UNIVERSITÄT OSNABRÜCK

INSTITUT FÜR INFORMATIK
WISSENSBASIERTE SYSTEME

BACHELORARBEIT

EIN KONZEPT ZUR KORREKTUR INKONSISTENTER WISSENSBASEN UND DIE UMSETZUNG IN ANSWER SET PROGRAMMING

TOBIAS STOLZMANN

18. DEZEMBER 2018

ERSTGUTACHTER: PROF. DR. JOACHIM HERTZBERG
ZWEITGUTACHTER: DR. SEBASTIAN STOCK

ZUSAMMENFASSUNG

Diese Arbeit beschreibt ein Verfahren zur Korrektur inkonsistenter Wissensbasen. Es löst Widersprüche durch möglichst wenige Änderungen auf und wird für abstrakte Logiken auf Basis eines Konsequenzoperators eingeführt. Elementare Eigenschaften, wie die Existenz einer Korrektur oder die Unveränderlichkeit einer konsistenten Wissensbasis bei der Korrektur, werden theoretisch bewiesen. Da eine Korrektur unter Umständen unvollständig ist, wird erklärt, wie sie sich konsistent vervollständigen lässt. Da sie im Allgemeinen nicht eindeutig ist, wird die Auswahl einer Korrektur auf Basis zusätzlicher Informationen erklärt. Diese Arbeit beschreibt die Umsetzung des Verfahrens in Answer Set Programming. Es werden zwei Beispiele besprochen: Die Buchwelt modelliert ein Szenario aus der Robotik. Die Korrektur inkonsistenter Sudokus demonstriert die Leistungsfähigkeit des Verfahrens.

ABSTRACT

This work describes a procedure for correcting inconsistent knowledge bases. It resolves contradictions with as few changes as possible and is introduced for abstract logic based on a consequence operator. Elementary properties, such as the existence of a correction or the unchangeability of a consistent knowledge base during correction, are theoretically proven. Since a correction may be incomplete, I explain how it can be completed consistently. Since it is not unique in general, the selection of a correction is explained on the basis of additional information. This work describes the implementation of the procedure in Answer Set Programming. Two examples are discussed: The books world models a scenario from robotics. The correction of inconsistent Sudokus demonstrates the performance of the procedure.

DANKSAGUNG

Mein Dank gilt den Menschen, die zugehört haben. Mein Respekt gilt den Menschen, die zugehört und nicht verstanden haben. Meine Bewunderung gilt den Menschen, die zugehört und verstanden haben, dass es nichts zu verstehen gab.

Diese Menschen verdienen Ruhm und Ehre:

- Mama hat Essen und saubere Kleidung bereitgestellt und jede Laune ausgehalten.
- Papa hat Beweise korrigiert und mehr Fragen gestellt, als es Antworten gibt.
- Oma Hanna hat mich – ohne es zu wissen – zur Korrektur inkonsistenter Sudokus inspiriert.
- Torsten Schaub, Martin Gebser und Javier Romero haben mich herzlich an der Universität Potsdam empfangen und jede Frage zum Answer Set Programming beantwortet.
- Sebastian Stock hat mich zuvorkommend betreut.
- Prof. Hertzberg hat mir das Thema zur Verfügung gestellt. Er hat mir die Freiheit gelassen, es nach Lust und Laune zu interpretieren, und mich zum richtigen Zeitpunkt gefragt, ob ich diese Freiheit wirklich nutzen möchte.

INHALT

1 EINLEITUNG

Motivation

Die Suche nach der Wahrheit ist ein erstrebenswertes und verbreitetes Ideal. Das gilt gleichsam im Alltag, in der Politik[1], vor Gericht, in der Wirtschaft und in der Wissenschaft. Wer die Wahrheit kennt, hat Vorteile. Wer sie spricht, tut Recht. Wer Lügen verbreitet ist bestenfalls unmündig, weil er aus Faulheit und Feigheit die Lügen anderer nachplappert, ohne seinen Verstand zu benutzen [Kan84, S. 481]. Die Wahrheit zu finden, so sie denn existiert, ist erstaunlich schwierig, denn was soll die Wahrheit sein, wenn verschiedene Quellen verschiedene Wahrheiten propagieren. Überwiegend unbestritten ist, dass die Wahrheit vollständig und widerspruchsfrei sein muss. Ein erster Schritt auf dem Weg zur Wahrheit ist es also, eine vollständige und widerspruchsfreie Wissensbasis zu finden.

Die klassische Logik ist sehr erfolgreich, wenn es darum geht, Widersprüche zu erkennen. Falls sie jedoch tatsächlich einen Widerspruch entdeckt, kennt sie keinen Ausweg, außer die gesamte Wissensbasis abzulehnen. Dies greift zu kurz, wenn man annimmt, dass jede Lüge ein wenig Wahrheit enthält, und wird zu Recht in [BHS05] kritisiert. Jeder Sensor macht gelegentlich einen Fehler und eine gespeicherte Information ist bisweilen veraltet. Das bedeutet jedoch nicht, dass alle anderen Informationen falsch sind. Ein Roboter, der sich rational verhält, sollte nicht blind, taub und ohne Gedächtnis handeln, nur weil er unvollständige oder falsche Informationen bekommt. Er sollte stattdessen Widersprüche korrigieren und Wissenslücken konsistent füllen. Der Mensch tut dies fast automatisch, wie der McGurk-Effekt [MM76] eindrucksvoll belegt.

Fragestellung

Die Frage, mit der sich diese Arbeit beschäftigt, lautet also: Wie lässt sich ein Widerspruch korrigieren und eine Wissenslücke füllen?

Ansatz

Eine Antwort lautet: durch Auswahl! [Res64] schlägt vor, die Aussagen einer Wissensbasis als Hypothesen zu betrachten, deren Wahrheitsstatus fragwürdig oder unbestimmt ist. Ausgehend von einer Hypothese, die als wahr angenommen wird, lässt sich für jede andere Hypothese entscheiden, ob sie beibehalten werden kann oder abgelehnt werden muss, weil sie im Widerspruch zu der bereits gewählten Hypothese steht. Durch wiederholte Auswahl einer Hypothese, die noch nicht abgelehnt oder ausgewählt wurde, lässt sich eine konsistente Wissensbasis berechnen. Dieses Verfahren kann auch zur Vervollständigung eingesetzt werden, wenn man nach Abschluss der Korrektur jede beliebige Aussage als Hypothese betrachtet.

1 Der Autor meint die Politik in ihrer Idealform.

Eine Korrektur dieser Art ist nicht eindeutig. Jede Entscheidung für oder gegen eine Hypothese verändert die Korrektur. Es ist daher zu untersuchen, was eine sinnvolle Entscheidung ausmacht. Eine intuitive Möglichkeit ist, die Entscheidungen so zu wählen, dass die Korrektur durch möglichst wenige Änderungen aus der ursprünglichen Wissensbasis hervorgeht. Die Lösung dieses Optimierungsproblems ist jedoch auch nicht eindeutig. Im Laufe dieser Arbeit wird sich zeigen, dass man Eindeutigkeit unter bestimmten Bedingungen erzwingen kann, wenn weitere Informationen zur Verfügung stehen. Es ist zum Beispiel denkbar, Hypothesen zu bevorzugen, die aktueller oder wahrscheinlicher sind oder weniger Kosten verursachen.

alternative Ansätze

Es gibt zahlreiche konkurrierende Ansätze, mit Widersprüchen und Wissenslücken umzugehen. Die Berechnung maximaler Extensionen in einer reiterschen oder poolschen Default-Theorie [Poo88; Bre89; Fre98] ist dem Ansatz dieser Arbeit sehr ähnlich. Belief Revision [DS03; DS07] beschäftigt sich mit der konsistenzerhaltenden Aktualisierung einer Wissensbasis. Parakonsistente Logiken [KL92] verändern die Art der logischen Schlüsse, um Widersprüche in einer Wissensbasis zu tolerieren, ohne diese zu korrigieren. Andere Ansätze betrachten die Abhängigkeiten zwischen verschiedenen Widersprüchen [BG02]. Diese Arbeit geht auf keinen dieser Ansätze näher ein.

Überblick

Diese Arbeit besteht aus folgenden Teilen: Kapitel 2 definiert zahlreiche Grundbegriffe. Kapitel 3 stellt ein theoretisches Konzept zur eindeutigen Korrektur und Vervollständigung einer Wissensbasis vor. Kapitel 4 beinhalten eine Einführung in das sogenannte Answer Set Programming (ASP), eine Variante der logischen Programmierung. Kapitel 5 beschäftigt sich mit der Lösung von Optimierungsproblemen in ASP. Kapitel 6 beschreibt die Umsetzung der Theorie aus Kapitel 3 in ASP. Kapitel 7 stellt zwei Beispiele vor: Die Buchwelt skizziert die Anwendbarkeit in der Robotik. Die Korrektur inkonsistenter Sudokus demonstriert die Leistungsfähigkeit des Verfahrens.

2

GRUNDLAGEN

Bevor man sich mit der Korrektur inkonsistenter Wissensbasen beschäftigen kann, ist es notwendig, einige Grundbegriffe einzuführen. Was ist eine Wissenbasis? Wann ist eine Wissensbasis konsistent und was bedeutet dies? Was ist eine Schlussfolgerung? Was ist ein Modell?

Die klassische Aussagen- oder Prädikatenlogik gibt auf viele dieser Fragen eine konkrete Antwort. Weil es jedoch noch andere Logiken gibt, ist es erstrebenswert, nach den übergeordneten Strukturen zu suchen. Dieses Kapitel betrachtet eine abstrakte formale Logik. Mithilfe eines Konsequenzoperators und einer Modellzuordnung, die gewisse Eigenschaften erfüllen müssen, ist es bereits möglich, viele übliche Begriffe zu definieren.

Ein wichtiges Ergebnis dieses Kapitels besteht darin, dass es keinen einheitlichen Inkonsistenzbegriff gibt, sondern mindestens drei:

1. Es gibt kein Modell. — modellinduzierte Inkonsistenz
2. Jede Aussage ist ableitbar. — absolute Inkonsistenz
3. Eine Aussage $p \wedge \neg p$ ist ableitbar. — relative Inkonsistenz

Eine Wissensbasis, die kein Modell hat, ist absolut inkonsistent. Eine absolut inkonsistente Wissenbasis ist relativ inkonsistent. Die Umkehrung gilt jedoch nur, wenn der Konsequenzoperator und die Modellzuordnung weitere Voraussetzungen erfüllen.

Dieses Kapitel verwendet Ergebnisse aus [Wal11] und erweitert sie immer dort, wo ein Beweis nicht zitiert wurde. Darüber hinaus sind die relative und die modellinduzierte Konsistenz neu.

