A Comparison of ASP-Based and SAT-Based Algorithms for the Contension Inconsistency Measure

Isabelle Kuhlmann, Anna Gessler, Vivien Laszlo, and Matthias Thimm

Artificial Intelligence Group, University of Hagen, Germany
{isabelle.kuhlmann, vivien.laszlo, matthias.thimm}@fernuni-hagen.de
anna.gessler.work@gmail.com

Abstract. We propose an algorithm based on satisfiability problem (SAT) solving for determining the contension inconsistency degree in propositional knowledge bases. In addition, we present a revised version of an algorithm based on answer set programming (ASP), which serves the same purpose. In an experimental analysis, we compare the two algorithms to each other, as well as to a naive baseline method. Our results demonstrate that both the SAT and the ASP approach expectedly outperform the baseline algorithm. Further, the revised ASP method is not only superior to the SAT approach, but also to its predecessors from the literature. Hence, it poses a new state of the art.

Keywords: Inconsistency Measurement · Answer Set Programming · Satisfiability Solving.

1 Introduction

The ubiquitous presence of conflicting information and the handling thereof constitutes a major challenge in Artificial Intelligence. The field of *inconsistency measurement* (see the seminal work by Grant [14], and the book by Grant and Martinez [15]) allows for an analytical perspective on the subject of inconsistency in formal knowledge representation formalisms. In inconsistency measurement, the aim is to quantitatively assess the *severity* of inconsistency in order to both guide automatic reasoning mechanisms and to help human modelers to identify issues and compare different alternative formalizations. For example, inconsistency measures have been used to estimate reliability of agents in multi-agent systems [9], to analyze inconsistencies in news reports [17], to support collaborative software requirements specifications [21], to allow for inconsistency-tolerant reasoning in probabilistic logic [23], and to monitor and maintain quality in database settings [4]. Hence, there is clearly a need for practically applicable approaches.

There are numerous inconsistency measures, based on different concepts, such as minimal inconsistent subsets (see, e.g., [16]) or maximal consistent sets (see, e.g., [3]), or non-classical semantics (see, e.g., [13]); see [26] for an overview.

A number of problems related to these measures lies on the first level of the polynomial hierarchy, which renders them complexity-wise most likely to be suitable for practical applications, compared to other measures where the associated problems are located higher up the polynomial hierarchy [29]. Further, a natural approach to computing these measures is using *satisfiability problem* (SAT) solving, which is widely used in applications such as the automatic verification of hardware specifications [31], or cryptanalysis [22]. Moreover, there exist highly optimized SAT solvers (see the results of the annual SAT competition[1] for an overview). In this paper, we present a SAT-based approach for inconsistency measurement. To be precise, we develop a SAT encoding for determining the *contension inconsistency measure* (which we also simply refer to as *contension measure*) [13] via binary search.

There already exist a couple of works that take an algorithmic perspective on inconsistency measurement. In [18] and [19], the authors present approaches for computing a number of inconsistency measures based on reductions to *answer set programming* (ASP). A total of three inconsistency measures [13, 5, 27], where the corresponding decision problems are all on the first level of the polynomial hierarchy, were selected. The three measures were implemented and compared to naive baseline implementations in an experimental analysis. As anticipated, the results showed that the ASP-based implementations were clearly superior.

We compare the newly proposed SAT-based approach with a revised version of the ASP approach that was presented in [19]. In an extensive experimental evaluation, we additionally compare the two methods to a naive baseline method, which is, to the best of our knowledge, the only other existing implementation of the contension measure. Yet, we focus our analysis on the comparison between the SAT approach and the ASP approach, as those are the more promising—and, in terms of performance, more comparable—methods. The results reveal that, as expected, the SAT approach is clearly superior to the naive one, however, it cannot compete with the ASP approach. In addition, we draw a comparison between the previous versions of the ASP-based method [18, 19] and the newly proposed one. We demonstrate that the latter is superior, and thus represents a new state of the art.

The remainder of this paper is organized as follows. In Section 2, we explain fundamental definitions regarding inconsistency measurement, ASP, and SAT solving. Sections 3 and 4, respectively, provide descriptions of the SAT-based and ASP-based approaches for the contension inconsistency measure. In Section 5, we describe our experimental evaluation, including an in-depth discussion of the results, and we conclude in Section 6. Due to space restrictions, all proofs are omitted in the main paper, but are provided in an appendix[2].

[1] http://www.satcompetition.org/
[2] https://e.feu.de/sum2022-appendix

2 Preliminaries

We define At to be a fixed set of propositions (also referred to as (propositional) *atoms*), and $\mathcal{L}(\mathsf{At})$ to be the corresponding propositional language. $\mathcal{L}(\mathsf{At})$ is constructed by applying the usual connectives, i.e., $\wedge$ (*conjunction*), $\vee$ (*disjunction*), and $\neg$ (*negation*). A finite set of (propositional) formulas $\mathcal{K} \subseteq \mathcal{L}(\mathsf{At})$ is called a (propositional) *knowledge base* (KB). Let $\mathbb{K}$ denote the set of all KBs. Let F be a formula or a set of formulas. We denote the set of propositions appearing in F, i.e., the *signature* of F, as $\mathsf{At}(F)$. Semantics of a propositional language is determined by *interpretations*.

