Inconsistent Cores for ASP: The Perks and Perils of Non-monotonicity

Johannes K. Fichte[*1], **Markus Hecher**[*2], **Stefan Szeider**[*3]

[1] TU Wien, Research Unit Databases and AI, Vienna, Austria
[2] Computer Science and Artificial Intelligence Lab, Massachusetts Institute of Technology, Cambridge, MA, United States
[3] TU Wien, Research Unit Algorithms and Complexity, Vienna, Austria
johannes.fichte@tuwien.ac.at, hecher@mit.edu, sz@ac.tuwien.ac.at

Abstract

Answer Set Programming (ASP) is a prominent modeling and solving framework. An inconsistent core (IC) of an ASP program is an inconsistent subset of rules. In the case of inconsistent programs, a smallest or subset-minimal IC contains crucial rules for the inconsistency. In this work, we study finding minimal ICs of ASP programs and key fragments from a complexity-theoretic perspective. Interestingly, due to ASP's non-monotonic behavior, also consistent programs admit ICs. It turns out that there is an entire landscape of problems involving ICs with a diverse range of complexities up to the fourth level of the Polynomial Hierarchy. Deciding the existence of an IC is, already for tight programs, on the second level of the Polynomial Hierarchy. Furthermore, we give encodings for IC-related problems on the fragment of tight programs and illustrate feasibility on small instance sets.

1 Introduction

Answer set programming (ASP) is a declarative programming paradigm with roots in non-monotonic reasoning and logic programming (Brewka, Eiter, and Truszczyński 2011). It has many applications in knowledge representation, artificial intelligence, and planning and supports compact problem modeling (Baral 2003; Pontelli et al. 2012). In ASP, a problem is encoded as a set of rules, called a logic program, and evaluated under the stable model semantics (Gelfond and Lifschitz 1988, 1991), where solutions are called *answer sets*. Solvers such as `clingo` (Gebser et al. 2011, 2014), `WASP` (Alviano et al. 2015a), or `DLV` (Alviano et al. 2017) have been developed.

A well-known concept for ASP are unsatisfiable cores, which are also widely used for guiding the search for optimal answer sets (Alviano, Dodaro, and Ricca 2015; Alviano et al. 2015b, 2018a). An *unsatisfiable core* of a given program Π is a subset C of literals that make the program $\Pi \cup \{\leftarrow \neg\ell \mid \ell \in C\}$ under the (literal) assumptions C inconsistent. Besides core-guided optimization, assumptions and unsatisfiable cores are relevant for cautious reasoning (Alviano et al. 2018b), explainability (Alviano et al. 2019), belief revision (Garcia et al. 2018), and forgetting rules (cf. Gonçalves, Knorr, and Leite 2021). However, the concept of an unsatisfiable core is quite strict as it does not support identifying the source of inconsistency in terms of rules but only in terms of atoms. Therefore, we consider the more general concept of an *inconsistent core (IC)*, which for a given program Π is a subset of rules $\Pi' \subseteq \Pi$ that is already inconsistent.

In the context of SAT, the counterpart of ICs are unsatisfiable subsets. There, one is mainly interested in finding *minimal unsatisfiable subsets (MUSes)* of a CNF formula F, which are subsets $F' \subseteq F$ of clauses that are unsatisfiable, but $F' \setminus \{c\}$ is satisfiable for any clause $c \in F'$. MUSes are actively used in product configuration, knowledge-based validation, and hardware and software design and verification (McCarthy 1980; Schlobach et al. 2007; Andraus, Liffiton, and Sakallah 2008; Soh and Inoue 2010; Bendík and Meel 2020; Endriss 2020). Finding MUSes is among the standard solving repertoires in the SAT community, with a long list of works on extracting them (Nadel 2010; Marques-Silva and Lynce 2011; Belov, Manthey, and Marques-Silva 2013; Lagniez and Biere 2013; Belov, Heule, and Marques-Silva 2014; Mencía et al. 2019).

In SAT, the complexity of finding and recognizing MUSes is well studied (Papadimitriou and Wolfe 1988; Szeider 2004; Fleischner, Kullmann, and Szeider 2002). For instance, deciding whether a formula is an MUS is $\mathrm{D^P}$-complete. However, the complexity of problems involving ICs in the setting of ASP is primarily unexplored, despite plenty of investigations on ASP problems of higher complexity (Bogaerts, Janhunen, and Tasharrofi 2016; Amendola, Ricca, and Truszczynski 2019) and considerations on strong inconsistency in ASP (Mencía and Marques-Silva 2020).

Contributions. Here, we chart the complexity map of various computational problems arising from ICs. It turns out that there is an entire landscape of problems involving ICs with a diverse range of complexities up to the fourth level of the *Polynomial Hierarchy (PH)*. Our main contributions are:

1. We establish the computational complexity of finding ICs, minimal inconsistent cores (MICs), and smallest ICs for ASP. Additionally, we provide ASP encodings to compute the respective set of rules for the considered problems.
2. We present detailed complexity results for reasoning problems using ICs, namely, credulous and skeptical

[*] These authors contributed equally.

Fragment	Existence Problems	Reasoning Problems			
	IC, EIC, RIC, MIC, SMIC	CRED$_{\text{MIC}}$	CRED$_{\text{SMIC}}$	SKEP$_{\text{MIC}}$	SKEP$_{\text{SMIC}}$
SAT / non-negative ASP	co-NP	Σ_2^P	Θ_2^P	Π_2^P	Θ_2^P
normal / tight ASP	Σ_2^P	Σ_3^P	Θ_3^P	Π_3^P	Θ_3^P
disjunctive ASP	Σ_3^P	Σ_4^P	Θ_4^P	Π_4^P	Θ_4^P

Table 1: Overview of our complexity results for existence and reasoning problems involving inconsistent cores. Each row shows results regarding a specific class of formulas or programs, respectively, denoted in the first column. The definitions of the classes are as usual and can be found in Section 2; for the less-known fragment of *non-negative programs*, disjunctions in the rule heads are allowed, and atoms in the negative body are forbidden. Problems consider ICs (inconsistent cores), EICs (essential inconsistent cores), MICs (minimal EICs), RICs (relevant EICs), and SMICs (smallest MICs), as defined in Section 3.

reasoning when querying for a subset of the rules. An overview of our complexity results is provided in Table 1.

3. We illustrate the feasibility of our encodings for ICs and MICs on a small set of instances.

Our results reveal that computing ICs can be significantly harder than MUSes: *non-monotonicity causes an additional source of complexity.* In fact, hardness is not caused by cycles (Lifschitz and Razborov 2006); instead, basic properties of non-monotonic reasoning, inherent in the definition of ASP, result in higher complexity. We can already see a significant gap between tight ASP and SAT. Based on that, non-monotonicity motivates the need for further flexibility that allows us to distinguish between rules that should always be in ICs of interest and optional rules.

2 Preliminaries

Computational Complexity. We assume familiarity with standard notions in computational complexity (Papadimitriou 1994), usual complexity classes and the Polynomial Hierarchy. In particular, $\Sigma_0^P = \Pi_0^P = \Delta_0^P = \text{P}$ and for $i \geq 1$ we use $\Sigma_i^P := \text{NP}^{\Sigma_{i-1}^P}$, $\Pi_i^P := \text{co-NP}^{\Pi_{i-1}^P}$, and $\Delta_i^P := \text{P}^{\Sigma_{i-1}^P}$.

