FernUniversität in Hagen
Faculty of Mathematics and Computer Science

AIG
Artificial Intelligence Group

Algorithmic Approaches for the Formula-Based Contension Inconsistency Measure

Master's Thesis

in partial fulfillment of the requirements for
the degree of Master of Science (M.Sc.)
in Praktische Informatik

submitted by
Marcel Traser

First examiner: Prof. Dr. Matthias Thimm
Artificial Intelligence Group

Advisor: Isabelle Kuhlmann
Artificial Intelligence Group

Statement

Ich erkläre, dass ich die Masterarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Masterarbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Masterarbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

I explicitly agree to have this thesis published on the webpage of the artificial intelligence group and endorse its public availability.

Software created for this work has been made available as open source; a corresponding link to the sources is included in this work. The same applies to any research data.

Darmstadt 27.07.25 .....

(Place, Date) (Signature)

Zusammenfassung

Eine große Herausforderung im Gebiet der künstlichen Intelligenz liegt im Umgang mit widersprüchlichen Informationen. Inkonsistenzmaße ermöglichen eine quantitative Bewertung des Konfliktgehalts einer Wissensbasis. Diese Masterarbeit widmet sich der Entwicklung und Evaluation von zwei neuen algorithmischen Ansätzen zur Berechnung des formelbasierten Contension-Inkonsistenzmaßes (I_{fc}), welches auf der dreiwertigen Logik von Priest basiert. Der erste Ansatz formuliert das Problem als Optimierungsproblem für (Maximum) Satisfiability (MaxSAT) Solver, während der zweite Ansatz auf dem Answer Set Programming (ASP) aufbaut. Die experimentelle Evaluation auf Basis von synthetischen und aus realen Anwendungen abgeleiteten Datensätzen zeigt einen klaren Kompromiss zwischen den beiden Ansätzen. Der MaxSAT-basierte Ansatz ist bei syntaktisch einfachen Wissensbasen deutlich schneller, wohingegen der ASP-Ansatz eine höhere Robustheit aufweist und bei strukturell komplexen Problemen mehr Instanzen lösen kann. Die Ergebnisse verdeutlichen, dass die praktischen Herausforderungen bei der Berechnung des I_{fc} vor allem die Syntaxsensitivität der Formeln betrifft. Die Arbeit liefert somit neben den zwei Algorithmen auch Einblicke darin, welcher Solver sich für den spezifischen Einsatz unter Verwendung der dreiwertigen Logik am besten eignet.

Abstract

A major challenge in the field of artificial intelligence is dealing with contradictory information. Inconsistency measures allow for a quantitative assessment of the conflict level of a knowledge base. This master's thesis is dedicated to the development and evaluation of two new algorithmic approaches for calculating the formula-based contension inconsistency measure (I_{fc}), which is based on Priest's three-valued logic. The first approach formulates the problem as an optimization problem for (Maximum) Satisfiability (MaxSAT) solvers, while the second approach is based on Answer Set Programming (ASP). The experimental evaluation, based on synthetic and real-world-derived datasets, shows a clear trade-off between the two approaches. The MaxSAT-based approach is significantly faster for syntactically simple knowledge bases, whereas the ASP approach exhibits greater robustness and can solve more instances in structurally complex problems. The results clarify that the practical challenges in computing the I_{fc} is primarily the syntax sensitivity of the formulas. Thus, in addition to the two algorithms, the work also provides insights into which solver is best suited for the specific application using three-valued logic.

Contents

List of Tables

List of Figures

1. Introduction

The increasing aggregation of data from heterogeneous sources challenges knowledge based systems in artificial intelligence with the handling of inconsistent information. A key approach to this issue is the quantitative measurement of contradictions through inconsistency measures. This Master's thesis is dedicated to the development and evaluation of two new algorithmic approaches for the calculation of a specific measure based on three-valued logic. This introductory chapter motivates the need of this research, defines the central objectives and research questions, and provides an overview of the structure of the entire work.

1.1. Motivation and Problem statement

In the age of digitalization and artificial intelligence, knowledge based systems collect a constantly growing amount of information from diverse and often heterogeneous sources [BH08]. Whether in autonomous robotics, medical diagnostics, or the integration of enterprise databases, the ability to handle large amounts of data and gather logical conclusions from them is very important [Thi19]. However, this data aggregation leads to a fundamental challenge: the emergence of contradictory information, which are called inconsistencies. Conflicts within a knowledge base occure very often, and they can significantly impair the reliability and functionality of a system by leading to faulty conclusions [KT20, Thi13].

Classical logic, which is based on the principle of non-contradiction, is not always suitable for handling such inconsistent knowledge bases, since any arbitrary statement can be derived from a single contradiction [GS16, Hun07, HK08]. Instead of summarily dismissing an inconsistent knowledge base as unusable, previous researches have established approaches that aim to deal with contradictions. A central step in this process is the quantification of the degree of inconsistency [HK08]. Inconsistency measures are functions that assign a numerical value to a knowledge base, representing its degree of conflict. These kinds of measures make the comparison of different knowledge bases possible.

A promising approach to defining such a measure is based on paraconsistent logics, which are able to tolerate contradictions [Thi13]. The formula-based contension inconsistency measure considered in this thesis uses Priest's three-valued logic [KGLT22]. This logic extends the classical truth values of *true* and *false* with a third value: *both*. In this system, a formula is considered satisfied if it evaluates to *true* or *both*. This makes it possible to satisfy even classically contradictory sets of formulas. The measure therefore quantifies inconsistencies by identifying the minimum number of formulas containing any atom valued both. Higher values therefore indicate a greater inconsistency.

1.2. Objectives of the study

The central goal of this Master's thesis is the design, implementation, and experimental evaluation of two new algorithmic approaches for the calculation of the formula-based contension inconsistency measure. To explore the strengths of different solution strategies for this problem, two established paradigms of declarative programming and constraint solving will be employed. Answer Set Programming (ASP) is a declarative programming paradigm that is particularly well-suited for modeling and solving combinatorial search and optimization problems. (Maximum) Satisfiability Solving ((Max)SAT) is an extension of the classical propositional satisfiability problem that allows for the formulation and solution of optimization problems.

Therefore, the research questions to be answered in this thesis are as follows:

- How can the problem of calculating the formula-based contension inconsistency measure be formulated as MaxSAT encoding?
- How can the problem of calculating the formula-based contension inconsistency measure be formulated as a logic program within the Answer Set Programming framework?
- How do the two developed approaches compare in performance? An experimental evaluation will show the performance and scalability of the algorithms using different benchmark instances.

The results of this work aim to provide two practical algorithms for calculating this specific inconsistency measure. They additionally provide deep insights into the suitability of ASP and MaxSAT for problems in paraconsistent logic.

1.3. Structure of the thesis

The work is primarily divided into a theoretical and a practical part. The theoretical part covers the logical foundations, a general definition of inconsistency measures, and the transition to the formula-based contension inconsistency measure. Furthermore, there is an introduction to Answer Set Programming and (Max)SAT solvers.

The practical part of the work builds on the theoretical explanations and aims to develop two different algorithmic approaches. The ASP-based approach is about formulating the problem as a declarative programming approach that produces multiple answer sets as solutions. Whereas the (Max)SAT-based approach formulates the problem as a (Max)SAT problem and implementation of a corresponding solver.

In a further step, the algorithms will be evaluated experimentally. For this purpose, a collection of datasets is used that has already been used in the research of [KGLT22]. In one run-through, all knowledge bases of a dataset are processed and

the computation time is determined for each. If a knowledge base can't be processed within a specified time it runs into a timeout and the algorithm skips to the next knowledge base. The algorithms can finally be compared based on their running time and the underlying dataset.

2. Theoretical Background

This chapter comes up with the theoretical foundations of classical logic, provides an introduction to Graham Priest's three-valued logic, and offers a general definition as well as an overview of existing inconsistency measures. These fundamentals are essential for the following chapters and form the necessary basic knowledge for the programming and implementation of the algorithms.

2.1. Logical Foundations

Mathematical logic is a subfield of mathematics that deals with the formal analysis of arguments and proofs. The main focus is on classical logic, which is based on three principles. The principle of bivalence states that a statement can either be *true* or *false* [Ebb18]. A statement can also not be *true* and *false* at the same time, which leads to the principle of non-contradiction. Finally, for every statement, either it itself or its negation holds [Men15, BL12].

2.1.1. Propositional Logic

Propositional logic is a subfield of mathematical logic that deals with statements, their truth values, and the logical connections between them. The simplest form of a statement is the atom, which can take on the truth value *true* or *false* [Ebb18]. The literal extends the atom by the negation of the same. Literals can also be viewed as the simplest form of a formula, which can be extended with other atoms or literals using logical operators to form new statements.

Definition 1 (Truth Values)**.** The truth values are $\{t, f\}$, representing *true* and *false*.

The most common logical operators are the $\neg$ negation, $\wedge$ conjunction, $\vee$ disjunction, $\rightarrow$ implication and $\leftrightarrow$ equivalence [Kle04]. The conjunction as logical AND, as well as the disjunction as logical OR, include or exclude individual atoms or other formulas as a condition for the fulfillment of the statement. The implication is a logical consequence which is derived from the state of another atom in the form of a condition. Note that an implication is *false* only if the premise is *true* and the conclusion is *false*. A logical equivalence requires the same truth value and the leading conditions for the formation of the truth value [Sip12, Lif19].

An atom p could be translated into the positive literal p and the negative literal $\neg p$. A formula $\neg p \wedge q$ is read as "not p and q". The formula becomes *true* if p is

not *true* and q is *true*. In any other case, the statement is to be evaluated as *false*. The tautology, as a special form of a statement, is always *true* regardless of the truth values of its constituent parts. A contradiction is the opposite of tautology and is always *false*.

Definition 2 (Alphabet of Propositional Logic)**.** Let $\mathcal{A}$ be a countable set of atomic propositions (e.g., p, q, r). The set of logical connectives is $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$. Parentheses (and) are used for grouping.

The alphabet defines all the valid symbols that can be used to construct logical statements. The atoms are the basic building blocks, and the connectives are the tools to combine them. From here we can provide a definition for constructing complex and syntactically correct statements from simple atoms.

Definition 3 (Formula)**.** The set of well-formed formulas $\mathcal{F}$ is defined inductively:

- Every $p \in \mathcal{A}$ is a formula.
- If $\varphi \in \mathcal{F}$, then $\neg\varphi \in \mathcal{F}$.
- If $\varphi, \psi \in \mathcal{F}$, then $(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \rightarrow \psi), (\varphi \leftrightarrow \psi) \in \mathcal{F}$.

A knowledge base represents a collection of statements, facts, or beliefs that are to be considered together [EIST09, Thi13]. For computational purposes, it is typically assumed to be a finite set [KT21].

Definition 4 (Knowledge base)**.** A knowledge base $\mathcal{K}$ is a finite set of formulas of propositional logic. Let $\mathbb{K}$ then be the set of knowledge bases $\mathcal{K}$.

Definition 5 (Set of Atoms)**.** Let $\mathrm{At}(\mathcal{F})$ denote the set of all atoms occurring in the set of formulas $\mathcal{F}$. If φ is a single formula, then $\mathrm{At}(\varphi)$ is equivalent to $\mathrm{At}(\{\varphi\})$, where $\{\varphi\}$ represents the singleton set containing only the formula φ.

This function can be used for extracting the set of all propositional variables that a formula or knowledge base contains.

Example 1. Let the alphabet of propositional logic be defined by the countable set of atomic propositions $\mathcal{A} = \{p, q, r\}$ and the usual logical connectives $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$, along with parentheses.

Consider the following knowledge base $\mathcal{K} \in \mathbb{K}$, consisting of three well-formed formulas $\varphi_1, \varphi_2, \varphi_3 \in \mathcal{F}$:

$$\mathcal{K} = \{\varphi_1, \varphi_2, \varphi_3\}$$

where

$$\varphi_1 = p \rightarrow (q \wedge r)$$
$$\varphi_2 = \neg q \vee r$$
$$\varphi_3 = \neg(p \leftrightarrow r)$$

The set of all atomic propositions occurring in $\mathcal{K}$ is:

$$\mathrm{At}(\mathcal{K}) = \{p, q, r\}$$

and, for instance, the set of atoms in φ_2 is:

$$\mathrm{At}(\varphi_2) = \{q, r\}$$

Definition 6 (Interpretation)**.** Let $\omega \colon \mathrm{At}(F) \to \{t, f\}$ be an interpretation, which assigns each atom α one of the truth values t or f.

An interpretation can be thought of as one possible state where every proposition is assigned a definite truth value.

The following rules formally define the semantics of the logical connectives. They specify exactly how the truth value of a complex formula is calculated from the truth values of its simpler components, based on the initial interpretation of the atoms.

Definition 7 (Truth Assignment)**.** The truth value of a formula φ under an interpretation ω is defined as following:

- $\omega(p) = t$ or f for $p \in \mathcal{A}$.
- $\omega(\neg\varphi) = t$ if $\omega(\varphi) = f$; otherwise, $\omega(\neg\varphi) = f$.
- $\omega(\varphi \wedge \psi) = t$ if $\omega(\varphi) = t$ and $\omega(\psi) = t$; otherwise, $\omega(\varphi \wedge \psi) = f$.
- $\omega(\varphi \vee \psi) = t$ if $\omega(\varphi) = t$ or $\omega(\psi) = t$; otherwise, $\omega(\varphi \vee \psi) = f$.
- $\omega(\varphi \to \psi) = f$ if $\omega(\varphi) = t$ and $\omega(\psi) = f$; otherwise, $\omega(\varphi \to \psi) = t$.
- $\omega(\varphi \leftrightarrow \psi) = t$ if $\omega(\varphi) = \omega(\psi)$; otherwise, $\omega(\varphi \leftrightarrow \psi) = f$.

A model, therefore, represents a scenario in which the entire knowledge base holds *true* [BHvMW09, Lif19]. If a knowledge base has at least one model, it is called satisfiable or consistent [Ebb18, KT20]. If it has no model, it is unsatisfiable or inconsistent [Gra78].

Definition 8 (Model)**.** A model is an interpretation that makes a given set of formulas *true*. It satisfies all formulas of a knowledge base $\mathcal{K}$. This means that every statement in $\mathcal{K}$ is *true* under this interpretation [NKTJ23].

In various scenarios, it can be useful or even necessary to extend the classical logic. Statements that, due to insufficient information or their contradictory nature, would have to be *true* and *false* at the same time, cannot be represented by classical logic, which is why further approaches are required [BL12]. In addition to fuzzy logic, in which gradual membership is determined based on a scale, there are various forms of multi-valued logic [KY95].

Example 2. Let $\mathcal{A} = \{p, q, r\}$ be the set of atoms, and consider the following knowledge base $\mathcal{K}$:

$$\mathcal{K} = \{\varphi_1, \varphi_2\}$$

with

$$\varphi_1 = p \rightarrow (q \wedge r)$$
$$\varphi_2 = \neg r \vee q$$

Define an interpretation $\omega \colon \mathcal{A} \rightarrow \{t, f\}$ by:

$$\omega(p) = t, \quad \omega(q) = t, \quad \omega(r) = t$$

We evaluate the truth values of the formulas in $\mathcal{K}$ under ω:

- $\omega(\varphi_1) = \omega(p \rightarrow (q \wedge r)) = t \rightarrow (t \wedge t) = t \rightarrow t = t$
- $\omega(\varphi_2) = \omega(\neg r \vee q) = \neg t \vee t = f \vee t = t$

Since both φ_1 and φ_2 evaluate to t under ω, it follows that ω is a *model* of $\mathcal{K}$

$$\omega \models \mathcal{K}$$

So we can say, that the knowledge base $\mathcal{K}$ is satisfiable.

2.1.2. Boolean Satisfiability Problem

The Boolean Satisfiability Problem (SAT) addresses the question of whether there is a truth assignment for a given propositional logic formula that makes the formula *true*. For example, the formula $A \wedge B$ is satisfiable, because there exists an interpretation ω with $\omega(A) = t$ and $\omega(B) = t$, such that $\omega(A \wedge B) = t$ [BHvMW09].

Generally, such satisfiability problems can be classified into complexity classes. Big $\mathcal{O}$ notation is used here to describe the asymptotic behavior of functions, particularly the upper bound of the runtime or memory requirements of an algorithm depending on the size of the input. The $\mathcal{O}$ stands for *order of* and characterizes how quickly the resource requirement grows in the worst case [Păt08]. The class P represents a set of problems that can be solved by a deterministic algorithm within polynomial runtime. An algorithm runs in polynomial time if its runtime is bounded by a polynomial in the size n of the input (e.g., $\mathcal{O}(n^k)$ for a constant k). So it grows relatively slowly with increasing input size. Algorithms that run in polynomial time are considered efficient [Sip12].

Determinism is defined by the fact that the output of an algorithm remains identical with each run for the same input. Non-deterministic algorithms, on the other hand, are characterized by random decisions or parallel processing and are considered to be more efficient than deterministic algorithms [Sip12]. In practice, the results of such algorithms can only be used to a limited extent due to the non-unique

output. The class NP accordingly comprises all problems of class P, as well as problems that can be solved by non-deterministic algorithms in polynomial time [Sip12]. Although SAT is in NP, the best-known deterministic algorithms for solving the general SAT problem are exponential in the worst case. This will result for example in runtimes on the order of $\mathcal{O}(2^n)$, where n is the number of variables.

Figure 1 shows class P as a subset of class NP. Furthermore, a distinction must be made between the classes NP-Hard and NP-Complete. A problem is NP-Hard if it is at least as difficult as the most difficult problems in class NP. NP-Complete comprises the most difficult problems in class NP and forms the intersection between NP and NP-Hard [BM07]. A central method in this context is reduction. A reduction transforms an instance of a problem A into an instance of a problem B such that a solution for the B-instance provides a solution for the A-instance. If this transformation itself can be performed in polynomial time, it is called a polynomial-time reduction $A \leq_p B$. It serves to compare the relative difficulty of problems: If a known hard problem A can be reduced to problem B, then B is at least as hard as A [BHvMW09]. Since every problem in NP can be reduced in polynomial time to any NP-Complete problem, they are considered the most difficult problems in NP. SAT is considered the first problem for which NP-Completeness was proven [NKTJ23], which implies that there is no deterministic algorithm with polynomial runtime for it.

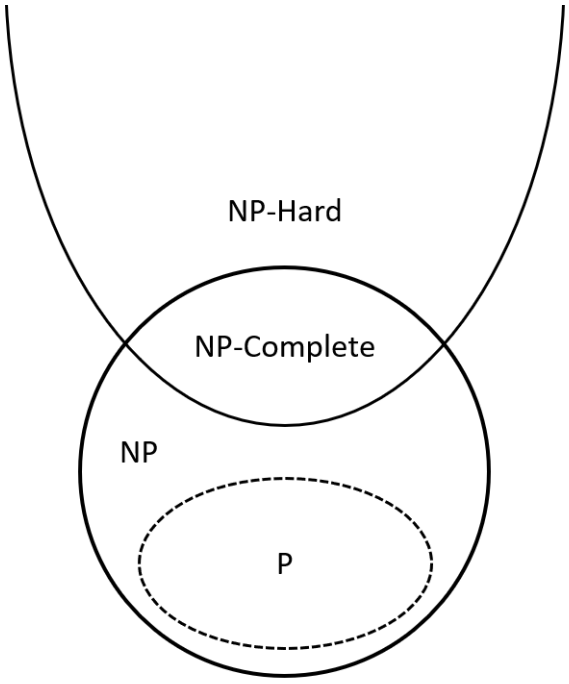


Figure 1: Complexity classes of decision problems

An unsolved question in mathematics is whether the classes NP and P might not be identical after all. To prove this, a deterministic and efficient algorithm is searched that can solve a problem of the NP-Complete class in polynomial time. All problems in class NP, including class P, could then be reduced to this algorithm [Sip12].