Definition 1. *A propositional* interpretation *is a function* $\omega : \mathsf{At} \to \{\mathsf{true}, \mathsf{false}\}$. *Let* $\Omega(\mathsf{At})$ *be the set of all interpretations.*

An interpretation *satisfies* an atom $x \in \mathsf{At}$ if and only if $\omega(x) = \mathsf{true}$. This is also denoted as $\omega \models x$, and ω is also referred to as a *model* of x. We extend this concept to formulas and sets of formulas in the usual manner.

2.1 Inconsistency Measurement

If there exists no model for a formula or a set of formulas F, i.e., if $\neg\exists\omega \models F$, then F is *inconsistent*. The intuition behind an *inconsistency measure* is that a higher value indicates a more severe inconsistency than a lower one. Besides, the minimal value (0) is supposed to model the absence of inconsistency, i.e., consistency. Let $\mathbb{R}^{\infty}_{\geq 0}$ be the set of non-negative real numbers, including infinity.

Definition 2. *An* inconsistency measure $\mathcal{I}$ *is a function* $\mathcal{I} : \mathbb{K} \to \mathbb{R}^{\infty}_{\geq 0}$ *that satisfies* $\mathcal{I}(\mathcal{K}) = 0$ *if and only if* $\mathcal{K}$ *is consistent, for all* $\mathcal{K} \in \mathbb{K}$.

We further define $\textsc{Upper}_{\mathcal{I}}$ to be the decision problem of deciding whether a given value $u \in \mathbb{R}^{\infty}_{\geq 0}$ is an upper bound of $\mathcal{I}(\mathcal{K})$ wrt. a given KB $\mathcal{K}$, and we define $\textsc{Value}_{\mathcal{I}}$ to be the functional problem of determining the value of $\mathcal{I}(\mathcal{K})$.

The *contension inconsistency measure* $\mathcal{I}_{\mathrm{c}}$ [13] is based on Priest's three-valued propositional logic [24], which extends the two classical truth values true (t) and false (f) by a third value, which indicates *paradoxical*, or *both true and false* (b). The truth tables of this logic are presented in Table 1. A *three-valued interpretation* $\omega^3 : \mathsf{At}(\mathcal{K}) \mapsto \{t, f, b\}$ assigns one of the three truth values to each atom in a KB $\mathcal{K}$. An interpretation ω^3 is a *three-valued model* of an atom, if it evaluates to either t or b. Again, we extend this concept to formulas, and sets of formulas. We denote the set of all three-valued models wrt. an arbitrary KB $\mathcal{K}$ as $\mathsf{Models}(\mathcal{K}) = \{\omega^3 \mid \forall\alpha \in \mathcal{K}, \omega^3(\alpha) = t \text{ or } \omega^3(\alpha) = b\}$.

Furthermore, we can divide the domain of an interpretation ω^3 into two sets, of which one contains those atoms that are assigned a classical truth value (t, f), and the other one contains those atoms that are assigned truth value b. The latter is defined as $\mathsf{Conflictbase}(\omega^3) = \{x \in \mathsf{At}(\mathcal{K}) \mid \omega^3(x) = b\}$. Consider the interpretation ω^3_B which sets all atoms in a KB $\mathcal{K}$ to b. Such an interpretation will always satisfy $\mathcal{K}$ (i.e., $\omega^3_B \in \mathsf{Models}(\mathcal{K})$). However, if we minimize the number

Table 1. Truth tables for Priest's propositional three-valued logic.

x	y	$x \wedge y$	$x \vee y$
t	t	t	t
t	b	b	t
t	f	f	t
b	t	b	t
b	b	b	b
b	f	f	b
f	t	f	t
f	b	f	b
f	f	f	f

x	$\neg x$
t	f
b	b
f	t

of b assignments (i.e., $|\mathsf{Conflictbase}(\omega^3)|$), it becomes evident which atoms are involved in a conflict, because they are exactly those atoms that cannot be set to t or f without rendering $\mathcal{K}$ unsatisfiable.

Definition 3. *We define the* contension inconsistency measure $\mathcal{I}_c$ *wrt. a knowledge base* $\mathcal{K}$ *as* $\mathcal{I}_c(\mathcal{K}) = \min\{|\mathsf{Conflictbase}(\omega^3)| \mid \omega^3 \in \mathsf{Models}(\mathcal{K})\}$.

The minimal number of atoms that are assigned b corresponds exactly to the number of atoms which are involved in a conflict, as the following example illustrates.

Example 1. Consider $\mathcal{K}_1 = \{x \wedge y, \neg x, y \vee z\}$. Let ω_1^3 be an interpretation with $\omega_1^3(y) = \omega_1^3(z) = t$, and $\omega_1^3(x) = b$, i.e., ω_1^3 is a model of $\mathcal{K}$, and $\mathsf{Conflictbase}(\omega_1^3) = \{x\}$. It is easy to see that x must be assigned b in order to make $\mathcal{K}_1$ satisfiable, and that no lower number of atoms being assigned b could achieve this. Hence, $\mathcal{I}_c(\mathcal{K}_1) = |\mathsf{Conflictbase}(\omega_1^3)| = |\{x\}| = 1$.

2.2 Satisfiability Solving

One of the major problems of propositional logic is the *Boolean Satisfiability Problem*, which is one of the most-studied problems of computer science, and which is NP-complete [7].