Interestingly, there is also a complexity class between $\Sigma_{i-1}^P/\Pi_{i-1}^P$ and Δ_i^P for $i \geq 2$. This class is sometimes denoted by Θ_i^P or $\Delta_i^{P[\log(n)]}$ and therefore seems similar to Δ_i^P, but only permits $\mathcal{O}(\log(n))$ many Σ_{i-1}^P-oracle calls for every instance of size n.

Quantified Boolean Formulas (QBFs). We define *propositional formulas* in the usual way; *literals* are variables or their negations. For a propositional formula F, we denote by $\mathsf{var}(F)$ the set of variables of F. Logical operators $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$ are used in the usual meaning. A *term* is a conjunction of literals, and a *clause* is a disjunction of literals. F is in *conjunctive normal form (CNF)* if F is a conjunction of clauses and F is in *disjunctive normal form (DNF)* if F is a disjunction of terms. In both cases, we identify F by its set of clauses or terms, respectively. We assume that a propositional formula is in CNF, unless stated otherwise. Let $\ell \geq 0$ be an integer. A *quantified Boolean formula* Q is of the form $Q_1 V_1 . Q_2 V_2 . \cdots Q_\ell V_\ell . F$ where $Q_i \in \{\forall, \exists\}$ for $1 \leq i \leq \ell$ and $Q_j \neq Q_{j+1}$ for $1 \leq j \leq \ell - 1$; and where the V_i are disjoint, non-empty sets of propositional variables with $\bigcup_{i=1}^{\ell} V_i = \mathsf{var}(F)$ and F is a propositional formula. We call ℓ the *quantifier depth* of Q and let $\mathsf{matrix}(Q) := F$.

An *assignment* is a mapping $\iota : X \rightarrow \{0,1\}$ defined on a set X of variables. Consider a propositional formula F and an assignment ι on $\mathsf{var}(F)$. Then, for F in CNF, $F[\iota]$ is the propositional formula obtained by removing every $c \in F$ with $x \in c$ and $\neg x \in c$ if $\iota(x) = 1$ and $\iota(x) = 0$, respectively, and by removing from every remaining clause $c \in F$ literals x and $\neg x$ with $\iota(x) = 0$ and $\iota(x) = 1$, respectively. Analogously, for F in DNF values 0 and 1 are swapped. For a given QBF Q and an assignment $\iota : X \rightarrow \{0,1\}$, $Q[\iota]$ is the QBF obtained from Q, where variables $x \in X$ are removed from preceding quantifiers accordingly, and $\mathsf{matrix}(Q[\iota]) := (\mathsf{matrix}(Q))[\iota]$. A propositional formula F *evaluates to true* if there exists an assignment ι for $\mathsf{var}(F)$ such that $F[\iota] = \emptyset$ if F is in CNF or $F[\iota] = \{\emptyset\}$ if F is in DNF. A QBF Q with $Q_1 = \exists$ *evaluates to true* (or is *valid*) if and only if there exists an assignment $\iota : V_1 \rightarrow \{0,1\}$ such that $Q[\iota]$ evaluates to true. If $Q_1 = \forall$, then $Q[\iota]$ evaluates to true if for every assignment $\iota : V_1 \rightarrow \{0,1\}$, $Q[\iota]$ evaluates to true. QSAT$_\ell$ refers to the *problem of deciding validity* for a given a QBF Q of quantifier depth ℓ. The problem is Σ_ℓ^P-complete if $Q_1 = \exists$, and Π_ℓ^P-complete if $Q_1 = \forall$ (Kleine Büning and Lettman 1999; Papadimitriou 1994; Stockmeyer and Meyer 1973).

Answer Set Programming (ASP). We follow standard definitions of propositional ASP (Brewka, Eiter, and Truszczyński 2011; Janhunen and Niemelä 2016). Let ℓ, m, n be non-negative integers such that $\ell \leq m \leq n$ and $a_1, \ldots, a_n$ be distinct propositional atoms. Moreover, we refer by *literal* to a *propositional variable (atom)* or the negation thereof. A *(logic) program* Π is a set of *rules* of the form $a_1 \vee \cdots \vee a_\ell \leftarrow a_{\ell+1}, \ldots, a_m, \neg a_{m+1}, \ldots, \neg a_n$. For a rule r, we let $H_r := \{a_1, \ldots, a_\ell\}$, $B_r^+ := \{a_{\ell+1}, \ldots, a_m\}$, and $B_r^- := \{a_{m+1}, \ldots, a_n\}$. We denote the sets of *atoms* occurring in a rule r or in a program Π by $\mathsf{at}(r) := H_r \cup B_r^+ \cup B_r^-$ and $\mathsf{at}(\Pi) := \bigcup_{r \in \Pi} \mathsf{at}(r)$, respectively. A rule r is *normal* if $|H_r| \leq 1$; r is *non-negative* if $|B_r^-| = 0$. Then, a program Π is *normal* (*non-negative*) if all its rules $r \in \Pi$ are normal (non-negative). The *dependency digraph* D_Π of Π is the directed graph defined on atoms $\bigcup_{r \in \Pi} H_r \cup B_r^+$, where for every rule $r \in \Pi$ atoms $a \in B_r^+$ and $b \in H_r$ are joined by an edge (a, b). A program Π is *tight* if there is no directed cycle in D_Π (Fages 1994).

An *interpretation* I is a set of atoms. I *satisfies* a rule r if $(H_r \cup B_r^-) \cap I \neq \emptyset$ or $B_r^+ \setminus I \neq \emptyset$. I is a *model* of Π if it satisfies all rules of Π, in symbols $I \models \Pi$. For brevity, we view propositional formulas as sets of formulas (e.g., clauses) that need to be satisfied, and analogously use the notion of interpretations, models, and satisfiability. The *Gelfond-Lifschitz (GL) reduct* of Π under I is the program Π^I obtained from Π by first removing all rules r with $B_r^- \cap I \neq \emptyset$ and then removing all $\neg z$ where $z \in B_r^-$ from the remaining rules r (Gelfond and Lifschitz 1991). I is an *answer set* of a program Π if I is a minimal model of Π^I. A program is *consistent* if it has at least one answer-set, otherwise it is *inconsistent*. CONSISTENCY, the problem of deciding whether an ASP program is consistent, is Σ_2^{P}-complete (Eiter and Gottlob 1995). If the input is restricted to normal programs, the complexity drops to NP-completeness (Bidoít and Froidevaux 1991; Marek and Truszczyński 1991). The answer sets of a tight program can be represented by the models of a propositional formula, obtainable in linear time via, e.g., Clark's completion (Clark 1977).

3 Minimal Inconsistent Cores (MIC)

Driven by the definition of unsatisfiable cores and corresponding practical considerations for ASP (Andres et al. 2012; Alviano and Dodaro 2017; Saikko et al. 2018), we formally define the concept of inconsistent cores and the central problems that are the focus of this work.

Definition 1 (Inconsistent Core (IC))**.** *Let* Π *be a given program. Then, an* inconsistent core (for Π) *is an inconsistent subset* $\Pi' \subseteq \Pi$.

Bear in mind that semantics of logic programs is non-monotonic, which has the consequences briefly shown below.