2.1 FORMALE LOGIK

Sei A eine endliche oder abzählbar unendliche Menge elementarer Aussagen und F eine endliche Menge von Funktionen. Sei $f*$ die Stelligkeit einer Funktion f. Die folgende Definition beschreibt die Syntax einer abstrakten formalen Logik:

formale Logik

Definition 2.1. Eine Menge L heißt *formale Logik*[1] genau dann, wenn gilt:

1. $A \subseteq L$
2. Für jedes $f \in F$ gilt:
   Wenn $A_1, \ldots, A_{f*} \in L$, dann $f(A_1, \ldots, A_{f*}) \in L$.

Aussage
Wissensbasis

Ein Element $p \in L$ heißt *Aussage*. Eine Teilmenge $P \subseteq L$ heißt *Wissensbasis*.

Die Syntax der klassischen Aussagenlogik ist eine formalen Logik, wie das folgende Beispiel zeigt:

Beispiel 2.1. Sei A eine Signatur und $F = \{w, f, \neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$. Dann ist die erzeugte formale Logik L die Syntax der Aussagenlogik.

Die Syntax der klassischen Prädikatenlogik lässt sich nicht direkt als formale Logik im Sinne der Definition 2.1 beschreiben, weil die Menge A nur Grundterme darstellen kann. Eine formale Logik kann folglich nicht mit Variablen umgehen. Die Wahl des Herbrand-Universums [Schoo, S. 77] als Menge A kann hier jedoch Abhilfe schaffen.

2.2 SCHLUSSFOLGERUNG

Eine Logik zeichnet sich dadurch aus, dass man logische Schlüsse ziehen kann. Eine Schlussfolgerung ist eine Aussage, die aus einer Wissensbasis folgt. Ein Konsequenzoperator Cn definiert alle Schlussfolgerungen $Cn(P)$ zu einer Wissensbasis P. Er muss folgende Bedingungen erfüllen:

Konsequenzoperator

Definition 2.2. Eine Funktion $Cn : 2^L \rightarrow 2^L$ heißt *Konsequenzoperator* genau dann, wenn für alle $P, P' \subseteq L$ gilt[2]:

1. $P \subseteq Cn(P)$ — Reflexivität
2. $Cn(Cn(P)) \subseteq Cn(P)$ — Idempotenz
3. Wenn $P \subseteq P'$, dann ist $Cn(P) \subseteq Cn(P')$. — Monotonie

Schlussfolgerung

Eine Aussage $p \in Cn(P)$ heißt *Schlussfolgerung*. Man schreibt $Cn(p)$ statt $Cn(\{p\})$, falls $p \in L$.

nichtmonotoner Konsequenzoperator

Logiken mit schwacher Negation haben keinen Konsequenzoperator, der monoton ist, weil Monotonie die Revision einer Aussage untersagt. Sie haben jedoch einen *nichtmonotonen Konsequenzoperator* C, bei dem die dritte Bedingung durch die vorsichtige Monotonie ersetzt wurde [Ker10, S. 26]:

1 [Wal11, S. 65] verwendet den Begriff *formale Sprache*. Dem Autor dieser Arbeit ist dieser Begriff zu weit gefasst.

2 2^L ist die Potenzmenge von L. Sie enthält alle Wissensbasen $P \subseteq L$.

3. Wenn $P \subseteq P' \subseteq C(P)$, dann ist $C(P) \subseteq C(P')$.

Die nachfolgenden Definitionen zur Konsistenz und Vollständigkeit sind für klassische und nichtmonotone Konsequenzoperatoren gleichermaßen sinnvoll. Jedoch gelten nicht alle Sätze für nichtmonotone Konsequenzoperatoren.

2.3 KONSISTENZ UND VOLLSTÄNDIGKEIT

Eine verbreitete Definition nennt eine Wissensbasis konsistent, falls sich aus ihr kein Widerspruch, also keine Aussage der Form $p \wedge \neg p$ ableiten lässt [BHS05, S. 2; Ert16, S. 34]:

Definition 2.3. Eine Wissensbasis P heißt *relativ konsistent* genau dann, wenn es kein $p \in A$ gibt, sodass $p \wedge \neg p \in Cn(P)$. Sonst heißt P *relativ inkonsistent*. *relative Konsistenz*

Diese Definition hat den offensichtlichen Nachteil, dass eine Logik eine Negation $\neg$ und eine Konjunktion $\wedge$ enthalten, also $\neg, \wedge \in F$ gelten muss. Folgende Definition kommt ohne diese Voraussetzung aus:

Definition 2.4. Eine Wissensbasis P heißt *absolut konsistent* genau dann, wenn $Cn(P) \neq L$. Sonst heißt P *absolut inkonsistent*. *absolute Konsistenz*

Relative und absolute Konsistenz sind im Allgemeinen nicht äquivalent, wie das folgende Beispiel zeigt:

Beispiel 2.2. Sei $P = \{p \wedge \neg p\}$ und $Cn(P) = P$. Dann ist P relativ inkonsistent und absolut konsistent.

Beweis. Es gilt $p \wedge \neg p \in Cn(P) = \{p \wedge \neg p\} \neq L$. □

Absolute Konsistenz ist jedoch ein Spezialfall der relativen Konsistenz:

Satz 2.1. *Falls P relativ konsistent ist, so ist P auch absolut konsistent.*

Beweis. Sei P relativ konsistent, also $p \wedge \neg p \notin Cn(P)$. Da $p \wedge \neg p \in L$, ist $Cn(P) \neq L$. P ist folglich absolut konsistent. □

Der Konsequenzoperator in Beispiel 2.2 hat ein gravierendes Problem. Er genügt dem Grundsatz *ex falso quodlibet*[3] nicht. Ein Konsequenzoperator genügt diesem Grundsatz, wenn aus einem einzigen Widerspruch jede beliebige Aussage folgt:

Definition 2.5. Ein Konsequenzoperator Cn genügt dem Grundsatz *ex falso quodlibet*, falls für jedes $p \in A$ gilt: $Cn(\{p \wedge \neg p\}) = L$. *ex falso quodlibet*

3 Die deutsche Übersetzung von *ex falso quodlibet* ist *aus Falschem folgt beliebiges*.

Genügt ein Konsequenzoperator diesem Grundsatz, so gilt auch die Umkehrung von Satz 2.1. In diesem Fall sind relative und absolute Konsistenz äquivalent:

Satz 2.2. Cn *genüge dem Grundsatz* ex falso quodlibet *und* P *sei absolut konsistent. Dann ist* P *auch relativ konsistent.*

Beweis. Zum Zwecke des Widerspruchs sei P relativ inkonsistent, also $p \wedge \neg p \in \mathrm{Cn}(P)$. Folglich ist $\{p \wedge \neg p\} \subseteq \mathrm{Cn}(P)$. Aus der Monotonie folgt $\mathrm{Cn}(\{p \wedge \neg p\}) \subseteq \mathrm{Cn}(\mathrm{Cn}(P))$. Aus der Idempotenz folgt $\mathrm{Cn}(\{p \wedge \neg p\}) \subseteq \mathrm{Cn}(P)$. Aus dem Grundsatz des *ex falso quodlibet* folgt $L \subseteq \mathrm{Cn}(P)$. Da $\mathrm{Cn}(P) \subseteq L$ gilt $\mathrm{Cn}(P) = L$. Das steht im Widerspruch zur Voraussetzung, dass P absolut konsistent sei. □

Neben der Konsistenz einer Wissensbasis lässt sich auch ihre Vollständigkeit untersuchen. Eine Wissensbasis heißt genau dann vollständig, wenn jede Aussage entweder aus der Wissensbasis ableitbar oder mit ihr unverträglich ist:

Vollständigkeit

Definition 2.6. P heißt *vollständig* genau dann, wenn für alle $p \in L$ gilt: Wenn $P \cup \{p\}$ konsistent ist, dann ist $p \in \mathrm{Cn}(P)$.

relative und absolute Vollständigkeit

Je nachdem welche Konsistenz verwendet wurde, unterscheidet man zwischen *relativer* und *absoluter Vollständigkeit*.

Es ist wichtig zu verstehen, dass eine Wissenbasis ihre logischen Schlüsse nicht enthalten muss. Tut sie es trotzdem, heißt sie Theorie:

Theorie

Definition 2.7. P ist eine *Theorie* genau dann, wenn $\mathrm{Cn}(P) \subseteq P$.

Eine besonders wünschenswerte Eigenschaft einer Wissensbasis ist die maximale Konsistenz. Eine Wissensbasis, die diese Eigenschaft erfüllt, ist konsistent, vollständig und deduktiv abgeschlossen, also eine Theorie:

absolute maximale Konsistenz

Definition 2.8. P ist *absolut maximal konsistent* genau dann, wenn P absolut konsistent ist und für alle $p \in L$ gilt: Wenn $P \wedge \{p\}$ konsistent ist, dann ist $p \in P$.

Satz 2.3. P *ist absolut maximal konsistent genau dann, wenn* P *eine absolut konsistente und vollständige Theorie ist.*

Beweis. siehe [Wal11, S. 72] □

Der folgende Satz garantiert, dass eine absolut konsistente Wissensbasis eine absolut maximal konsistente Erweiterung hat, wenn man die Wissensbasis und den verwendeten Konsequenzoperator mit endlichen Bausteinen beschreiben kann:

finitär

Definition 2.9. Ein Konsequenzoperator Cn heißt *finitär* genau dann, wenn für alle $P \subseteq L$ gilt: $\mathrm{Cn}(P) = \bigcup_{P' \subseteq P \text{ endlich}} \mathrm{Cn}(P')$.

Definition 2.10. Sei Cn ein Konsequenzoperator. Eine Wissensbasis $P \subseteq L$ heißt *endlich axiomatisierbar* bezüglich Cn genau dann, wenn es ein endliches $P' \subseteq L$ gibt, sodass $Cn(P) = Cn(P')$.

endlich axiomatisierbar

Satz 2.4. *Sei* Cn *ein finitärer Konsequenzoperator,* L *endlich axiomatisierbar und* P *absolut konsistent. Dann existiert ein absolut maximal konsistentes* P' *mit* $P \subseteq P'$.

Lindenbaumscher Ergänzungsatz

Beweis. siehe [Wal11, S. 73] □

2.4 MODELL

In den vorherigen Abschnitten wurden abstrakte Logiken rein syntaktisch betrachtet. Dieser Abschnitt betrachtet abstrakte Semantiken. Sei $B \neq \emptyset$ eine Menge möglicher Belegungen. Eine Modellzuordnung Mod ordnet einer Aussage p alle gültigen Belegungen $Mod(p)$ zu:

Definition 2.11. Eine Funktion $Mod : L \to 2^B$ heißt *Modellzuordnung*. Ein Element $M \in Mod(p)$ heißt *Modell*. Man schreibt $M \in Mod(P)$ genau dann, wenn $M \in Mod(p)$ für alle $p \in L$ gilt und erweitert Mod so zu einer Funktion $Mod : 2^L \to 2^B$.