Definition 4. *The* Boolean Satisfiability Problem *(SAT) is the problem of deciding if there exists an interpretation that satisfies a given propositional formula.*

A *SAT solver* is a program that solves SAT for a given formula. There exist numerous high-performance SAT solvers (see [10] for a recent overview). Note that the input formula of a SAT solver must be in Conjunctive Normal Form (CNF), i.e., it must be a conjunction of clauses. Although every propositional formula can be transformed to CNF, a naive conversion using Boolean transformation rules may result in a formula which is exponentially larger than the original formula. For this reason, in this work we use the Tseitin method [30] for converting formulas to CNF, which yields an equisatisfiable formula in CNF, with only a linear increase in size. We further have a concept of modeling *cardinality constraints* in SAT, which represent that at least, at most, or exactly some number

k out of a set of propositional atoms are allowed to be true. Using the formal definition of Abio et al. [1], we define a cardinality constraint to be of the form $a_1 + \ldots + a_n \bowtie k$, where $a_1, \ldots, a_n$ are propositional atoms with $|\mathsf{At}| = n$, k is a natural number, and $\bowtie \in \{<, \leq, =, \geq, >\}$. The meaning of the $+$ operator is that for every true atom the number 1 is added and for every false atom the number 0 is added, thereby counting the number of true atoms. To encode $\mathcal{I}_\mathrm{c}$, we merely require *at-most-k constraints*, i.e., constraints of the form $a_1 + \ldots + a_n \leq k$. A straightforward approach to realize at-most-k constraints is to add all clauses which are disjunctions of subsets $A_i \subseteq \{\neg a_1, \ldots, \neg a_n\}$ with $|A_i| = k + 1$ to the SAT encoding. Since this method creates $\binom{n}{k+1}$ clauses (*binomial encoding*), it does not scale well, and it is often not suitable for practical applications. There are, however, more efficient approaches, such as the *sequential counter encoding* [25], which is used in our experiments (see Section 5).

2.3 Answer Set Programming

Answer set programming (ASP) [11, 20, 8] is a declarative problem solving approach targeted at difficult search problems. Thus, rather than modeling instructions on how to solve a problem, a representation of the problem itself is modeled. More precisely, a problem is modeled as an *extended logic program* which consists of a set of *rules* of the form

$$r = H \texttt{ :- } A_1, \ldots, A_n, \texttt{not } B_1, \ldots, \texttt{not } B_m. \tag{1}$$

where H, A_i with $i \in \{1, \ldots, n\}$, and B_j with $j \in \{1, \ldots, m\}$ are classical literals. ASP rules consist of a *head* and a *body*, separated by "`:-`", of which one may be empty. Wrt. a rule r, we denote the sets of literals contained in the head and body as $\mathsf{head}(r)$, and $\mathsf{body}(r)$, respectively. In Eq. (1), $\mathsf{head}(r) = \{H\}$, and $\mathsf{body}(r) = \{A_1, \ldots, A_n, B_1, \ldots, B_m\}$. We refer to a rule with an empty body as *fact*, and to one with an empty head as *constraint*. An extended logic program is *positive* if it does not contain any default negation (`not`). A set of literals L is called *closed under* a positive program P if and only if for any rule $r \in P$, $\mathsf{head}(r) \in L$ whenever $\mathsf{body}(r) \subseteq L$. A set L is consistent if it does not contain both A and $\neg A$ for some literal A. We denote the smallest of such sets wrt. a positive program P, which is always uniquely defined, as $\mathsf{Cn}(P)$. Wrt. an arbitrary program P, a set L is called an *answer set* of P if $L = \mathsf{Cn}(P^L)$, with

$$\begin{aligned} P^L = \{&H \texttt{ :- } A_1, \ldots, A_n \mid \\ &H \texttt{ :- } A_1, \ldots, A_n, \texttt{not } B_1, \ldots, \texttt{not } B_m. \in P, \{B_1, \ldots, B_m\} \cap L = \emptyset\}. \end{aligned}$$

In addition to the "basic" rules (as described above), modern ASP dialects allow for more complex structures. An example of such is the *cardinality constraint*, which is of the form

$$l\{A_1, \ldots, A_n, \texttt{not } B_1, \ldots, \texttt{not } B_m\}u,$$

where l is a lower bound, u is an upper bound, and $A_1, \ldots, A_n, B_1, \ldots, B_m$ are literals. It can be interpreted as follows: if at least l and at most u of the

literals are included in an answer set, the cardinality rule is satisfied by that answer set. ASP solvers also offer the option to express cost functions involving minimization and/or maximization in order to solve optimization problems [11]. In this work, we only require a specific type of optimization statements of the form `#minimize`$\{A_1, \ldots, A_n\}$. Such a *minimize statement* instructs the ASP solver to include only a minimal number of the literals $A_1, \ldots, A_n$ in any answer set. We refer to an answer set that corresponds to the minimization (i.e., that contains a minimal number of $A_1, \ldots, A_n$) as an *optimal answer set*.

As of yet, we introduced a basic *propositional* syntax and semantics of ASP, in order to define the different language concepts in a concise manner. Note, however, that we model $\mathcal{I}_c$ in ASP using *first-order* predicates and functions. This serves the purpose of a) easing readability, and b) facilitating an improved automated grounding process. The use of first-order concepts allows us to set variables which range over constant symbols. Following the Clingo syntax [20], we use capitalized identifiers for variables and non-capitalized ones for constants. Replacing the variables in a rule by the corresponding constant symbols is referred to as *grounding*. In addition, we express the arity n of a predicate or function `f` as `f`$/n$.