Example 2. *Consider the programs* $\{\leftarrow \neg a\} \subseteq \{a \leftarrow; \leftarrow \neg a\}$ *and observe that the first is inconsistent while the second is consistent.*

So, a program might be consistent, but a subset of the rules can form an IC. This is different from propositional formulas and stems from the non-monotonicity of logic programs, indicating that compared to the answer sets of a program, we might obtain additional answer sets after adding rules to the program. However, the converse is not true: if a program is an IC, then obviously, some subset is an IC as well.

Interestingly, the non-monotonicity is precisely, why every subset has to be analyzed for finding an IC, already for a normal (and even tight) logic program, which, surprisingly, is in stark contrast to finding unsatisfiable subsets of propositional CNF formulas. This result is established in the following theorem.

Theorem 3. *The problem of deciding whether a tight logic program* Π *admits an* IC *is* Σ_2^{P}*-complete.*

Proof (Sketch). Membership: We sketch an algorithm that shows Σ_2^{P} membership. First, we guess a set $R \subseteq \Pi$. Then, we check whether R is inconsistent, which can be done in co-NP, i.e., it requires one call to the NP oracle. Consequently, the algorithm is in Σ_2^{P}.

Hardness: We reduce from the Σ_2^{P}-complete problem QSAT$_2$. Take any instance $I = \exists V_1.\forall V_2.F$ with $F = \{d_1, \ldots, d_n\}$ in DNF. From this, we define a program Π, constructed below using fresh variables $u, u_1, \ldots, u_n$ as well as $\tilde{a}$ for every variable $a \in V_2$. We define $\Pi := \{\, a \leftarrow \mid a \in V_1 \,\} \cup \{\, b \leftarrow \neg\tilde{b}; \tilde{b} \leftarrow \neg b \mid b \in V_2 \,\} \cup \{\, u_i \leftarrow \bar{l_j} \mid 1 \leq i \leq n, d_i = l_1 \wedge \ldots \wedge l_o, 1 \leq j \leq o \,\} \cup \{u \leftarrow u_1, \ldots, u_n\} \cup \{\leftarrow \neg u\}$ with $\bar{l} := a$ if $l = \neg a$ for some $a \in \mathsf{var}(F)$ and otherwise, if $l \in V_1$ then $\bar{l} := \neg a$ and $\bar{l} := \tilde{a}$ if $l \in V_2$. Then, we briefly sketch that $\exists V_1.\neg\exists V_2.\neg F$ is valid if and only if Π is inconsistent for a selection of rules $a \leftarrow$. $\Longrightarrow$: Let $\alpha : V_1 \rightarrow \{0, 1\}$ be an assignment such that $(\exists V_1.\neg\exists V_2.\neg F)[\alpha]$ evaluates to true. Then we have that $\Pi \setminus \{\, a \leftarrow \ \mid a \in V_1, \alpha(a) = 0 \,\}$ is inconsistent. $\Longleftarrow$: Assume that Π admits an IC $\Pi' \subseteq \Pi$. From this, we define an assignment α, where for every $a \in V_1$ we set $\alpha(a) := 1$ whenever $\{a \leftarrow\} \in \Pi'$ and $\alpha(a) := 0$ otherwise. Then, assuming that $(\exists V_1.\neg\exists V_2.\neg F)[\alpha]$ evaluates to false contradicts that Π' is an IC. $\square$

Observe that when slightly adapting the decision problem, such that affirmative answers are expected for inconsistent programs only, instead of Σ_2^{P}-completeness as in Theorem 3, we obtain co-NP-completeness. So, in this case, checking inconsistency suffices.

The non-monotonicity of ASP motivates also the need for further flexibility that allows us to distinguish between rules that should always be in ICs of interest and rules that are optional. Let therefore Π be a given program, $E \subseteq \Pi$ a subset of (essential) rules, and $\Pi' \subseteq \Pi$ an IC. Then, if $E \subseteq \Pi'$, we call Π' an *essential* IC *(*EIC*)*.

Among essential ICs, one might be curious about finding a minimal one. We study different notions of minimality. We say Π' is a *minimal* EIC *(*MIC*)* if every strict subset $\Pi'' \subsetneq \Pi'$ with $\Pi'' \supseteq E$ is consistent. Further, we say Π' is a *relevant* EIC *(*RIC*)* if Π' is a EIC of Π, but $\Pi' \setminus \{r\}$ is consistent for every $r \in \Pi' \setminus E$. Finally, Π' is a *smallest* MIC *(*SMIC*)* if there is no EIC Π'' of Π and E with $|\Pi''| < |\Pi'|$.

Example 4. *Consider the program* $\Pi := \{a \leftarrow b, \neg c; b \vee d \leftarrow; d \vee a \leftarrow; \leftarrow a, \neg c; \leftarrow d; c \leftarrow\}$. *Observe that* Π *is consistent, since* $\{a, b, c\}$ *is an answer set, i.e., there is no* EIC *of* Π *containing* $E = \Pi$. *However,* Π *admits an* IC, *e.g.,* $\Pi \setminus \{c \leftarrow\}$. *There are two* MIC*s of* Π *containing* $\{\leftarrow d\}$, *namely* $\Pi_1 = \{a \leftarrow b, \neg c; b \vee d \leftarrow; \leftarrow a, \neg c; \leftarrow d\}$ *and* $\Pi_2 = \{d \vee a \leftarrow; \leftarrow a, \neg c; \leftarrow d\}$. *However, the latter is the only* SMIC. *There is no* EIC *of* Π *containing* $\{c \leftarrow\}$.

Corollary 5 ($\star$)**.** *The problem of deciding whether a tight logic program* Π *admits an* (S)MIC *for a set* $E \subseteq \Pi$ *is* Σ_2^{P}*-complete.*

SAT– A Monotonic ASP Fragment. The difference between SAT and ASP in terms of the computation of unsatisfiable subsets/inconsistent cores reveals that non-monotonicity can significantly harden the computation and cause an additional source of complexity. Therefore, since there is already a gap between tight ASP and SAT, we study our problems of interest also for SAT. To this end, propositional formulas in CNF can be viewed as the fragment of

[1]Proofs for theorems marked by "$\star$" are in an extended version.

non-negative programs, i.e., where negation is not permitted[1]. Consequently, we capture propositional formulas by also considering the fragment of non-negative programs.

Observation 6 ($\star$). *A propositional formula can be represented as a non-negative program such that there is a bijection between the models and the answer sets.*

Interestingly, for the fragment of non-negative programs, deciding the existence of an inconsistent core is easier.

Observation 7 ($\star$). *The problem of deciding whether a non-negative program* Π *admits an* IC *is co*-NP*-complete.*

Characterizing the Computation of MICs

We define decision problems that express different reasoning problems involving MICs for a given program Π. Credulous and skeptical reasoning problems ask whether some or all (S)MIC, respectively, contain certain rules $Q \subseteq \Pi$ of interest, called the *query*.

Definition We define credulous and skeptical reasoning problems as follows.

Problem:	$\text{CRED}_{\text{(S)MIC}}$
Input:	Program Π, $E \subseteq \Pi$, query $Q \subseteq \Pi$
Output:	Does there exist an (S)MIC $\Pi' \supseteq E$ of Π, where $\Pi' \supseteq Q$ holds?

