit all non-hole positions of the boards exactly once and return to the initial position (which may be chosen). The objective of correction was to remove as few holes as possible so that the tour is possible. Random hole positions were generated for fixed N and H. Instances with $N = 8$, $H = 4$ and $N = 8$, $H = 10$ were considered.

Graceful graphs. Given a graph (V, E) the task is to determine whether it is possible to label its vertices with distinct integers in the range $0..|E|$ so that each edge is labeled with the absolute difference between the labels of its vertices and so that all edge labels are distinct (such graph is called *graceful*). The correction, for a given graph that is not graceful, tries to find its subgraph that maintains as many edges as possible (a single-edge graph is graceful). Considered instances were generated with $|V| = 10$ and $|E| = 50$ and $|V| = 30$ and $|E| = 20$.

Permutation Pattern Matching. The problem's input are two permutations T and P. The task is to determine whether T contains a subsequence that is order-isomorphic to P. For the purpose of correcting we consider T and P so that P is not an order-isomorphic subsequence of T with the objective of adding and

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

Family	**a**	**p**	**u**	**x**	**VBS**
knight [8,10] (95)	74	75	**78**	60	80
knight [8,4] (51)	7	**13**	**13**	7	14
patterns [16,10] (100)	**100**	**100**	**100**	**100**	100
patterns [20,15] (100)	**100**	**100**	**100**	**100**	100
solitaire [12] (18)	**18**	**18**	**18**	17	18
solitaire [14] (16)	**12**	9	11	4	13
graceful graphs [10,50] (100)	57	**75**	63	62	83
graceful graphs [30,20] (57)	56	**57**	**57**	55	57
total (537)	424	**447**	440	405	465

Table 1. Number of solved instances for each instance class. Parameters of the class given in square brackets and total number of the instances in a class in parentheses.

Δ	0	1	2	3	4	5	6	7	8–20
a	300	19	38	11	21	8	17	1	9
p	328	16	39	12	16	7	9	1	19
u	332	20	30	14	15	7	8	1	13

Table 2. Distribution of how difference between $|M_a| + |M_r|$ for a–p and for x.

removing elements to P so that P satisfies the condition. Considered instances were generated with $|T| = 16$ and $|P| = 10$ and $|T| = 20$ and $|P| = 15$.

Table 1 shows the number of solved instances by each of the approaches for the different benchmark families. The last column shows the *virtual best solver (VBS)*, which is calculated by picking the best solver for each of the instances. While the progression-based algorithm (p) is clearly in the lead, it is not a winner for all the instances (or even classes of instances). Indeed, the virtual best solver enables us to solve almost 20 more instances compared to the progression-based approach. The strength of progression shows on benchmarks with larger target sets. Such is the case for the class graceful graphs [10,50], where the target set is the set of edges E with $|E| = 50$. On the other hand, the approaches a and u do not aim at minimizing the number of calls to the ASP solver but represent calls that are likely to be easier. In u, the solver only must make sure that one fact from the target set is set to true. In a, the solver can in fact choose which fact should be set to true. These algorithms are the two best ones for solitaire [16]. Fig. 1 shows a cactus plot across all the considered families (300 easiest instances were cut off for better readability). This plot confirms data in Table 1. Progression provides the most robust solution but at the same time, there is a significant gap between progression and the virtual best solver.

Out of the considered algorithms, cardinality maximization (x) performs the worst. The scatter plot in Fig 2 compares progression to cardinality maximization. There are some instances where maximization performs well but overall progression dominates maximization; by orders of magnitude in a number of cases.

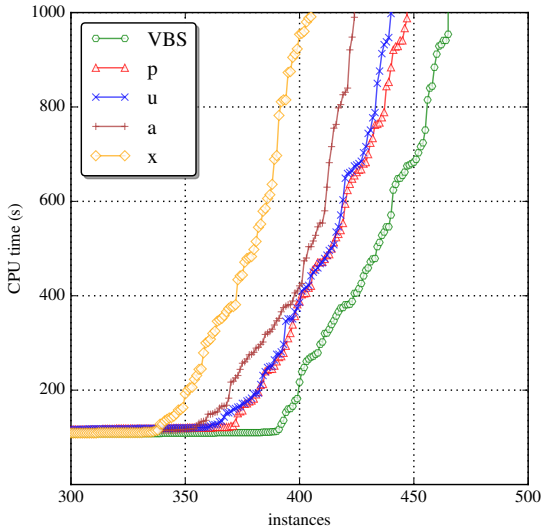

Fig. 1. Cactus plot of the overall results with the first 300 easiest instances cut off.