Example 2. Our aim is to model a KB $\mathcal{K}_2$ with $\mathsf{At}(\mathcal{K}_2) = \{a, b\}$ in ASP. To represent the concept of an atom, we introduce the predicate `atom/1`. We also use the constant symbols `a` and `b` to represent the atoms a and b, and create the facts `atom(a)` and `atom(b)`. If we now wish to use `atom/1` in a rule, we do not have to explicitly state `atom(a)` and `atom(b)`, but we can simply use a variable, e.g., `X`, and write `atom(X)`. During the grounding process, `X` will then be replaced by `a` and `b`.

3 An Algorithm for $\mathcal{I}_c$ Based on SAT

In order to compute the value of $\mathcal{I}_c(\mathcal{K})$ wrt. a KB $\mathcal{K}$, i.e., in order to solve $\textsc{Value}_{\mathcal{I}_c}$, we use the standard approach to solve the functional problem $\textsc{Value}_{\mathcal{I}_c}$ by iterative calls to a SAT solver which determines the answers to the decision problem $\textsc{Upper}_{\mathcal{I}_c}$ [29]. The range of $\mathcal{I}_c(\mathcal{K})$ is clearly defined (with 0 being the minimal and $|\mathsf{At}(\mathcal{K})|$ the maximal value), which enables us to use binary search to find the exact value. To be precise, we start with $u = \lfloor |\mathsf{At}(\mathcal{K})|/2 \rfloor$, and determine a SAT encoding for $\textsc{Upper}_{\mathcal{I}_c}$ wrt. the KB $\mathcal{K}$ and the value u as the upper bound. If u is in fact an upper bound of $\textsc{Upper}_{\mathcal{I}_c}$, we continue the binary search in the lower interval, if it is not, we continue in the upper interval. After $\log_2(|\mathsf{At}(\mathcal{K})|)$ iterative calls to a SAT solver, we know the lowest possible value for which $\textsc{Upper}_{\mathcal{I}_c}$ returns true, i.e., we know the solution to $\textsc{Value}_{\mathcal{I}_c}$.

In the following, we illustrate how to construct a set of formulas $S(\mathcal{K}, u)$ wrt. a KB $\mathcal{K}$ and a non-negative integer value u, which is satisfied if and only if u is an upper bound of $\mathcal{I}_c(\mathcal{K})$. To encode Priest's tree-valued logic in propositional logic, we require additional variables. To begin with, for every atom x in the original signature $\mathsf{At}(\mathcal{K})$, we introduce three new atoms x_t, x_b, x_f (S1) to represent the

three truth values t, b, f. In order to ensure that only one of such atoms is true wrt. some $x \in \mathsf{At}(\mathcal{K})$, we add the following rule:

$$(x_t \vee x_f \vee x_b) \wedge (\neg x_t \vee \neg x_f) \wedge (\neg x_t \vee \neg x_b) \wedge (\neg x_b \vee \neg x_f) \tag{S2}$$

In addition, we must model the evaluation of formulas in three-valued logic (see Table 1). We introduce three variables $v^t_\phi, v^f_\phi, v^b_\phi$ (S3) for every sub-formula ϕ of every formula $\alpha \in \mathcal{K}$ to represent when each of the three possible valuations of ϕ occurs. For each of these atoms we add an equivalence relation which defines the evaluations based on the operator of the sub-formula. Thus, we need to model conjunction, disjunction, and negation. To encode a conjunction $\phi_c = \psi_{c,1} \wedge \psi_{c,2}$, we need to model that ϕ_c is t if both conjuncts are t, ϕ_c is f if at least one of the conjuncts is f, and ϕ_c is b if at least one of the conjuncts is b and the other one is not f:

$$v^t_{\phi_c} \leftrightarrow v^t_{\psi_{c,1}} \wedge v^t_{\psi_{c,2}} \tag{S4}$$
$$v^f_{\phi_c} \leftrightarrow v^f_{\psi_{c,1}} \vee v^f_{\psi_{c,2}} \tag{S5}$$
$$v^b_{\phi_c} \leftrightarrow (v^b_{\psi_{c,1}} \vee v^b_{\psi_{c,2}}) \wedge \neg v^f_{\psi_{c,1}} \wedge \neg v^f_{\psi_{c,2}} \tag{S6}$$

In the same fashion, we can encode that a disjunction is f if both of its disjuncts are f, it is t if at least one of the disjuncts is t, and it is b if at least one disjunct is b and the other one is not t:

$$v^t_{\phi_d} \leftrightarrow v^t_{\psi_{d,1}} \vee v^t_{\psi_{d,2}} \tag{S7}$$
$$v^f_{\phi_d} \leftrightarrow v^f_{\psi_{d,1}} \wedge v^f_{\psi_{d,2}} \tag{S8}$$
$$v^b_{\phi_d} \leftrightarrow (v^b_{\psi_{d,1}} \vee v^b_{\psi_{d,2}}) \wedge \neg v^t_{\psi_{d,1}} \wedge \neg v^t_{\psi_{d,2}} \tag{S9}$$

Negations $\phi_n = \neg\psi_n$ are encoded as follows:

$$v^t_{\phi_n} \leftrightarrow v^f_{\psi_n} \qquad v^f_{\phi_n} \leftrightarrow v^t_{\psi_n} \qquad v^b_{\phi_n} \leftrightarrow v^b_{\psi_n} \tag{S10–12}$$

Further, we add variables for each sub-formula ϕ_a which represents an individual atom x:

$$v^t_{\phi_a} \leftrightarrow x_t \qquad v^f_{\phi_a} \leftrightarrow x_f \qquad v^b_{\phi_a} \leftrightarrow x_b \tag{S13–15}$$

Moreover, we need to represent a formula $\alpha \in \mathcal{K}$ being satisfied in three-valued logic. This is the case when the sub-formula which contains the entire formula evaluates to t or b. Thus, we add the formula $v^t_\alpha \vee v^b_\alpha$ (S16). Finally, we add a cardinality constraint representing that at most u of the b-atoms can be true: $\texttt{at\_most\_u}(\mathsf{At}_b)$ (S17). We define $S(\mathcal{K}, u)$ to be comprised of (S1–17).