In the following chapters, the presented inconsistency measures and their algorithms will be examined more closely for their complexity class, because this plays a significant role in comparability and evaluation [KGLT22, Thi18].

2.1.3. Priest's Three-Valued Logic

Graham Priest, with his three-valued logic in the subfield of non-classical logic, broke the principle of bivalence and defined a third value, *both*. Concretely, this means that a statement can be both *true* (t) and *false* (f) and thus *both* (b). Although the idea of a third truth value may seem unusual at first, there are still some scenarios in which it is at least not clearly recognizable which truth value is correct. Three-valued logic creates a way to deal with such uncertainties or contradictions in statements [Pri79].

Definition 9 (Truth Values in Three-Valued Logic)**.** The set of truth values is $\{t, f, b\}$, where b represents *both true and false*.

Definition 10 (Interpretation in Three-Valued Logic)**.** Let $\omega^3 \colon \mathrm{At}(\mathcal{F}) \to \{t, f, b\}$ be a three-valued interpretation, which values each atom α with one of the truth values t, f or b.

Example 3. Consider the following statement made by a person:

> "I am lying."

If this statement is true, then the person is lying, which implies the statement is false. But if it is false, then the person is not lying, and hence the statement is true. This creates a classical paradox which cannot be resolved using only the classical truth values t and f.

In three-valued logic, we can assign the truth value b to the statement to capture this contradiction. Let the atom a represent the statement "I am lying". Then we define the interpretation ω^3 as:

$$\omega^3(a) = b$$

This assignment reflects the fact that the statement is simultaneously true and false and avoids the inconsistency in classical logic.

The syntax of three-valued logic is similar to that of classical logic and uses the usual connectives. A truth table (Table 1) illustrates the relationships.

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
t	t	f	t	t	t	t
t	f	f	f	t	f	f
t	b	f	b	t	b	b
f	t	t	f	t	t	f
f	f	t	f	f	t	t
f	b	t	f	b	t	b
b	t	b	b	t	t	b
b	f	b	f	b	b	b
b	b	b	b	b	t	t

Table 1: Truth Table

For the three truth values, a value ordering can be established: $f \prec b \prec t$, where $\prec$ means that the left side is less *true* than the right side. In classical logic, the statement $p \wedge \neg p$ would lead to a clear contradiction [NKTJ23]. According to Priest, p could also be assigned the value *both* which resolves the inconsistency [Pri79].

Definition 11 (Truth Assignment in Three-Valued Logic)**.** The truth value of a formula φ under an interpretation ω^3 is defined as follows:

- $\omega^3(p) = t, f,$ or b for $p \in \mathcal{A}$.
- $\omega^3(\neg\varphi) = t$ if $\omega^3(\varphi) = f$; f if $\omega^3(\varphi) = t$; b if $\omega^3(\varphi) = b$.
- $\omega^3(\varphi \wedge \psi) = \min(\omega^3(\varphi), \omega^3(\psi))$ with the order $f \prec b \prec t$.
- $\omega^3(\varphi \vee \psi) = \max(\omega^3(\varphi), \omega^3(\psi))$.
- $\omega^3(\varphi \rightarrow \psi) = \begin{cases} t, & \text{except if} \\ & \omega_3(\varphi) = t \text{ and } \omega_3(\psi) = f, \text{ then } \omega_3(\varphi \rightarrow \psi) = f; \\ & \omega_3(\varphi) = t \text{ and } \omega_3(\psi) = b \text{ then } \omega_3(\varphi \rightarrow \psi) = b. \\ & \omega_3(\varphi) = b \text{ and } \omega_3(\psi) = f \text{ then } \omega_3(\varphi \rightarrow \psi) = b. \end{cases}$
- $\omega_3(\varphi \leftrightarrow \psi) = \begin{cases} t, & \text{if } \omega_3(\varphi) = \omega_3(\psi); \\ b, & \text{if either } \omega_3(\varphi) = b \text{ or } \omega_3(\psi) = b). \\ f, & \text{else.} \end{cases}$

Definition 12 (Three-Valued Model)**.** A three-valued interpretation ω^3 is a model of a knowledge base $\mathcal{K}$ if $\omega^3(\varphi) \in \{t, b\}$ for all $\varphi \in \mathcal{K}$

Example 4. Let $\mathcal{A} = \{p, q\}$ be a set of atoms, and define the three-valued interpretation ω^3 as follows:

$$\omega^3(p) = b, \quad \omega^3(q) = f$$

Consider the knowledge base:

$$\mathcal{K} = \{\varphi_1, \varphi_2\} \quad \text{with} \quad \begin{array}{l} \varphi_1 = \neg p \vee q \\ \varphi_2 = p \rightarrow q \end{array}$$

We now evaluate each formula under ω^3:

- $\omega^3(\varphi_1) = \omega^3(\neg p \vee q) = \max(\omega^3(\neg p), \omega^3(q)) = \max(b, f) = b$
- $\omega^3(\varphi_2) = \omega^3(p \rightarrow q) = b \rightarrow f = b$

Since both formulas evaluate to b, and b is accepted as a satisfying value in three-valued logic (i.e., $\omega^3(\varphi) \in \{t, b\}$) we can say:

$$\omega^3 \in \text{Models}(\mathcal{K})$$

So we can say that ω^3 is a three-valued model of $\mathcal{K}$.

Table 1 also makes it clear that three-valued logic overrides the classical rules of inference and is not readily applicable in every scenario. Thus, in many areas of application such as philosophy, semantics, and computer science, in addition to the opportunities for analyzing paradoxes, modeling ambiguous concepts or developing fault-tolerant systems, some criticisms are also mentioned [Pri79].

Firstly, the interpretation of the value *both* can be difficult and lead to philosophical debates, as the result may contradict human intuition. Secondly, the logic gains in a wider range of truth-values, which is particularly evident from the truth table. Analyses and an understanding of the statements could generally be more difficult. Last but not least, the use of the value *both* results in a loss of information, as statements are now no longer unambiguous.

2.2. Basics of SAT and MaxSAT

The Boolean satisfiability problem (SAT) aims to determine whether a given propositional logic formula can be made *true* by assigning appropriate truth values to its variables [BHvMW09]. SAT solvers are specialized algorithms that assess the satisfiability of formulas.

In contrast, MaxSAT solvers address the optimization variant of SAT. Rather than simply determining if a formula is satisfiable, MaxSAT aims to find a truth-value assignment that satisfies the maximum number of clauses in a given CNF formula [BHvMW09]. To model this, a formula is often divided into two types of clauses. Hard clauses must be satisfied in any valid solution while soft clauses are optional. The goal is to satisfy as many of them as possible.

Additionally, soft clauses can be weighted to instruct the solver which rules to prioritize [NKTJ23]. MaxSAT is an NP-hard problem, meaning that no known polynomial-time algorithm can solve all instances [BHvMW09]. Despite this, modern MaxSAT

solvers apply demanding techniques to solve practical instances efficiently. Exact MaxSAT solvers typically use branch-and-bound strategies, where the search space is systematically explored and pruned using upper and lower bounds. Additional methods include variable selection heuristics, clause learning, and efficient data structures [BHvMW09].

A common technique for solving MaxSAT problems is the reduction to SAT [ABL13]. This involves reformulating the optimization problem so that it can be solved by one or more calls to a standard SAT solver. A well-known approach for this uses blocking variables, where soft clauses are modified with helping variables. The objective then shifts from maximizing the number of satisfied clauses to minimizing the number of activated blocking variables, which is often achieved through iterative calls to a SAT solver.

2.2.1. Davis-Putnam-Logemann-Loveland Algorithm

SAT and MaxSAT solvers are originally based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [Lif19], which was designed to solve decision problems (SAT). It is based on backtracking, systematically searching through all possible truth assignments.

Its operation can be broken down into three core operations applied in a recursive process. Starting from an empty truth assignment ω, this process is repeated:

1. Deterministic Simplification: Using Unit Propagation, the last unassigned literal within a clause is assigned the truth value which is necessary to satisfy the clause [DLL62, Wor24]. Let $C = (l \vee l_1 \vee \cdots \vee l_n)$ be a clause in F such that under the two-valued assignment ω, it leads to: $\omega(l_1) = \cdots = \omega(l_n) = f$, and the literal l is still unassigned. Then the assignment ω is extended with the assignment for the atom of l that makes l *true*. If a literal l appears in the yet unsatisfied clauses of F, but its complementary literal $\neg l$ does not, the assignment ω is extended with the assignment for the atom of l that makes l *true*. This process is called Pure Literal Elimination [GKSS08, BHvMW09].

2. Decision: If no further simplification is possible, an unassigned variable is chosen, and a truth value is speculatively assigned to it.

3. Backtracking: In the final step, this randomized assignment is evaluated. If a conflict occurs, meaning that a clause cannot be satisfied, the last decision is undone, and the alternative truth value is tried. This process is called backtracking [GKSS08]. If the algorithm tried all possible assignments without finding a solution, the formula is considered unsatisfiable.

Modern SAT and MaxSAT solvers use Conflict-Driven Clause Learning (CDCL) which is an extension of DPLL. The algorithm is enhanced with the ability to analyze conflicts and learn new clauses from them, which are then added to the set of

formulas. This can significantly reduce the time required to find a solution [GKSS08, Wor24].

Since DPLL/CDCL solves decision problems and MaxSAT is designed for optimization problems, some adaptations are necessary. The most common is the Branch and Bound method. Here, the backtracking search is extended with a cost function and a so-called pruning rule [BHvMW09].

The cost of a given truth assignment is the number of falsified clauses C_i in a formula F.

$$cost(\omega) = |\{C_i \in F \mid \omega(C_i) = f\}|$$

The goal is to minimize this cost. During the search, the solver stores the best solution with the lowest cost found so far. This is called the upper bound (UB). The lower bound (LB) for a partial assignment ω_p represents the cost already incurred by the assignments made so far.

$$LB(\omega_p) = |\{C_i \in F \mid \omega(C_i) = f\}|$$

The central optimization of the method is the pruning rule. At any point in the search, if $LB(\omega_p) \geq UB$, the current search branch is canceled, because it cannot compute a better solution, and backtracking is initiated.

2.2.2. Conjunctive Normal Form

A normal form describes a syntactic restriction for formulas. For each formula there must be a equivalent formula of its normal form like disjunctive normal form (DNF) or negative normal form (NNF) [BM07].

A necessary prerequisite for applying the DPLL algorithm is that the input formula F is expressed in Conjunctive Normal Form (CNF) [DLL62]. A formula in CNF is a conjunction of clauses C_i where each clause is a disjunction of literals l_{ij}:

$$F_{CNF} = C_1 \wedge C_2 \wedge \cdots \wedge C_m, \quad \text{where } C_i = (l_{i1} \vee l_{i2} \vee \cdots \vee l_{ik})$$

There are several methods to transform F into CNF. A naive CNF transformation recursively rewrites logical connectives by applying equivalence laws until only conjunctions of disjunctions remain.

1. Eliminate equivalences and implications:
   $A \rightarrow B \equiv \neg A \vee B$,
   $A \Leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$

2. Push negations inward using De Morgan's laws:
   $\neg(A \wedge B) \equiv \neg A \vee \neg B$,
   $\neg(A \vee B) \equiv \neg A \wedge \neg B$

3. Apply the distributive law:
   $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$

Although simple to implement, this method can lead to an exponential blow-up in the size of the resulting CNF due to formula duplication.

In contrast, the Tseitin transformation offers a more efficient alternative by introducing new propositional variables for subformulas. Each subformula is associated with a new variable, and constraints are added to ensure logical equivalence between the original formula and the introduced variables. The key advantage of the Tseitin transformation is that it preserves satisfiability and ensures the resulting CNF grows only linearly in the size of the original formula [Tse68]. However, it does not preserve logical equivalence, only satisfiability [KKS$^+$22].

Definition 13 (Tseitin Transformation [Tse68, KKS$^+$22])**.** Let φ be a propositional formula. The Tseitin transformation, denoted $T(\varphi)$, recursively constructs a formula in CNF that is equisatisfiable to φ. The process is as follows:

1. A new variable h_ψ is introduced for each non-atomic subformula ψ of φ.
2. For each such subformula ψ, a set of clauses C_ψ is generated to enforce the logical equivalence $h_\psi \leftrightarrow \psi$. The structure of C_ψ depends on the main connective of ψ:
   - If $\psi = a \wedge b$, the clauses for $h_\psi \leftrightarrow (a \wedge b)$ are:
     $$(\neg h_\psi \vee a) \wedge (\neg h_\psi \vee b) \wedge (h_\psi \vee \neg a \vee \neg b)$$
   - If $\psi = a \vee b$, the clauses for $h_\psi \leftrightarrow (a \vee b)$ are:
     $$(h_\psi \vee \neg a) \wedge (h_\psi \vee \neg b) \wedge (\neg h_\psi \vee a \vee b)$$
   - If $\psi = \neg a$, the clauses for $h_\psi \leftrightarrow \neg a$ are:
     $$(\neg h_\psi \vee \neg a) \wedge (h_\psi \vee a)$$
   - If $\psi = a \rightarrow b$, the clauses for $h_\psi \leftrightarrow (a \rightarrow b)$ are:
     $$(h_\psi \vee a) \wedge (h_\psi \vee \neg b) \wedge (\neg h_\psi \vee \neg a \vee b)$$

   Here, a and b can be either original variables from φ or the helper variables corresponding to their respective subformulas.
3. The final transformed formula, $T(\varphi)$, is the conjunction of the helper variable for the entire formula (h_φ) and all generated equivalence clauses:
   $$T(\varphi) = (h_\varphi) \wedge \bigwedge_{\psi \in \mathrm{Sub}(\varphi)} C_\psi$$
   where $\mathrm{Sub}(\varphi)$ is the set of all non-atomic subformulas of φ.

The Weighted Conjunctive Normal Form (WCNF) is an extension of CNF for MaxSAT problems. The individual clauses are weighted so that a prioritization is achieved. Hard clauses are valued the highest possible weight, therefore they have to be fulfilled. However, the assigned weight for soft clauses has to be smaller than the highest possible value and can be injured. The goal of a solver is now to fulfill all hard clauses and to maximize the sum of all weights of satisfied soft clauses.

2.2.3. Types of MaxSAT Solver

There is a multitude of MaxSAT solvers, which all differ in their algorithms and specializations for solving certain types of problems. These solvers can be divided into the following categories:

- Branch and Bound Solvers: This approach extends the backtracking search based on DPLL/CDCL with a cost function and a pruning rule to efficiently prune the solution space.
- Core-guided Solvers: The solver first attempts to find a solution that satisfies all clauses (hard and soft). If this fails, the SAT solver returns a set of clauses that causes the contradiction [ABL13]. The algorithm then strategically relaxes one or more soft clauses from this core. This process is repeated until a satisfiable set of clauses is found and the solver can prove that the cost, or in other words the number of relaxed soft clauses, is minimal [BHvMW09, IMM19].
- Implicit Hitting Set Solvers: Unsatisfiable cores are collected iteratively. The problem is then reformulated as finding a minimal set of soft clauses that hits each of these conflict sets (cores) [BHvMW09]. This minimum hitting set with the lowest cost corresponds to the optimal solution of the MaxSAT problem.

One of the most well-known and powerful representatives of modern MaxSAT solvers is RC2. Its core technique is an efficient encoding of relaxable cardinality constraints, which are used for the stepwise relaxation of soft clauses [IMM19]. As a core-guided solver, it is based on an efficient implementation of strategies such as the MaxSAT Using Unsatisfiable Cores (MSU3) or One-Literal Learning (OLL) algorithms [IMM19, BHvMW09].

RC2 has regularly been ranked as one of the top performers, because of its high performance [IMM19]. A significant advantage for practical application is its availability within popular libraries like pySAT, which enables its direct use in a Python environment.

2.3. Introduction to ASP

Answer Set Programming (ASP) is a declarative programming paradigm based on the stable model semantics of logic programs [Lif19]. In contrast to procedural programming languages, where the programmer specifies the algorithm to solve

a problem step-by-step, in ASP one describes the problem itself in terms of logical rules. An answer set solver, a specialized software program, then automatically computes the answer sets of the program [SPBS03, Lif19].

The focus lies on describing the problem, not on the solution procedure. This makes ASP particularly elaboration tolerant so new requirements or constraints can be easily integrated by adding new rules [SPBS03].

2.3.1. Grounder and Solver

Even if ASP solver don't need a CNF to proceed knowledge bases, a grounded program still is mandatory. Grounding describes a process where variables get substituted through constants to transform the program from a predicate logic into a propositional form [BET11, KGLT22].Therefore all terms are constituted, which result out of the constants and function symbols defined in the program. For each rule these terms will be the ground instances [EIST09].

Grounding is decisive for the performance of the ASP solver. A naive grounding would be, similar to a naive CNF transformation, increase the number of rules in an exponential way. This is the reason why there are more effective techniques needed, like e.g. the lazy grounding, on which only solution relevant rules are generated [EIST09, SPBS03].

The Potsdam Answer Set Solving Collection (Potassco) includes various tools for ASP like the grounder Gringo, the conflict driven answer set solver Clasp and Clingo.

Clasp is therefore used to find the final answer sets. It needs the preprocessed grounded logic programme and makes use of techniques like clause learning and back-jumping [SPBS03]. ASP-Solver like Clasp make also use of the Davis-Putnam-Logemann-Loveland algorithm, even if it was primarly designed for computing SAT [BET11, Lif19].

Clingo integrates both Gringo and Clasp into a single tool, which performs grounding and solving in one unified process. As such, Clingo can directly process logic programs that include variables, constants and basic logical constructs [Lif19].

2.3.2. Syntax

A logic program consists of a set of rules that are formed as the following [SPBS03]:

$$p_1 \mid \ldots \mid p_k \leftarrow q_1, \ldots, q_m, \text{ not } q_{m+1}, \ldots, \text{ not } q_n.$$

where p_i and q_j are literals and not denotes the negation. The head of the rule consists of the disjunction of the literals p_1 through p_k while the body is made up of the positive literals p_1 to q_m and the negated literals not q_{m+1} through not q_n. There are some special cases of rules.

Facts for example are rules with a non-empty head and an empty body. They unconditionally state that the head is *true*. The fact

$$atom(a)$$

declares that the constant *a* is an instance of the predicate *atom* [Lif19, BET11]. Constraints are rules with an empty head, used to eliminate answer sets that satisfy certain conditions:

$$\leftarrow condition_1,\ \text{not}\ condition_2.$$

This rule forbids any answer set in which $condition_1$ is *true* and $condition_2$ is not *true* [SPBS03, EIST09].