Problem:	$\text{SKEP}_{\text{(S)MIC}}$
Input:	Program Π, $E \subseteq \Pi$, $Q \subseteq \Pi$
Output:	Does every (S)MIC $\Pi' \supseteq E$ of Π fulfill $\Pi' \supseteq Q$?

Example 8. *Credulous and skeptical reasoning enables reasoning based on the computation of certain* MIC*s of interest (fulfilling a query). Recall* Π*,* Π_1*, and* Π_2 *from Example 4. Assume* $E := \{a \leftarrow b, \neg c\}$*. Then, the problem* $\text{CRED}_{\text{SMIC}}$ *for* Π_1*,* E *and query* Π_1 *actually is answered affirmatively, since* $E \not\subseteq \Pi_2$*. Further, observe that every* IC *of* Π *relies on the rule* $\{\leftarrow d\}$*. Consequently, the problem* SKEP_{MIC} *for* Π *and query* $\{\leftarrow d\}$ *is also answered affirmatively.*

Interestingly, non-monotonicity as briefly discussed in Example 2 is also the reason, why $\text{CRED}_{\text{(S)MIC}}$ and $\text{SKEP}_{\text{(S)MIC}}$ are harder for a given non-empty query Q. So, in case a non-empty query is used (see Section 4), these problems are harder compared to the case when using a trivial query. Observe that, therefore, the query Q in general is crucial; it is different from the set E of essential rules and cannot be merged into E.

Encodings for Normal Programs

In this section, we present ASP encodings for computing MICs and SMICs. First, we start with the case of tight programs, which can then be extended to normal programs as well. Unfortunately, due to the hardness of the studied problems, it is not possible to further lift these encodings to disjunctive programs, which we discuss in the next section.

[1]Disjunctions in the fragment of non-negative programs can be analogously turned into rules without disjunction using negation (shifting). However, their behavior for IC-problems is different.

Computing EICs. With ASP-Core-2 (Calimeri et al. 2020), an extended syntax for ASP that is supported by main solvers, we can easily compute EICs. Note that the plain (decision) problem of deciding whether there exists an IC for a propositional formula boils down to deciding unsatisfiability, which is co-NP-complete. However, as already discussed in Theorem 3, the situation is completely different for normal (or tight) logic programs, due to the non-monotonicity and without restriction to inconsistent programs.

Listing 1 shows an encoding for tight logic programs. We assume that the given program Π is specified using the binary predicates body and head, where $\mathsf{body}(r,l)$ or $\mathsf{head}(r,a)$ indicate that the body of r contains literal l or its head contains atom a, respectively. Further, the essential rules are given using e, i.e., $\mathsf{e}(r)$ specifies that $r \in \Pi$ is essential. Then, the output of Listing 1 is given using predicate ic, which is a unary predicate giving a rule r part of the computed IC. The idea of this encoding is to guess ICs that contain essential rules. Then, we guess among all assignments and ensure that every assignment does not satisfy the guessed IC, which uses saturation (Eiter and Gottlob 1995).

```
% INPUT: Program using body/2, head/2 and
     essential rules by means of e/1.
% OUTPUT: IC using ic/1.

% Auxiliary predicates, '';'' denotes ''or''
rule(R)   ← body(R,_) ; head(R,_).
atom(|A|) ← body(_,A) ; head(_,A).

% Pick essential rules
ic(R) ← e(R).
% Pick subset of rules
{ ic(R) : rule(R) }.

% Guess assignment
assign(A) ∨ assign(-A) ← atom(A).

% Check inconsistency
sat(R, A) ← head(R,A).
sat(R,-A) ← body(R,A).
bodyusat(R) ← assign(-X), body(R,X).
incons ← ic(R), assign(-X) : sat(R,X).
incons ← assign(X), X>0, bodyusat(R) :
     ic(R),head(R,X).
← not incons.
% Saturate
assign(A)  ← incons, atom(A).
assign(-A) ← incons, atom(A).
#show ic/1.
```

Listing 1: Encoding for EICs of tight programs

Computing RICs. By slightly modifying the encoding of Listing 1, we compute relevant ICs. Therefore, we need to add the following encoding to the listing above. The idea of Listing 2 is to check that when we remove an arbitrary rule from the IC, it is not an IC any more.

```
% IC atoms
iat(|A|) ← atom(A), body(R,A) : ic(R).
iat(|A|) ← atom(A), head(R,A) : ic(R).
```

```
% Guess assignment for IC without R
assign(A,R) ∨ assign(-A,R) ← iat(A), ic(R).
% Check consistency for IC without R
← ic(R), ic(R2), R ≠ R2,
        assign(-A,R) : sat(R2,A).
% Check provability for IC without R
← assign(X,R), X > 0, bodyusat(R2) :
     ic(R2),R2≠R,head(R2,X).
```

Listing 2: Encoding for RICs of tight programs

Computing SMICs. For computing an SMIC, we take the encoding of Listing 1 and the following line to minimize the cardinality of computed ICs. This can be achieved using cost minimization (Gebser et al. 2012; Calimeri et al. 2020), an extension that can reach problems up to Δ_3^{P}.

```
% Minimize the IC -> SMIC
#minimize{ 1,C : ic(C)}.
```

Listing 3: Obtain SMICs by Listing 1 and Minimizing ICs

Extending Encodings to Normal Programs. The encodings above can be extended to normal programs by using level mappings (e.g., Lin and Zhao 2004; Janhunen 2006). There are known implementations for converting normal programs to tight programs, see, e.g., lp2lp or lp2atomic[2].

Extending to Credulous and Skeptical Reasoning. The encodings above also address the problems $\mathrm{CRED}_{\mathrm{SMIC}}$ and $\mathrm{SKEP}_{\mathrm{SMIC}}$, since the query Q can be directly addressed via credulous and skeptical reasoning over the answer sets, respectively. Thereby, the query needs to be specified on top of atoms over the ic predicate and one asks whether Q is contained in some answer set or any answer set for $\mathrm{CRED}_{\mathrm{SMIC}}$ or $\mathrm{SKEP}_{\mathrm{SMIC}}$, respectively.

4 Complexity Landscape

Next, we present detailed complexity results for the reasoning problems above. Thereby, we give an analysis of credulous reasoning, which we then extend to skeptical reasoning.

Credulous reasoning

SAT and Normal ASP. First, we show the complexity for reasoning problems on propositional satisfiability and the related ASP fragment of non-negative programs.

Proposition 9 ($\star$). *The problem* $\mathrm{CRED}_{\mathrm{MIC}}$ *for a propositional formula F, a set $E \subseteq F$, and a query $Q \subseteq F$ is Σ_2^{P}-complete.*

Corollary 10 ($\star$). *The problem* $\mathrm{CRED}_{\mathrm{MIC}}$ *for a non-negative program Π, a set $E \subseteq \Pi$, and a query $Q \subseteq \Pi$ is Σ_2^{P}-complete.*

Interestingly, for normal (and tight) programs, credulous reasoning is on the third level of the Polynomial Hierarchy.

Theorem 11 ($\star$). *The problem* $\mathrm{CRED}_{\mathrm{MIC}}$ *for a normal logic program Π, a set $E \subseteq \Pi$, and a query $Q \subseteq \Pi$ is Σ_3^{P}-complete.*

Computing SMICs is even slightly harder, as shown below.

[2]See http://www.tcs.hut.fi/Software/lp2sat/.