Theorem 1. *For a given value u, the encoding $S(\mathcal{K}, u)$ is satisfiable if and only if $\mathcal{I}_c(\mathcal{K}) \leq u$.*

4 An Algorithm for $\mathcal{I}_c$ Based on ASP

There already exist two ASP-based approaches for computing $\mathcal{I}_c$ in the literature. In [18], the authors introduce a method similar to our SAT approach (see Section 3), which uses ASP encodings for the problem $\textsc{Upper}_{\mathcal{I}_c}$, in order to find $\textsc{Value}_{\mathcal{I}_c}$ via binary search. A revised version of this approach is proposed in [19], which calculates $\textsc{Value}_{\mathcal{I}_c}$ directly within ASP by means of a minimize statement. Note that in both versions, only propositional language concepts are used, which leads to a program that is already ground (i.e., the program is essentially ground manually, instead of by a grounder). In the present work, we demonstrate yet another revision of the ASP approach, which is very similar to the one in [19], but makes use of first-order predicates and variables, which enables an automated, and internally optimized, grounding procedure.

We address the problem of computing $\mathcal{I}_c(\mathcal{K})$ wrt. a KB $\mathcal{K}$ by constructing an extended logic program as follows. First, we define some facts that describe the composition of $\mathcal{K}$. Every atom $x \in \mathsf{At}(\mathcal{K})$ is represented in ASP as `atom(`x`)` (A1), and every formula $\alpha \in \mathcal{K}$ as `kbMember(`α`)` (A2). Further, every conjunction $\phi_c = \psi_{c,1} \wedge \psi_{c,2}$ is encoded as `conjunction(`ϕ_c`,`$\psi_{c,1}$`,`$\psi_{c,2}$`)` (A3). In the same fashion, every disjunction $\phi_d = \psi_{d,1} \vee \psi_{d,2}$ is encoded as `disjunction(`ϕ_d`,`$\psi_{d,1}$`,`$\psi_{d,2}$`)` (A4). Each negation, i.e., each $\phi_n = \neg\psi_n$, is represented as `negation(`ϕ_n`,`ψ_n`)` (A5). Further, each formula ϕ_a that consists of an individual atom x is encoded as `formulaIsAtom(`ϕ_a`,`x`)` (A6). Moreover, we represent the truth values of Priest's three-valued logic (t, f, b) as `tv(`t`)`, `tv(`f`)`, and `tv(`b`)` (A7).

To encode the actual functionality of the contension measure, we need to create a rule which "guesses" a three-valued interpretation. To achieve this, we model that each atom is assigned exactly one truth value by using the cardinality constraint

```
1{truthValue(A,T): tv(T)}1:- atom(A).                    (A8)
```

As with the SAT approach, we need to represent the role of the operators $\wedge$, $\vee$, and $\neg$ in three-valued logic. In order for a conjunction $\phi_c = \psi_{c,1} \wedge \psi_{c,2}$ to be t, both of its conjuncts need to be t:

```
truthValue(F,t):- conjunction(F,G,H),
                  truthValue(G,t), truthValue(H,t).      (A9)
```

For a conjunction to be f, on the other hand, it is sufficient if only one of the conjuncts is f:

```
truthValue(F,f):- conjunction(F,G,H),
                  1{truthValue(G,f), truthValue(H,f)}.   (A10)
```

Finally, a conjunction is b if it is neither t nor f:

```
truthValue(F,b):- conjunction(F,_,_),
                  not truthValue(F,t), not truthValue(F,f).  (A11)
```

In the same fashion, a disjunction $\phi_d = \psi_{d,1} \vee \psi_{d,2}$ is only f if both of its disjuncts are f (A12), it is t if at least one disjunct is t (A13), and it is b if it is neither t nor f (A14). A negation is t in three-valued logic if its base formula is f, i.e.,

```
truthValue(F,t):- negation(F,G), truthValue(G,f).        (A15)
```

and vice versa, and it is b if its base formula is also b. Hence, the other two cases (A16–17) follow accordingly. Moreover, if a (sub-)formula consists of a single atom, it must have the same truth value as the referred atom:

```
truthValue(F,T):- formulaIsAtom(F,G),
                  truthValue(G,T), tv(T).        (A18)
```

In order to compute $\mathcal{I}_c$, we still need to ensure that the ASP solver finds an interpretation that satisfies all formulas $\alpha \in \mathcal{K}$. Consequently, every $\alpha \in \mathcal{K}$ must evaluate to either t or b—in other words, no formula must evaluate to f. We realize this using the following integrity constraint:

```
:- truthValue(F,f), kbMember(F).        (A19)
```

Finally, as our aim is to find the minimal number of atoms being evaluated to b, we add

```
#minimize{1,A: truthValue(A,b), atom(A)}.        (A20)
```

We define P_c to be the union of all rules (A1–20) defined above. Further, let ω_M^3 be the three-valued interpretation represented by an answer set M of $P_c(\mathcal{K})$.