While Algorithms 1–3 give us an advantage over the cardinality maximization approach, the data so far does not show how the minimal corrections differ from the minimum-cardinality corrections in size. This is shown in Table 2. The table shows a distribution of how much the value $|M_a| + |M_r|$ differs from the value obtained from the approach x. For instance, row p, column 3 shows that in 12 cases the value differs by 3 from the minimum one. Naturally, these data only come from instances where x and the given approach finished with success. The distributions have a characteristic "heavy tail". In the majority of cases, the actual minimum is obtained (over 300 for all the approaches). Outliers exist but they are small in numbers.

6 Related Work

The proposed approach is closely related to the work on debugging on answer set programs, which has a long history [1,2,7,8,10–13,15,21,25,30,37,39,42,45–47]. Namely, the approach of Brain et al., which also may produce new facts in a correction [8]. Such facts, however, may only be heads of existing rules. Hence, our approach to data debugging by corrections gives the user a tighter control over the given set of inputs.

Nevertheless, the existing work on ASP debugging (in particular [8,21,45]), also needs to deal with removing redundancy. For such, the mentioned works use the cardinality-maximal/minimal sets. Hence, our approach to maximal subset consistency could be applied instead. Our experimental evaluation suggests that this could improve efficiency of those approaches.

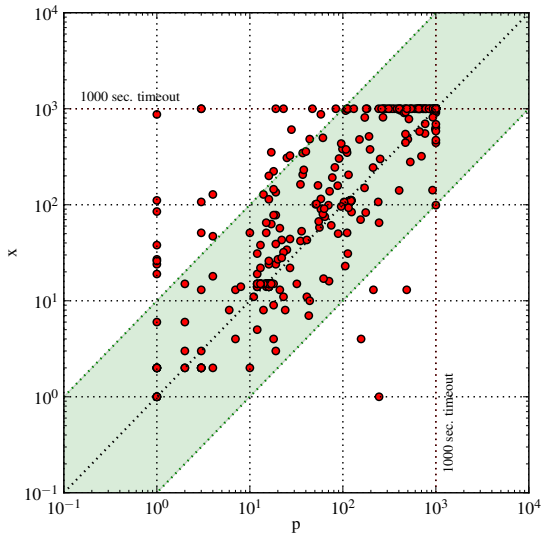

Fig. 2. Scatter plot comparing approaches x and p (logarithmic scale).

Other approaches to maximality in ASP exist. In particular, Gebser et al. [20] use meta-modeling techniques to optimize given criteria and Nieves and Osorio propose calculation of maximal models using ASP [35]. Both approaches, however, hinge on *disjunctive logic programming (DLP)*. Hence, these approaches require solving a problem in the second level of the polynomial hierarchy. Intuitively, this means worst-case exponential calls to an NP oracle. In contrast, all our approaches require *polynomial* calls to an NP oracle. However, we do not claim that theoretically there does not exist a non-disjunctive approach to the problem.

Some of the algorithms described in the paper are inspired by work on computing MSS/MCSes of propositional formulas in conjunctive normal form, which in term can be traced back to Reiter's seminal work on model-based diagnosis [40]. The well-known grow procedure is described for example in [4] and more recently in [36]. However, it is used in many other settings, including backbone computation [33] or even prime implicant computation [14]. There has been recent renewed interest in the computation of MCSes [31, 32]. Additional algorithms include the well-known QuickXplain algorithm [18, 27] or dichotomic search [26]. Clearly, and similarly to the case of ASP, MaxSAT can be used for computing and enumerating MCSes [29].

Although the paper considers a restricted sample of algorithms for computing MCS algorithms, adapted from [4, 31, 32], other options would include the additional approaches summarized above.

Our approach to effectively modifying the debugged program is conceptually akin to belief revision and abduction reasoning in ASP [16, 41, 49]. However, our

main focus is to compare the computational efficiency of the different presented algorithms.

Minimization in the context of ASP is addressed in asprin [9] and recent work of Faber et al. [17]. It is the subject of future work to explore the context presented in these works.

7 Summary and Future Work

Motivated by debugging of data to ASP programs, the paper studies the problem of minimal (irreducible) corrections of inconsistent programs and the problem of maximal consistency. The two problems are closely related. Indeed, a minimal correction to an inconsistent program can be calculated by computing a maximal consistency with respect to only slightly modified program.

We show that algorithms for calculating maximally satisfiable sets (MSSes) in propositional logic can be adapted to ASP. This is an interesting result as unlike propositional logic, ASP is non-monotonic. For three MSS algorithms we show how they are ported to ASP (Algorithms 1–3). A similar approach, however, could be used for other MSS algorithms.