A key feature of ASP is the distinction between two types of negation. Default negation is denoted by *not* and expresses that a literal cannot be proven. Classical negation, on the other hand, is represented by a leading minus sign and allows for the explicit modeling of negative facts [BET11, Lif19].

Another important construct in ASP are choice rules. They allow selecting subsets of atoms into the answer set. The general form is:

$$LB\{p_1, \ldots, p_k\}UB \leftarrow \text{body}.$$

This states that if the body is satisfied, the literals between LB (lower bounds) and UB (upper bounds) of the atoms from the set $\{p_1, \ldots, p_k\}$ must be included in the answer set [Lif19].

Clingo's concrete syntax also supports aggregates, which allow reasoning over collections of terms. Common aggregates include `#count`, `#sum`, `#min`, and `#max`. For example, the number of active students can be counted with:

$$number\_active\_students(N) \leftarrow N = \#count\{S : student(S), active(S)\}.$$

Based on aggregates, optimization statements can be used to find solutions that are optimal with respect to some criterion. The `#minimize` and `#maximize` constructs instruct the solver to find answer sets that minimize or maximize the value of a specified expression [BET11].

Finally, it is conventional to write predicate symbols and constants in lowercase letters, while variables are written in uppercase [Lif19, KGLT22]. This distinction is important to avoid unintended variable binding and to ensure correct grounding of rules.

2.3.3. Semantics of Stable Models

The stable model semantics was introduced by Gelfond and Lifschitz [GL88] and forms the foundation of how ASP programs are interpreted and solved. It defines the answer sets, which are consistent sets of atoms that satisfy all rules of the program. In contrast to classical logic, ASP is non-monotonic, meaning that adding new rules can change the set of answer sets.

The mathematical definition of a stable model is based on the concept of the Gelfond-Lifschitz reduct [EIST09].

Definition 14 (Stable Models [EIST09, MT99])**.** An interpretation M of a program P is a stable model of P if M is equal to the least model of the reduct P_M of P with M.

The construction of the reduct P_M for a variable free (grounded) program P and an interpretation M is made in two steps. First, all rules that contain a default negated literal *not* α in the body where the atom α is in the interpretation M ($a \in M$) have to be removed. The intuition here is that if α is in M then *not* α is considered false, and thus this rule cannot apply. After that, removing all remaining default-negated literals *not* α from the bodies of the remaining rules is neccessary. Here, it is assumed that α is not in M ($\alpha \notin M$), so that *not* α is considered *true* and can be removed.

The result of this process is a positive program, meaning that it contains no default negations. A positive program always has a unique least model [Lif19, EIST09]. The stability of the interpretation M comes from the fact that this initially assumed interpretation M is exactly the same as the least model of the reduced program P_M. This means that M verifies or confirms itself [Lif19].

For programs with variables, the definition is extended to the grounding of the program. A stable model of a program P is a stable model of its ground instantiation $grnd(P)$ [EIST09].

Stable models should fulfill several properties [EIST09, MT99]:

- Every stable model M of a program P is a model of P. This means that it is compatible with all rules of the program.
- A stable model M is a minimal model of P. It contains the minimal necessary set of facts, which have to be *true*, to consist with the scenario of the program.
- Various stable models of a program are not comparable. This means that if M_1 and M_2 are two different stable models, neither $M_1 \subset M_2$ nor $M_2 \subset M_1$ applies.
- Stable models generalize the semantics of positive (the least model of a positive program is its only stable model) stratified programs (the perfect model of a stratified program is its only stable model) .

In ASP, a program can have zero, one, or multiple stable models. This multi-model nature can be useful for modeling search and optimization problems, where each answer set represents a distinct solution. When multiple stable models exist, there are two main modes of reasoning [EIST09]:

- Brave Reasoning: An atom α is a brave consequence if it is contained in at least one stable model of the program.
- Cautious Reasoning: An atom α is a cautious consequence if it is contained in every stable model of the program.

2.4. Inconsistency Measures

The meaning of the word consistency in logic largely corresponds to that of freedom from contradiction. A set of statements is therefore consistent if no contradiction

can be derived from them. So it is not possible to derive both a statement and its negation at the same time [Hun71].

Inconsistency measures are used to quantitatively measure the degree of logical inconsistency [HK08]. For example, they can provide information about the number of logical contradictions within a collection of statements, which also allows for more in-depth analysis or localization [Gra78]. Such a collection of statements is also called a knowledge base, which can be expanded by deriving rules between the existing statements.

2.4.1. Definition and Properties

A knowledge base $\mathcal{K} = \{A \wedge B, \neg B\}$ is inconsistent, as its statements can not be satisfied simultaneously using classical logic. Inconsistency arises due to the constraints on atom B. It is implicitly asserted by the first and explicitly negated by the second, making it impossible to satisfy all statements concurrently [Hun71].

To handle this inconsistency in $\mathcal{K}$, there are several possibilities:

- Deletion: The simplest method is to delete formulas until the knowledge base becomes consistent. In this example either the statement $A \wedge B$, or $\neg B$ could be deleted [GH11].
- Weakening: Instead of deleting formulas it is possible to weaken them by making them less restrictive. The statement $A \wedge B$ could be transformed into $A \vee B$, so that the contradiction is resolved [GH11].
- Splitting: Splitting a formula into its constituent atoms can isolate these components. The statement $A \wedge B$ can be split into atoms A and B. The knowledge base therefore looks like: $\mathcal{K} = \{A, B, \neg B\}$. Even if the knowledge base remains inconsistent, subsequent resolution steps might be simplified e.g. deleting of some isolated atoms [GH11].
- Paraconsistent logics: Finally, paraconsistent logics, such as Priest's three-valued logic also can be used to dissolve inconsistencies. If B is valued the truth value *both*, the inconsistency gets tolerated.

Each of these handling methods typically results in some degree of information loss. Choosing one of these methods depends on the application, the type of inconsistency and the goals to be achieved. Inconsistency measures aim to quantify such inconsistencies, providing an indication of how inconsistent a knowledge base is. In the following, we formally define an inconsistency measure as a function that assigns a non-negative real number to a knowledge base, representing its degree of inconsistency.

Definition 15 (Non-Negative Extended Reals)**.** Let $\mathbb{R}^{\infty}_{\geq 0} = \{x \in \mathbb{R}^{\infty}_{\geq 0}\} \cup \{\infty\}$ be the set of all non-negative real numbers extended by positive infinity.

Definition 16 (Inconsistency Measure [Thi18, NKTJ23])**.** Let $I : \mathbb{K} \to \mathbb{R}^{\infty}_{\geq 0}$ be a inconsistency measure, where $I(\mathcal{K}) = 0$ iff $\mathcal{K}$ is consistent for each $\mathcal{K} \in \mathbb{K}$

In some cases a knowledge base contains formulas, which can be seen as kind of irrelevant to the measure of inconsistency. These formulas can be differentiated between safe and free formulas. Both of them make use of the concept of minimal inconsistent subsets, which can be thought of the smallestpossible group of formulas leading to a contradiction.

Definition 17 (Minimal Inconsistent Subset [HK08] [Thi18])**.** A subset $M \subseteq \mathcal{K}$ of a knowledge base $\mathcal{K}$ is a Minimal Inconsistent Subset (MIS) if it satisfies the following two conditions:

1. M is inconsistent. ($M \vdash \bot$)
2. Every subset of M is consistent. (For all $M' \subset M$, $M' \nvdash \bot$)

In other words, a MIS is a smallest possible group of formulas from $\mathcal{K}$ that collectively lead to an inconsistency. Removing any single formula from this group makes the remaining group consistent.

Definition 18 (Free Formulas [Thi18])**.** A formula $\varphi \in \mathcal{K}$ is called free if $\varphi \notin \cup MI(\mathcal{K})$. Therefore $Free(\mathcal{K})$ be the set of all free formulas of $\mathcal{K}$.

In other words, a free formula is not involved in any conflict and therefore is no element of any MI of $\mathcal{K}$.

Definition 19 (Safe Formulas [Thi18])**.** A formula $\varphi \in \mathcal{K}$ is called safe if it is consistent and $At(\varphi) \cap At(\mathcal{K} \setminus \{\varphi\}) = \emptyset$. Therefore $Safe(\mathcal{K})$ be the set of all safe formulas of $\mathcal{K}$.

In other words, a safe formula extends the definition of free formulas by a state of isolation. Because there is no intersection of atoms with the rest of the knowledge base, a safe formula can't be involved in any conflict.

Example 5. Let the knowledge base be:

$$\mathcal{K} = \{\varphi_1, \varphi_2, \varphi_3, \varphi_4\}$$

with

$$\begin{aligned}\varphi_1 &= p\\ \varphi_2 &= \neg p\\ \varphi_3 &= q \to r\\ \varphi_4 &= \neg q\end{aligned}$$

The subset $\{\varphi_1, \varphi_2\}$ is inconsistent and minimal, because no subset is inconsistent. So:

$$MI(\mathcal{K}) = \big\{\{\varphi_1, \varphi_2\}\big\}$$

Only formulas involved in an MIS are considered as conflicted. So φ_3 and φ_4 are not part of any MIS:

$$Free(\mathcal{K}) = \{\varphi_3, \varphi_4\}$$

Now we check whether a formula is consistent and has no atom in common with the rest of the knowledge base.

- $\varphi_3 = q \to r$ has the atoms q, r. φ_4 also contains q, so it's not safe.
- $\varphi_4 = \neg q$ shares q with φ_3, so it's not safe either.
- φ_1 and φ_2 share the atom p, so they are both not safe, too.

Therefore we can say:

$$Safe(\mathcal{K}) = \emptyset$$

We can use a simple inconsistency measure I that counts the number of MIS to illustrate our definition for this example:

$$I(\mathcal{K}) = 1$$

There are many different properties a inconsistency measure should satisfy und which can help evaluating and comparing to each other. The following postulates therefore are decisive [Thi18]:

- Consistency (CO): $I(\mathcal{K}) = 0$
  According to the definition of inconsistency measures, a consistent knowledge base schould have an inconsistency value of zero, while inconsistent knowledge bases should have a value greater than zero.
- Normalization (NO): $0 \leq I(\mathcal{K}) \leq 1$
  The inconsisteny value always should be in the unit interval, so that a comparison of different inconsistency values gets simplified.
- Monotony (MO): If $\mathcal{K} \subseteq \mathcal{K}'$ then $I(\mathcal{K}) \leq I(\mathcal{K}')$
  Adding formulas to a knowledge base should not decrease the the inconsistency value.
- Free-formula independence (IN): If $\varphi \in Free(\mathcal{K})$ then $I(\mathcal{K}) = I(\mathcal{K} \backslash \{\varphi\})$
  Removing formulas, which do not contribute to the inconsistency, should not affect the inconsistency value.

- Dominance (DO): If $\varphi \not\models \bot$ and $\varphi \models \psi$ then $I(\mathcal{K} \cup \{\varphi\}) \geq I(\mathcal{K} \cup \{\psi\})$
  Replacing a consistent formula through a weaker one should not increase the inconsistency value. If φ is consistent ($\bot$) and ψ follows logically of φ then the incosnsitency value from $\mathcal{K}$ unified with φ is higher or equal to the incosnsitency value of $\mathcal{K}$ unified with ψ.

Postulates in general describe a desirable behavior and are considered as policies. Thus they are not undisputed and can implie or be in conflict to each other. In addition to the mentioned postulates, Thimm added some more which originally were used in the context of probabilistic logic and got transferred to propositional logic. Table 2 gives an overview of all postulates and their computation.

Postulate	**Definition**
Consistency (CO)	$I(\mathcal{K}) = 0$
Normalization (NO)	$0 \leq I(\mathcal{K}) \leq 1$
Monotony (MO)	If $\mathcal{K} \subseteq \mathcal{K}'$ then $I(\mathcal{K}) \leq I(\mathcal{K}')$
Free-formula independence (IN)	If $\varphi \in Free(\mathcal{K})$ then $I(\mathcal{K}) = I(\mathcal{K} \backslash \{\varphi\})$
Dominance (DO)	If $\varphi \not\models \bot$ and $\varphi \models \psi$ then $I(\mathcal{K} \cup \{\varphi\}) \geq I(\mathcal{K} \cup \{\psi\})$
Safe-formula independence (SI)	If $\varphi \in Safe(\mathcal{K})$ then $\mathcal{I}(\mathcal{K}) = \mathcal{I}(\mathcal{K} \backslash \{\varphi\})$
Super-additivitiy (SA)	If $\mathcal{K} \cap \mathcal{K}' = \emptyset$ then $\mathcal{I}(\mathcal{K} \cup \mathcal{K}') \geq \mathcal{I}(\mathcal{K}) + \mathcal{I}(\mathcal{K}')$
Penalty (PY)	If $\varphi \notin Free(\mathcal{K})$ then $\mathcal{I}(\mathcal{K}) > \mathcal{I}(\mathcal{K} \backslash \{\varphi\})$
MI-seperability (MI)	If $MI(\mathcal{K} \cup \mathcal{K}') = MI(\mathcal{K}) \cup MI(\mathcal{K}')$ and $MI(\mathcal{K}) \cap MI(\mathcal{K}') = \emptyset$ then $\mathcal{I}(\mathcal{K} \cup \mathcal{K}') = \mathcal{I}(\mathcal{K}) + \mathcal{I}(\mathcal{K}')$
MI-normalization (MN)	If $M \in MI(\mathcal{K})$ then $\mathcal{I}(M) = 1$
Attentuation (AT)	$M, M' \in MI(\mathcal{K})$ and $\|M\| > \|M'\|$ implies $\mathcal{I}(M) < \mathcal{I}(M')$
Equal conflict (EC)	$M, M' \in MI(\mathcal{K})$ and $\|M\| = \|M'\|$ implies $\mathcal{I}(M) = \mathcal{I}(M')$
Almost consistency (AC)	Let M_1, M_2 be a sequence of minimal incosnsitent sets M_i with $\lim_{i\to\infty} \|M_i\| = \infty$, then $\lim_{i\to\infty} \mathcal{I}(M_i) = 0$
Contradiction (CD)	$\mathcal{I}(\mathcal{K}) = 1$ if and only if for all $\emptyset \neq \mathcal{K}' \subseteq \mathcal{K}$, $\mathcal{K}' \models \bot$
Free-formula Dilution (FD)	If $\varphi \in Free(\mathcal{K})$ then $\mathcal{I}(\mathcal{K}) \leq \mathcal{I}(\mathcal{K} \backslash \{\varphi\})$
Irrelevance of Syntax (SY)	If $K \equiv_b \mathcal{K}'$ then $\mathcal{I}(\mathcal{K}) = \mathcal{I}(\mathcal{K}')$
Exchange (EX)	If $\mathcal{K}' \not\models \bot$ and $\mathcal{K}' \equiv \mathcal{K}''$ then $\mathcal{I}(\mathcal{K} \cup \mathcal{K}') = \mathcal{I}(\mathcal{K} \cup \mathcal{K}'')$
Adjunction Invariance (AI)	$\mathcal{I}(\mathcal{K} \cup \{\varphi, \psi\}) = \mathcal{I}(\mathcal{K} \cup \{\varphi \wedge \psi\})$

Table 2: Overview of rationality postulates

2.4.2. Overview of Existing Inconsistency Measures

There is a multitude of inconsistency measures, which differ in the concepts they are based on and the way they capture the severity of contradictions. In general, these measures can be roughly categorized as follows.

- Based on Minimal Inconsistent Subsets (MISs): These measures typically count or analyze the minimal sets of formulas within a knowledge base that are contradictory. The more MISs, or the smaller they are, the more inconsistent the base might be considered [HK08].
- Based on Maximal Consistent Subsets (MCSs): These approaches focus on the largest subsets of formulas that are free of contradictions [GH11]. The inconsistency is often measured by how much information needs to be removed to achieve consistency, or by the number of such consistent subsets [Hun07].
- Based on Multi-valued Logics or Non-classical Semantics: These measures use logics that allow for more than two truth values, such as three-valued to quantify the degree of contradiction [Thi19].
- Based on Distances between Truth Assignments: These measures consider the distance between different models of the knowledge base. For instance, how many variable assignments need to change to satisfy different parts of the knowledge base, or the distance to the closest satisfying assignment of a consistent version [NKTJ23, TW19].
- Based on Proofs or Derivations: These measures value inconsistency based on the properties of logical proofs or derivations from the knowledge base, such as the length or number of proofs required to derive a contradiction [Thi18].
- Based on Variable Forgetting: These approaches quantify inconsistency by determining how many or which variables need to be forgotten to restore consistency to the knowledge base [TW19].
- Simple or Baseline Measures: These are often straightforward, sometimes binary, measures that provide a basic assessment of inconsistency, like simply checking if a contradiction is derivable [Thi18].

A well-established classification of inconsistency measures distinguishes between the syntactic and the semantic approach [HK08]. This distinction is based on the level of representation the measure operates on. Syntactic measures work on the formula level and typically analyze structural properties of the knowledge base. They often rely on the identification of Minimal Inconsistent Subsets (MIS) [Thi19].

Semantic measures, on the other hand, operate on the language level by considering the interpretations or models of the knowledge base. These approaches analyze the inconsistency by evaluating how difficult or impossible it is to construct models that satisfy the knowledge base [GH11, Thi19].

The contension inconsistency measure, for example, falls into the semantic category, as it evaluates the minimal number of atoms that must be assigned a non-classical truth value to achieve consistency and is based on Priest´s three-valued logic. Each atom in a knowledge base is thus assigned one of the three truth values. The goal is to find an assignment that minimizes the number of atoms with a truth value *both*. The contension value is the minimum number of atoms that must be assigned the value *both* to make the knowledge base consistent [GH11, Thi18].

The set of all three-valued interpretations to fulfill each formula in our knowledge base $\mathcal{K}$ is defined as $Models(\mathcal{K}) = \{\omega^3 \mid \forall \alpha \in K : \omega^3(\alpha) = t \vee \omega^3(\alpha) = b\}$. The set of all Atoms valued as b under an interpretation ω^3 is then defined as $Conflictbase(\omega^3)$.

Definition 20 (Conflict Base)**.** The conflict base of a three-valued interpretation ω^3 is defined as

$$\mathrm{Conflictbase}(\omega^3) = \{\alpha \in \mathrm{At}(\mathcal{K}) \mid \omega^3(\alpha) = b\}.$$

Example 6. Let $\mathcal{A} = \{p\}$ be the set of atoms, and consider the following knowledge base $\mathcal{K}$:

$$\mathcal{K} = \{\varphi_1, \varphi_2\}$$

with

$$\varphi_1 = p$$
$$\varphi_2 = \neg p$$

We can see that this knowledge base is inconsistent. We now consider a three-valued interpretation $\omega^3 \colon \mathcal{A} \to \{t, f, b\}$ defined by:

$$\omega^3(p) = b$$

Then, we evaluate the formulas under ω^3 using Priest's three-valued logic:

- $\omega^3(\varphi_1) = \omega^3(p) = b$
- $\omega^3(\varphi_2) = \omega^3(\neg p) = \neg b = b$

Since both formulas evaluate to either t or b, it follows that ω^3 satisfies all formulas in $\mathcal{K}$

$$\omega^3 \in \mathrm{Models}(\mathcal{K})$$

We can now compute the conflict base:

$$\mathrm{Conflictbase}(\omega^3) = \{a\}$$

Definition 21 (Contension Inconsistency Measure)**.** The contension inconsistency measure is defined as

$$I_c(\mathcal{K}) = \min\{|\mathrm{Conflictbase}(\omega^3)| \mid \omega^3 \in \mathrm{Models}(\mathcal{K})\},$$

where $\mathrm{Models}(\mathcal{K})$ is the set of all three-valued interpretations satisfying $\mathcal{K}$.