Theorem 2. *Let M be an optimal answer set of $P_c(\mathcal{K})$. Then $|(\omega_M^3)^{-1}(b)| = \mathcal{I}_c(\mathcal{K})$.*[3]

5 Experimental Analysis

The central aspect of our experimental evaluation is a comparison between the SAT-based and ASP-based approaches introduced in Sections 3 and 4, as well as a naive baseline algorithm. The latter, which is provided by *TweetyProject*[4], is implemented by first converting the KB to CNF, followed by iterating through all subsets of atoms (with increasing cardinality), deleting all clauses in which one of the atoms of the current set appears (thus, effectively setting their three-valued truth value to b). At each interation we check whether the resulting KB is consistent by means of a SAT solver (here, CaDiCal sc2021[5] [6]). If it is, the cardinality of the current set of atoms is returned.

Although the ASP approach has been proven to be clearly superior to the naive one in terms of runtime (see [19]), and we can expect the same for the

[3] For any function $\varphi : X \mapsto Y$ and $y \in Y$ we define $\varphi^{-1}(y) = \{x \in X \mid f(x) = y\}$

[4] https://e.feu.de/tweety-contension

[5] https://github.com/arminbiere/cadical

SAT approach, we still draw this comparison in order to concretely quantify this assumption. Besides, the naive algorithm is, to the best of our knowledge, the only existing alternative to compute $\mathcal{I}_c$. However, as the result of comparing the SAT and ASP approaches is far less predictable (both SAT and ASP are established formalisms for dealing with problems on the first level of the polynomial hierarchy), we examine the two methods more closely. We consider how the runtimes of the approaches are composed, e.g., how much time the respective solvers require, or the time it takes to compute the encodings. Moreover, we draw a comparison between previous versions of the ASP approach [18, 19] and the newly proposed one.

5.1 Experimental Setup

As there is, to the best of our knowledge, no dedicated benchmark dataset for inconsistency measurement, we need to compile our own dataset. One option to achieve this is to generate completely synthetic data; another one is to "translate" benchmark data from a different research field. Therefore, we use both a synthetic and a "translated" dataset. The **SRS** dataset[6] consists of synthetic KBs generated by the *SyntacticRandomSampler*[7] provided by TweetyProject. This dataset corresponds exactly to the union of datasets A and B in [19]. Hence, the SRS dataset contains a total of 1800 KBs of varying complexity. The smallest instances have a signature size of 3, and contain between 5 and 15 formulas, the biggest ones have a signature size of 30, and contain between 50 and 100 formulas. As the formulas are created randomly, and independently of one another, most KBs are highly inconsistent[8]. The **ML** dataset[9] consists of a total of 1920 KBs learned from the *Animals with Attributes*[10] (AWA) dataset, which is widely used in the area of machine learning. It describes 50 types of animals using 85 binary attributes. Following [28], we used the Apriori algorithm [2] to mine association rules from the AWA dataset for a given minimal confidence value c and minimal support value s. These rules were then interpreted as propositional logic implications. We finally selected one animal at random and added all its attributes as facts, likely making the KB inconsistent, as even rules with low confidence values were interpreted as strict implications. We set

$$c \in \{0.6, 0.65, 0.70, 0.75, 0.8, 0.85, 0.90, 0.95\},$$
$$s \in \{0.6, 0.65, 0.70, 0.75, 0.8, 0.85, 0.90, 0.95\},$$

and allowed a maximum of 4 literals per rule.

Both the SAT-based and the ASP-based approach are implemented in C++. The SAT solver we use is CaDiCal sc2021 (as with the naive method), and the

[6] Download: https://e.feu.de/srs-dataset
[7] https://e.feu.de/tweety-syntactic-random-sampler
[8] Overview of inconsistency values: https://e.feu.de/sum2022-tables
[9] Download: https://e.feu.de/ml-dataset
[10] http://attributes.kyb.tuebingen.mpg.de

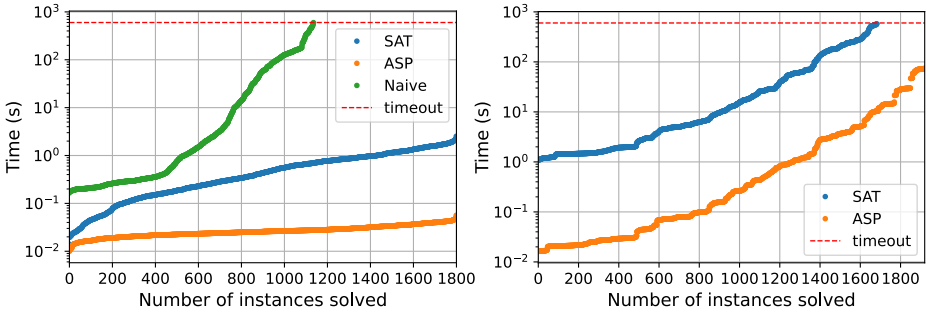

Fig. 1. Runtime comparison of the ASP-based, SAT-based, and naive approach on the SRS dataset (left). Further runtime comparison of the ASP-based and SAT-based methods on the ML dataset (right). Timeout: 10 minutes.