Theorem 12. *The problem* $\mathrm{CRED}_{\mathrm{SMIC}}$ *for a normal logic program Π, set $E \subseteq \Pi$ and query $Q \subseteq \Pi$ is Θ_3^{P}-complete.*

Proof (Sketch). Membership is obtained using binary search over the costs $1, \ldots, n$, which in the worst case requires $\mathcal{O}(\log n)$ many Σ_2^{P} oracle calls. In each call with cost k, we need to guess a set R with $E \subseteq R \subseteq \Pi$ such that $|R| = k + |E|$, and check whether R is an IC. This is indeed in Σ_2^{P}, cf. Theorem 3. If we found the smallest cost k, where R is an IC, we again search via a Σ_2^{P} oracle call an IC R' with $E \subseteq R' \subseteq \Pi$ and $|R| = k + |E|$, but such that $Q \subseteq R'$. If such an R' exists, we answer affirmatively, otherwise the answer is no.

Hardness: We show hardness by reducing from PARITY(QSAT_2), where we have been given a sequence of QBF formulas $I_1, \ldots, I_n$ each of the form $\exists V_1^i.\forall V_2^i.F_i$ with F_i being in DNF and consisting of at most n_i terms such that $\mathsf{var}(I_j) \cap \mathsf{var}(I_k) = \emptyset$ for $1 \leq j < k \leq n$. This sequence is a positive instance of PARITY(QSAT_2), whenever there is an i that is odd with $1 \leq i \leq n-1$ such that $I_1, \ldots, I_i$ are satisfiable, and $I_{i+1}, \ldots, I_n$ are unsatisfiable. This problem PARITY(QSAT_2) is known to be Θ_3^{P}-complete (Eiter and Gottlob 1997; Wagner 1987). Without loss of generality, we assume n to be even, which can be easily obtained by adding a trivial invalid formula I_{n+1} at the end, if this is not the case. Further, we assume that I_1 is valid, which can be achieved by adding two trivially valid formulas at the beginning of the sequence. We refer by $V := \mathsf{var}(F_1 \cup \ldots \cup F_n)$ and use auxiliary variables $\tilde{v}$ for every $v \in V \setminus \{\, V_1^i \mid 1 \leq i \leq n \,\}$ as well as s_i and u_i to indicate validity and invalidity for instance I_i with $1 \leq i \leq n$. We construct a program E by:

$v \vee \tilde{v} \leftarrow$ for every $v \in V \setminus \{\, V_1^i \mid 1 \leq i \leq n \,\}$,
$s_i \leftarrow l_1, \ldots, l_o$ for every $1 \leq i \leq n, (l_1 \wedge \ldots \wedge l_o) \in I_i$,

Then, we define rules P for obtaining invalidity, which have a strong "penalty" that depends on the number of rules U defined afterwards. The set P consist of the following rules, using penalty atoms p_i^k for every $1 \leq i \leq n$ as well as $1 \leq k \leq 2\,|U| - i$.

$u_i \leftarrow p_i^1, \ldots, p_i^k, s_{i-1}$ for every $2 \leq i \leq n, k = 2\,|U| - i$,
$u_i \leftarrow p_i^1, \ldots, p_i^k, u_{i-1}$ for every $2 \leq i \leq n, k = 2\,|U| - i$,
$p_i^k \leftarrow$ for every $1 \leq i \leq n, 1 \leq k \leq 2\,|U| - i$.

Finally, we define U consisting of the following rules, which use penalty atoms p_1 and p_2.

$v \leftarrow$ for every $v \in V_1^i, 1 \leq i \leq n$,
$\leftarrow s_n$; $\leftarrow u_n, s_i, \neg s_j, s_k$ for every $1 \leq i < j < k \leq n$,
$\leftarrow u_n, s_i, u_{i+1}$ for every $1 \leq i \leq n, i = 2k$,
$\leftarrow u_n, p_1, p_2$, and $p_1 \leftarrow$; $p_2 \leftarrow$.

We define $\Pi := E \cup U \cup P \cup Q$ with $Q := \{\leftarrow u_n, p_1, p_2\}$. The construction is such that under every decision $v \leftarrow$ for $v \in V_1^i$ we prefer those that derive less atoms u_i and those of larger index i. So, whenever the same number of atoms u_i is derived, we "prefer" those of larger index. Then, the remaining rules of U ensure that it is cheaper (due to penalty atoms p_1, p_2) to obtain a negative ("no") instance. In other words, the only case where the SMIC contains Q is, if indeed the instance is a positive instance. So, we have that $I_1, \ldots, I_n$ is a positive instance, whenever Q is required in an SMIC of Π containing E. $\square$

Proposition 13 ($\star$)**.** *The problem* $\textsc{Cred}_{\text{SMIC}}$ *for a propositional formula F, set $E \subseteq F$, and a query $Q \subseteq F$ is Θ_2^{P}-complete.*

Disjunctive Asp. For arbitrary programs including disjunctions, already the decision of whether the program admits an IC is on the third level of the Polynomial Hierarchy.

Theorem 14 ($\star$)**.** *The problem of deciding whether a logic program Π admits an* IC *is Σ_3^{P}-complete.*

Then, the complexity of reasoning on top of minimal MICs is increased by one level.

Theorem 15 ($\star$)**.** *The problem* $\textsc{Cred}_{\text{MIC}}$ *for a logic program Π, a set $E \subseteq \Pi$, and a query $Q \subseteq \Pi$ is Σ_4^{P}-complete.*

Again, computing smallest MICs is slightly harder.

Theorem 16. *The problem* $\textsc{Cred}_{\text{SMIC}}$ *for a logic program Π, a set $E \subseteq \Pi$, and query $Q \subseteq \Pi$ is Θ_4^{P}-complete.*

Proof (Sketch). Membership: Showing membership works analogously to Theorem 12.

Hardness: Let $I_1, \ldots, I_n$ be an instance of PARITY(QSAT$_3$) each of the form $\exists V_1^i.\forall V_2^i.\exists V_3^i.F_i$, where F_i consists of n_i clauses. Then, $I_1, \ldots, I_n$ is positive whenever there is an i that is odd with $1 \leq i \leq n-1$ such that $I_1, \ldots, I_i$ are valid, and $I_{i+1}, \ldots, I_n$ are invalid. We refer by $V := \mathsf{var}(F_1 \cup \ldots \cup F_n)$ and use auxiliary variables $\tilde{v}$ for every $v \in V \setminus \{ V_1^i \mid 1 \leq i \leq n \}$ as well as s_i and u_i to indicate validity and invalidity for instance I_i with $1 \leq i \leq n$. Further, for every u_i, we use auxiliary variable f_i for storing that the evaluation of $\forall V_2^i.\exists V_3^i.F_i$ is false, i.e., that we have $\exists V_2^i.\forall V_3^i.\neg F_i$. Similar to the proof of Theorem 12, we construct program E as follows.

$v \vee \tilde{v} \leftarrow$ for every $v \in V \setminus \{ V_1^i \mid 1 \leq i \leq n \}$,

$f_i \leftarrow \bar{l}_1, \ldots, \bar{l}_o$ for every $1 \leq i \leq n, (l_1 \vee \ldots \vee l_o) \in I_i$, where for $l = \neg a$: $\bar{l} := a$, for $l \in V_1^i$: $\bar{l} := \neg a$, and $\bar{l} := \tilde{a}$ if $l \in V \setminus V_1^i$,

$v \leftarrow f_i$ for every $1 \leq i \leq n, v \in V_3^i$,

$\tilde{v} \leftarrow f_i$ for every $1 \leq i \leq n, v \in V_3^i$, and

$s_i \leftarrow \neg f_i$ for every $1 \leq i \leq n$.