Example 7. Given the conflict base:

$$\mathrm{Conflictbase}(\omega^3) = \{a\}$$

As there is no interpretation in $\mathrm{Models}(\mathcal{K})$ with fewer than one atom assigned b, the contension inconsistency measure is:

$$I_c(\mathcal{K}) = 1$$

2.4.3. The Formula-Based Contension Inconsistency Measure

The key difference between the contension inconsistency measure I_c and its formula-based variant I_{fc} lies in the unit of measurement. I_c evaluates how many atoms must be assigned the truth value *both* in order to satisfy all formulas in the knowledge base [KT20]. Thus, it focuses on atomic conflict participation. In contrast, I_{fc} evaluates how many formulas in the knowledge base must contain at least one atom assigned *both* in order for the whole knowledge base to be satisfiable. Hence, it focuses on formulas affected by conflict.

While both measures are based on Priest's three-valued logic and rely on the existence of *both*-assignments to resolve contradictions, I_{fc} shifts the perspective. Instead of counting conflicted atoms, it counts formulas whose satisfaction depends on conflicted atoms. I_c requires modeling the minimal number of atoms assigned *both*, independently of how often each appears in formulas. I_{fc} on the other hand, is formulawise satisfiability-oriented. Each formula is checked individually whether it needs a *both*-valued atom to be satisfied.

This leads to syntactic sensitivity in I_{fc} as logically equivalent transformations can change the number of formulas and how they are affected by *both*-assignments. In contrast, I_c is not affected under such a transformation.

The definition of the formula-based inconsistency measure is therefore the minimum number of formulas in a knowledge base that contain at least one atom with the truth value *both*. So, the set of all formulas φ in $\mathcal{K}$ with a minimum number of atoms valued with the truth value b under an interpretation ω^3 is defined as $ConflictFormulas(\omega_3)$.

Definition 22 (Conflict Formulas)**.** The set of formulas containing at least one conflicting atom is

$$\mathrm{ConflictFormulas}(\omega_3) = \{\varphi \in \mathcal{K} \mid \exists \alpha \in \mathrm{At}(\varphi) : \omega_3(\alpha) = b\}.$$

Example 8. Let $\mathcal{A} = \{p\}$ be again the set of atoms, and consider the known knowledge base $\mathcal{K}$:

$$\mathcal{K} = \{\varphi_1, \varphi_2\}$$

with

$$\varphi_1 = p$$
$$\varphi_2 = \neg p$$

We define the three-valued interpretation ω^3 by:

$$\omega^3(p) = b$$

We can now compute the conflicted formulas:

$$\text{ConflictFormulas}(\omega^3) = \{\varphi_1, \varphi_2\}$$

Definition 23 (Formula-Based Contension Inconsistency Measure)**.** The formula-based contension inconsistency measure is defined as

$$I_{fc}(\mathcal{K}) = \min\{|\text{ConflictFormulas}(\omega^3)| \mid \omega^3 \in \text{Models}(\mathcal{K})\}.$$

where $\text{Models}(\mathcal{K})$ is the set of all three-valued interpretations satisfying K.

Example 9. Given the conflicted formulas:

$$\text{ConflictFormulas}(\omega^3) = \{\varphi_1, \varphi_2\}$$

As there is no interpretation in $\text{Models}(\mathcal{K})$ with fewer formulas containing any atom valued b, the formula-based contension inconsistency measure is:

$$I_{fc}(\mathcal{K}) = 2$$

Matthias Thimm presented a comparison of many different inconsistency measures in [Thi18]. He used the 18 rationality postulates to create a basis for evaluating these measures. Each measure is then considered to satisfy or violate a postulate. In this section, this evaluation framework is partial expanded to include the formula-based contension inconsistency measure.

To enable a direct comparison of the contension inconssitency measure I_c and its formula-based variation I_{fc} their compliance with some of these postulates is listed in Table 3. Since I_{fc} evaluates inconsistency on the formula level, the measure is sensitive to the syntactic structure of formulas. Therefore, the evaluation of the postulates is restricted to a selected subset of five postulates that are formula-sensitive, so their satisfaction may depend on the syntactic representation of formulas in the knowledge base.

Postulate	I_c	I_{fc}
Free-formula independence (IN)	✓	✗
Dominance (DO)	✓	✗
Safe-formula independence (SI)	✓	✓
Exchange (EX)	✓	✗
Adjunction Invariance (AI)	✓	✗

Table 3: Compliance of inconsistency measures with rationality postulates

The postulate *IN* says that a free Formula φ can be removed from a knowledge base K without changing the inconsistency value. But even if φ is free and therefore not affected in any conflicts, the formula can contain atoms, which are valued *both* due to conflicts occurring in any other formula. This scenario would then decrease our inconsistency value.

Proof. Let the knowledge base $\mathcal{K} = \{\varphi_1, \varphi_2, \varphi_3\}$ with

$$\varphi_1 = A, \quad \varphi_2 = \neg A, \quad \varphi_3 = A \vee B.$$

The subset $\{\varphi_1, \varphi_2\}$ is inconsistent, and no proper subset of it is inconsistent, so it is a MIS. The formula φ_3 is not part of any MIS, so it is a free formula.

According to the postulate IN, removing a free formula does not change the inconsistency value:

$$I_{fc}(\mathcal{K}) = I_{fc}(\mathcal{K} \setminus \{\varphi_3\}).$$

Since $\varphi_3 = A \vee B$ contains the atom A, which is assigned the value *both* due to the conflict between φ_1 and φ_2, the formula increases the I_{fc} value. The postulate IN is therefore violated. □

DO means that substituting a formula φ with a weaker logical formula ψ should not increase the inconsistency value. Even if there is no possibility to add new conflicts to our knowledge base, there is a potential way that ψ contains atoms which were not included in φ but are affected in conflicts in other formulas. This scenario then would increase our inconsistency value.

Proof. Let $\varphi_1 = A$ and $\varphi_2 = A \vee C$, where φ_2 is logically weaker than φ_1 since $\varphi_1 \models \varphi_2$.

Let $\mathcal{K}_1 = \{\varphi_1, C, \neg C\}$ and $\mathcal{K}_2 = \{\varphi_2, C, \neg C\}$.

Both knowledge bases are inconsistent due to the conflict of the atom C but neither φ_1 nor φ_2 are affected in this conflict.

The $I_{fc}(\mathcal{K}_1) = 2$ because φ_1 does not contain any atom C. But for φ_2 there is an atom C, which leads to $I_{fc}(\mathcal{K}_2) = 3$. Therefore $I_{fc}(\mathcal{K}_1) < I_{fc}(\mathcal{K}_2)$ violates the postulate DO. □

The postulate *SI* says that if φ is a safe formula, removing it should not change the inconsistency value. Similarly to *IN*, this formula could still contain atoms which are assigned the truth value *both*, therefore removing it would decrease our value of I_{fc}.

Proof. Let $\mathcal{K}$ be a knowledge base and $\varphi \in Safe(\mathcal{K})$.

By definition of safe formulas, φ is consistent and satisfies

$$At(\varphi) \cap At(\mathcal{K} \setminus \{\varphi\}) = \emptyset.$$

Since φ is consistent, it does not contribute to any minimal inconsistent subsets of $\mathcal{K}$.

Because the atoms of φ are disjoint from the atoms of the rest of $\mathcal{K}$, there is no way that φ shares any conflicting atoms with other formulas.

Therefore, removing φ from $\mathcal{K}$ does not reduce the number of conflicting atoms or minimal inconsistent subsets, so

$$I_{fc}(\mathcal{K}) = I_{fc}(\mathcal{K} \setminus \{\varphi\}).$$

So the postulate SI is satisfied. □

The postulate *EX* says that adding a knowledge base K_1 to K should result in the same inconsistency value as adding K_2 to K, if K_1 and K_2 are logically equivalent. But despite this equivalence K_1 and K_2 can differ in its containing atoms just like in the number of formulas.

AI specifies the equality of the inconsistency value by adding two separate formulas and the conjunction of these formulas. It is clear to identify a significant violation of I_{fc} due to the different number of formulas.

Proof. Consider the knowledge base

$$\mathcal{K} = \{\varphi_1, \varphi_2\}$$

with

$$\varphi_1 = \neg A, \quad \varphi_2 = \neg B,$$

Now, let the two consistent extensions be:

$$\mathcal{K}_1 = \{B, A\}, \quad \mathcal{K}_2 = \{B \wedge A\},$$

where both $\mathcal{K}_1$ and $\mathcal{K}_2$ are consistent and logically equivalent.

Then,

$$I_{fc}(\mathcal{K} \cup \mathcal{K}_1) = 4,$$

and

$$I_{fc}(\mathcal{K} \cup \mathcal{K}_2) = 3,$$

Therefore,

$$I_{fc}(\mathcal{K} \cup \mathcal{K}_1) \neq I_{fc}(\mathcal{K} \cup \mathcal{K}_2),$$

which violates the EX and AI postulates. □

In conclusion I_{fc} satisfies fewer postulates than I_c which suggests that I_{fc} is less robust or behaves less predictably according to these rationality criteria.

The formulas of a knowledge base could be reformulated into logically equivalent formulas to modify the inconsistency value. The formula $\{A, \neg A\}$ could be transferred to $\{A \wedge \neg A\}$ which would decrease I_{fc} from the value 2 to 1. This may lead to a higher sensitivity of the used syntax.

This characteristic implies that the formula-based contension inconsistency measure should be less used to evaluate the logical contradictions than measuring the extent to the formulas of it.

In addition to these postulates, Thimm also compared the inconsistency measures using the term Expressivity. This is all about the information the used inconsistency measure can give us about a given knowledge base and how it can identify differences between deviating knowledge bases. In case of the I_{fc} the primary information we get might be the quantified affected formulas. Larger but fewer formulas may lead to weaker expressivity regarding the granularity or perceived magnitude of inconsistency, because each formula is counted as a single unit.

Computational Complexity as the last part of the evaluation criteria refers to the scalability of computational effort with the size of the input and the classification into complexity classes. As I_{fc} brings an optimization problem in finding the minimal number of formulas containing *both* valued atoms, which is part of SAT, the problem of computing it is NP-hard. The computation can therefore lead to impractical computation times or require computation resources, especially for large knowledge bases.

3. Algorithmic Approaches

After the definition and explanation of the theoretical foundations, there will be a description of the Framework and the algorithmic approaches. Considered are the resulting encodings, illustrated by an example for each approach, and the Correctness and Limitations for the modeled I_{fc}.

The project has a modular structure, clearly separating data processing, algorithmic encoding, and the execution of experiments. It consists of three main components: the main script, the parser and data model, and the two solver encoders. The whole code can be found as a repository on GitHub [1].

3.1. Implementation Framework

This section explains the technical structure of the implementation and the experimental procedure used to evaluate the algorithmic approaches. It provides an overview of the system architecture, data modeling, and solver orchestration. The Code is an adaption of the works from Isabelle Kuhlmann provided in GitHub [2].

3.1.1. Project Structure and Data Modeling

The project architecture is designed in a modular fashion to ensure a clear separation of concerns. The entire process from the knowledge base to the final results follows a logical pipeline, which is visualized in Appendix E.

The central script, *main.py*, acts as the main controller. It iterates through knowledge bases in the file system, initiates solving runs for each approach, and man-

[1] Repository link for this project: https://github.com/Marcel104/Abschlussarbeit

[2] Repository link of the project "algorithms_for_inconsistency_measurement_in_PL" from Isabelle Kuhlmann: https://github.com/aig-hagen/algorithms_for_inconsistency_measurement_in_PL/tree/main

ages logging. The scripts *parser.py*, *formula.py*, and *knowledge base.py* are responsible for reading and transforming the textual formulas into a structured, object-oriented representation. For each algorithmic approach, a dedicated encoder exists. Each encoder receives the object-oriented data model and translates it into the specific input format required by the corresponding solver. The resulting programs or formulas are passed to the respective solver libraries. Solver outputs are interpreted and standardized for consistent logging.

The foundation for all algorithmic processing is the correct and flexible representation of logical formulas. The module *parser.py* implements a parser that transforms textual formulas into syntax trees. This process involves tokenization and tree construction. A regular expression is used to split the input string into tokens, recognizing atoms, operators, parentheses, and constants. The parser implements the Shunting Yard algorithm to convert the infix token sequence into postfix notation while respecting operator precedence and associativity. A syntax tree is then recursively built from this postfix sequence.

The syntax tree is represented by instances of the *Formula* class (from *formula.py*). Each node in the tree is such an object, which may recursively contain other *Formula* instances as children. This class is the core of the data model and provides crucial methods such as `get_atoms()` for extracting all unique atoms from a formula, and `to_cnf()`, which offers two options for CNF conversion.

The *Kb* class serves as a container that holds a list of *Formula* instances, representing a knowledge base. A class diagram illustrating the data model is provided in Appendix F.

Result extraction is implemented in a solver-specific manner. The function `asp_encode_and_solve` uses the `on_model` callback function from the Clingo library. For each optimal model found, this callback extracts the cost and the atoms of the `val/2` and `f_inconsistent/1` predicates to reconstruct the solution details. The MaxSAT functions use the RC2 solver from the PySAT library. After computing the result using `rc2.compute()`, the cost is directly read from the solver object. To retrieve solution details, the returned model is analyzed and integer variables are translated back into their boolean interpretations using a `reverse_varmap`. The violated soft clauses reveal the inconsistent formulas.

3.1.2. Formula Representation with an Abstract Syntax Tree

To process logical formulas in a structured and rule-based way, each formula in the knowledge base can be represented as an abstract syntax tree (AST). This tree structure enables recursive analysis and transformation, which is neccessary for the ASP and MaxSAT encodings. Each node in an AST represents either a logical connective or a propositional atom [BHvMW09, BM07]. With the help of its hierarchical structure, the tree can be run through.

Consider the knowledge base

$$\mathcal{K} = \{A \wedge B, \neg A\}$$

which consists of the two formulas $\varphi_1 = A \wedge B$ and $\varphi_2 = \neg A$. The corresponding syntax trees are:

Figure 2: Syntax trees of the formulas in the knowledge base $\mathcal{K}$

The AST is implemented via the Formula class defined in formula.py. Each node is an instance of this class, with a type field denoting the logical connective and optional left, right, or atom fields describing its substructure. For example, the formula $A \wedge B$ is represented as

```
Formula(
    type=FormulaType.AND,
    left=Formula(type=FormulaType.ATOM, atom="A"),
    right=Formula(type=FormulaType.ATOM, atom="B")
)
```

while the formula $\neg A$ can be modeled as

```
Formula(
    type=FormulaType.NOT,
    left=Formula(type=FormulaType.ATOM, atom="A")
)
```

The class provides further recursive transformation methods, for example to normalize the formula and convert it into CNF. The ASP-based encoding directly uses the AST to generate structural facts [KT20]. The benefit of such an AST representation is that it preserves the original syntactic structure of the formulas [BM07]. This is very important for the syntactically sensitive I_{fc} measure.

3.2. Approach Based on MaxSAT Solving

This section presents the first of the two algorithmic approaches developed in this thesis, which is based on Maximum Satisfiability (MaxSAT). The core of this approach is an encoding that translates the problem of computing the I_{fc} measure from the three-valued logic into a Weighted Conjunctive Normal Form (WCNF) formula. By leveraging the optimization capabilities of a MaxSAT solver, we can find a solution whose cost directly corresponds to the desired inconsistency value. The following subsections will detail the formal model of this encoding, its practical implementation, a concrete example, and a proof of its correctness.

3.2.1. Modelling the I_{fc} for MaxSAT

Given a knowledge base

$$K = \{\varphi_1, \ldots, \varphi_n\}$$

we first apply a *naive* transformation of each formula φ_i into conjunctive normal form (CNF):

$$\mathrm{CNF}(\varphi_i) = \{C_{i1}, \ldots, C_{im}\}$$

where each clause C_{ij} is a disjunction of literals. In this work, we apply a naive CNF transformation that directly expands the structure of each formula. Although this approach is less efficient than Tseitin's method, it preserves logical equivalence and maintains the original association between formulas and their constituent literals.

For each atom $\alpha \in \mathrm{At}(K)$, we introduce three new Boolean variables to represent its truth value in a three-valued logic:

- α^t: atom α is assigned *true*
- α^f: atom α is assigned *false*
- α^b: atom α is assigned *both*

Additionally, we introduce a Boolean variable F_i for each formula φ_i, which indicates whether the formula is inconsistent under the current interpretation.

The following hard clauses must be satisfied in any valid solution:

- $\alpha^t \vee \alpha^f \vee \alpha^b$: ensures that at least one truth value is assigned to each atom.
- $\neg\alpha^t \vee \neg\alpha^f$, $\neg\alpha^t \vee \neg\alpha^b$, $\neg\alpha^f \vee \neg\alpha^b$: ensures that no more than one truth value is assigned.

To link formula inconsistency with atom values, we add for each $\alpha \in \mathrm{At}(\varphi_i)$ the hard clause:

$$F_i \vee \neg\alpha^b$$

This means that if an atom in φ_i is assigned the value *both*, the formula must be considered inconsistent.

Each clause C_{ij} from the CNF of φ_i is extended to:

$$C_{ij} \vee \alpha_1^b \vee \alpha_2^b \vee \cdots \vee \alpha_k^b$$

where $\alpha_1, \ldots, \alpha_k$ are the atoms occurring in C_{ij}. This construction allows a clause to be satisfied even when it is otherwise violated, provided that at least one involved atom is assigned the value *both*.

Finally, for each formula φ_i, we add a *soft clause*:

$$\neg F_i$$

This clause may be violated by the solver, but doing so leads to a cost. The solver thus aims to minimize the number of formulas considered inconsistent by minimizing the number of violated soft clauses.

The number of formulas φ_i for which $F_i = \text{true}$ therefore corresponds to the formula-based contension inconsistency measure.

The cost of an optimal solution to the generated WCNF problem corresponds exactly to the value of $I_{fc}(\mathcal{K})$. This can be justified as follows: A satisfying assignment ω for the hard clauses of the generated formula in WCNF corresponds to a valid three-valued interpretation ω^3 for the knowledge base $\mathcal{K}$. The added boolean variables α^t, α^f and α^b for each Atom $\alpha \in At(\mathcal{K})$ and the associated generated constraints guarantee that α is assigned exactly one of the values in ω^3.

Furthermore, The hard clauses ensure, that every formula $\varphi \in \mathcal{K}$ is satisfied in Priest´s logic, therefore no formula has the value *false*. A formula only could be violated if it is classically unsatisfied and none of its atoms are assigned the value *both*. Therefore every satisfying assignment of the hard clauses in F equals $\omega^3 \in Models(\mathcal{K})$.