Modellzuordnung
Modell

Das folgende Beispiel demonstriert, wie man die Semantik der klassischen Aussagenlogik als Modellzuordnung darstellen kann:

Beispiel 2.3. Seien A, F und L wie in Beispiel 2.1 und $B = A$. Wähle für jedes $p \in L$ ein $Mod(p) \subseteq 2^B$ so, dass für jedes $M \in Mod(p)$ gilt: p ist wahr genau dann, wenn alle Variablen $p^+ \in M$ wahr und alle Variablen $p^- \in B \setminus M$ falsch sind.

Jede Modellzuordnung erfüllt die Eigenschaft der Antitonie. Je mehr Aussagen eine Wissensbasis enthält, desto weniger Modelle können ihr zugeordnet werden:

Satz 2.5. *Wenn* $P \subseteq P'$, *dann* $Mod(P') \subseteq Mod(P)$.

Beweis. siehe [Wal11, S. 74] □

Jede Modellzuordnung induziert einen Konsequenzoperator:

Definition 2.12. Es sei Mod eine Modellzuordnung. Dann heißt eine Funktion $Cn_{Mod} : 2^L \to 2^L$ mit $Cn_{Mod}(P) = \{p \mid Mod(P) \subseteq Mod(p)\}$ *modellinduzierter Konsequenzoperator*.

modellinduzierter Konsequenzoperator

Satz 2.6. Cn_{Mod} *ist ein Konsequenzoperator.*

Beweis. siehe [Wal11, S. 75] □

Es gibt einen Zusammenhang zwischen der Konsistenz einer Wissensbasis und der Existenz von Modellen:

Satz 2.7. *Die Wissenbasis* P *sei absolut konsistent bezüglich des modellinduzierten Konsequenzoperators* $\mathrm{Cn}_{\mathrm{Mod}}$. *Dann ist* $\mathrm{Mod}(P) \neq \emptyset$.

Beweis. Zum Zwecke des Widerspruchs sei $\mathrm{Mod}(P) = \emptyset$. Laut Definition 2.12 gilt $\mathrm{Cn}_{\mathrm{Mod}}(P) = \{p \mid \mathrm{Mod}(P) \subseteq \mathrm{Mod}(p)\} = \{p \mid \emptyset \subseteq \mathrm{Mod}(p)\} = L$. Das steht im Widerspruch zur Voraussetzung, dass P absolut konsistent sei. □

Die Umkehrung von Satz 2.7 gilt im Allgemeinen nicht, wie das folgende Beispiel zeigt:

Beispiel 2.4. Für jedes $p \in L$ sei $\mathrm{Mod}(p) = 42$. Dann ist jedes $P \subseteq L$ absolut inkonsistent bezüglich des modellinduzierten Konsequenzoperators $\mathrm{Cn}_{\mathrm{Mod}}$.

Beweis. Laut Definition 2.12 gilt $\mathrm{Cn}_{\mathrm{Mod}}(P) = \{p \mid \mathrm{Mod}(P) \subseteq \mathrm{Mod}(p)\} = \{p \mid \{42\} \subseteq \{42\}\} = L$. □

Die Umkehrung von Satz 2.7 lässt sich jedoch zeigen, wenn man eine zusätzliche Bedingung für die Modellzuordnung Mod hinzufügt:

Satz 2.8. *Es sei* P *eine Wissensbasis mit* $\mathrm{Mod}(P) \neq \emptyset$. *Weiter sei*

$$\bigcap_{p \in L} \mathrm{Mod}(p) = \emptyset.$$

Dann ist P *absolut konsistent bezüglich des modellinduzierten Konsequenzoperators* $\mathrm{Cn}_{\mathrm{Mod}}$.

Beweis. Zum Zwecke des Widerspruchs sei P absolut inkonsistent, also $\mathrm{Cn}(P) = L$. Aus Definition 2.12 folgt nun, dass für jedes $p \in \mathrm{Cn}(P) = L$ gilt: $\mathrm{Mod}(P) \subseteq \mathrm{Mod}(p)$. Daraus folgt $\mathrm{Mod}(P) \subseteq \bigcap_{p \in L} \mathrm{Mod}(p)$, ein Widerspruch zur Voraussetzung. □

Die zusätzliche Bedingung in Satz 2.8 ist sehr häufig erfüllt. Es genügt zum Beispiel, wenn die Logik eine Negation $\neg$ mit der üblichen Semantik besitzt, da $\mathrm{Mod}(p) \cap \mathrm{Mod}(\neg p) = \emptyset$ für jedes $p \in L$ gilt.

Folgendes Korollar fasst die Gemeinsamkeiten der verschiedenen Konsistenzbegriffe zusammen:

Korollar 2.9. *Sei* P *eine Wissensbasis,* $\mathrm{Cn}_{\mathrm{Mod}}$ *erfülle den Grundsatz* ex falso quodlibet *und* $\bigcap_{p \in L} \mathrm{Mod}(p) = \emptyset$. *Dann sind folgende Aussagen äquivalent:*

1. P *ist absolut konsistent bezüglich* $\mathrm{Cn}_{\mathrm{Mod}}$.
2. P *ist relativ konsistent bezüglich* $\mathrm{Cn}_{\mathrm{Mod}}$.
3. $\mathrm{Mod}(P) \neq \emptyset$

Beweis. folgt aus Satz 2.1, 2.2, 2.7 und 2.8 □

3

KONZEPT

Dieses Kapitel beschreibt ein Verfahren, um aus einer beliebigen Wissensbasis eine konsistente, vollständige und eindeutige Wissensbasis herzuleiten. Das Verfahren besteht aus drei Schritten:

1. Bei der Korrektur werden Widersprüche durch möglichst wenige Änderungen aufgelöst.
2. Bei der Vervollständigung werden Wissenslücken konsistent geschlossen.
3. Da Korrektur und Vervollständigung im Allgemeinen kein eindeutiges Ergebnis haben, wird auf Basis weiterer Informationen eine Auswahl getroffen.

Aus Kapitel 2 ist bekannt, dass es keine einheitliche Definition der Konsistenz gibt. Dieses Kapitel unterscheidet relative, absolute und modellinduzierte Konsistenz trotzdem nicht. Der Leser kann eine Konsistenz seiner Wahl nutzen, sofern eine Definition oder ein Satz keine Einschränkung vorsieht[1].

Sofern nicht explizit beschrieben, ist der Konsequenzoperator in diesem Kapitel nicht zwingend monoton.

Dieses Kapitel wurde von [Res64] inspiriert.

3.1 KORREKTUR

Eine Korrektur ist eine konsistente Wissensbasis P^*, die durch möglichst wenige Veränderungen aus einer Wissensbasis P hervorgeht:

Definition 3.1. Sei P eine Wissensbasis. Eine Wissensbasis P^* heißt *Korrektur* von P genau dann, wenn gilt: *Korrektur*

1. P^* ist konsistent.
2. $\mathrm{Rev}(P^*) := P \setminus P^* \cup P^* \setminus P$ ist minimal. Das heißt, für jede konsistente Wissenbasis $\check{P}^*$ gilt: Wenn $\mathrm{Rev}(\check{P}^*) \subseteq \mathrm{Rev}(P^*)$, dann $\mathrm{Rev}(P^*) \subseteq \mathrm{Rev}(\check{P}^*)$.

Eine Korrektur ist nur aussagekräftig, wenn ein Widerspruch lokal beschränkt auftritt und sich tatsächlich durch wenige Änderungen beheben lässt.

1 Der Leser kann natürlich auch die Bedingungen aus Korollar 2.9 erfüllen und alle Definitionen verwenden.

Das folgende Beispiel ist eine Adaption der bekannten *Nixon-Raute* aus [RN10, S. 459]. Es zeigt, wie eine Korrektur funktioniert und dass sie im Allgemeinen nicht eindeutig ist:

Nixon-Raute

Beispiel 3.1. Sei L die erzeugte formale Logik zu

$$A = \{\, \text{pacifist}(x), \text{quaker}(x), \text{republican}(x) \mid x \in \{\text{bush}, \text{nixon}\} \,\} \quad \text{und}$$
$$F = \{\, w, f, \neg, \wedge, \vee, \rightarrow, \leftrightarrow \,\}.$$

Weiter seien Cn und Mod wie in der Aussagenlogik und

$$R = \{ \quad \text{quaker}(x) \rightarrow \text{pacifist}(x), \quad \text{republican}(x) \rightarrow \neg\,\text{pacifist}(x) \quad \mid x \in \{\text{bush}, \text{nixon}\} \,\}.$$

Die Wissensbasis $P = \{\text{quaker}(\text{nixon}), \text{republican}(\text{nixon})\} \cup R$ ist inkonsistent. Die Wissensbasen $P_1^* = \{\text{quaker}(\text{nixon})\} \cup R$ und $P_2^* = \{\text{republican}(\text{nixon})\} \cup R$ sind Korrekturen von P. Die Wissensbasis $P_3^* = \{\text{quaker}(\text{nixon}), \text{republican}(\text{bush})\} \cup R$ ist keine Korrektur von P.

Eine Korrektur P^* von P unterteilt alle beteiligten Aussagen in drei disjunkte Mengen: Alle Aussagen, die bei der Korrektur erhalten wurden, sind in $P \cap P^*$ enthalten. Alle Aussagen, die bei der Korrektur entfernt wurden, sind in $P \setminus P^*$ enthalten. Alle Aussagen, die bei der Korrektur hinzugefügt wurden, sind in $P^* \setminus P$ enthalten. Folgende Definition nutzt diesen Zusammenhang und erweitert ihn auf beliebige Wissensbasen:

Glaube und Zweifel

Definition 3.2. Sei P und P' Wissensbasen. Eine Aussage $p \in L$ wird von P' genau dann

1. *geglaubt*, wenn $p \in P \cap P'$,
2. *bezweifelt*, wenn $p \in P \setminus P'$ und
3. *ergänzt*, wenn $p \in P' \setminus P$.