Then, similar to above, we define rules P for obtaining invalidity, with a "penalty" that depends on the number of rules U defined thereafter. The set P consist of the following rules, using penalty atoms p_i^k for every $1 \leq i \leq n$ as well as $1 \leq k \leq 2|U| - i$.

$u_i \leftarrow p_i^1, \ldots, p_i^k, s_{i-1}$ for every $2 \leq i \leq n, k = 2|U| - i$,

$u_i \leftarrow p_i^1, \ldots, p_i^k, u_{i-1}$ for every $2 \leq i \leq n, k = 2|U| - i$,

$p_i^k \leftarrow$ for every $1 \leq i \leq n, 1 \leq k \leq 2|U| - i$.

Finally, we define U consisting of the following rules, which use penalty atoms p_1 and p_2.

$v \leftarrow$ for every $v \in V_1^i, 1 \leq i \leq n$,

$\leftarrow s_n; \leftarrow u_n, s_i, \neg s_j, s_k$ for every $1 \leq i < j < k \leq n$,

$\leftarrow u_n, s_i, u_{i+1}$ for every $1 \leq i \leq n, i = 2k$,

$\leftarrow u_n, p_1, p_2$, and $p_1 \leftarrow; p_2 \leftarrow$.

We define $\Pi := E \cup U \cup P \cup Q$ with $Q := \{\leftarrow u_n, p_1, p_2\}$. Similar to the proof of Theorem 12, $I_1, \ldots, I_n$ is a positive instance whenever Q is required in an SMIC of Π that contains E. $\square$

Instance Set	Prob	$t[s]$	$t_{\text{avg}}[s]$	size
(S1): reach	EIC	10672.9	8.1	632
	RIC	10658.8	8.1	632
	SMIC	10823.3	8.3	84
(S2): edge_col	EIC	16038.8	13.4	1352
	RIC	16040.9	13.4	1352
	SMIC	16260.4	13.6	207
(S3): vertex_col	EIC	5996.9	5.0	272
	RIC	5996.3	5.0	272
	SMIC	6010.1	5.0	63

Table 2: Solved instances by instance set and problem. $t[s]$ refers to the total running time on the solved instances in hours, $t_{\text{avg}}[s]$ refers to the average running time of an instance, and size refers to the median size of the found IC.

Complexity for Skeptical SMIC

For skeptical reasoning, we obtain the following results.

Theorem 17 ($\star$)**.** *The problem* $\textsc{Skep}_{\text{MIC}}$ *for a normal logic program Π, a set $E \subseteq \Pi$, and a query $Q \subseteq \Pi$ is Π_3^{P}-complete.*

For arbitrary programs, complexity increases by one level.

Theorem 18 ($\star$)**.** *The problem* $\textsc{Skep}_{\text{MIC}}$ *for a logic program Π, a set $E \subseteq \Pi$ and a query $Q \subseteq \Pi$ is Π_4^{P}-complete.*

Further, skeptical reasoning on smallest MICs also increases by one level compared to tight or normal programs.

Theorem 19 ($\star$)**.** *The problem* $\textsc{Skep}_{\text{SMIC}}$ *for a (normal) program Π, $E \subseteq \Pi$, and query $Q \subseteq \Pi$ is (Θ_3^{P}) Θ_4^{P}-complete.*

5 Preliminary Empirical Results

Our primary interest is to obtain a basic understanding and an initial practical indicator for the hardness and solvability of the various problem versions arising from inconsistent cores in Asp. To study whether one can obtain inconsistent cores, we evaluated our encodings as given in Listings 1–3 using the solver `clingo`[3] We follow standard guidelines for empirical evaluations (van der Kouwe et al. 2018).

Design of Experiment. We study three questions:

Q1 are solvers capable of outputting EICs, RICs, and SMICs on tight/normal instances using our encodings?

Q2 is there a notable difference in computing SMICs over RICs in terms of runtime vs. size on these instances? and

Q3 how large are the computed cores?

Solver, Encodings, and Instances. In contrast to SAT, e.g., (Belov, Manthey, and Marques-Silva 2013; Lagniez and Biere 2013), no dedicated solvers or instances for IC-related problems exist. For tight/normal programs, we use our encoding from Section 3 in combination with level mappings (Lin and Zhao 2004; Janhunen 2006) and a state-of-the-art Asp solver such as `clingo`, which we use in version 5.5.1. We refrain from establishing alternative encodings or using other solvers for performance comparison because we aim for a starting point to find ICs and not to establish the most competitive solving technique. Our focus

[3]See https://github.com/daajoe/asp_micer for more details.

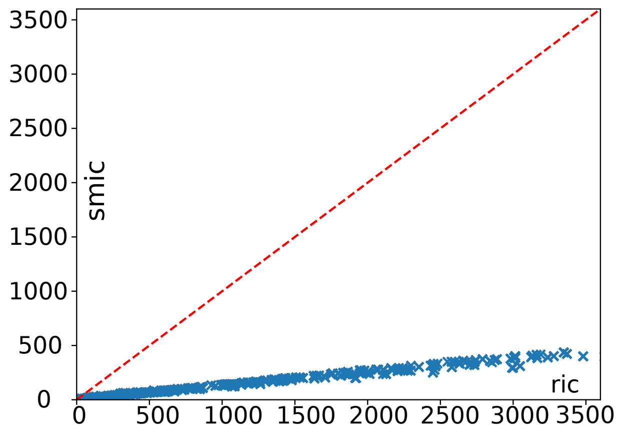

Figure 1: Comparing individually the size of solved instances for RICs and SMICs on Set (S1). The dashed line states identity, below the line indicates that SMIC is smaller.

on selecting a set of input programs is to consider (a) a sufficiently large number of tight/normal instances and avoid compilations of extended rules and (b) show ASP-specific features. Therefore, we take as instances three encodings on real-world graphs of public transport networks from all over the world, used in the PACE'16 and '17 challenges (Dell et al. 2017) and recent works on ASP (Eiter, Hecher, and Kiesel 2021). Set (S1) consists of grounded instances on an encoding of a prototypical ASP domain with reachability and use of transitive closure; (S2) an edge coloring encoding; and (S3) a vertex coloring encoding. In total, the transit instances consist of 561 full networks and 2553 subgraphs. For each instance, we assume the station with the smallest and largest index to be the start and end stations, respectively. We restricted the input to instances of at most 5KB due to a potentially high number of cycles, resulting in 1310 instances in total for (S1) and 1199 instances in total for (S2) and (S3). We selected essential rules based on structural properties. For (S1), we selected the rules on choices for edges and for (S2) and (S3) we picked choices for edges or vertices. Consideration (a) from above prevents us from using ASP competition instances. Instances from early competitions are likely not large enough to see notable differences between the problems (Gebser et al. 2007); later competition instances widely employ optimization or focus on ASP systems (Gebser, Maratea, and Ricca 2015).