The optimization goal of the MaxSAT solver is to minimize the sum of the weights of the violated soft clauses. In the encoding there is exactly one soft clause fpr each $\varphi_i \in \mathcal{K}$ with a weight of 1. A soft clause is violated if F_i is assigned the value *true*. The set of hard clauses $F_i \vee \neg\alpha^b$ for each atom $\alpha \in At(\varphi)$ ensures that F_i is forced to be *true* if at least one of its constituent atoms is assigned the value *both*.

Consequently, minimizing the cost of violated soft clauses is equivalent to minimizing the number of the variables F_i that are assigned *true*. This, in turn, is equivalent to minimizing the number of formulas affected by a *both* assignment. The number of formulas φ_i for which $F_i = true$ corresponds to the formula-based contension inconsistency measure.

3.2.2. Implementation of the MaxSAT Approach

The formal model is implemented in Python within the MaxSat Encoder class in *solver_Max_SAT.py*. This class is responsible for converting a Kb object into a WCNF object, which is the standard input format for the pySAT library, using the *encode* method.

In a first step, the class is initialized. A central dictionary, *atom_vars*, maps tuples of *atom_name: truth_value* to unique integer variables required by the SAT solver. A counter *var_counter* ensures that each new variable is unique.

According to this, the class provides additional methods like *new_var* (returns the increased *var_counter*), *get_var* (returns the truth value of the original atom) and some methods to extract clauses from a CNF formula or literals and atoms from a clause.

The encoder now iterates through all unique atoms present in the knowledge base. For each atom, it retrieves the three corresponding integer variables for *true*, *false*, and *both* and adds hard clauses to guarantee that there is exactly one of those variables satisfied to the WCNF object.

After that, the encoder iterates through each formula of the knowledge base. First, the *to_cnf* method of the *formula* instance is called to get a list of clauses representing the formula. For each clause, a new list of literals is created where each original literal is mapped to its corresponding three-valued variable. For example α becomes α^t and $\neg\alpha$ becomes α^f. This new extended clause is added as a hard clause. Additionally a new helper variable $finc$ is created for the formula. To link $finc$ to the α^b variables of the atoms in the original formulas, again hard clauses are added. Finally, the soft clause $\neg finc$ with a weight of 1 is added.

The encoder method returns the WCNF object which is then passed to the solver in *main.py*. In this case the solver RC2 from the library pysat is instantiated. We can start solving by using the *compute* method and can use the returned model and cost value for further analysis.

3.2.3. Example of the MaxSAT Encodings

To walk through the modeling and implementation with an example, the knowledge base $K = \{A \wedge B, \neg A\}$ is given. The knowledge base thus contains the two formulas $\texttt{f0} = A \wedge B$ and $\texttt{f1} = \neg A$. The set of atoms in $\mathcal{K}$ is $At(\mathcal{K}) = \{A, B\}$.

All variables and clauses are expressed in DIMACS format, which is the standard input format for most SAT and MaxSAT solvers. In this format, each clause is a list of integers that represent variables. A positive number denotes a variable set to *true*, while a negative number denotes the variable set to *false* [BHvMW09]. DIMACS does not represent propositional logic directly, it encodes it through integers for efficient processing by solvers.

The mapping of the three-valued variables of each atom and the variables for each formula to an integer will look like shown in table 4.

Atom	**Mapping**
A_{true}	1
A_{false}	2
A_{both}	3
B_{true}	4
B_{false}	5
B_{both}	6
$f0_{inconssitent}$	7
$f1_{inconssitent}$	8

Table 4: Mapping of the three-valued variables to Integers

To ensure that every atom is assigned exactly one of the truth-values, the following hard clauses are added:

$$[A_{true}, A_{false}, A_{both}]$$
$$[\neg A_{true}, \neg A_{false}]$$
$$[\neg A_{true}, \neg A_{both}]$$
$$[\neg A_{false}, \neg A_{both}]$$

$$[B_{true}, B_{false}, B_{both}]$$
$$[\neg B_{true}, \neg B_{false}]$$
$$[\neg B_{true}, \neg B_{both}]$$
$$[\neg B_{false}, \neg B_{both}]$$

Because formula `f0` consists of the Atoms A and B, one of these atoms valued *both* would lead into inconsistency of this formula, whereas formula `f0` only contains of Atom B.

$$[f0_{inconsistent}, \neg A_{both}]$$
$$[f0_{inconsistent}, \neg B_{both}]$$
$$[f1_{inconsistent}, \neg A_{both}]$$

To model the logical structure of our knowledge base, the WCNF of the formulas is used. Therefore $\mathcal{K}$ is transformed into $W_{\mathcal{K}} = \{A, B, \neg A\}$. For each clause a new hard clause is created, containing the variable of its occuring atoms (A_{true}, B_{true}, A_{false}) and the variable representing the value *both* of each atom.

$$[A_{true}, A_{both}]$$
$$[B_{true}, B_{true}]$$
$$[A_{false}, A_{both}]$$

It is easy to see, that the first and the second of these three hard clauses leads to an inconsistency. Since the variables A_{true} and A_{false} cant be *true* simultaneously, A_{both} has to be set to *true* instead. Finally the soft clauses are added, where the mapped integer variables of the both original formulas are negated to ensure that an occuring inconsistency injures these clauses.

$$[\neg f0_{inconsistent}]$$
$$[\neg f1_{inconsistent}]$$

In the end, the solver will identifiy both formulas as inconsistent while valueing the atom A *both*. The cost of injuring both soft clauses leads to an I_{fc} of 2.

3.2.4. Correctness of the MaxSAT Encoding for Computing I_{fc}

In this section, the correctness of our MaxSAT encoding for computing the $I_{fc}(K)$ is proven. We denote the cost of an optimal solution to the WCNF formula W_K generated from a knowledge base K by $\mathrm{cost}(W_K)$. The goal is to show that

$$\mathrm{cost}(W_K) = I_{fc}(K).$$

This equality is established by proving two bounds: soundness, which shows $\mathrm{cost}(W_K) \geq I_{fc}(K)$, and completeness, which shows $\mathrm{cost}(W_K) \leq I_{fc}(K)$.

First, we show that the cost of an optimal solution is at least $I_{fc}(K)$.

An optimal model M for the WCNF formula W_K must satisfy all hard clauses. This model defines a valid three-valued interpretation for K where all formulas are satisfied in Priest's logic. The cost of this model, $\mathrm{cost}(W_K)$, is determined by the number of violated soft clauses. Since there is exactly one soft clause $\neg F_i$ for each formula φ_i, the cost is the number of F_i variables assigned *true*.

Our encoding, specifically through the hard clauses $(F_i \vee \neg\alpha^b)$, ensures that F_i is forced to be *true* if any atom α within the formula φ_i is assigned the value *both* (i.e., if $M(\alpha^b) =$ true). An optimal solver will not set any F_i to *true* unless required. Therefore, the cost of the optimal solution M corresponds exactly to the number of formulas containing at least one atom assigned the value *both* in that specific solution.

By definition, $I_{fc}(K)$ is the minimal possible number of such affected formulas over all possible satisfying three-valued interpretations. The solution found by the MaxSAT solver represents just one such interpretation. Therefore, its cost cannot be lower than the theoretical minimum. Hence:

$$\mathrm{cost}(W_K) \geq I_{fc}(K).$$

Next, we show that the cost of an optimal solution is no more than $I_{fc}(K)$.

Let $B_{\min} \subseteq \mathrm{At}(K)$ be a set of atoms that, when assigned the value *both*, satisfies the knowledge base K and produces the minimal number of affected formulas, which is exactly $I_{fc}(K)$. We can construct a valid assignment $M_{\min}$ for our WCNF formula based on this optimal set $B_{\min}$.

- For every atom $\alpha \in B_{\min}$, we set its corresponding variable α^b to *true*. For all other variables, we assign *true* or *false* according to a corresponding satisfying three-valued interpretation. This ensures that the hard clauses for exactly one variable to be *true* are satisfied.

- For every formula variable F_i, we set it to *true* if its corresponding formula φ_i contains an atom from $B_{\min}$ (i.e., $\mathrm{At}(\varphi_i) \cap B_{\min} \neq \emptyset$), and to *false* otherwise.

This constructed assignment $M_{\min}$ satisfies all hard clauses of the encoding. The cost incurred by this assignment is the number of F_i variables set to *true*. By our construction, this number is $|\{\varphi \in K \mid \mathrm{At}(\varphi) \cap B_{\min} \neq \emptyset\}|$, which equals $I_{fc}(K)$.

Since we have successfully constructed a valid assignment with a total cost of exactly $I_{fc}(K)$, the cost of an optimal solution found by the solver can not be higher. Thus:

$$\mathrm{cost}(W_K) \leq I_{fc}(K).$$

Combining the two inequalities:

$$\mathrm{cost}(W_K) \geq I_{fc}(K) \quad \text{and} \quad \mathrm{cost}(W_K) \leq I_{fc}(K),$$

we conclude that

$$\mathrm{cost}(W_K) = I_{fc}(K),$$

which establishes the correctness of our MaxSAT encoding.

3.2.5. Limitations of the Tseitin Transformation for I_{fc} and Possible Adaptations

The Tseitin transformation is a well-established method for converting propositional formulas into equisatisfiable CNF. It introduces helping variables to represent subformulas, and avoids an exponential blow-up in the formula size. While this makes it highly efficient for SAT and MaxSAT solving, it leads to specific challenges in the context of the I_{fc}.

The key requirement for computing I_{fc} is the ability to identify inconsistencies that come up within specific formulas because of conflicting truth assignments to their atomic literals. But Tseitin's transformation does not preserve logical equivalence, it only ensures equisatisfiability [KKS+22]. This means that:

- The transformed CNF contains helping variables that do not correspond to original atoms.
- The structure of the original formula is no longer available.
- It becomes hard to trace whether a *both* assignment to a literal in a formula still affects that specific formula after transformation. However, the violation of the *DO* postulate makes it necessary to ensure this traceability, since even logically weaker formulas can introduce new conflicts when their atoms overlap with those in other formulas.

As a result, the transformed formula set cannot directly support the definition of I_{fc}.

To enable the use of Tseitin-transformed formulas in the computation of I_{fc} the encoding must be extended with additional constructs. The core idea is to bridge the gap between helping CNF clauses and the original formulas. This can be achieved by:

- Recording, for each original formula φ_i the set of its original literals
- Ensuring that for every literal A in a formula φ_i the assignment of *both* to A marks the formula as incosnsitent.
- Adding rules that explicitly link the truth assignments of original atoms to the inconsistency variable associated with their parent formulas.
- Avoiding the use of helping variables in the definition of formula inconsistency. Only original atoms should contribute to whether a formula is considered inconsistent.

In fact, this procedure relies the encodings in the use of the naive CNF transformation. The point is that helping variables always have to be valued *true* and therefore do not need to be translated into the three-valued logic. This is justified by the semantics of the Tseitin transformation: each helping variable introduced represents the truth value of a subformula and is constrained by implications that ensure its value is determined by the truth values of its constituents. In any satisfying assignment, these constraints enforce the helping variables to be *true* if and only if the subformulas they represent are satisfied. Consequently, helping variables cannot themselves become *both*, and they cannot cause a formula to be marked as inconsistent. This ensures that the three-valued interpretation only needs to be applied to the original atoms.

As an example, the knowledge base $\mathcal{K} = \{A \wedge B, \neg A\}$ is considered again. The atoms are mapped to three-valued variables in DIMACS format, as shown in Table 4. Likewise, hard clauses are generated to represent the three-valued semantics of the atoms and to capture the inconsistency of formulas based on the *both* assignment for each atom. The soft clauses are added as usual.

The CNF transformation using Tseitin, together with the extension involving the *both* variables for each original atom, results in the following clauses:

$$
\begin{aligned}
[\neg h_1, A_{true}, A_{both}] \\
[\neg h_1, B_{true}, B_{both}] \\
[h_1, A_{false}, B_{false}, A_{both}, B_{both}] \\
[h_1, A_{true}, A_{both}] \\
[h_1] \\
[h_2] \\
[f0_{inconsistent}] \\
[f1_{inconsistent}]
\end{aligned}
$$

Here, variables h_1 and h_2 are helper variables h introduced by the Tseitin transformation.

3.3. Approach Based on Answer Set Programming

ASP is well-suited for modeling the formula-based Contension measure. It naturally supports recursive rules, which makes it ideal for evaluating logical formulas based on their structure. The use of the *#minimize* statement enables the modeling of the task as an optimization problem that targets the number of inconsistent formulas. Moreover, the declarative nature of ASP allows us to describe the structure and the conflicts of our knowledge base, whereby the solver will deliver us the corresponding answer sets [SPBS03]. Although ASP is based on classical two-valued logic, its flexible predicate representation makes it possible to simulate the three-valued semantics required by the formula-based contension inconssitency measure.

3.3.1. Motivation for ASP Based on MaxSAT Limitations

ASP can be understood as a powerful extension of the classical SAT problem. Both are tools describing a problem using logical rules, and a solver finds a solution that follows those rules. But ASP adds extra features from logic programming that make it more flexible. While SAT focuses on finding truth assignments that satisfy all clauses, ASP is based on stable model semantics, allowing for default negation and recursive definitions.

The basic ideas of MaxSAT and ASP are very similar. The equivalents to hard clauses in MaxSAT are integrity constraints in ASP. For optimization, MaxSAT uses soft clauses, while ASP does the same thing with its `#minimize` statement.

However, ASP offers several key advantages that could make it a better choice for this project. First, ASP moves from a propositional to a predicate-based representation. In the MaxSAT approach, each atom requires three distinct propositional variables to represent its three-valued state. Additional hard clauses are then needed to enforce that exactly one of these variables is assigned *true*. By contrast, the ASP approach abstracts this concept by using a single predicate `val(Atom, Value)`, to express the relationship between an atom and its truth value. The condition, which guarantess that excatly one truth value is assigned to each atom, is then captured by a single choice rule. This transforms the encoding from a variable-based model to a more abstract and scalable one [BET11, ABL13].

Second, a significant advantage of ASP is its ability to handle recursive structures naturally. The MaxSAT approach requires transforming each formula into CNF, a process that flattens the formula's native structure and can lead to an exponential increase in size with a naive transformation. In contrast, ASP can represent the syntactic tree of a formula directly using recursive rules. A conjunction can be defined in terms of its conjuncts, which can themselves be complex formulas. This avoids the loss of structural information and allows the evaluation logic of Priest's semantics to be modeled in a way that directly mirrors its recursive definition[SPBS03].

Finally, the `not` operator allows to create rules that apply when something else cannot be proven. In this case, this is extremely useful for the truth value *both*. A formula is evaluated to *both* if it can be proven neither *true* nor *false*. This *otherwise* condition is difficult to model in classical logic but is a natural and declarative feature of ASP [BET11, EIST09].

3.3.2. Modelling the I_{fc} in ASP

To process our knowledge base, we decompose each formula into its logical components. This is achieved using recursively definable ASP terms that represent each part of a formula as a separate predicate. Each formula and subformula is assigned a unique identifier, allowing us to reference and evaluate them individually. To specify the meaning of these predicates, connectivity rules are added. Our solver thus gets an instruction on how to process this part of code. In this specific case we add connectivity rules for the cases of conjunction, disjunction, negation and the case that our formula is treated like an atom. Implications and equivalences are simplified in it´s previous basic operations.

Given a knowledge base

$$\mathcal{K} = \{\varphi_1, \ldots, \varphi_n\}$$

with the set of all atoms occurring in $\mathcal{K}$

$$\mathrm{At}(\mathcal{K}) = \{\alpha_1, \ldots, \alpha_m\}.$$

First, the knowledge base is translated into a set of foundational ASP facts. For each atom $\alpha \in \mathrm{At}(\mathcal{K})$, a fact atom$(\alpha)$ is generated. For each formula $\varphi \in \mathcal{K}$, a fact kb$(rep(\varphi))$ is added to mark it as a member of the knowledge base.

In addition, the structure of each formula $\varphi \in \mathcal{K}$ is translated into further facts. We use $rep(\psi)$ to denote the unique identifier for any subformula ψ. The Python component is responsible for this syntactic decomposition by generating facts that represent the formula structure.

- If φ is an atom α:

$$\text{formula\_is\_atom}(rep(\alpha), \alpha)$$

- If $\varphi = \neg\psi$:

$$\text{negation}(rep(\psi), rep(\neg\psi))$$

- If $\varphi = \psi_1 \wedge \cdots \wedge \psi_k$:

$$\begin{aligned} \text{conjunction}(rep(\varphi)) \\ \text{num\_conjuncts}(rep(\varphi), k) \\ \text{conjunct\_of}(rep(\psi_j), rep(\varphi)) \ \text{ for each } j \in \{1, \ldots, k\} \end{aligned}$$

- If $\varphi = \psi_1 \vee \cdots \vee \psi_k$:

$$\begin{aligned} &\text{disjunction}(rep(\varphi)) \\ &\text{num\_disjuncts}(rep(\varphi), k) \\ &\text{disjunct\_of}(rep(\psi_j), rep(\varphi)) \ \text{ for each } j \in \{1, \ldots, k\} \end{aligned}$$

The union of these generated facts for all formulas in the knowledge base, forms the structural foundation of the ASP program. In a previous step, implications and equivalences are translated into their basic operations.

The predicate $\text{val}(X, Y)$ denotes that the atom or formula X is assigned the truth value Y.

For each atom $\alpha \in \text{At}(\mathcal{K})$, exactly one truth value must be assigned. This explores the search space of all possible three-valued interpretations. This is encoded as the choice rule:

$$1\{\text{val}(\alpha, t); \text{val}(\alpha, f); \text{val}(\alpha, b)\}1 \quad \text{for all } \alpha \in \text{At}(\mathcal{K}).$$

The truth values of complex formulas are derived from their subformulas according to the semantics of Priest´s logic. For a formula φ with subformulas ψ_j:

- Negation ($\varphi = \neg\psi$):

$$\begin{aligned} \text{val}(rep(\varphi), t) &\leftarrow \text{val}(rep(\psi), f) \\ \text{val}(rep(\varphi), f) &\leftarrow \text{val}(rep(\psi), t) \\ \text{val}(rep(\varphi), b) &\leftarrow \text{val}(rep(\psi), b) \end{aligned}$$

- Conjunction ($\varphi = \psi_1 \wedge \cdots \wedge \psi_k$):

$$\begin{aligned} \text{val}(rep(\varphi), t) &\leftarrow \forall j : \text{val}(rep(\psi_k), t) \\ \text{val}(rep(\varphi), f) &\leftarrow \exists j : \text{val}(rep(\psi_k), f) \\ \text{val}(rep(\varphi), b) &\leftarrow \neg\text{val}(rep(\varphi), t) \wedge \neg\text{val}(rep(\varphi), f) \end{aligned}$$

- Disjunction ($\varphi = \psi_1 \vee \cdots \vee \psi_k$):

$$\begin{aligned} \text{val}(rep(\varphi), t) &\leftarrow \exists j : \text{val}(rep(\psi_k), t) \\ \text{val}(rep(\varphi), f) &\leftarrow \forall j : \text{val}(rep(\psi_k), f) \\ \text{val}(rep(\varphi), b) &\leftarrow \neg\text{val}(rep(\varphi), t) \wedge \neg\text{val}(rep(\varphi), f) \end{aligned}$$

All formulas in the knowledge base must be satisfied, so their truth value must be t or b. This is enforced with a hard constraint that forbids any formula in $\mathcal{K}$ from being evaluated to *false*.

$$\leftarrow \text{val}(\varphi, f), \text{kb}(\varphi)$$

The predicate $\text{kb}(\varphi)$ marks that the formula φ is a member of the initial knowledge base $\mathcal{K}$. An answer set of the program up to this point corresponds to a valid Priest-model of $\mathcal{K}$.