Eine Korrektur minimiert die Menge der revidierten Aussagen $\text{Rev}(P^*)$. Das sind genau alle Aussagen, die bezweifelt oder ergänzt werden. Zugleich maximiert eine Korrektur die Menge $P \cap P^*$ der Aussagen, die geglaubt werden, wie der nächste Satz zeigt:

Lemma 3.1. *Es gilt*[2]*:*

$$A \setminus B_1 \subseteq A \setminus B_2 \Leftrightarrow A \cap B_2 \subseteq A \cap B_1$$

2 Das Metasymbol $\Leftrightarrow$ steht für „genau dann, wenn“, $\Rightarrow$ steht für „hat zur Folge, dass“.

Beweis.

$$
\begin{aligned}
& A \setminus B_1 \subseteq A \setminus B_2 \\
\Leftrightarrow\ & \forall x : x \in A \setminus B_1 \rightarrow x \in A \setminus B_2 \\
\Leftrightarrow\ & \forall x : x \in A \wedge x \notin B_1 \rightarrow x \in A \wedge x \notin B_2 \\
\Leftrightarrow\ & \forall x : \neg(x \in A \wedge x \notin B_1) \vee x \in A \wedge x \notin B_2 \\
\Leftrightarrow\ & \forall x : x \notin A \vee x \in B_1 \vee x \in A \wedge x \notin B_2 \\
\Leftrightarrow\ & \forall x : x \in B_1 \vee x \notin A \vee x \in A \wedge x \notin B_2 \\
\Leftrightarrow\ & \forall x : x \in B_1 \vee (x \notin A \vee x \in A) \wedge (x \notin A \vee x \notin B_2) \\
\Leftrightarrow\ & \forall x : x \in B_1 \vee x \notin A \vee x \notin B_2 \\
\Leftrightarrow\ & \forall x : x \notin B_2 \vee x \notin A \vee x \in B_1 \\
\Leftrightarrow\ & \forall x : x \notin B_2 \vee (x \notin A \vee x \in A) \wedge (x \notin A \vee x \in B_1) \\
\Leftrightarrow\ & \forall x : x \notin B_2 \vee x \notin A \vee x \in A \wedge x \in B_1 \\
\Leftrightarrow\ & \forall x : x \notin A \vee x \notin B_2 \vee x \in A \wedge x \in B_1 \\
\Leftrightarrow\ & \forall x : \neg(x \in A \wedge x \in B_2) \vee x \in A \wedge x \in B_1 \\
\Leftrightarrow\ & \forall x : x \in A \wedge x \in B_2 \rightarrow x \in A \wedge x \in B_1 \\
\Leftrightarrow\ & \forall x : x \in A \cap B_2 \rightarrow x \in A \cap B_1 \\
\Leftrightarrow\ & A \cap B_2 \subseteq A \cap B_1
\end{aligned}
$$

□

Satz von Glaube und Zweifel

Satz 3.2. *Sei* P *eine Wissensbasis und* P^* *eine Korrektur von* P. *Dann ist* $P \cap P^*$ *maximal. Das heißt, für jede konsistente Wissensbasis* $\hat{P}^*$ *gilt: Wenn* $P \cap P^* \subseteq P \cap \hat{P}^*$, *dann* $P \cap \hat{P}^* \subseteq P \cap P^*$.

Beweis. Sei P eine Wissensbasis und P^* eine Korrektur von P. Laut Definition 3.1 ist dann $P \setminus P^* \cup P^* \setminus P$ minimal. Da $P \setminus P^*$ und $P^* \setminus P$ disjunkt sind, ist auch $P \setminus P^*$ minimal. Das heißt, für jede konsistente Wissensbasis $\check{P}^*$ gilt: Wenn $P \setminus \check{P}^* \subseteq P \setminus P^*$, dann $P \setminus P^* \subseteq P \setminus \check{P}^*$. Mit Lemma 3.1 folgt daraus: Wenn $P \cap P^* \subseteq P \cap \check{P}^*$, dann $P \cap \check{P}^* \subseteq P \cap P^*$. Setze $\hat{P}^* = \check{P}^*$ und $P \cap P^*$ ist maximal. □

Wenn eine Wissensbasis konsistent ist, sind keine Änderungen nötig, um Widersprüche zu korrigieren. In diesem Fall gibt es genau eine Korrektur und sie entspricht der ursprünglichen Wissenbasis:

Korrektur konsistenter Wissensbasen

Satz 3.3. *Sei* P *konsistent. Dann ist* $P^* = P$ *die einzige Korrektur von* P.

Beweis. $P^* = P$ ist laut Voraussetzung konsistent und $\mathrm{Rev}(P^*) = \mathrm{Rev}(P) = P \setminus P \cup P \setminus P = \emptyset$. Sei $\check{P}^*$ eine konsistente Wissensbasis und $\mathrm{Rev}(\check{P}^*) \subseteq \mathrm{Rev}(P^*)$, also $\mathrm{Rev}(\check{P}^*) = \emptyset$. Dann ist $\check{P}^* = P$. Folglich ist $P^* = P$ die einzige Wissensbasis, welche die Voraussetzungen aus Definition 3.1 erfüllt. □

Eine Korrektur fügt einer Wissensbasis keine Aussagen hinzu, wenn der verwendete Konsequenzoperator monoton ist. In diesem Fall ist die Korrektur eine Teilmenge der ursprünglichen Wissensbasis:

Lemma 3.4. *Sei* P *inkonsistent und* $P \subseteq P'$*. Der Konsequenzoperator, der dem Konsistenzbegriff zugrunde liegt, sei monoton. Dann ist* P' *inkonsistent.*

Beweis. Je nach Definition der Inkonsistenz folgt das Lemma entweder aus der Monotonie des Konsequenzoperators oder aus der Antitonie der Modellzuordnung. □

Korrektur bei monotonem Konsequenzoperator

Satz 3.5. *Sei* P *eine Wissensbasis und* P^* *eine Korrektur von* P*. Der Konsequenzoperator, der dem Konsistenzbegriff zugrunde liegt, sei monoton. Dann ist* $P^* \setminus P = \emptyset$ *und damit* $P^* \subseteq P$*.*

Beweis. Zum Zwecke des Widerspruchs sei $P^* \setminus P \neq \emptyset$ und $p \in P^* \setminus P$. Laut Lemma 3.4 ist $P^* \setminus \{p\}$ konsistent, denn wäre $P^* \setminus \{p\}$ inkonsistent, so wäre P^* im Widerspruch zu Definition 3.1 inkonsistent. Es gilt: $p \in \mathrm{Rev}(P^* \setminus \{p\}) = P \setminus (P^* \setminus \{p\}) \cup P^* \setminus \{p\} \setminus P$, da $p \notin P$. Es gilt weiter: $p \in \mathrm{Rev}(P^*) = P \setminus P^* \cup P^* \setminus P$. Mit $\check{P}^* = P^* \setminus \{p\}$ steht dies im Widerspruch zur Minimalität von P^*. □

Aus Beispiel 3.1 ist bekannt, dass eine Korrektur nicht zwangsläufig eindeutig ist. Das folgende Lemma beschreibt einen Fall, in dem keine Korrektur existiert:

Lemma 3.6. *Sei* $\emptyset$ *inkonsistent. Der Konsequenzoperator, der dem Konsistenzbegriff zugrunde liegt, sei monoton. Dann ist jede Wissensbasis* $P \subseteq L$ *inkonsistent. Insbesondere gibt es keine Korrektur* P^* *zu* P*.*

Beweis. folgt aus Lemma 3.4 □

Dieser Fall ist für die Praxis uninteressant, denn eine Korrektur ergibt wenig Sinn, wenn es überhaupt keine konsistente Wissenbasis gibt. Ist $\emptyset$ konsistent, so hat jede Wissensbasis mit abzählbarer Potenzmenge[3] eine Korrektur:

Existenz einer Korrektur

Satz 3.7. *Sei* P *eine Wissensbasis und* 2^P *abzählbar. Der Konsequenzoperator, der dem Konsistenzbegriff zugrunde liegt, sei monoton. Es gibt eine Korrektur* P^* *zu* P *genau dann, wenn* $\emptyset$ *konsistent ist.*

Beweis. „⇒“: Sei P eine Wissensbasis und P^* eine Korrektur von P. Dann ist P^* konsistent. Da $\emptyset \subseteq P^*$, ist $\emptyset$ nach Lemma 3.4 konsistent, denn wäre $\emptyset$ inkonsistent, so wäre auch P^* inkonsistent.
„⇐“: Sei P eine Wissensbasis, 2^P abzählbar und $\emptyset$ konsistent. Laut

3 Die Frage, ob eine Wissensbasis mit überabzählbarer Potenzmenge im Allgemeinen eine Korrektur hat, muss in dieser Arbeit offen bleiben. Der Autor hat trotz intensiver Suche keine Wissensbasis gefunden, die keine Korrektur besitzt. Er konnte jedoch auch keine allgemeinere Version des Satzes beweisen.

Satz 3.5 gilt $P^* \subseteq P$ für jede Korrektur P^* von P. Falls es eine Korrektur gibt, so ist sie folglich in 2^P enthalten. Da 2^P abzählbar ist, gibt es eine bijektive Folge $(P_i)_{i\in\mathbb{N}}$ auf 2^P. Durch Umordnung lässt sich garantieren, dass $P_j \subseteq P_i \Rightarrow i \leqslant j$ für jedes $i, j \in \mathbb{N}$ gilt. Dann folgt $P \cap P_j \subseteq P \cap P_i \Rightarrow i \leqslant j$ und mit Lemma 3.1 $P \setminus P_i \subseteq P \setminus P_j \Rightarrow i \leqslant j$. Da $P_i \setminus P = P_j \setminus P = \emptyset$ gilt $\mathrm{Rev}(P_i) \subseteq \mathrm{Rev}(P_j) \Rightarrow i \leqslant j$. P^* sei nun das erste Glied der Folge (P_i), das konsistent ist. Da $\emptyset \in 2^P$ konsistent und (P_i) surjektiv ist, gibt es ein solches P^*. $\mathrm{Rev}(P^*)$ ist minimal, denn gäbe es ein konsistentes $\check{P}^*$ mit $\mathrm{Rev}(\check{P}^*) \subsetneq \mathrm{Rev}\, P^*$, so stände $\check{P}^*$ in der Folge (P_i) vor P^* und P^* wäre nicht das erste konsistente Glied der Folge (P_i). P^* ist also eine Korrektur von P. □

Der Satz garantiert insbesondere, dass jede endliche Wissensbasis eine Korrektur hat, wenn $\emptyset$ konsistent ist. Der Beweis enthält einen Algorithmus, um eine Korrektur zu berechnen.