ASP solver we use is Clingo 5.5.1[11] [12]. For the computation of cardinality constraints in SAT we use sequential counter encoding, and for transforming formulas to CNF we use Tseitin's method. All experiments were run on a computer with 125 GB RAM and an Intel Xeon E5-2690 CPU which has a basic clock frequency of 2.90 GHz.

5.2 Results

We first consider the overall runtime per KB of all three approaches wrt. the SRS dataset. Figure 1 (left) shows a *cactus plot* of the measured runtimes, i.e., wrt. each method it shows the runtimes wrt. each KB of the dataset, sorted from low to high, with a timeout set to 10 minutes. We can see that both the SAT and the ASP approach fare quite well compared to the naive method. While the latter produces a total of 664 timeouts, the former are able to compute all inconsistency values easily within the time limit. However, even though the SAT method clearly outperforms the naive method, it cannot match the ASP approach. A comparison of the SAT method and the ASP method wrt. the more challenging ML dataset shows the same pattern (see the right part of Figure 1). Here, the SAT method actually times out in 237 cases.

We now proceed to a more detailed examination of how the SAT and ASP runtimes are composed. To achieve this, we measure the amount of time required to compute the SAT/ASP encoding, the respective solving time, and, in the SAT case, the time required to transform the formulas to CNF. Note that wrt. SAT, we measure the *total* time needed for encoding, solving, and transforming, as the iterative nature of the approach requires multiple calls. Further, in the ASP case, "solving" includes the grounding process, and in both the ASP and the SAT case, it includes initializing the solver, and feeding it the program/clauses. Figure 2 visualizes how the runtimes of both approaches are composed on average (regarding the SRS dataset). The category "other" included in the figure covers

[11] https://potassco.org/clingo/

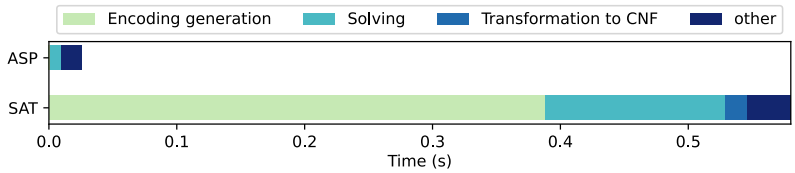

Fig. 2. Overview of the average runtime composition of the ASP-based and SAT-based approaches wrt. the SRS dataset.

factors such as loading the KB. As the cactus plot in Figure 1 (left) already indicates, the average runtime of the ASP-based method is several times shorter than the runtime of the SAT-based method (0.026 seconds vs. 0.580 seconds). With regard to the ASP approach, it is noticeable that the encoding generation only takes up a tiny fraction of the overall runtime (with 0.0008 seconds it is barely even visible in Figure 2), while the "other" category takes up more than half the runtime. However, since the average total runtime of this method is very low in general, this observation should be taken with a grain of salt, as the ratio of the different runtime shares could shift with an increasing size and complexity of the KBs at hand. One striking observation wrt. the SAT approach is that the encoding generation represents the largest fraction of the overall runtime. This is mainly due to the fact that a new cardinality constraint is required for each iteration, and its calculation can be costly even when using a non-trivial method. The transformation to CNF, on the other hand, hardly contributes to the overall runtime. It should also be noted that the pure solving time (excluding any preprocessing) is only 0.0064 seconds on average, which demonstrates the extent to which modern SAT solvers are optimized. Hence, we see that SAT solvers are in fact able to solve $\textsc{Upper}_{\mathcal{I}_c}$ quite fast, however, the iterative nature of the approach leads to a large overhead, in particular wrt. the generation of cardinality constraints.

Yet another aspect we aim to investigate is how well the newly proposed revision of the ASP approach performs in comparison to its predecessors in [18] and [19] (see Section 4 for an overview of the two approaches). We apply exactly those Java implementations which were used in the two corresponding papers. To conduct our analysis, we use the SRS dataset. The results of this experiment, which are illustrated in Figure 3, show that the new version of the method in fact outperforms the older ones. The first ASP approach [18], which is based on a binary search procedure, clearly performs the poorest, and hits the timeout of 10 minutes in 600 cases. The second version of the approach [19] yields more consistent results, nevertheless it performs on average roughly 7 times slower than the newest version (0.266 vs. 0.037 seconds). Although the new version might have an advantage by being implemented in C++, both rely on the same ASP solver. In fact, the solving time itself is around 3 times shorter wrt. the new ASP version compared to the previous one (0.010 vs. 0.028 seconds on average).

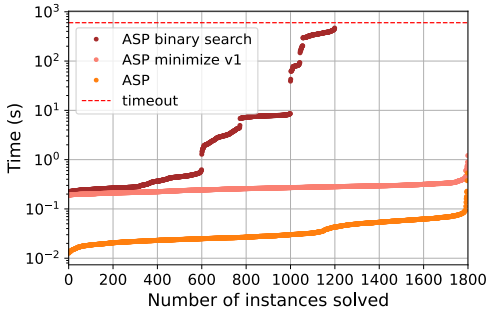

Fig. 3. Runtime comparison of the different versions of the ASP approach on the SRS dataset. "ASP binary search" refers to the version from [18], "ASP minimize v1" to the version from [19], and "ASP" to the new version. Timeout: 10 minutes.