The formula-based Contension measure seeks to minimize the number of formulas in $\mathcal{K}$ that are involved in a conflict. A formula is defined as conflict-involved if it contains at least one atom assigned the truth value *both*. This is captured by the predicate f_inconsistent(φ):

$$\text{f\_inconsistent}(\varphi) \leftarrow \text{kb}(\varphi), \text{formula\_contains\_atom}(\varphi, \alpha), \text{val}(\alpha, b)$$

The overall objective is to find a model of $\mathcal{K}$ that minimizes the cardinality of this set. This is expressed through the minimize statement:

$$\#\text{minimize}\{1, \varphi : \text{f\_inconsistent}(\varphi)\}$$

The final value of $I_{fc}(\mathcal{K})$ is the number of f_inconsistent(φ) predicates that hold in an optimal answer set. The ASP solver thus searches for a three-valued interpretation that satisfies all formulas while minimizing the number of formulas containing atoms with the value *both*.

Table 5 provides a conceptual mapping between logical constructs, their ASP representation, and the corresponding elements in the Python implementation:

Concept	**ASP Representation**	**Python Identifier**
Atom	`atom(a)`	`ATOM`
Truth value assignment	`val(a, t/f/b)`	`TRUTH_VALUE_PREDICATE`
Formula is an atom	`formula_is_atom(f,a)`	`FORMULA_IS_ATOM`
Negation	`negation(sub, f)`	`NEGATION`
Conjunction	`conjunction(f)`	`CONJUNCTION`
Disjunction	`disjunction(f)`	`DISJUNCTION`
Part of a conjunction	`conjunct_of(sub, f)`	`CONJUNCT_OF`
Part of a disjunction	`disjunct_of(sub, f)`	`DISJUNCT_OF`
Number of subformulas	`num_conjuncts(f, n)`, `num_disjuncts(f, n)`	`NUM_CONJUNCTS`, `NUM_DISJUNCTS`
Formula belongs to KB	`kb(f)`	`KB_MEMBER`
Formula contains atom	`formula_contains_atom(f,a)`	`add_formula_atom_links()`
Formula is inconsistent	`f_inconsistent(f)`	`F_INCONSISTENT`
Implication/Equivalence	translated to `OR/AND/NOT`	`is_implication()`, `is_equivalence()`

Table 5: Conceptual mapping between logical concepts, ASP predicates, and Python implementation

This structured representation results in a fully declarative semantic model of the knowledge base, implemented entirely within ASP. The Python component is responsible for the syntactic decomposition and for generating the ASP program, which correctly implements the three-valued evaluation and the optimization process for the inconsistency measure.

3.3.3. Implementation of the ASP Approach

The implementation of the ASP encodings can be found in the class *ASPEncoder* in *solver_ASP.py*. The class contains the necessary function *pl_to_asp* for converting a formula into a tree structure. In a recursive call, each formula and subformula is examined for its operators, whereby implications and equivalences are eliminated, and then transformed into rules that syntactically represent the composition of the formula. Conjunctions, disjunctions, negations, and the case where a formula consists merely of an atom are directly included as such rules in the program. The *pl_to_asp* function thus provides the foundation for the solver to process the knowledge base and for the further logic to be programmed in ASP. It is initially called by the function *handle_formulas_in_kb* for each formula in the knowledge base.

The *encode* function makes use of this possibility and, in addition to the facts for mapping the formula structure, establishes further facts for the existing truth-values and the atoms present in the knowledge base. Furthermore, the assignment of atoms to formulas is ensured by using the function *formula_atom_links* to find the contained atoms for each formula and to add the fact *formula_contains_atom*.

The truth evaluation of the connectivity rules is performed with the help of other helper functions. If a conjunction, disjunction, or negation is present in one of the formulas of the knowledge base, the corresponding function *add_conjunction_rules*, *add_disjunction_rules* or *add_negation_rules* is called. In all three cases, three constraints are added that represent which conditions must be met for the assignment of truth values.

Furthermore, the constraints for the evaluation of the truth-values are established, with which a formula is not allowed to be *false* and an atom must be assigned exactly one truth value. Additionally, the evaluation of a formula as inconsistent is added for whenever an atom occurring in it (*formula_contains_atom*) is assigned the value both.

Finally, the encoder returns the created program, which is passed on to the grounder and solver Clingo in *main.py*. With the help of the following lines of code, the solver is instructed to search for exactly one optimal model and to stop the search as soon as it has found it and proven its optimality.

If a model is found, *on_model* is called to read out the costs as I_{fc}, as well as to determine the inconsistent formulas and the atoms assigned the truth value *both*.

3.3.4. Example of the ASP Encodings

One more time, the knowledge base $K = \{A \wedge B, \neg A\}$ is given. The knowledge base thus contains the two formulas `f0` $= A \wedge B$ and `f1` $= \neg A$. The set of atoms in $\mathcal{K}$ is $At(\mathcal{K}) = \{A, B\}$. Since constants in ASP are written in lowercase, the atoms will be represented as $At(\mathcal{K}) = \{a1, a2\}$.

The ASP encoder can therefore define the facts for the three-valued logic, the formulas, and the atoms. To easily determine later which atoms appear in a formula of the knowledge base, *formula_contains_atom* facts are generated.

```
tv(t).
tv(f).
tv(b).
kb(f0)
kb(f1)

atom(a)
atom(b)

formula_contains_atom(f0,a2).
formula_contains_atom(f0,a1).
formula_contains_atom(f1,a1).
```

The necessary facts are also generated to represent the structure of the formulas. Formula `f0` consists of a conjunction of the atoms $a1$ and $a2$. Two subformulas `f0_0` and `f0_1` are created, each containing one atom.

```
conjunction(f0).
num_conjuncts(f0,2).
conjunct_of(f0_0,f0).
formula_is_atom(f0_0,a1).
conjunct_of(f0_1,f0).
formula_is_atom(f0_1,a2).
```

Formula `f1`, on the other hand, consists only of the negated atom $a1$, for which the negated subformula `f1_n` is created.

```
negation(f1_n,f1).
formula_is_atom(f1_n,a1).
```

Added to this are the constraints for the use of truth values. The first rule is an integrity constraint, which ensures that all models satisfy the knowledge base. The second rule is a choice rule, which instructs the solver to find a unique truth assignment for each atom.

```
:- val(X, f), kb(X).

1{val(X,Y) : tv(Y)}1 :- atom(X).
```

The correct assignment of truth values across the formula structure is ensured by further constraints, starting with the constraint for the case where a formula consists only of one atom

```
24    val(X,Z):tv(Z):- formula_is_atom (X,Y), val (Y,Z).
```

and the occurring logical operators (conjunction and negation). These rules recursively define how the truth value of a complex formula is determined from the values of its parts.

```
25    val(Y,t):conjunction(Y):-N{val(X,t):
26        conjunct_of(X,Y)}N,num_conjuncts(Y,N).
27    val(Y,f):conjunction(Y):-1{val(X,f)},conjunct_of(X,Y).
28    val(X,b):-conjunction(X),not val(X,t),not val(X,f).
29
30    val(Y,t):-negation(X,Y),val(X,f).
31    val(Y,f):-negation(X,Y),val(X,t).
32    val(Y,b):-negation(X,Y),val(X,b).
```

As the last constraint, the evaluation for the inconsistency of a formula is added.

```
33    f_inconsistent(F) :- kb(F),
34    formula_contains_atom(F,A), val(A,b).
```

Finally, the minimize statement is added.

```
35    #minimize { 1,F : f_inconsistent(F) }.
```

The satisfaction of the formulas can only be achieved through the assignment $a1 = b$.

- An assignment of `val(a1, t)` would result in `val(f1, f)`, which is forbidden.
- An assignment of `val(a1, f)` would result in `val(f0, f)`, which is also forbidden.

Thus, the rule `val(X,b):-conjunction(X),not val(X,t),not val(X,f).` applies for *f0*. Furthermore, the constraint `val(Y,b):-negation(X,Y),val(X,b).` applies for the evaluation of *f1*. Consequently, both formulas are included in `f_inconsistent(F)`, which marks them as inconsistent.

4. Experimental Evaluation

In this section, the results of the algorithmic approaches are reviewed and evaluated. First, the Setup is described to allow a comparison between different environments,

following a short description of the used datasets. The second section compares the WCNF encodings, coming from the naive and Tseitin approach. In this comparison, the number of hard clauses will be calculated whereby a higher number can be the reason for possible performance issues. After that the results of the used datasets are compared, seperated by the used solver, where the runtime is crucial for the computation of the I_{fc}. The last section gives an interpretation of these results, aiming to give an answer to the third research question.

4.1. Description of the Experimental Setup

The code described in the previous chapters is programmed in Python. The MaxSAT Solver RC2 was imported from the library PySAT, while the Potsdam Answer Set Solving Collection (Potassco) provides the library for the grounder and solver Clingo.

The experiments were run on a machine using 16GB RAM and the AMD Ryzen 5 6600HS, which features 6 Cores and operates with a base clock speed of 3.3 GHz and can boost up to 4.5 GHz.

The used datasets contain different complex knowledge bases. Every knowledge base consists of formulas, connected with the common logical operators like conjunction, disjunction, negation, implication and equivalence. As there is no dedicated benchmark dataset, new synthetic data were created and additionally, datasets from other research fields were translated [KGLT22].

- The SRS dataset contains 1800 synthetic generated knowledge bases. These were generated with the help of the Syntactic Random Sampler (SRS) from the TweetyProject[3] [KT21]. The complexity of the knowledge bases varies strongly; the smallest ones include 5 to 15 formulas with a signature size of 3, while the largest ones include 50 to 100 formulas with a signature size of 30 [KGLT22].

- The machine learning (ML) dataset contains 1920 knowledge bases, generated from the Animals with Attributed (AWA) dataset. This dataset describes 50 animal types with the help of 85 binary attributes. With the help of the Apriori algorithm, association rules were mined, which then were interpreted as logical implications. Additionally, the attributes for any random animal were added as facts. This probably leads to a high inconssitency.

- The argumentation (ARG) dataset consists of 326 knowledge bases and an average signature size of 827, whereby each of these knowledge bases consists of clauses in CNF derived of a standard SAT encoding with added constraints [NKTJ23]. It is based on the benchmarks of the International Competition on Computational Models of Argumentation 2019 [NKTJ23, KT21].

[3] *https://tweetyproject.org/*

For the encoding and solving time of each solver, a timeout of 1,000 seconds was established. If one gets into this timeout, the algorithm passes over to the remaining approach or the following dataset.

4.2. Comparison of the WCNF Encodings

In the naive approach, formulas are transformed into CNF by fully distributing conjunctions over disjunctions. In the worst case, this can lead to an exponential number of clauses [BHvMW09]:

$$C_{naiv}(\varphi) = \mathcal{O}(2^d)$$

where d denotes the depth or the maximum number of nested disjunctions over conjunctions in the formula φ. The total number of CNF clauses for a knowledge base $\mathcal{K} = \{\varphi_1, \ldots, \varphi_m\}$ is then given by:

$$C_{naiv}(\mathcal{K}) = \sum_{i=1}^{m} C_{naiv}(\varphi_i)$$

Based on the implemented WCNF encoding, the following additional hard clauses are generated:

- 4 hard clauses for each unique atom in $\mathcal{K}$
- One hard clause for each formula–atom pair in $\mathcal{K}$
- One hard clause for each clause in the CNF transformation

The total number of hard clauses then can be estimated as:

$$H_{naiv}(\mathcal{K}) = 4 \cdot |At(\mathcal{K})| + \sum_{i=1}^{m} |At(\varphi_i)| + C_{naiv}(\mathcal{K})$$

In the Tseitin-based approach, each subformula ψ is replaced by an auxiliary variable x_ψ and encoded using a constant number of clauses:

$$C_{tseitin}(\varphi) = c \cdot s$$

where s is the number of logical operators in φ and $c \leq 3$ is a constant indicating the maximum number of clauses per operator in the Tseitin encoding. The total number of CNF Tseitin clauses for a knowledge base $\mathcal{K} = \{\varphi_1, \ldots, \varphi_m\}$ is then:

$$C_{tseitin}(\mathcal{K}) = c \cdot \sum_{i=1}^{m} s_i$$

The WCNF encoding yields the following hard clauses:

- 4 hard clauses for each unique atom in $\mathcal{K}$
- One hard clause for each formula–atom pair in $\mathcal{K}$
- The Tseitin encoding $C_{tseitin}$
- One final clause to enforce the top-level variable of each Tseitin-encoded formula

The total number of hard clauses H can thus be computed as:

$$H_{tseitin}(\mathcal{K}) = 4 \cdot |At(\mathcal{K})| + \sum_{i=1}^{m} |At(\varphi_i)| + \sum_{i=1}^{m} (3s_i + 1)$$

The number of soft clauses S is the same for both approaches and corresponds to the number of formulas in $\mathcal{K}$:

$$S(\mathcal{K}) = m$$

Therefore, the total number of WCNF clauses W for both encodings is given by:

$$W_{naiv}(\mathcal{K}) = H_{naiv}(\mathcal{K}) + m$$

$$W_{tseitin}(\mathcal{K}) = H_{tseitin}(\mathcal{K}) + m$$

On the SRS dataset, the naive approach has fewer hard clauses than the Tseitin-based encoding (figure 3). The explanation lies in the structure of the formulas in the dataset.

Although the naive transformation may be exponential, this only occurs in formulas that involve deep nesting of disjunctions over conjunctions. Many formulas in the SRS dataset are already close to CNF or contain only simple combinations of logical operators. Therefore, the naive CNF generation results in a relatively small number of clauses.

In contrast, the Tseitin transformation introduces additional auxiliary variables and encoding overhead, even for small or simple subformulas. Each logical operator adds up to 3 hard clauses, and each top-level formula adds one more clause to enforce its truth. As a result, the Tseitin-based encoding may produce more clauses in cases where the naive transformation remains compact.

To understand when the Tseitin approach becomes more efficient, consider formulas of the form:

$$(A_1 \wedge A_2) \vee (A_3 \wedge A_4) \vee \cdots \vee (A_{2d-1} \wedge A_{2d})$$

This disjunction of d conjunctions forces the naive transformation to distribute conjunctions over disjunctions. The result is:

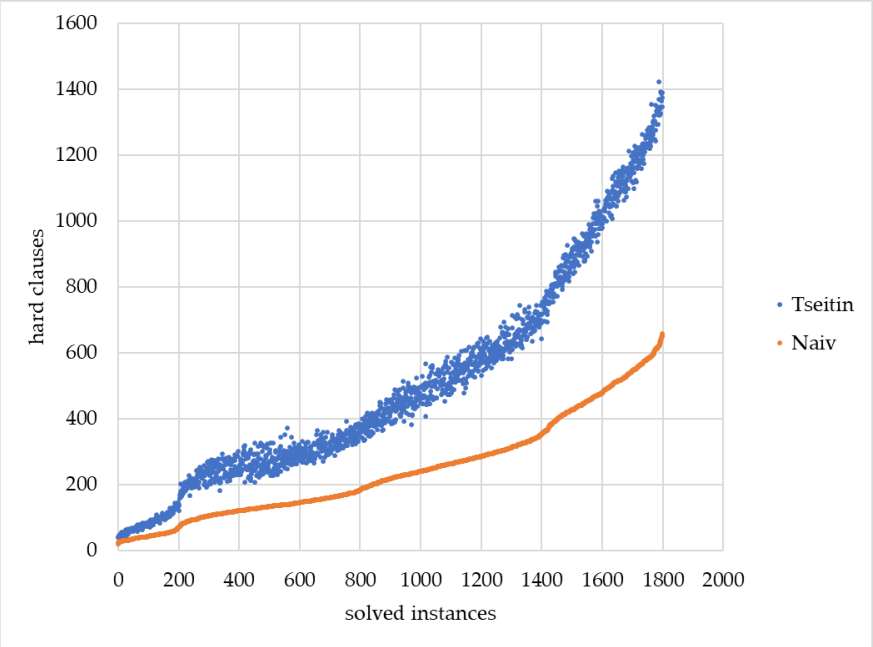


Figure 3: WCNF hard clauses comparison

$$C_{naiv}(d) \in \mathcal{O}(2^d)$$

since distributing conjunctions over disjunctions leads to clause explosion [BHvMW09]. The Tseitin encoding, however, grows linearly in d, because each conjunction and disjunction introduces only a small constant number of clauses:

$$C_{tseitin}(d) \in \mathcal{O}(d)$$

This behavior is reflected in Table 6, which shows that while the naive encoding starts with fewer clauses, it becomes exponentially more expensive. The break-even point occurs at around 5 disjunctions with each two conjunctions, after that the Tseitin transformation is more efficient.

4.3. Comparison of the Results

The results for the runs of each dataset and solver (here ASP and naive MaxSAT) were logged in a CSV file. Table 7 shows the comparison of the number of solved instances and the cumulative runtime for each dataset.

The results reveal a clear trade-off between the two approaches. While the naive MaxSAT solver exhibits significantly lower runtimes on the instances it can solve, the ASP approach proves to be more robust, solving a higher number of instances on the more challenging datasets (ML and ARG) within the 1000-second timeout.

Conjunctions (d)	**Naive**	**Tseitin**
1	12	14
2	24	30
3	38	46
4	56	62
5	82	78
6	124	94
7	198	110

Table 6: Break-even-point of the number of WCNF hard clauses for both CNF transformations

	ASP		**MaxSAT**	
Dataset	instances solved	sum of runtime (s)	instances solved	sum of runtime (s)
SRS *(1800)*	1800	30,25	1800	6,17
ML *(1920)*	1520	1,567.63	1088	12.97
ARG *(326)*	186	4,585.55	162	11.75

Table 7: Comparison of the results of both solvers

On the SRS dataset, which appears to consist of structurally simpler problems, both solvers solve all 1800 instances. The MaxSAT approach, with a cumulative runtime of 6.17 s, is nearly five times faster than the ASP approach (30.25 s). On the ML dataset, in contrast, the ASP solver solves 432 more instances than the MaxSAT solver. This extremely low cumulative runtime of the MaxSAT solver (12.97 s) suggests that it primarily solves the easy instances very quickly before timing out early on the more difficult ones. A similar behavior is observed on the ARG dataset. It should be noted here that for the ASP approach, 41 instances were terminated not due to a timeout, but because of a memory overflow. The reason of this may be the high complexity of the encoding for these specific instances.

To analyze the origin of the runtime differences, the time for encoding and the solving time of the solver were considered separately. Figure 4 shows the average times for the ARG dataset.