The algorithms for maximal consistency let us then calculate minimal corrections. For these corrections we assume a general scheme where rules may be added or removed in order to restore consistency. For evaluation we return to the initial motivation of the paper and that is the calculation of corrections to data that make a program inconsistent. A number of instances from various problem classes were considered. The progression-based algorithm (Algorithm 3) turned out to be the most effective overall. Nevertheless, it was not a winner for each of the considered classes. In contrast, the maximum-cardinality approach clearly performed the worst. Here, however, we should point out that we used the default implementation of minimization in clingo and it is the subject of future work to evaluate other algorithms. (A number of MaxSAT algorithms are being adapted to Max-ASP [3, 38]). Overall, the evaluation suggests that a portfolio comprising the different algorithms would provide the best solution.

The paper opens a number of avenues for future work. Irredundancy is needed in other approaches to debugging, e.g. [8, 21, 45]. It is the subject of future work to evaluate the proposed algorithms also in these contexts. The prototype used for the evaluation uses the ASP solver in a black-box fashion. It is likely that integrating the algorithms directly into an ASP solver would give further performance boost. In this respect we believe it would be beneficial to make use of the existing asprin framework [9].

Acknowledgments. This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013 and by FCT funding of LASIGE Research Unit, reference UID/CEC/00408/2013.

References

1. Alviano, M., Dodaro, C.: Anytime answer set optimization via unsatisfiable core shrinking. TPLP 16(5-6), 533–551 (2016), https://doi.org/10.1017/S147106841600020X
2. Alviano, M., Dodaro, C., Ricca, F.: Anytime computation of cautious consequences in Answer Set Programming. TPLP 14(4-5), 755–770 (2014)
3. Andres, B., Kaufmann, B., Matheis, O., Schaub, T.: Unsatisfiability-based optimization in clasp. In: ICLP (Technical Communications). pp. 211–221 (2012)
4. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Practical Aspects of Declarative Languages (PADL) (2005)
5. Baral, C.: Knowledge representation, reasoning and declarative problem solving. Cambridge University Press (2003)
6. Belov, A., Janota, M., Lynce, I., Marques-Silva, J.: Algorithms for computing minimal equivalent subformulas. Artif. Intell. 216, 309–326 (2014)
7. Brain, M., De Vos, M.: Debugging logic programs under the answer set semantics. In: ASP Workshop (2005)
8. Brain, M.: Declarative problem solving using answer set semantics. In: International Conference on Logic Programming (ICLP). vol. 4079, pp. 459–460 (2006)
9. Brewka, G., Delgrande, J.P., Romero, J., Schaub, T.: asprin: Customizing Answer Set Preferences without a headache. In: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence. pp. 1467–1474 (2015)
10. Brummayer, R., Järvisalo, M.: Testing and debugging techniques for answer set solver development. TPLP 10(4-6) (2010)
11. Cabalar, P., Fandinno, J.: An algebra of causal chains. In: Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP) (2013)
12. Console, L., Friedrich, G., Dupré, D.T.: Model-based diagnosis meets error diagnosis in logic programs (extended abstract). In: First International Workshop on Automated and Algorithmic Debugging. pp. 85–87 (1993)
13. Damásio, C.V., Pires, J.M., Analyti, A.: Unifying justifications and debugging for answer-set programs. In: International Conference on Logic Programming (ICLP) (2015), http://ceur-ws.org/Vol-1433/tc_84.pdf
14. Déharbe, D., Fontaine, P., Berre, D.L., Mazure, B.: Computing prime implicants. In: Formal Methods in Computer-Aided Design (FMCAD). pp. 46–52. IEEE (2013)
15. Dodaro, C., Gasteiger, P., Musitsch, B., Ricca, F., Shchekotykhin, K.M.: Interactive debugging of non-ground ASP programs. In: Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 279–293 (2015)
16. Eiter, T., Gottlob, G., Leone, N.: Abduction from logic programs: Semantics and complexity. Theoretical computer science 189(1-2), 129–177 (1997)
17. Faber, W., Vallati, M., Cerutti, F., Giacomin, M.: Solving set optimization problems by cardinality optimization with an application to argumentation. In: European Conference on Artificial Intelligence (ECAI). pp. 966–973. IOS Press (2016)
18. Felfernig, A., Schubert, M., Zehentner, C.: An efficient diagnosis algorithm for inconsistent constraint sets. AI EDAM 26(1), 53–62 (2012)
19. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Wanko, P.: Theory solving made easy with Clingo 5. In: Technical Communications of the 32nd International Conference on Logic Programming, ICLP. pp. 2:1–2:15 (2016)
20. Gebser, M., Kaminski, R., Schaub, T.: Complex optimization in answer set programming. TPLP 11(4-5), 821–839 (2011)

21. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: Conference on Artificial Intelligence (AAAI) (2008)
22. Gelfond, M., Leone, N., Pfeifer, G. (eds.): Logic Programming and Nonmonotonic Reasoning (LPNMR), vol. 1730 (1999)
23. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: ICLP/SLP. pp. 1070–1080 (1988)
24. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Comput. 9(3/4), 365–386 (1991)
25. Greßler, A., Oetsch, J., Tompits, H.: Harvey : A system for random testing in ASP. In: Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 229–235 (2017)
26. Hemery, F., Lecoutre, C., Sais, L., Boussemart, F.: Extracting MUCs from constraint networks. In: European Conference on Artificial Intelligence (ECAI). pp. 113–117 (2006)
27. Junker, U.: QuickXplain: Preferred explanations and relaxations for over-constrained problems. In: AAAI. pp. 167–172 (2004)
28. Lee, J.: A model-theoretic counterpart of loop formulas. In: International Joint Conference on Artificial Intelligence (IJCAI) (2005)
29. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning 40(1), 1–33 (2008)
30. Lloyd, J.W.: Declarative error diagnosis. New Generation Computing 5(2), 133–154 (1987)
31. Marques-Silva, J., Heras, F., Janota, M., Previti, A., Belov, A.: On computing minimal correction subsets. In: International Joint Conference on Artificial Intelligence (IJCAI) (2013)
32. Marques-Silva, J., Janota, M., Belov, A.: Minimal sets over monotone predicates in Boolean formulae. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification (CAV). pp. 592–607 (2013)
33. Marques-Silva, J., Janota, M., Lynce, I.: On computing backbones of propositional theories. In: European Conference on Artificial Intelligence (ECAI). pp. 15–20 (2010)
34. Niemelä, I., Simons, P., Soininen, T.: Stable model semantics of weight constraint rules. In: Gelfond et al. [22], pp. 317–331
35. Nieves, J.C., Osorio, M.: Generating maximal models using the stable model semantics. In: LA-NMR (2007)
36. Nöhrer, A., Biere, A., Egyed, A.: Managing SAT inconsistencies with HUMUS. In: VaMoS. pp. 83–91 (2012)
37. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: On debugging non-ground answer-set programs. TPLP 10(4-6) (2010)
38. Oikarinen, E., Järvisalo, M.: Max-ASP: Maximum satisfiability of answer set programs. In: Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 236–249 (2009)
39. Pereira, L.M., Damásio, C.V., Alferes, J.J.: Debugging by diagnosing assumptions. In: Automated and Algorithmic Debugging: First International Workshop (AADEBUG). pp. 58–74 (1993)
40. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. 32(1), 57–95 (1987)
41. Sakama, C., Inoue, K.: An abductive framework for computing knowledge base updates. Theory and Practice of Logic Programming 3(6), 671–715 (2003)

42. Schulz, C., Satoh, K., Toni, F.: Characterising and explaining inconsistency in logic programs. In: Logic Programming and Nonmonotonic Reasoning (LPNMR). pp. 467–479 (2015)
43. Simons, P.: Extending the stable model semantics with more expressive rules. In: Gelfond et al. [22], pp. 305–316
44. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artif. Intell. 138(1-2), 181–234 (2002)
45. Syrjänen, T.: Debugging inconsistent answer set programs. In: Workshop on Nonmonotonic Reasoning (NMR). pp. 77–83 (2006)
46. Ulbricht, M., Thimm, M., Brewka, G.: Measuring inconsistency in answer set programs. In: Logics in Artificial Intelligence - 15th European Conference, JELIA. pp. 577–583 (2016)
47. Viegas Damásio, C., Analyti, A., Antoniou, G.: Justifications for logic programming. In: LPNMR. pp. 530–542 (2013)
48. Zhu, C.S., Weissenbacher, G., Malik, S.: Post-silicon fault localisation using maximum satisfiability and backbones. In: Formal Methods in Computer-Aided Design (FMCAD). pp. 63–66 (2011)
49. Zhuang, Z., Delgrande, J.P., Nayak, A.C., Sattar, A., et al.: Reconsidering AGM-style belief revision in the context of logic programs. In: European Conference on Artificial Intelligence (ECAI). pp. 671–679 (2016)