3.2 VERVOLLSTÄNDIGUNG

Eine Wissensbasis ist im Allgemeinen nicht vollständig. Sie enthält möglicherweise Wissenslücken. Das gilt auch für Korrekturen. Es ist daher sinnvoll, sich Gedanken zu machen, wie man Wissenslücken schließen kann, ohne Widersprüche zu erzeugen. Die einfachste Möglichkeit, einige Wissenslücken einer Wissensbasis P zu schließen, besteht darin, ihre logischen Schlussfolgerungen $\mathrm{Cn}(P)$ zu betrachten:

Definition 3.3. Sei P eine konsistente Wissenbasis. Eine Wissensbasis $P^\uparrow = \mathrm{Cn}(P)$ heißt *partielle Vervollständigung* von P.

partielle Vervollständigung

Offensichtlich existiert eine partielle Vervollständigung immer und sie ist eindeutig. Sie ist jedoch keineswegs vollständig, wie das folgende Beispiel zeigt:

Beispiel 3.2. Seien A, F, L und R wie in Beispiel 3.1. Weiter sei $P = \{\mathrm{quaker}(\mathrm{nixon})\} \cup R$ und $P^\uparrow$ die partielle Vervollständigung von P. Nun ist $P^\uparrow \cup \{\mathrm{republican}(\mathrm{bush})\}$ konsistent, aber $\mathrm{republican}(\mathrm{bush}) \notin P^\uparrow$. Laut Definition 2.6 ist $P^\uparrow$ daher nicht vollständig.

Eine partielle Vervollständigung zeichnet sich dadurch aus, dass sie keine Aussage ergänzt, die sich aufgrund des vorhandenen Wissens in Zweifel ziehen lässt. Eine totale Vervollständigung integriert darüber hinaus Spekulationen, ist jedoch vollständig:

Definition 3.4. Sei P eine absolut konsistente Wissenbasis. Eine Wissenbasis $P^{\bar{\uparrow}}$ heißt *totale Vervollständigung* von P genau dann, wenn gilt:

totale Vervollständigung

1. $\mathrm{Cn}(P) \subseteq P^{\bar{\uparrow}}$
2. $P^{\bar{\uparrow}}$ ist absolut maximal konsistent.

Satz 3.8. *Eine totale Vervollständigung* $P^{\tilde{\uparrow}}$ *ist vollständig.*

Beweis. folgt aus Satz 2.3 □

Der Lindenbaumsche Ergänzungssatz 2.4 garantiert die Existenz einer totalen Vervollständigung. Sie ist jedoch im Allgemeinen nicht eindeutig, wie das folgende Beispiel zeigt:

Beispiel 3.3. Seien A, F, L und R wie in Beispiel 3.1 und P wie in Beispiel 3.2. Dann gibt es zu P mindestens zwei totale Vervollständigungen P_1 und P_2 mit $\text{republican}(\text{bush}) \in P_1$ und $\neg\,\text{republican}(\text{bush}) \in P_2$.

Stellt man sich eine Wissensbasis P als das verfügbare Wissen über eine zugrundeliegende Welt vor, so kann man sagen: Die partielle Vervollständigung $P^{\uparrow}$ beschreibt, wie die Welt sein muss. Eine totale Vervollständigung $P^{\tilde{\uparrow}}$ beschreibt, wie die Welt sein kann. Da $P^{\uparrow} \subseteq P^{\tilde{\uparrow}}$ ist auch die Wissensbasis $P^{\tilde{\uparrow}} \setminus P^{\uparrow}$ interessant, denn sie enthält alle spekulativen Aussagen, die $P^{\tilde{\uparrow}}$ zu P ergänzt, ohne dass es dafür eine besondere Rechtfertigung gibt. Diese Aussagen könnte man, falls die Möglichkeit dazu besteht, überprüfen, um P zu verbessern.

3.3 AUSWAHL

Die Beispiele 3.1 und 3.3 zeigen, dass weder eine Korrektur noch eine totale Vervollständigung im Allgemeinen eindeutig ist. Das ist ein Problem, wenn man auf der Suche nach der einen Wahrheit ist:

„The very most that logic can tell us is to show that a revision is necessary; it cannot give us an answer to the question which revision ought to be made." [Res64, S. 18]

Das Problem lässt sich verallgemeinern, wenn man eine endliche Menge $\mathbb{P} \subseteq 2^L$ betrachtet. $\mathbb{P}$ enthält zum Beispiel alle Korrekturen P^* oder totale Vervollständigungen $P^{\tilde{\uparrow}}$ einer Wissensbasis P. Das Problem besteht nun darin, eine Wissensbasis aus $\mathbb{P}$ auszuwählen.

Um eine Auswahl treffen zu können, sind weitere Informationen nötig. Man könnte zum Beispiel eine Wissensbasis auswählen, welche die meisten, neuesten oder wahrscheinlichsten Informationen enthält oder möglichst geringe Kosten verursacht. Im Allgemeinen wählt man ein Element, das maximal bezüglich einer Ordnung $<$ auf $\mathbb{P}$ ist:

partielle Ordnung

Definition 3.5. Sei A eine Menge. Eine Relation $< \subseteq A \times A$ heißt *partielle Ordnung* auf A genau dann, wenn für alle $a, b, c \in A$ gilt:

1. $a \not< a$ — Irreflexivität
2. Aus $a < b$ und $b < c$ folgt $a < c$. — Transitivität

Der Ausdruck $a < b$ ist dabei eine Kurzschreibweise für $(a, b) \in <$. $a \not< b$ steht für $(a, b) \notin <$.

Definition 3.6. Eine partielle Ordnung $<$ heißt *totale Ordnung* genau dann, wenn für alle $a, b \in A$ gilt: *totale Ordnung*

3. $a < b$ oder $a = b$ oder $b < a$ Totalität

Definition 3.7. Sei $<$ eine partielle Ordnung auf A. Ein Element $a \in A$ heißt *maximal*, wenn es kein $b \in A$ mit $a < b$ gibt. *maximal*

Ist $\mathbb{P}$ endlich[4] und $<$ eine partielle Ordnung auf $\mathbb{P}$, so garantiert das Lemma von Zorn [Zie17, S. 76] die Existenz von mindestens einem maximalen Element in $\mathbb{P}$. Die Menge der maximalen Elemente von $\mathbb{P}$ ist eine Teilmenge von $\mathbb{P}$. Demzufolge verringert eine partielle Ordnung die Anzahl der Möglichkeiten und vereinfacht so die Auswahl.

Ist $\mathbb{P}$ endlich und $<$ eine totale Ordnung auf $\mathbb{P}$, so existiert genau ein maximales Element in $\mathbb{P}$. Eine totale Ordnung entscheidet also die Auswahl.

Das folgende Beispiel beschreibt eine Möglichkeit, um aus einer totalen Ordnung auf einer Menge M eine totale Ordnung auf 2^M herzuleiten. Folglich genügt bereits eine Ordnung auf L, um ein Element aus $\mathbb{P} \subseteq 2^L$ auszuwählen:

Beispiel 3.4. Sei M eine Menge und $\prec$ eine totale Ordnung auf M. Für jedes $A, B \in 2^M$ sei $A < B$ genau dann, wenn ein $b \in B \setminus A$ existiert, sodass für alle $a \in A \setminus B$ $a \prec b$ gilt. Die Relation $<$ ist eine totale Ordnung auf 2^M. *lexikografische Ordnung*

In Kapitel 5 werden weitere Möglichkeiten beschrieben, Ordnungen zu erzeugen.

Das folgende Beispiel zeigt, dass eine Korrektur nach Definition 3.1 den maximalen Elementen aus der Menge der konsistenten Wissensbasen entspricht, wenn man eine bestimmte partielle Ordnung verwendet:

Beispiel 3.5. Sei $P \subseteq L$ eine Wissensbasis und $\mathbb{P} \subseteq 2^L$ die Menge der konsistenten Wissenbasen. Zu jedem $P_1, P_2 \in \mathbb{P}$ sei $P_1 < P_2$ genau dann, wenn $\mathrm{Rev}(P_2) \subsetneq \mathrm{Rev}(P_1)$. Die Relation $<$ ist eine partielle Ordnung auf $\mathbb{P}$. Ein $P^* \in \mathbb{P}$ ist genau dann eine Korrektur von P, wenn P^* maximal ist.

4 Das Lemma von Zorn gilt eigentlich für deutlich allgemeinere Mengen. Das führt an dieser Stelle jedoch zu weit.

4

ANSWER SET PROGRAMMING

Um eine Wissensbasis im Computer darzustellen, Schlussfolgerungen zu berechnen und die Konsistenz der Wissensbasis zu prüfen, ist eine logische Programmiersprache besonders geeignet. Dieses Kapitel stellt die logische Programmiersprache Answer Set Programming (ASP) vor.

ASP ist keine Programmiersprache im traditionellen Sinn. Um ein Problem zu lösen, ist es nicht notwendig, eine Lösungvorschrift zu programmieren, die von einem Computer ausgeführt werden kann. Stattdessen werden nur die Regeln des Problems modelliert. Sie können anschließend von einem Computerprogramm aufgelöst werden. Es berechnet ein Ergebnis bestehend aus stabilen Modellen[1] so, dass jede Regel erfüllt ist. Diese Art der Programmierung heißt *deklarativ* und wird von ASP strikt umgesetzt.

deklarativ

In diesem Kapitel werden die Ergebnisse aus [Geb+12] zusammengefasst, die für diese Arbeit relevant sind. Ein kurzer Überblick über ASP im Allgemeinen ist in [BET11] zu finden.

4.1 SYNTAX

Ein Programm besteht in ASP aus Regeln. Eine Regel besteht aus Literalen und Atomen.

Sei A eine endliche Menge und $L = A \cup \{\sim a \mid a \in A\}$. Ein Element $a \in A$ heißt *Atom* und beschreibt eine elementare Aussage, die entweder wahr oder falsch ist. Das Symbol $\sim$ steht für die schwache Negation[2]. Ein Element $l \in L$ heißt *Literal*. Es ist entweder ein Atom a oder die schwache Negation $\sim a$ eines Atoms a.

Atom

Literal

Der Kopf einer Regel ist ein Atom, der Rumpf besteht aus Literalen:

Definition 4.1. Sei $0 \leqslant m \leqslant n$ und $a_i \in A$ für jedes $0 \leqslant i \leqslant n$. Ein Ausdruck der Form

$$a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n$$

heißt *Regel*.

Regel

Ist r eine Regel, so bezeichnet $\text{head}(r) = a_0$ den *Kopf* von r und $\text{body}(r) = \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}$ den *Rumpf* von r.

Kopf

Rumpf

1 Die stabilen Modelle heißen auch *Antwortmengen*. Sie geben der *Antwortmengenprogrammierung* (engl. *Answer Set Programming*) ihren Namen.

2 An dieser Stelle genügt es, die schwache Negation als Symbol ohne Bedeutung zu betrachten. Die intuitive Bedeutung, dass $\sim a$ genau dann wahr ist, wenn es keinen Beweis für a gibt, ergibt sich aus Definition 4.3. Der Unterschied zwischen starker und schwacher Negation wird in Beispiel 4.7 genauer betrachtet.