Platform, Measure, and Resource Enforcements. We gathered results on RHEL7.7 Linux with kernel 3.10.0-1127.19.1.el7. We evaluated the encodings on machines with 2 sockets equipped with Intel E5-2680v3 CPUs of 12 physical cores and 64GB RAM each at 2.50GHz base frequency. We run at most 10 solvers on one node, set a timeout of 600s, and limited available RAM to 6GB per instance and solver. Note that we excluded grounding times to exceed the 600s and did not restrict the runtime for the grounder as we are primarily interested in the runtime of the solving process.

Experimental Results. We list our results in Table 2 as the number of solved instances, runtime, and size of the obtained ICs. For the set (S1), we can obtain results for all instances and see only a small difference between the different problems. This is expected as we restrict instances to smaller ones. We see a small difference between the number of solved instances between problems, but grounding time increases significantly, whereas the size of ICs drops to 1/10 when computing SMICs. For the sets (S2) and (S3), we observe that we require higher total runtime but only a slightly higher individual runtime. Grounding time increases notably. Again, the size of the ICs is much larger than SMIC, and while the running times of RIC and SMIC are similar, the size is almost one order lower. These results answer our Q1 affirmatively, and we can obtain a notable number of ICs, RICs, and SMICs using the presented encoding and state-of-the-art ASP solvers. To address Q3, we turn our attention again to Table 2. We can see that the median size depends on the problem. To answer Q3, we directly compare runtime and size of RICs for Set (S1) in Figure 1. The runtime for finding RICs and SMICs are overall quite similar. However, SMICs can be notably smaller.

Summary. Our results provide an initial idea on the hardness of computing EICs, RICs, and SMICs. The practical results align with theoretical expectations; encodings for EIC are easier to solve than RIC and SMIC. Similar to the theoretical complexity, we see that runtimes of RIC and SMIC are in similar range. For future experiments, considering additional instances and more fine-tuned encodings or using QBF solvers to obtain solutions for higher-level problems might be interesting.

6 Conclusion and Future Work

We introduced the notions of inconsistent cores (ICs) of an ASP program and studied their computational complexity. We expect that inconsistent cores can be useful for debugging programs, where smallest inconsistent cores containing certain essential rules are required. Further, computing the smallest inconsistent core containing required facts can be interesting when investigating large search spaces. Finding minimal ICs of ASP programs is far more complex than SAT, even for key fragments of programs with close connections to SAT. Interestingly, deciding the existence of an IC is already for tight programs on the second level of the PH. Due to the non-monotonic behavior of ASP, also consistent programs admit ICs. From this property, an entire landscape of problems involving ICs naturally arises. We see a diverse range of complexities up to the fourth level of the PH. In addition, we give encodings for problems on the fragment of tight programs and illustrate feasibility on small instance sets. Our results are summarized in Table 1.

We believe that our initial analysis provides only a starting point and asks for more detailed investigations into the computational complexity of ICs in ASP. We expect interesting insights from considering restricted classes of programs similar to the detailed trichotomy of decision and reasoning problems in answer set programming by Truszczynski (2011). There are also stronger notions of inconsistent cores, dealing with non-monotonicity differently (Ulbricht, Thimm, and Brewka 2020), where we expect synergies.

Acknowledgments

This work has been supported by the Austrian Science Fund (FWF), Grants J 4656 and P32830, the Society for Research Funding in Lower Austria (GFF NÖ) grant ExzF-0004, as well as the Vienna Science and Technology Fund, Grant WWTF ICT19-065.

References

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.; Wallner, J.; and Xiao, G. 2013. The Fourth Answer Set Programming Competition: Preliminary Report. In *LPNMR'13*, volume 8148, 42–53. Springer.

Alviano, M.; Calimeri, F.; Dodaro, C.; Fuscà, D.; Leone, N.; Perri, S.; Ricca, F.; Veltri, P.; and Zangari, J. 2017. The ASP System DLV2. In *Procs. of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'17)*, volume 10377 of *LNCS*, 215–221. Springer.

Alviano, M.; and Dodaro, C. 2017. Unsatisfiable Core Shrinking for Anytime Answer Set Optimization. In *Procs. of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI 2017)*, 4781–4785. ijcai.org.

Alviano, M.; Dodaro, C.; Fichte, J. K.; Hecher, M.; Philipp, T.; and Rath, J. 2019. Inconsistency Proofs for ASP: The ASP – DRUPE Format. *TPLP*, 19(5-6): 891—907.

Alviano, M.; Dodaro, C.; Järvisalo, M.; Maratea, M.; and Previti, A. 2018a. Cautious reasoning in ASP via minimal models and unsatisfiable cores. *TPLP*, 18(3-4): 319–336.

Alviano, M.; Dodaro, C.; Järvisalo, M.; Maratea, M.; and Previti, A. 2018b. Cautious reasoning in ASP via minimal models and unsatisfiable cores. *Theory and Practice of Logic Programming*, 18(3-4): 319–336.

Alviano, M.; Dodaro, C.; Leone, N.; and Ricca, F. 2015a. Advances in WASP. In *Proc. of the 13th Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, 40–54. Springer.

Alviano, M.; Dodaro, C.; Marques-Silva, J.; and Ricca, F. 2015b. Optimum stable model search: algorithms and implementation. *J. Logic Comput.*, 30(4): 863–897.

Alviano, M.; Dodaro, C.; and Ricca, F. 2015. A MaxSAT algorithm using cardinality constraints of bounded size. In *Procs. of the 24th International Joint Conference on Artificial Intelligence (AAA'15)*.

Amendola, G.; Ricca, F.; and Truszczynski, M. 2019. Beyond NP: Quantifying over Answer Sets. *Theory and Practice of Logic Programming*, 19(5-6): 705–721.

Andraus, Z. S.; Liffiton, M. H.; and Sakallah, K. A. 2008. Reveal: A Formal Verification Tool for Verilog Designs. In *Logic for Programming, Artificial Intelligence, and Reasoning*, 343–352. Springer.

Andres, B.; Kaufmann, B.; Matheis, O.; and Schaub, T. 2012. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, volume 17 of *LIPIcs*, 211–221. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Baral, C. 2003. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, Cambridge. ISBN 0-521-81802-8.

Belov, A.; Heule, M. J. H.; and Marques-Silva, J. 2014. MUS Extraction Using Clausal Proofs. In *Theory and Applications of Satisfiability Testing – SAT 2014*, 48–57. Springer.

Belov, A.; Manthey, N.; and Marques-Silva, J. 2013. Parallel MUS Extraction. In *Theory and Applications of Satisfiability Testing – SAT 2013*, 133–149. Springer. ISBN 978-3-642-39071-5.

Bendík, J.; and Meel, K. S. 2020. Approximate Counting of Minimal Unsatisfiable Subsets. In *Computer Aided Verification*, 439–462. Springer International Publishing. ISBN 978-3-030-53288-8.

Bidoít, N.; and Froidevaux, C. 1991. Negation by default and unstratifiable logic programs. *Theoretical Computer Science*, 78(1): 85–112.

Bogaerts, B.; Janhunen, T.; and Tasharrofi, S. 2016. Stable-unstable semantics: Beyond NP with normal logic programs. *Theory and Practice of Logic Programming*, 16(5-6): 570–586.

Brewka, G.; Eiter, T.; and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM*, 54(12): 92–103.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. ASP-Core-2 Input Language Format. *Theory Pract. Log. Program.*, 20(2): 294–309.