6 Conclusion

In the course of this work, we addressed the problem of computing the contension inconsistency measure from an algorithmic perspective. To be specific, we introduced a SAT-based approach, as well as a revised version of an ASP-based approach. We have subjected the two methods to extensive experimental analysis and have learned the following. SAT is generally a suitable formalism to compute $\mathcal{I}_c$ (as our SAT-based method clearly outperforms a naive baseline approach), however, due to its iterative nature, it cannot compete with the ASP-based method. In particular, our new version of the ASP approach outperforms not only the SAT-based one, but also its previous two versions from the literature. Besides, the performance of the ASP approach can now be more accurately assessed—a comparison with a SAT-based method is more appropriate than merely with a naive algorithm.

There are still numerous aspects to be examined in future work. For instance, wrt. SAT, one can compare different SAT solvers, different methods of converting formulas to CNF, different techniques of generating cardinality constraints, or exploit approaches to MaxSAT. Furthermore, both SAT and ASP approaches for other inconsistency measures could be developed and compared. Moreover, other formalisms, such as *Quantified Boolean Formulas* (QBF), might be interesting for computing inconsistency measures. One could also consider measures of higher complexity.

References

1. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints. In: 19th International Conference on Principles and Practice of Constraint Programming. pp. 80–96. CP'13 (2013)
2. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proceedings VLDB'94. pp. 487–499 (1994)

3. Ammoura, M., Raddaoui, B., Salhi, Y., Oukacha, B.: On measuring inconsistency using maximal consistent sets. In: European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty. pp. 267–276. Springer (2015)
4. Bertossi, L.: Measuring and computing database inconsistency via repairs. In: 12th International Conference on Scalable Uncertainty Management. pp. 368–372 (2018)
5. Besnard, P.: Forgetting-based inconsistency measure. In: 10th International Conference on Scalable Uncertainty Management. pp. 331–337. Springer (2016)
6. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
7. Biere, A., Heule, M., Maaren, H., Walsh, T.: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, IOS Press (2009)
8. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Communications of the ACM **54**(12), 92–103 (2011)
9. Cholvy, L., Perrussel, L., Thevenin, J.M.: Using inconsistency measures for estimating reliability. International Journal of Approximate Reasoning **89**, 41–57 (2017)
10. Department of Computer Science, University of Helsinki , Helsinki: Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions (2021)
11. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer set solving in practice. Synthesis Lectures on Artificial Intelligence and Machine Learning **6**(3), 1–238 (2012)
12. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. Theory and Practice of Logic Programming **19**(1), 27–82 (2019)
13. Grant, J., Hunter, A.: Measuring consistency gain and information loss in stepwise inconsistency resolution. In: Proceedings ECSQARU'11. pp. 362–373 (2011)
14. Grant, J.: Classifications for inconsistent theories. Notre Dame Journal of Formal Logic **19**(3), 435–444 (1978)
15. Grant, J., Martinez, M.V. (eds.): Measuring Inconsistency in Information, Studies in Logic, vol. 73. College Publications (2018)
16. Hunter, A., Konieczny, S.: Measuring inconsistency through minimal inconsistent sets. In: Proceedings KR'08. pp. 358–366 (2008)
17. Hunter, A.: How to act on inconsistent news: Ignore, resolve, or reject. Data & Knowledge Engineering **57**(3), 221–239 (2006)
18. Kuhlmann, I., Thimm, M.: An algorithm for the contension inconsistency measure using reductions to answer set programming. In: 14th International Conference on Scalable Uncertainty Management. pp. 289–296. Springer (2020)
19. Kuhlmann, I., Thimm, M.: Algorithms for inconsistency measurement using answer set programming. In: 19th International Workshop on Non-Monotonic Reasoning (NMR). pp. 159–168 (2021)
20. Lifschitz, V.: Answer Set Programming. Springer Berlin (2019)
21. Martinez, A.B.B., Arias, J.J.P., Vilas, A.F.: On measuring levels of inconsistency in multi-perspective requirements specifications. In: Proceedings of the 1st Conference on the Principles of Software Engineering (PRISE'04). pp. 21–30 (2004)
22. Mironov, I., Zhang, L.: Applications of SAT solvers to cryptanalysis of hash functions. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 102–115. Springer (2006)
23. Potyka, N., Thimm, M.: Inconsistency-tolerant reasoning over linear probabilistic knowledge bases. International Journal of Approximate Reasoning **88**, 209–236 (2017)

24. Priest, G.: The logic of paradox. Journal of Philosophical logic pp. 219–241 (1979)
25. Sinz, C.: Towards an optimal cnf encoding of boolean cardinality constraints. In: International Conference on Principles and Practice of Constraint Programming. pp. 827–831. Springer (2005)
26. Thimm, M.: On the evaluation of inconsistency measures. In: Measuring Inconsistency in Information, vol. 73. College Publications (2018)
27. Thimm, M.: Stream-based inconsistency measurement. International Journal of Approximate Reasoning **68**, 68–87 (2016)
28. Thimm, M., Rienstra, T.: Approximate reasoning with ASPIC+ by argument sampling. In: Proceedings of the Third International Workshop on Systems and Algorithms for Formal Argumentation (SAFA'20). pp. 22–33 (2020)
29. Thimm, M., Wallner, J.P.: On the complexity of inconsistency measurement. Artificial Intelligence **275**, 411–456 (2019)
30. Tseitin, G.S.: On the complexity of derivation in propositional calculus. Structures in Constructive Mathematics and Mathematical Logic pp. 115–125 (1968)
31. Vizel, Y., Weissenbacher, G., Malik, S.: Boolean satisfiability solvers and their applications in model checking. Proceedings of the IEEE **103**(11), 2021–2035 (2015)