Die intuitive Lesart einer Regel r ist, dass ihr Kopf $\text{head}(r)$ genau dann wahr sein muss, wenn ihr Rumpf $\text{body}(r)$ gilt. Der Rumpf $\text{body}(r)$ gilt, wenn $a_1, \ldots, a_m$ beweisbar wahr sind und $a_{m+1}, \ldots, a_n$ falsch sein können, es also keinen Beweis gibt, dass $a_{m+1}, \ldots, a_n$ wahr sind.

Fakt Falls $\text{body}(r) = \emptyset$, ist r ein *Fakt* und man schreibt statt $a_0 \leftarrow$ auch a_0. Ein Fakt muss stets wahr sein.

Programm **Definition 4.2.** Ein *Programm* P ist eine endliche Menge von Regeln.

Zu einer Menge $X \subseteq L$ von Literalen sei $X^+ = \{a \in A \mid a \in X\}$ und $X^- = \{a \in A \mid \sim a \in X\}$. Der Rumpf $\text{body}(r)$ einer Regel r besteht also aus einem positiven Teil $\text{body}(r)^+ = \{a_1, \ldots, a_m\}$ und einem negativen Teil $\text{body}(r)^- = \{a_{m+1}, \ldots, a_n\}$.

positiv Eine Regel r heißt *positiv*, wenn $\text{body}(r)^- = \emptyset$. Ein Programm P heißt *positiv*, wenn jede Regel $r \in P$ positiv ist.

4.2 SEMANTIK

Das Ergebnis eines Programms ist in ASP eine Menge stabiler Modelle. Die Idee, stabile Modelle zu berechnen, stammt ursprünglich aus [GL88]. Sie wird im Folgenden erklärt.

Ein Modell ist, wie in Beispiel 2.3, eine Teilmenge der Atome. Sie enthält genau die Atome, die als wahr angenommen werden. Ein Modell muss in ASP gültig sein, also jede Regel des Programms erfüllen:

Modell **Definition 4.3.** Sei P ein Programm über die Atome aus A. Eine Menge $M \subseteq A$ heißt gültiges *Modell* von P genau dann, wenn für jede Regel $r \in P$ gilt:

$$\text{body}(r)^+ \subseteq M \wedge \text{body}(r)^- \cap M = \emptyset \rightarrow \text{head}(r) \in M$$

Ist P positiv, so existiert ein eindeutiges, $\subseteq$-minimales Modell von P. Dieses Modell enthält genau die Atome, die aufgrund der Anwendung der Regeln in P wahr sein müssen. Es wird daher in Anlehnung an Definition 2.12 mit $\text{Cn}(P)$ bezeichnet[3] und ist das eindeutige Ergebnis des Programms P.

Ist P nicht positiv, so existiert im Allgemeinen kein eindeutiges, $\subseteq$-minimales Modell von P. Wählt man jedoch eine Menge $M \subseteq A$, so

3 Es gibt wesentliche Unterschiede zwischen der hier verwendeten Definition und dem Konsequenzoperator aus Kapitel 2. Dazu gehört, dass $\text{Cn}(P)$ ein Modell ist und kein Programm. Betrachtet man ein Modell jedoch als ein Programm, das nur aus Fakten besteht, so erfüllt Cn die Eigenschaften eines Konsequenzoperators sinngemäß:

1. Besteht P nur aus Fakten, so ist Cn reflexiv.
2. Cn ist idempotent.
3. Ist P positiv, so ist Cn monoton.

M	P_1^M	$Cn(P_1^M)$
$\emptyset$	$\{a \leftarrow a, b \leftarrow\}$	$\{b\}$
$\{a\}$	$\{a \leftarrow a\}$	$\emptyset$
$\{b\}$	$\{a \leftarrow a, b \leftarrow\}$	$\{b\}$
$\{a, b\}$	$\{a \leftarrow a\}$	$\emptyset$

Tabelle 4.1: $P_1 = \{a \leftarrow a, b \leftarrow \sim a\}$ hat ein stabiles Modell $\{b\}$

M	P_2^M	$Cn(P_2^M)$
$\emptyset$	$\{a \leftarrow, b \leftarrow\}$	$\{a, b\}$
$\{a\}$	$\{a \leftarrow\}$	$\{a\}$
$\{b\}$	$\{b \leftarrow\}$	$\{b\}$
$\{a, b\}$	$\emptyset$	$\emptyset$

Tabelle 4.2: $P_2 = \{a \leftarrow \sim b, b \leftarrow \sim a\}$ hat zwei stabile Modelle $\{a\}$ und $\{b\}$

kann aus jedem Programm P ein positives Programm P^M hergeleitet werden. Dazu entfernt man alle Regeln in P, bei denen ein Atom aus M im negativen Teil des Rumpfes vorkommt. Danach reduziert man den Rumpf jeder Regel auf seinen positiven Teil:

Definition 4.4. Sei P ein Programm über die Atome aus A und $M \subseteq A$. Dann heißt

$$P^M = \{head(r) \leftarrow body(r)^+ \mid r \in P \wedge body(r)^- \cap M = \emptyset\}$$

Reduktion von P auf M. *Reduktion*

Mithilfe der Reduktion lassen sich nun stabile Modelle definieren. Ein Modell ist stabil, wenn es sich bei der Anwendung der Regeln aus P nicht verändert:

Definition 4.5. Sei P ein Programm über die Atome aus A. Eine Menge $M \subseteq A$ heißt *stabiles Modell* von P genau dann, wenn $Cn(P^M) = M$. *stabiles Modell*

Ein Programm kann kein, ein oder mehrere stabile Modelle haben, wie die Tabellen 4.1, 4.2 und 4.3 zeigen. Man erkennt ein stabiles Modell in der Tabelle gemäß Definition 4.5 daran, dass ein Eintrag in den Spalten M und $Cn(P^M)$ übereinstimmt.

M	P_3^M	$Cn(P_3^M)$
$\emptyset$	$\{a \leftarrow\}$	$\{a\}$
$\{a\}$	$\emptyset$	$\emptyset$

Tabelle 4.3: $P_3 = \{a \leftarrow \sim a\}$ hat kein stabiles Modell

4.3 SPRACHERWEITERUNGEN

Um die Anwendungsmöglichkeiten zu vergrößern, definiert ASP zahlreiche Spracherweiterungen. Man unterscheidet echte von unechten Spracherweiterungen. Eine echte Spracherweiterung erweitert die Syntax und die Semantik von ASP. Sie vergrößert die Menge der Probleme, die modelliert werden können. Eine unechte Spracherweiterung erweitert nur die Syntax von ASP. Sie vereinfacht es, Probleme zu modellieren. Die Abschnitte 4.3.1 bis 4.3.7 stellen unechte, der Abschnitt 4.3.8 eine echte Spracherweiterung vor.

4.3.1 *Variablen erster Ordnung*

Die bisherigen Definitionen erlauben es nicht, eine Regel zu verwenden, die Variablen enthält. Im Folgenden wird eine Konstruktion vorgestellt, die ASP um Prädikate und Variablen erster Ordnung erweitert. Eine Regel wird dabei als ein Schema verstanden, das seine Grundinstanzen repräsentiert. Die Konstruktion wurde in ähnlicher Form bereits in Beispiel 3.1 verwendet und hat Parallelen zur Herbrand-Theorie [Schoo, S. 77–84].

Definition 4.6. Sei $n \in \mathbb{N}$. Ein Ausdruck der Form

$$p(V_1, \ldots, V_n)$$

Prädikat heißt n-stelliges *Prädikat*.

Variable p ist der *Name* des Prädikats. Ein V_i heißt *Variable erster Ordnung*.

Sei P eine endliche Menge von Prädikaten. Jedes Prädikat $p \in P$ *Grundinstanz* beschreibt eine Menge von *Grundinstanzen* $\mathrm{grd}(p) \subseteq A$. Ein Regelschema besteht aus Prädikaten. Es beschreibt alle Regeln, bei denen die Prädikate durch ihre Grundinstanzen ersetzt wurden:

Definition 4.7. Sei $0 \leqslant m \leqslant n$ und $p_i \in P$ für jedes $0 \leqslant i \leqslant n$. Ein Ausdruck r der Form

$$p_0 \leftarrow p_1, \ldots, p_m, \sim p_{m+1}, \ldots, \sim p_n$$

Regelschema heißt *Regelschema*. Er beschreibt die Menge

$$\mathrm{grd}(r) = \{a_0 \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \mid a_i \in \mathrm{grd}(p_i)\}$$

der *Grundinstanzen* von r.

Ein Programm darf ab jetzt auch Regelschemata enthalten. Ein Programm, das nur aus gewöhnlichen Regeln besteht, also keine Regelschemata enthält, ist eine *Grundinstanz*. Jedes Programm hat eine Grundinstanz:

Definition 4.8. Sei P ein Programm. Das Programm

$$\mathrm{grd}(P) = \bigcup_{r \in P} \mathrm{grd}(r).$$

heißt *Grundinstanz* von P.

Die stabilen Modelle eines Programms P sind per Definition die stabilen Modelle seiner Grundinstanz $\mathrm{grd}(P)$.

In der Praxis ist es in der Regel möglich, ein Programm zu lösen, ohne die Grundinstanzen aller Prädikate zu kennen. Dabei werden die fehlenden Informationen hergeleitet. Das dazu nötige Verfahren wird in [Geb+12, S. 52–58] beschrieben und in den folgenden Beispielen eingesetzt:

Beispiel 4.1. Das Programm *Schrödingers Katze*

$$P = \left\{ \begin{array}{c} \mathrm{cat}(\mathrm{schroed}) \\ \mathrm{dead}(X) \leftarrow \mathrm{cat}(X), {\sim}\,\mathrm{alive}(X) \\ \mathrm{alive}(X) \leftarrow \mathrm{cat}(X), {\sim}\,\mathrm{dead}(X) \end{array} \right\}$$

hat die Grundinstanz

$$\mathrm{grd}(P) = \left\{ \begin{array}{c} \mathrm{cat}(\mathrm{schroed}) \\ \mathrm{dead}(\mathrm{schroed}) \leftarrow \mathrm{cat}(\mathrm{schroed}), {\sim}\,\mathrm{alive}(\mathrm{schroed}) \\ \mathrm{alive}(\mathrm{schroed}) \leftarrow \mathrm{cat}(\mathrm{schroed}), {\sim}\,\mathrm{dead}(\mathrm{schroed}) \end{array} \right\}$$

und die stabilen Modelle $\{\mathrm{cat}(\mathrm{schroed}), \mathrm{alive}(\mathrm{schroed})\}$ und $\{\mathrm{cat}(\mathrm{schroed}), \mathrm{dead}(\mathrm{schroed})\}$.