Calimeri, F.; Ianni, G.; and Ricca, F. 2014. The third open answer set programming competition. *TPLP*, 14: 117–135.

Clark, K. L. 1977. Negation as Failure. In *Logic and Data Bases*, Advances in Data Base Theory, 293–322. Plemum Press.

Dell, H.; Komusiewicz, C.; Talmon, N.; and Weller, M. 2017. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In *IPEC'17*, LIPIcs, 30:1—30:13. Dagstuhl Publishing.

Eiter, T.; and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, 15(3–4): 289–323.

Eiter, T.; and Gottlob, G. 1997. The Complexity Class $\mathrm{Theta}_2\mathrm{P}$: Recent Results and Applications in AI and Modal Logic. In *Fundamentals of Computation Theory, 11th International Symposium, FCT '97, Kraków, Poland, September 1-3, 1997, Procs.*, volume 1279 of *Lecture Notes in Computer Science*, 1–18. Springer.

Eiter, T.; Hecher, M.; and Kiesel, R. 2021. Treewidth-Aware Cycle Breaking for Algebraic Answer Set Counting. In *Procs. of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, 269–279.

Endriss, U. 2020. Analysis of One-to-One Matching Mechanisms via SAT Solving: Impossibilities for Universal Axioms. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020*, 1918–1925. AAAI Press.

Fages, F. 1994. Consistency of Clark's completion and existence of stable models. *Methods Log. Comput. Sci.*, 1(1): 51–60.

Fleischner, H.; Kullmann, O.; and Szeider, S. 2002. Polynomial-time recognition of minimal unsatisfiable formulas with fixed clause-variable difference. *Theoretical Computer Science*, 289(1): 503–516.

Garcia, L.; Lefèvre, C.; Papini, O.; Stéphan, I.; and Würbel, E. 2018. Possibilistic ASP Base Revision by Certain Input. In *Procs. of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, 1824–1830. AAAI Press.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Morgan & Claypool.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo = ASP + Control: Preliminary Report. *CoRR*, abs/1405.3694.

Gebser, M.; Kaminski, R.; König, A.; and Schaub, T. 2011. Advances in gringo series 3. In *Proc. of the Int. Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, 345–351. Springer.

Gebser, M.; Liu, L.; Namasivayam, G.; Neumann, A.; Schaub, T.; and Truszczyński, M. 2007. The First Answer Set Programming System Competition. In *Procs. of the 9th Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *LNCS*, 3–17. Springer.

Gebser, M.; Maratea, M.; and Ricca, F. 2015. The Design of the Sixth Answer Set Programming Competition. In *Procs. of the 13th Intl Conference Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, 531–544. Springer.

Gelfond, M.; and Lifschitz, V. 1988. The Stable Model Semantics For Logic Programming. In *ICLP/SLP'88*, volume 2, 1070–1080. MIT Press.

Gelfond, M.; and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Comput.*, 9(3/4): 365–386.

Gonçalves, R.; Knorr, M.; and Leite, J. 2021. Forgetting in Answer Set Programming – A Survey. *TPLP*, 1–43.

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

Janhunen, T.; and Niemelä, I. 2016. The Answer Set Programming Paradigm. *AI Magazine*, 37(3): 13–24.

Kleine Büning, H.; and Lettman, T. 1999. *Propositional logic: deduction and algorithms*. Cambridge University Press, Cambridge. ISBN 978-0521630177.

Lagniez, J.-M.; and Biere, A. 2013. Factoring Out Assumptions to Speed Up MUS Extraction. In *Theory and Applications of Satisfiability Testing – SAT 2013*, 276–292. Springer.

Lifschitz, V.; and Razborov, A. A. 2006. Why are there so many loop formulas? *ACM Trans. Comput. Log.*, 7(2): 261–268.

Lin, F.; and Zhao, Y. 2004. ASSAT: computing answer sets of a logic program by SAT solvers. *Artif. Intell.*, 157(1-2): 115–137.

Marek, W.; and Truszczyński, M. 1991. Autoepistemic logic. *J. of the ACM*, 38(3): 588–619.

Marques-Silva, J.; and Lynce, I. 2011. On Improving MUS Extraction Algorithms. In *Theory and Applications of Satisfiability Testing - SAT 2011*, 159–173. Springer.

McCarthy, J. 1980. Circumscription—A form of non-monotonic reasoning. *Artificial Intelligence*, 13(1): 27–39. Special Issue on Non-Monotonic Logic.

Mencía, C.; Kullmann, O.; Ignatiev, A.; and Marques-Silva, J. 2019. On Computing the Union of MUSes. In *SAT*, volume 11628 of *Lecture Notes in Computer Science*, 211–221. Springer.

Mencía, C.; and Marques-Silva, J. 2020. Reasoning About Strong Inconsistency in ASP. In Pulina, L.; and Seidl, M., eds., *Proc. of the 23rd Intl. Conference on Theory and Applications of Satisfiability Testing (SAT'20)*, 332–342. Springer.

Nadel, A. 2010. Boosting minimal unsatisfiable core extraction. In *Formal Methods in Computer Aided Design*, 221–229.

Papadimitriou, C. H. 1994. *Computational Complexity*. Addison-Wesley. ISBN 0-470-86412-5.

Papadimitriou, C. H.; and Wolfe, D. 1988. The Complexity of Facets Resolved. *J. of Computer and System Sciences*, 37(1): 2–13.

Pontelli, E.; Son, T.; Baral, C.; and Gelfond, G. 2012. Answer Set Programming and Planning with Knowledge and World-Altering Actions in Multiple Agent Domains. In *Correct Reasoning – Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265, 509–526. Springer.

Saikko, P.; Dodaro, C.; Alviano, M.; and Järvisalo, M. 2018. A Hybrid Approach to Optimization in Answer Set Programming. In *Procs. of the Sixteenth International Conference on Principles of Knowledge Representation and Reasoning (KR 2018)*, 32–41. AAAI Press.

Schlobach, S.; Huang, Z.; Cornet, R.; and van Harmelen, F. 2007. Debugging Incoherent Terminologies. *Journal of Automated Reasoning*, 39(3): 317–349.

Soh, T.; and Inoue, K. 2010. Identifying Necessary Reactions in Metabolic Pathways by Minimal Model Generation. In *Proc. of ECAI'10*, 277–282.

Stockmeyer, L. J.; and Meyer, A. R. 1973. Word problems requiring exponential time. In *Procs. of STOC'73*, 1–9. Assoc. Comput. Mach., New York.

Szeider, S. 2004. Minimal unsatisfiable formulas with bounded clause-variable difference are fixed-parameter tractable. *J. of Computer and System Sciences*, 69(4): 656–674.

Truszczynski, M. 2011. Trichotomy and dichotomy results on the complexity of reasoning with disjunctive logic programs. *TPLP*, 11(6): 881–904.

Ulbricht, M.; Thimm, M.; and Brewka, G. 2020. Handling and measuring inconsistency in non-monotonic logics. *Artif. Intell.*, 286: 103344.

van der Kouwe, E.; Andriesse, D.; Bos, H.; Giuffrida, C.; and Heiser, G. 2018. Benchmarking Crimes: An Emerging Threat in Systems Security. *CoRR*, abs/1801.02381.

Wagner, K. W. 1987. More Complicated Questions About Maxima and Minima, and Some Closures of NP. *Theor. Comput. Sci.*, 51: 53–80.