It becomes clear that for both approaches, the encoding time is negligible compared to the solving time. The main part of the runtime for the ASP approach is the solving phase. This means that the performance difference does not lie in the preprocessing step, but rather in the efficiency of the underlying search algorithms of the solvers for this class of problems.

The runtime distribution diagrams in Figures 5 to 7 allow for a more in-depth analysis of the solvers' behavior. The x-axis shows the number of solved instances, sorted by increasing runtime, which is shown on the logarithmic y-axis.

For the SRS dataset (Figure 5), the curves confirm the high performance of both

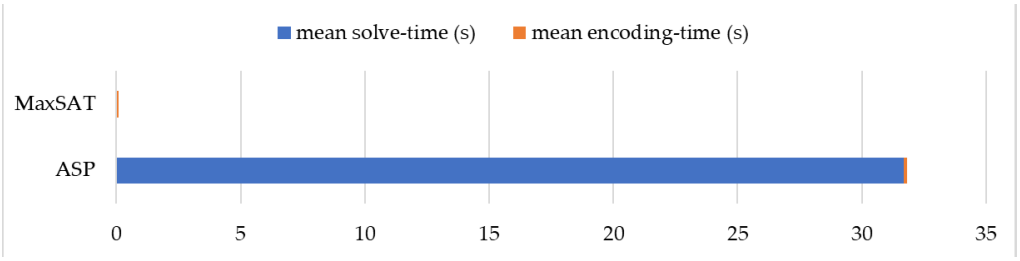


Figure 4: Comparison of the mean encoding and solve time of the ARG dataset

approaches. All instances are solved in less than 0.1 seconds. Nevertheless, a steeper rise in the curve for the ASP solver is already identificable. The SRS dataset is categorized by the number of unique atoms and the minimum and maximum of formulas in each knowledge base. These categories were collected in folders, by which the name of the folder is created on the mentioned information separated by an underline. For a more detailed view, table 8 and table 9 show the mean, sum and the minimum and maximum runtime of each folder.

Folder	Mean	Sum	Min	Max
sig3_5_15	5.19	1,038.49	2.79	10.76
sig5_15_25	8.88	1,776.83	5.73	19.69
sig10_15_25	8.42	1,684.64	5.61	16.1
sig15_15_25	9.25	1,850.34	6.12	17.81
sig15_25_50	15.70	3,140.71	8.40	26.60
sig20_25_50	18.45	3,689.15	9.97	32.35
sig25_25_50	21.12	4,224.80	9.99	32.90
sig25_50_100	30.95	6,190.80	15.08	50.91
sig30_50_100	33.28	6,656.47	17.27	69.92

Table 8: Runtime measures of the ASP approach over the SRS dataset (in ms)

It is recognizable, that some folders have a smaller mean runtime, although they have the identical number of minimal and maximal formulas but a higher number of unique atoms. These circumstances would raise the expectation of a higher runtime. This may come from more extensive and nested formulas within these folders or a larger amount of formulas near the maximum.

The trend of the SRS dataset intensifies on the ML dataset (Figure 6). The curve for the ASP solver rises steadily, until it reaches the timeout value. The last solved instance required nearly 100 seconds, which shows that the solver works on very hard instances for a long time. In contrast, the behavior of the MaxSAT solver is almost binary: it solves a large number of instances extremely quickly but then reaches a point where the runtime increases so sharply that almost all remaining instances

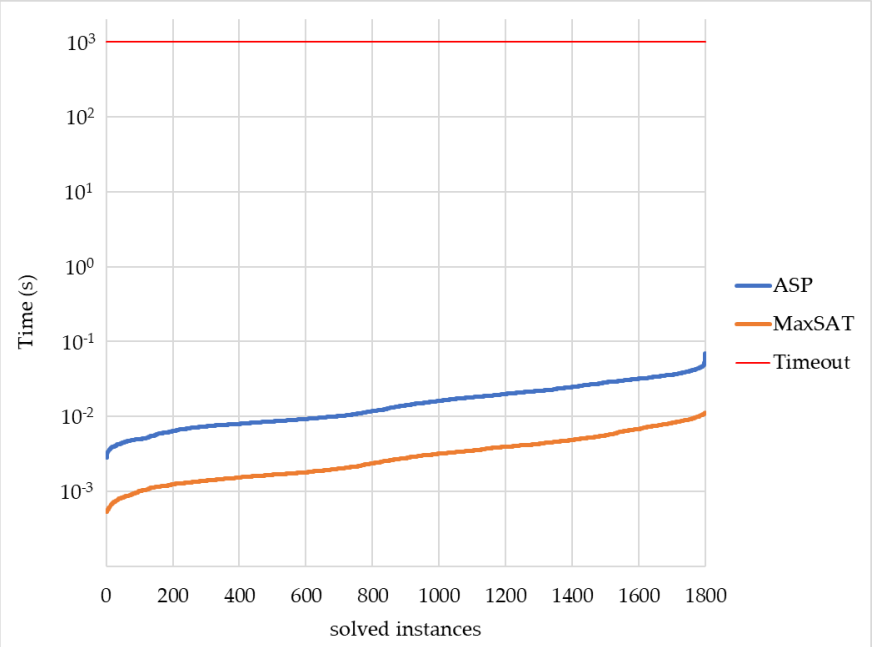


Figure 5: Runtime of the SRS Dataset

Folder	Mean	Sum	Min	Max
sig3_5_15	1.05	209.34	0.53	2.38
sig5_15_25	1.94	388.52	1.06	3.87
sig10_15_25	1.62	323.11	0.93	5.36
sig15_15_25	1.62	324.74	1.03	3.50
sig15_25_50	3.34	667.15	1.69	5.96
sig20_25_50	3.60	719.71	2.07	6.32
sig25_25_50	3.87	773.04	1.93	6.17
sig25_50_100	6.69	1,338.61	3.28	10.77
sig30_50_100	7.13	1,425.12	3.17	11.42

Table 9: Runtime measures of the (naive) MaxSAT approach over the SRS dataset (in ms)

time out. This explains why the MaxSAT solver solves fewer instances but has a very low cumulative runtime.

Figure 7 for the ARG dataset confirms this impression. Although the MaxSAT solver starts faster on the easiest instances, its curve is significantly steeper, leading it to solve fewer instances overall compared to the more resilient ASP solver.

A solver-independent evaluation in the form of diagrams showing the frequency of I_{fc} per dataset can be found in Appendix A for the SRS dataset, Appendix B

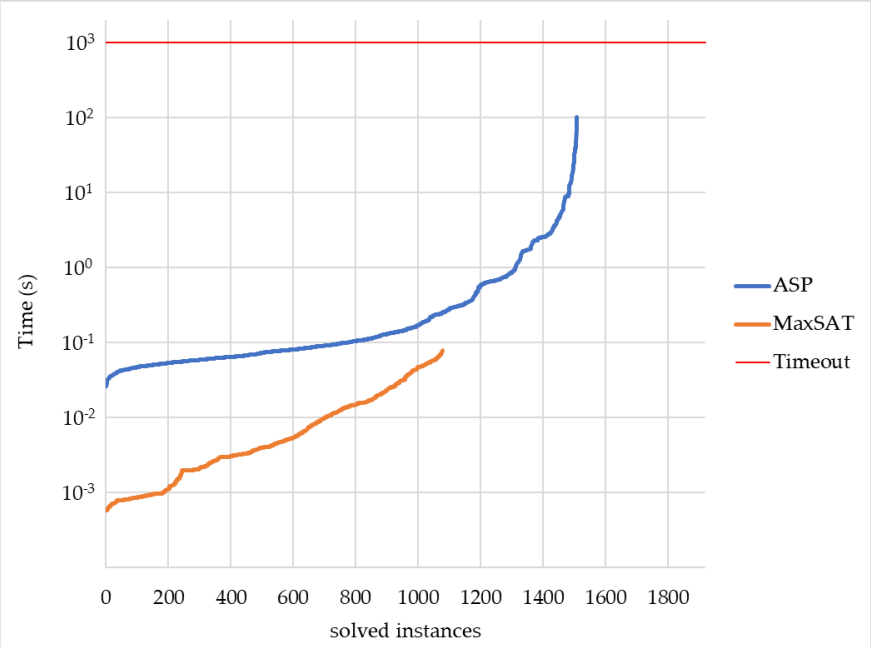


Figure 6: Runtime of the ML Dataset

for the ML dataset, and Appendix C for the ARG dataset. Appendix D provides a complementary tabular overview for the SRS dataset. It is worth mentioning that the I_{fc} value of 1 can never occur, as there always have to be at least two formulas, affected by the inconsistency. Otherwise it would need to be a safe formula, which has to be consistent by definition.

An examination of the diagrams reveals that I_{fc} varies significantly and can reach extremely high values. Although lower values are particularly common, the results clearly demonstrate the magnitude the inconsistency measure can attain when a large number of formulas contain inconsistent atoms.

4.4. Interpretation

The data analysis of the two algorithmic approaches shows a clear picture. The MaxSAT solver is significantly faster than the ASP-based approach in a direct comparison of simple knowledge bases. Although the encoding makes no significant time difference in these cases, it is the sticking point of the issue for the MaxSAT approach.

The MaxSAT solver RC2 demonstrates its strength on the SRS dataset, where it solves all 1800 instances in just 6.17 seconds, so it is about five times faster than the ASP approach. This high efficiency results from the specialized architecture of modern SAT solvers, which are optimized for processing clauses in CNF. However,

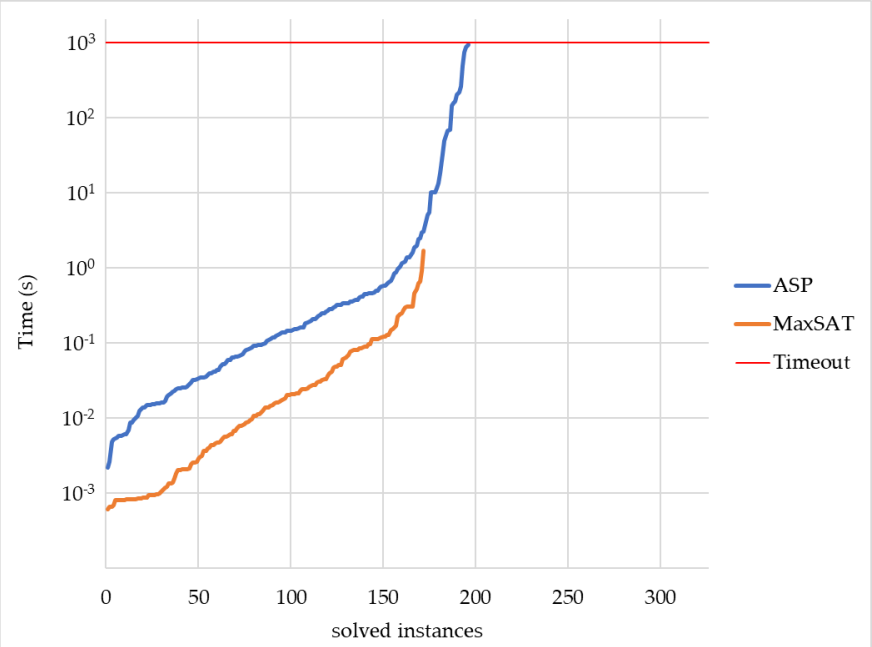


Figure 7: Runtime of the ARG Dataset

as soon as the complexity of the knowledge bases increases, as in the ML and ARG datasets, the picture reverses. Here, the MaxSAT approach solves significantly fewer instances than the ASP approach before it times out. The cactus plots in Figure 6 and Figure 7 visualize this behavior. The runtime curve of the MaxSAT solver remains extremely flat for many instances but then rises almost vertically upon reaching a complexity threshold. The low cumulative time for the ML and ARG datasets is therefore not a sign of consistent efficiency but reflects that the solver processes the simple cases very quickly and capitulates early on the more difficult ones.

In contrast, the ASP approach demonstrates greater robustness against structural complexity. Although it requires a higher baseline effort for simple instances, its runtime curve on more complex datasets rises faster. This indicates that the solver makes a stepwise progress even with increasingly difficult problems, rather than failing abruptly. This robustness allowed it to solve significantly more instances on the ML and ARG datasets within the time limit. However, the memory overflows that occurred on some very complex ARG instances suggest that the grounding process of Clingo can reach its limits with very large and nested rules.

The core of these performance differences lies in the different representation of the problem. The MaxSAT approach is dependent on the quality of the CNF transformation. Surprisingly, the naive transformation appears to be more efficient here than the Tseitin transformation. The reason is that the formulas in the used datasets did not consist of that many nested disjunctions over conjunctions that lead to an

exponential explosion of clauses. For syntactically simpler formulas, the overhead of the Tseitin transformation is greater than the benefit because of the additional helper variables. So the efficiency of computing I_{fc} with MaxSAT strongly depends on the syntactic form of the input. This mirrors the sensitivity to syntax of the measure itself, which is the reason for the violation of postulates such as Adjunction Invariance (AI) and Exchange (EX).

This is where the strength of the ASP approach lies. It does not require conversion into a flat set of clauses but can directly represent the recursive tree structure of the formulas through declarative rules. The modeling by means of predicate logic, such as `val(Atom, Value)`, and the use of powerful constructs like Choice Rules to represent the three-valued semantics are more robust against syntactic complexity. The logical structure of the problem is passed directly to the solver, which is optimized to handle such rule-based programs.

In summary, it can be sayed that the choice of the optimal algorithm for calculating the formula-based contension inconsistency measure depends on many circumstances. Regarding the objectives of efficiency and scalability mentioned in this work, the MaxSAT approach shows a high efficiency for large but syntactically simple knowledge bases. The ASP approach, on the other hand, offers better scalability with respect to the structural complexity of the formulas. The decision for an approach in practice thus strongly depends on the expected nature of the knowledge base to be analyzed. The experimental results show that the practical computation of I_{fc} reflects the theoretical nature of the measure. This means that both are sensitive to the syntactic representation of the information.

5. Conclusion and Future Work

This chapter summarizes the key findings of this thesis, answers the research questions from the beginning, and provides an outlook on possible future researches that build on these results.

5.1. Conclusion

The goal of this master's thesis was the development, implementation, and experimental evaluation of two new algorithmic approaches for computing the formula-based contension inconsistency measure I_{fc}. For this purpose, Answer Set Programming (ASP) and the principle of (Maximum) Satisfiability (MaxSAT) were used.

To answer the first research question, an encoding for the Weighted Conjunctive Normal Form (WCNF) was developed that translates Priest's three-valued logic into classical logic. Each atomic truth value is represented by a Boolean variable, and hard clauses ensure that each atom is assigned exactly one value. The satisfaction of formulas in the knowledge base is also ensured by hard clauses, which allow satisfaction through the assignment of *both* to the involved atoms. Finally, a soft clause is introduced for each formula, whose violation results into a cost. Minimizing these

costs using a MaxSAT solver corresponds to minimizing the number of formulas affected by a conflict, which is the value of I_{fc}.

The second research question deals with the equivalent approach using ASP. The ASP method takes advantage of the ability to directly represent the syntactic tree structure of formulas through recursive rules. The knowledge base is transformed into a set of facts describing its structure. Using predicates and recursive rules, Priest's three-valued semantics is reconstructed. A choice rule assigns one of the three truth values to each atom. The optimization task is implemented through the `#minimize` statement, which minimizes the number of `f_inconsistent` facts.

To answer the third research question, the experimental evaluation showed that there is no universally better approach. The MaxSAT-based method is faster for large but syntactically simple knowledge bases but the performance heavily depends on the quality of the CNF transformation. As soon as formulas become structurally more complex the performance worses significantly, which leads to many timeouts. Surprisingly, the naive CNF transformation turned out to be more efficient than the Tseitin transformation, since the test data did not contain the deep nesting that would lead to an exponential clause explosion.

In contrast, the ASP-based approach proved to be more robust against structural complexity. Although it has a higher base cost for simple instances, the runtime scales better with complex formulas. The key advantage here is the ability to directly process the recursive structure of formulas without converting them into clauses.

In summary, the computation of I_{fc} mirrors the theoretical properties of the measure itself. Both approaches are sensitive to the syntactic representation of the knowledge base. The choice of the optimal algorithm therefore strongly depends on the expected nature of the knowledge bases which have to be analyzed.

5.2. Outlook for Future Research

The findings of this thesis open up many possibilities for future research. One interesting result is the observation that the naive CNF transformation was better suited for the datasets used than the Tseitin transformation. Future work could investigate this systematically by generating datasets with deep nested formulas. This would allow a precise determination of the break-even point described in Table 6 and the derivation of a general formula property at which the linear overhead of the Tseitin transformation becomes essential. This would require experiments on more powerful hardware and with longer timeouts.

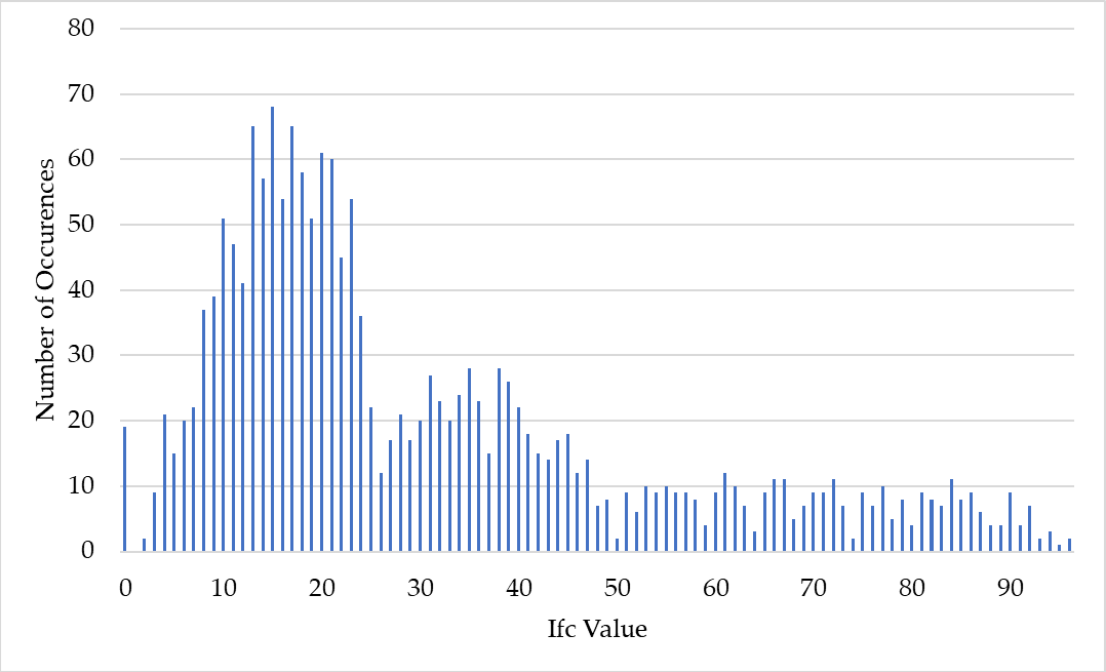
Since both approaches have clear strengths and weaknesses, the development of a hybrid solver could be a possible next step. Such an algorithm could first analyze a knowledge base based on syntactic features like the maximum nesting depth, the number of atoms and formulas or the ratio of conjunctions to disjunctions to finally decide which of the two approaches might be more performant.