Beispiel 4.2. Das Programm

$$P = \left\{ \begin{array}{c} \mathrm{arc}(1,1) \\ \mathrm{arc}(1,2) \\ \mathrm{edge}(X,Y) \leftarrow \mathrm{arc}(X,Y), \mathrm{arc}(Y,X) \end{array} \right\}$$

hat die Grundinstanz

$$\mathrm{grd}(P) = \left\{ \begin{array}{c} \mathrm{arc}(1,1) \\ \mathrm{arc}(1,2) \\ \mathrm{edge}(1,1) \leftarrow \mathrm{arc}(1,1), \mathrm{arc}(1,1) \\ \mathrm{edge}(2,2) \leftarrow \mathrm{arc}(2,2), \mathrm{arc}(2,2) \\ \mathrm{edge}(1,2) \leftarrow \mathrm{arc}(1,2), \mathrm{arc}(2,1) \\ \mathrm{edge}(2,1) \leftarrow \mathrm{arc}(2,1), \mathrm{arc}(1,2) \end{array} \right\}.$$

Alle weiteren Spracherweiterungen werden nur für gewöhnliche Regeln und Programme eingeführt. Sie lassen sich jedoch ohne Schwierigkeiten auf Prädikate und Variablen erster Ordnung übertragen. Desweiteren wird ein Regelschema manchmal als Regel bezeichnet.

4.3.2 *Integritätsbedingung*

Die Integritätsbedingung (engl. *integrity constraint*) modelliert eine Aussage der Form „Es kann nicht sein, dass ... gilt.“. Sie entfernt jedes stabile Modell aus dem Ergebnis eines Programms, das die Bedingung in ihrem Rumpf erfüllt:

Definition 4.9. Sei $1 \leqslant m \leqslant n$ und $a_i \in A$ für jedes $1 \leqslant i \leqslant n$. Eine Regel der Form

$$\leftarrow a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n$$

Integritätsbedingung heißt *Integritätsbedingung*.

Eine Integritätsbedingung wird in eine Regel

$$x \leftarrow a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n, {\sim}x$$

übersetzt, wobei x ein neues Atom, also $x \notin A$ ist.

4.3.3 *Auswahlregel*

Die Auswahlregel (engl. *choice rule*) macht es möglich, Entscheidungen über Teilmengen von Atomen auszudrücken. Ist der Rumpf einer Auswahlregel erfüllt, so kommt ein stabiles Modell für jede Teilmenge der Atome im Kopf infrage:

Definition 4.10. Sei $1 \leqslant m \leqslant n \leqslant o$ und $a_i \in A$ für jedes $1 \leqslant i \leqslant o$. Eine Regel der Form

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, {\sim}a_{n+1}, \ldots, {\sim}a_o$$

Auswahlregel heißt *Auswahlregel*.

Die Auswahlregel ist geeignet, um auf einfache Weise viele stabile Modelle zu erzeugen, wie das folgende Beispiel zeigt:

Beispiel 4.3. Das Programm $P = \{\ \{a, b\}\ \}$ hat vier stabile Modelle, nämlich die Elemente von $2^{\{a,b\}}$.

Die Übersetzung einer Auswahlregel erzeugt $2m + 1$ Regeln. Sie ist in [Geb+12, S. 18] beschrieben.

4.3.4 *Kardinalitätsregel*

Die Kardinalitätsregel (engl. *cardinality rule*) ist eine Verallgemeinerung der Auswahlregel. Sie ermöglicht es, bei der Auswahl die Anzahl der Atome durch eine untere Schranke l und eine obere Schranke u zu begrenzen:

Definition 4.11. Sei $1 \leqslant m \leqslant n \leqslant o$ und $a_i \in A$ für jedes $1 \leqslant i \leqslant o$. Weiter sein $0 \leqslant l \leqslant u \leqslant m$. Eine Regel der Form

$$l\,\{a_1, \ldots, a_m\}\,u \leftarrow a_{m+1}, \ldots, a_n, {\sim}a_{n+1}, \ldots, {\sim}a_o$$

heißt *Kopfform der Kardinalitätsregel*.

Kopfform der Kardinalitätsregel

Das Programm P_2 aus Tabelle 4.2 lässt sich mit der Kardinalitätsregel deutlich vereinfachen, wie das folgende Beispiel zeigt:

Beispiel 4.4. Das Programm $P = \{\ 1\ \{a, b\}\ 1\ \}$ hat zwei stabile Modelle, nämlich $\{a\}$ und $\{b\}$.

Alle bisherigen Definitionen haben nur den Kopf einer Regel verändert[4]. Die zweite Form der Kardinalitätsregel verändert den Rumpf und entspricht der Aussage „a_0 ist wahr, wenn mindestens l und höchstens u Literale aus $\{a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n\}$ wahr sind.“:

Definition 4.12. Sei $0 \leqslant m \leqslant n$ und $a_i \in A$ für jedes $0 \leqslant i \leqslant n$. Weiter sein $0 \leqslant l \leqslant u \leqslant n$. Eine Regel der Form

$$a_0 \leftarrow l\,\{a_1, \ldots, a_m, {\sim}a_{m+1}, \ldots, {\sim}a_n\}\,u$$

heißt *Rumpfform der Kardinalitätsregel*.

Rumpfform der Kardinalitätsregel

Die Kombination der Kopfform mit der Rumpfform ergibt die allgemeine Kardinalitätsregel. Der Rumpf der Rumpfform wird dabei mehrmals verwendet, sodass mehrere Bedingungen der Form „Mindestens l und höchstens u Literale aus S sind wahr.“ möglich sind:

Definition 4.13. Sei $S_i \subseteq L$ und $0 \leqslant l_i \leqslant u_i \leqslant |S_i|$ für jedes $0 \leqslant i \leqslant n$. Weiter sei $S_0 \subseteq A$. Ein Ausdruck $l_i\ S_i\ u_i$ heißt *Kardinalitätsbedingung*. Eine Regel der Form

Kardinalitätsbedingung

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \ldots, l_n\ S_n\ u_n$$

heißt *Kardinalitätsregel*.

Kardinalitätsregel

Es ist möglich, einzelne Schranken wegzulassen. Eine untere Schranke wird dann implizit zu $l_i = 0$, eine obere Schranke zu $u_i = |S_i|$ angenommen. Folgendes Beispiel nutzt dies:

Beispiel 4.5. Eine Regel nach Definition 4.1 lässt sich wie folgt als Kardinalitätsregel darstellen:

$$1\,\{a_0\} \leftarrow 1\,\{a_1\}, \ldots, 1\,\{a_m\}, 1\,\{{\sim}a_{m+1}\}, \ldots, 1\,\{{\sim}a_n\}$$

Mithilfe der folgenden Umformung lässt sich das Negationssymbol $\sim$ vollständig entfernen:

$$1\,\{a_0\} \leftarrow 1\,\{a_1\}, \ldots, 1\,\{a_m\}, \{a_{m+1}\}\,0, \ldots, \{a_n\}\,0$$

Die Übersetzung einer Kardinalitätsregel erzeugt $O(n^2)$ Regeln. Sie ist in [Geb+12, S. 18–21] beschrieben.

4 Es gibt keine Rumpfform der Auswahlregel, da eine Regel der Form $a_0 \leftarrow \{a_1, \ldots, a_m, a_{m+1}, \ldots, a_n\}$ wenig Sinn ergibt, denn $\{a_1, \ldots, a_m, a_{m+1}, \ldots, a_n\}$ ist stets erfüllt und die Regel daher äquivalent zu dem Fakt a_0.

4.3.5 *Gewichtsregel*

Die Gewichtsregel (engl. *weight rule*) ist eine Verallgemeinerung der Kardinalitätsregel. Sie erlaubt es, den Literalen ein Gewicht zuzuordnen. Die untere Schranke l und die obere Schranke u bezieht sich dann, statt auf die Anzahl der Atome, auf die Summe ihrer Gewichte.

Definition 4.14. Sei $S_i \subseteq L \times \mathbb{Z}$ und $l_i, u_i \in \mathbb{Z}$ für jedes $0 \leqslant i \leqslant n$. Weiter sei $S_0 \subseteq A \times \mathbb{Z}$. Ein Ausdruck $l_i\ S_i\ u_i$ heißt *Gewichtsbedingung*. Eine Regel der Form

Gewichtsbedingung

$$l_0\ S_0\ u_0 \leftarrow l_1\ S_1\ u_1, \ldots, l_n\ S_n\ u_n$$

heißt *Gewichtsregel*.

Gewichtsregel

Eine Gewichtsbedingung beschreibt man gewöhnlich mit einem Ausdruck der Form

$$l\ \{a_1 = w_1, \ldots, a_n = w_n\}\ u.$$

Es ist möglich einzelne Gewichte wegzulassen. Ein Gewicht wird dann implizit zu $w_i = 1$ angenommen. Eine Gewichtsregel ohne Gewichte ist eine Kardinalitätsregel.

Aggregate

Die Implementierung der Gewichtsregel in Form sogenannter *Aggregate* ist in [Geb+12, S. 114–116] beschrieben. Obwohl es große Gemeinsamkeiten zwischen der Kardinalitätsregel und der Gewichtsregel gibt, ist die Umsetzung der Gewichtsregel deutlich komplizierter und ineffizienter.

4.3.6 *Bedingtes Literal*

Das bedingte Literal (engl. *conditional literal*) ermöglicht es, Aussagen über abstrakte Objekte zu modellieren. Es kann ein oder mehrere gewöhnliche Literale in einer Regel ersetzen.

Definition 4.15. Sei $l_i \in L$ ein Literal für jedes $0 \leqslant i \leqslant n$. Ein Ausdruck der Form

$$l_0 : l_1 : \cdots : l_n$$

heißt *bedingtes Literal*.

bedingtes Literal

Die Bedeutung eines bedingten Literals ist vom Kontext abhängig. Ein bedingtes Literal wird im fertigen Programm durch die Literale in $\{l_0 \mid l_1, \ldots, l_n\}$ ersetzt. Das Kopfliteral l_0 wird also in Abhängigkeit vom Wahrheitsstatus der Bedingungsliterale $l_1, \ldots, l_n$ instanziiert.

Das bedingtes Literal ist besonders mächtig, wenn man es mit Variablen erster Ordnung kombiniert, wie das folgende Beispiel zeigt:

Beispiel 4.6. Das Programm

$$P = \left\{ \begin{array}{c} \text{theory}(a) \\ \text{theory}(b) \\ 1\ \{\text{truth}(X) : \text{theory}(X)\}\ 1 \end{array} \right\}$$

hat die Grundinstanz

$$\text{grd}(P) = \left\{ \begin{array}{c} \text{theory}(a) \\ \text{theory}(b) \\ 1\ \{\text{truth}(a), \text{truth}(b)\}\ 1 \end{array} \right\}$$

und die stabilen Modelle $\{\text{theory}(a), \text{theory}(b), \text{truth}(a)\}$ und $\{\text{theory}(a), \text{theory}(b), \text{truth}(b)\}$.