The algorithms presented here were evaluated on synthetic or derived datasets. A crucial next step would be to apply them in real world scenarios in order to measure

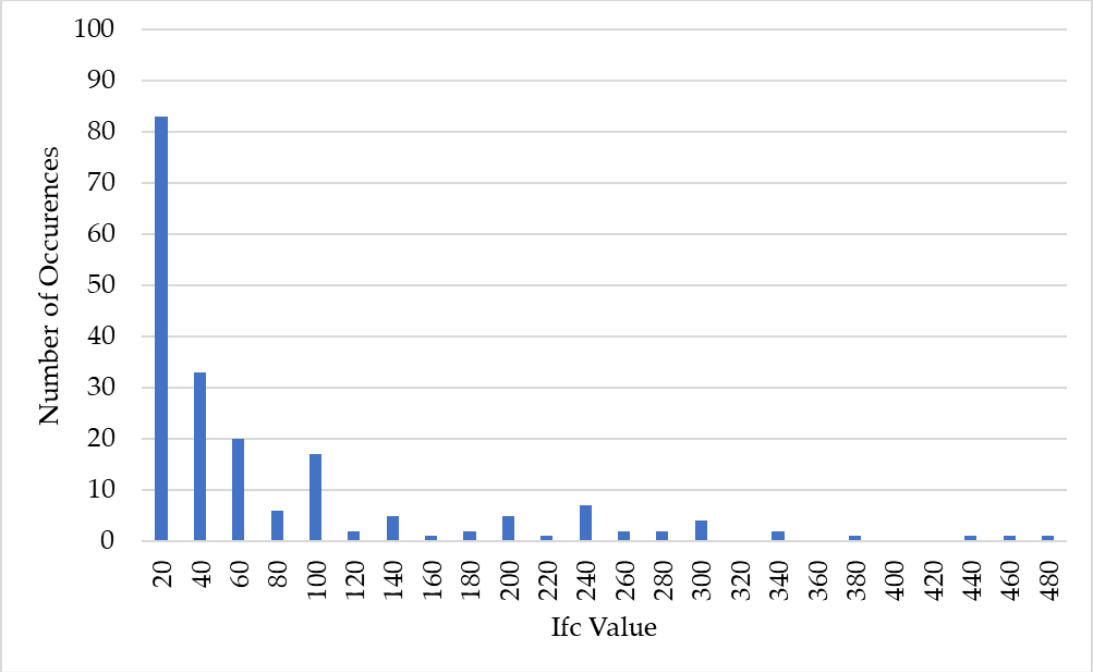
their performance on real data. This would not only demonstrate the algorithms in practice but also underline the practical relevance of I_{fc}. In addition, the algorithms already provide some extra information like the minimal sets of atoms assigned *both* and the formulas affected by them. These information could be processed further to give a better understanding of the occuring real world inconsistencies and recommendations how to handle with them.

6. Appendix

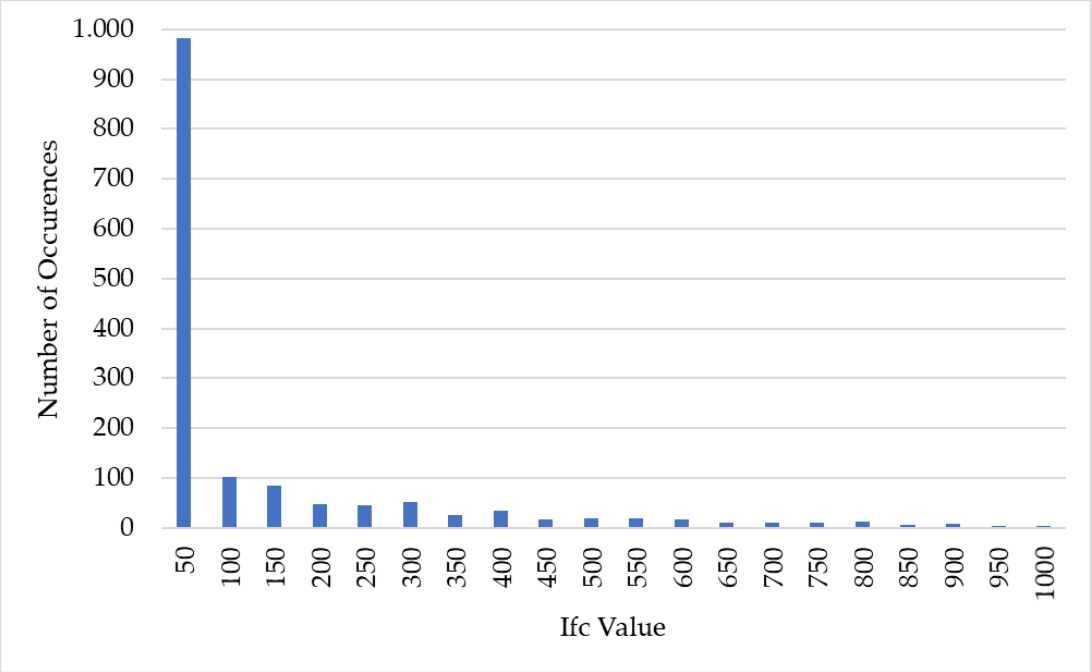
A. Number of Occurences for each I_{fc} value in the SRS dataset as diagram



B. Number of Occurences for upper bound I_{fc} value in the ML dataset as diagram



C. Number of Occurences for upper bound I_{fc} value in the ARG dataset as diagram

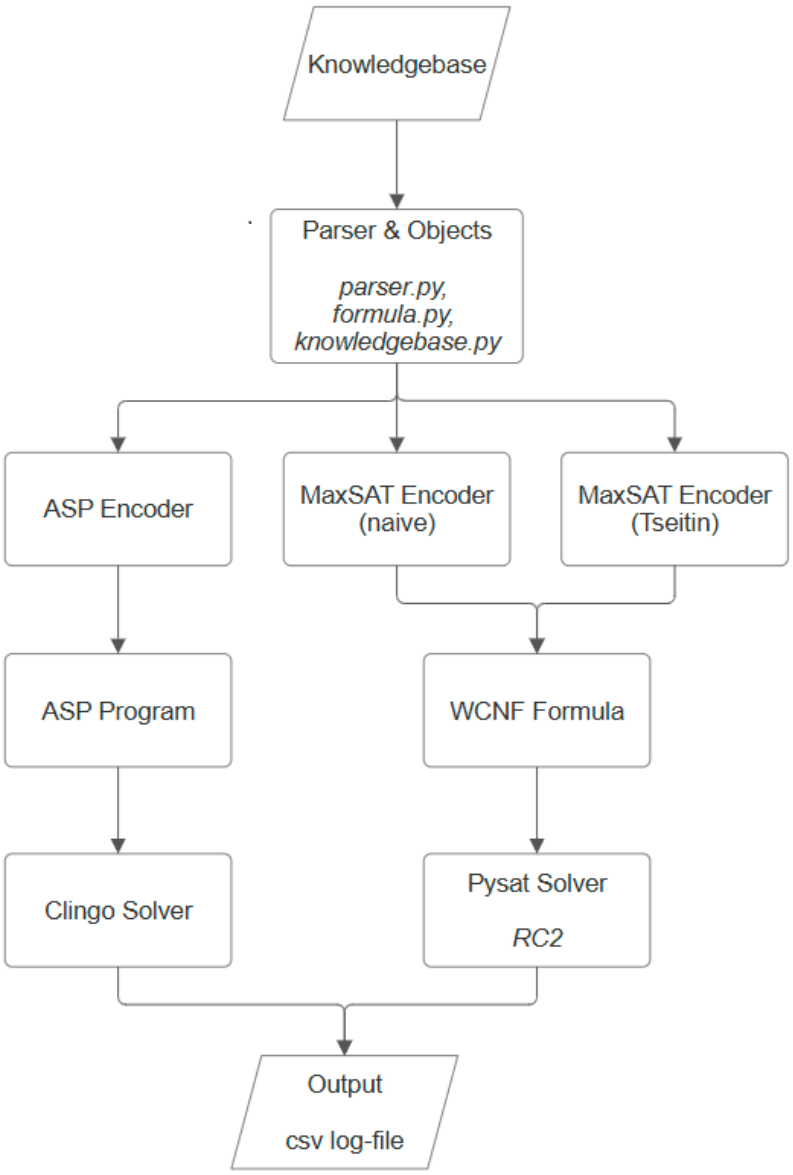


D. Number of Occurences for each I_{fc} value in the SRS dataset

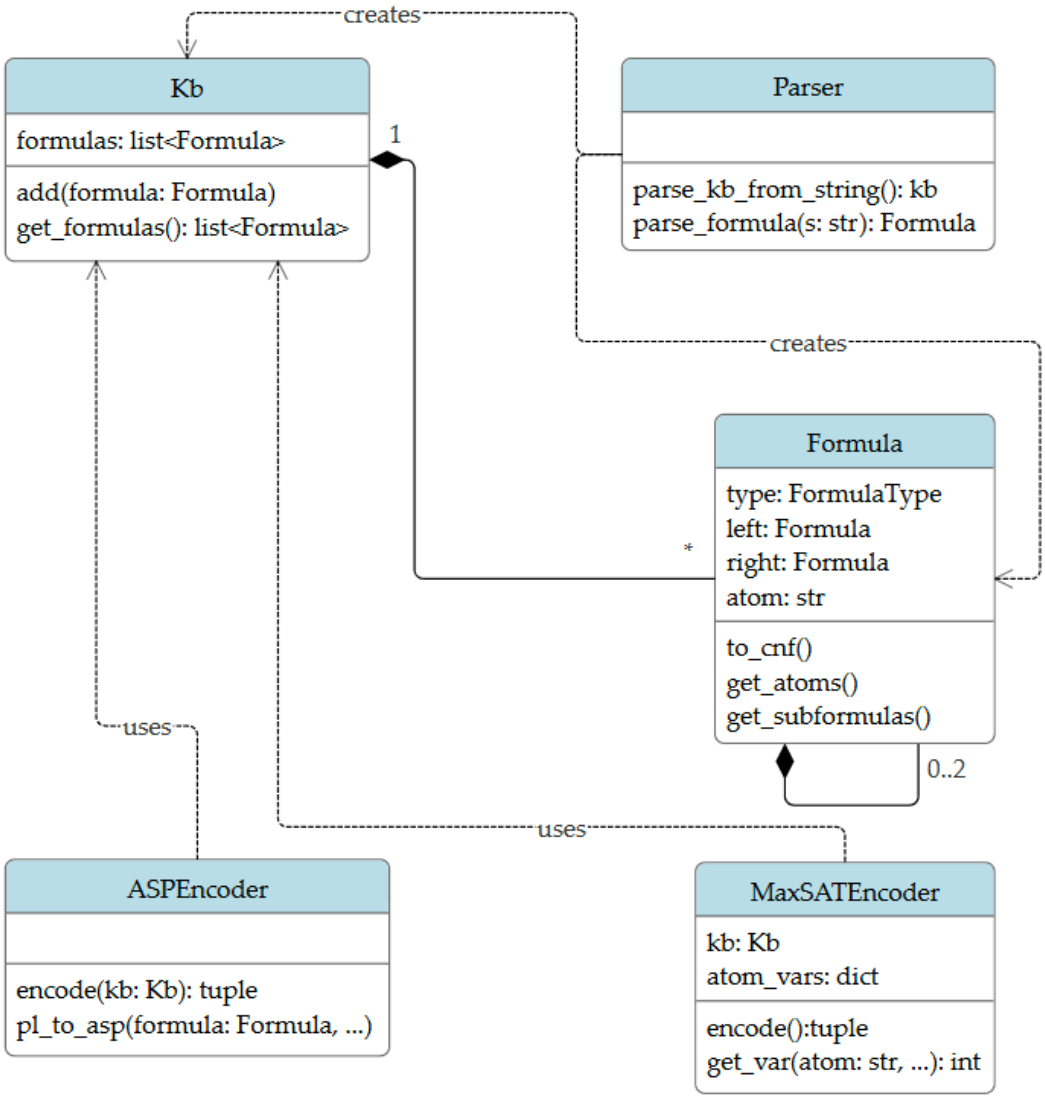
I_{fc}	**Occurences**	I_{fc}	**Occurences**	I_{fc}	**Occurences**
0	19	33	20	65	9
2	2	34	24	66	11
3	9	35	28	67	11
4	21	36	23	68	5
5	15	37	15	69	7
6	20	38	28	70	9
7	22	39	26	71	9
8	37	40	22	72	11
9	39	41	18	73	7
10	51	42	15	74	2
11	47	43	14	75	9
12	41	44	17	76	7
13	65	45	18	77	10
14	57	46	12	78	5
15	68	47	14	79	8
16	54	48	7	80	4
17	65	49	8	81	9
18	58	50	2	82	8
19	51	51	9	83	7
20	61	52	6	84	11
21	60	53	10	85	8
22	45	54	9	86	9
23	54	55	10	87	6
24	36	56	9	88	4
25	22	57	9	89	4
26	12	58	8	90	9
27	17	59	4	91	4
28	21	60	9	92	7
29	17	61	12	93	2
30	20	62	10	94	3
31	27	63	7	95	1
32	23	64	3	96	2

Table 10: Number of occurences for each I_{fc} value in the SRS dataset

E. Blockdiagram - System Architecture and Data Flow



F. Classdiagram - Data Modeling



References

[ABL13] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Sat-based maxsat algorithms. *Artificial Intelligence*, 196(2):77–105, 2013.

[BET11] Gerhard Brewka, Thomas Eiter, and Miroslaw Truszczynski. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[BH08] Philippe Besnard and Anthony Hunter. *A Logic-Based Theory of Deductive Arguments*. Artificial Intelligence. Springer, 2008.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of satisfiability*. IOS Press, 2009.

[BL12] Jc Beall and Shay Allen Logan. *Logic: The basics*. Routledge, 2012.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, Berlin, Heidelberg, 2007.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[Ebb18] Heinz-Dieter Ebbinghaus. *Einführung in die mathematische Logik*. Springer, Berlin, Heidelberg, 6 edition, 2018.

[EIST09] Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits. Answer set programming: A primer. In Chitta Baral, Evgeny Dantsin, and Michael Gelfond, editors, *Reasoning Web. Semantic Technologies for Information Systems*, volume 5689 of *Lecture Notes in Computer Science*, pages 40–110. Springer, 2009.

[GH11] John Grant and Anthony Hunter. Measuring consistency gain and information loss in stepwise inconsistency resolution. *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, 6717, 2011.

[GKSS08] Carla P. Gomes, Henry A. Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In *Foundations and Trends® in Artificial Intelligence*, volume 2, pages 235–391. Now Publishers, 2008.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming (ICLP)*, pages 1070–1080, 1988.

[Gra78] John Grant. Classifications for inconsistent theories. *Notre Dame Journal of Formal Logic*, 19(3):435–444, 1978.

[GS16] Dov M. Gabbay and Karl Schlechta. Making inconsistency respectable 1: A logical framework for inconsistency in reasoning. *Journal of Applied Logic*, 14:1–31, 2016.

[HK08] Anthony Hunter and Sébastian Konieczny. Measuring inconsistency through minimal inconsistent sets. *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning*, pages 358–366, 2008.

[Hun71] Geoffrey Hunter. *Metalogic: An Introduction to the Metatheory of Standard First Order Logic*. University of California Press, 1971.

[Hun07] Anthony Hunter. *Elements of Argumentation*. MIT Press, Cambridge, MA, 2007.

[IMM19] Alexey Ignatiev, Antonio Morgado, and João Marques-Silva. RC2: an Efficient MaxSAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):53–64, 2019.

[KGLT22] Isabelle Kuhlmann, Anna Gessler, Vivien Laszlo, and Matthias Thimm. A comparison of asp-based and sat-based algorithms for the contension inconsistency measure. 2022.

[KKS+22] Elias Kuiter, Sebastian Krieter, Chico Sundermann, Thomas Thüm, and Gunter Saake. Tseitin or not tseitin? the impact of cnf transformations on feature-model analyses. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 110:1–110:13. ACM, 2022.

[Kle04] Kevin C. Klement. Propositional logic. In *Internet Encyclopedia of Philosophy*. 2004.

[KT20] Isabelle Kuhlmann and Matthias Thimm. An algorithm for the contension inconsistency measure using reductions to answer set programming. In *Scalable Uncertainty Management (SUM 2020)*, volume 12322 of *Lecture Notes in Computer Science*, pages 289–296. Springer, Cham, 2020.

[KT21] Isabelle Kuhlmann and Matthias Thimm. Algorithms for inconsistency measurement using answer set programming. In *Proceedings of the 14th International Conference on Non-Monotonic Reasoning (NMR)*. University of Koblenz–Landau, 2021.

[KY95] George J Klir and Bo Yuan. *Fuzzy sets and fuzzy logic: Theory and applications*. Prentice Hall PTR, 1995.

[Lif19] Vladimir Lifschitz. *Answer Set Programming*. Springer, 2019.

[Men15] Elliott Mendelson. *Introduction to mathematical logic*. CRC press, 2015.

[MT99] Victor W. Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In Krzysztof R. Apt, Victor W. Marek, Mirosław Truszczyński, and David S. Warren, editors, *The Logic Programming Paradigm: A 25-Year Perspective*, Artificial Intelligence, pages 375–398. Springer Verlag, Berlin, Heidelberg, 1999. Department of Computer Science, University of Kentucky.

[NKTJ23] Andreas Niskanen, Isabelle Kuhlmann, Matthias Thimm, and Matti Järvisalo. Maxsat-based inconsistency measurement. 2023.

[Păt08] Mihai Pătraşcu. On the possibility of faster SAT algorithms. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '08)*, pages 1068–1075, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[Pri79] Graham Priest. The logic of paradox. *Journal of Philosophical Logic*, 8(1):219–241, 1979.

[Sip12] Michael Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 2012.

[SPBS03] Tran Cao Son, Enrico Pontelli, Marcello Balduccini, and Torsten Schaub. Answer set planning: A survey. *Theory and Practice of Logic Programming*, 2003.

[Thi13] Matthias Thimm. Measuring inconsistency in probabilistic knowledge bases. *Artificial Intelligence*, 197:1–24, 2013.

[Thi18] Matthias Thimm. On the evaluation of inconsistency measures. In *Measuring Inconsistency in Information*, volume 73, pages 19–60. College Publications, 2018.

[Thi19] Matthias Thimm. Inconsistency measurement. In Dov Gabbay, John Woods, Guillermo Simari, Sanjay Modgil, Henry Prakken, and Matthias Thimm, editors, *Handbook of Formal Argumentation*, pages 453–512. College Publications, 2019.

[Tse68] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning: Classical Papers on Computational Logic 1967–1970*, pages 466–483. Springer, 1968. Originally published in Russian in 1968; English translation in 1983.

[TW19] Matthias Thimm and Johannes P. Wallner. On the complexity of inconsistency measurement. In *Artificial Intelligence*, volume 275, pages 411–456, 2019.

[Wor24] James Worrell. The dpll algorithm. Lecture notes, University of Oxford, April 2024.