Die Kardinalitätsregel $1\ \{\text{truth}(X) : \text{theory}(X)\}\ 1$ modelliert, dass genau eine Theorie wahr sein muss, unabhängig davon, welche Theorien existieren.

4.3.7 *Starke Negation*

Die starke Negation ist sehr hilfreich, wenn man die Ungültigkeit einer Aussage explizit machen möchte. Sie lässt sich darstellen, wenn man zu jedem Atom $a \in A$ ein neues Atom $\neg a$ ergänzt und eine Integritätsbedingung hinzufügt, dass a und $\neg a$ nicht gleichzeitig in einem stabilen Modell auftreten können:

Definition 4.16. Sei $a, \neg a \in A$. Dann heißt $\neg a$ *starke Negation* von a und umgekehrt, wenn P eine Regel $\leftarrow a, \neg a$ enthält[5]. *starke Negation*

Das Symbol $\neg$ sei im Folgenden für die starke Negation reserviert. Dass heißt, wenn $\neg a \in A$, dann ist $a \in A$ und P enthält eine Regel $\leftarrow a, \neg a$.

Der Unterschied zwischen der starken und der schwachen Negation wird im folgenden Beispiel illustriert. Er besteht darin, wie ein Programm mit negativen Voraussetzungen umgeht, die nicht explizit sind:

Beispiel 4.7. Das Programm $P_1 = \{\text{cross} \leftarrow \sim \text{train}\}$ hat das eindeutige stabile Modell $\{\text{cross}\}$, während $P_2 = \{\text{cross} \leftarrow \neg\, \text{train}\}$ das eindeutige stabile Modell $\emptyset$ hat.

Die Programme $P_3 = P_1 \cup \{\neg\, \text{train}\}$ und $P_4 = P_2 \cup \{\neg\, \text{train}\}$ haben beide das eindeutige stabile Modell $\{\text{cross}, \neg\, \text{train}\}$.

5 [Geb+12, S. 24] verwendet eine abweichende Semantik, die A als stabiles Modell zulässt. Dazu wird P um eine Regel $a \leftarrow b, \neg b$ für jedes $a \in A$ und $b \in A^+$ ergänzt, wobei A^+ alle positiven Atome aus A enthält. Ein Atom heißt hier positiv, wenn es kein Präfix $\neg$ hat. Der Autor dieser Arbeit denkt, dass A als stabiles Modell unzulässig sein sollte, weil es Atome a und $\neg a$ enthält, die sich widersprechende Aussagen repräsentieren.

Während die schwache Negation nahelegt, einen Bahnübergang zu überqueren, wenn man *nicht weiß*, dass ein Zug kommt, empfiehlt die starke Negation, den Bahnübergang nur zu überqueren, wenn man *weiß*, dass *kein* Zug kommt.

4.3.8 *Disjunktion*

Eine Regel nach Definition 4.1 beschreibt eine Implikation ohne Auswahlmöglichkeit. Ist der Rumpf der Regel erfüllt, so muss ihr Kopfatom wahr sein. Die folgende Definition erweitert den Kopf um weitere Atome, von denen mindestens eines wahr sein muss, wenn der Rumpf erfüllt ist:

Definition 4.17. Sei $1 \leqslant m \leqslant n \leqslant o$ und $a_i \in A$ für jedes $1 \leqslant i \leqslant o$. Eine Regel der Form

$$a_1; \ldots; a_m \leftarrow a_{m+1}, \ldots, a_n, {\sim}a_{n+1}, \ldots, {\sim}a_o$$

disjunktive Regel heißt *disjunktive Regel*.

disjunktives Programm **Definition 4.18.** Ein *disjunktives Programm* ist eine endliche Menge disjunktiver Regeln.

Eine disjunktive Regel r lässt sich im Allgemeinen nicht auf andere Regeln zurückführen. Um ihre Semantik zu erfassen, muss der Modellbegriff aus Definition 4.3 überarbeitet werden. Dazu sei $\text{head}(r) = \{a_1, \ldots, a_m\}$.

Modell **Definition 4.19.** Sei P ein disjunktives Programm über die Atome aus A. Eine Menge $M \subseteq A$ heißt gültiges *Modell* von P genau dann, wenn für jede Regel $r \in P$ gilt:

$$\text{body}(r)^+ \subseteq M \wedge \text{body}(r)^- \cap M = \emptyset \rightarrow \text{head}(r) \cap M \neq \emptyset$$

Reduktion *stabiles Modell* Der Begriff der *Reduktion* aus Definition 4.4 bleibt unverändert. Gleiches gilt für das *stabile Modell* aus Definition 4.5: Ein $M \subseteq A$ heißt stabiles Modell von P, wenn M das $\subseteq$-minimale Modell von P^M ist.

Das stabile Modell eines positiven disjunktiven Programms ist, im Gegensatz zu dem stabilen Modell eines positiven Programms, nicht zwingend eindeutig, wie das folgende Beispiel zeigt:

Beispiel 4.8. Das disjunktive logische Programm $P = \{a; b, b; c\}$ hat zwei stabile Modelle, nämlich $\{a, c\}$ und $\{b\}$.

Beweis. Zu jeder Teilmenge $M \subseteq A = \{a, b, c\}$ ist $P^M = P$. Die Teilmengen $\emptyset$, $\{a\}$, $\{c\}$, $\{a, b\}$ und $\{b, c\}$ sind keine Modelle von P^M. $\{a, c\}$, $\{a, b, c\}$ und $\{b\}$ sind Modelle von P^M. $\{a, b, c\}$ ist kein minimales Modell von P^M, da $\{b\} \subseteq \{a, b, c\}$. Da $\{a, c\} \nsubseteq \{b\}$ und $\{b\} \nsubseteq \{a, c\}$, sind beide Modelle minimal und damit stabile Modelle von P. □

Betrachtet man nur die umgangssprachliche Beschreibung der disjunktiven Regel („Mindestens ein Atom im Kopf muss wahr sein, wenn der Rumpf erfüllt ist."), könnte man zu dem Schluss kommen, dass sie sich auf eine Kardinalitätsregel zurückführen lässt. Dies ist jedoch ein Trugschluss, wie das folgende Beispiel zeigt:

Beispiel 4.9. Das Programm $P = \{\ 1\ \{a, b\}, 1\ \{b, c\}\ \}$ hat fünf stabile Modelle, nämlich $\{a, b\}$, $\{a, c\}$, $\{b\}$, $\{b, c\}$ und $\{a, b, c\}$.

minimierende Disjunktion

In ASP ist eine Disjunktion im Allgemeinen weder inklusiv noch exklusiv. Sie ist minimierend, wie das folgende Beispiel zeigt:

Beispiel 4.10. Das Programm $P_1 = \{a; b\}$ hat die stabilen Modelle $\{a\}$ und $\{b\}$. Das Programm $P_2 = P_1 \cup \{a, b\}$ hat das stabile Modell $\{a, b\}$.

inklusive und exklusive Disjunktion

Die inklusive und exklusive Disjunktion lässt sich jeweils mithilfe einer Kardinalitätsregel modellieren: $1\ \{a_1, \ldots, a_n\}$ entspricht der inklusiven, $1\ \{a_1, \ldots, a_n\}\ 1$ der exklusiven Disjunktion der Atome $a_1, \ldots, a_n$.

4.4 IMPLEMENTIERUNG: GRINGO + CLASP = CLINGO

Grounder
Solver

Eine Implementierung von ASP muss zwei Aufgaben erfüllen, die gewöhnlich auf zwei verschiedene Programme aufgeteilt werden. Ein *Grounder* erzeugt die Grundinstanzen eines logischen Programms. Ein *Solver* berechnet daraus die stabilen Modelle. Diese Arbeit verwendet den Grounder *gringo* in Version 5.3.0 und den Solver *clasp* in Version 3.3.4 aus der Potsdam Answer Set Solving Collection (Potassco). Dieser Abschnitt beinhaltet eine kurze Einführung. Details können dem Handbuch [Geb+17] entnommen werden.

Die Eingabesprache von *gringo* ähnelt der bisher verwendeten Notation. Variablen wie `X`, `Y` und `Pet` beginnen mit einem Großbuchstaben, Konstanten wie `a`, `b`, `101` und `dalmatiner` nicht. Prädikate wie `p(X)`, `q(Y,Z)` und `q(a,Z)` bestehen aus Variablen und Konstanten. Sie sind Grundinstanzen wie `q(a,b)`, wenn sie keine Variablen enthalten. Tabelle 4.4 enthält eine Gegenüberstellung der Junktoren. Eine Regel wie `dead(Cat) :- not alive(Cat).` endet mit einem Punkt. Die Spracherweiterungen aus Abschnitt 4.3 werden unterstützt. Ihre Syntax ist vergleichbar.

Direktive

Eine *Direktive* gibt dem Grounder eine Anweisung, wie das Programm ausgeführt werden soll. Sie beginnt mit einer Raute. Die Direktive `#show.` unterdrückt sämtliche Ausgaben. `#show p/2.` zeigt das 2-stellige Prädikat `p` an. `#show p(X,Y) : q(X)` zeigt `p(X,Y)` an, wenn die Bedingung `q(X)` erfüllt ist. Weitere Direktiven werden in Kapitel 5 eingeführt.

Das Programm `graph.lp` in Listing 4.1 modelliert einen Graph mit 6 Knoten und 17 Kanten. Die Regel in Zeile 1 ist eine Kurzschreibweise für die Fakten `node(1)` bis `node(6)`.

	Theorie	Potassco
Implikation	$\leftarrow$	`:-`
Konjunktion	,	,
Disjunktion	;	;
Schwache Negation	$\sim$	`not`
Starke Negation	$\neg$	`-`

Tabelle 4.4: Junktoren der ASP in der Theorie und in Potassco

```
1 node(1..6).
2
3 edge(1,2). edge(1,3). edge(1,4).
4 edge(2,4). edge(2,5). edge(2,6).
5 edge(3,1). edge(3,4). edge(3,5).
6 edge(4,1). edge(4,2).
7 edge(5,3). edge(5,4). edge(5,6).
8 edge(6,2). edge(6,3). edge(6,5).
```

Listing 4.1: Ein Graph mit 6 Knoten und 17 Kanten (`graph.lp`)

```
1 col(r). col(g). col(b).
2
3 1 { color(X,C) : col(C) } 1 :- node(X).
4 :- edge(X,Y), color(X,C), color(Y,C).
5
6 #show.
7 #show color/2.
```

Listing 4.2: Das Graphfärbungsproblem mit 3 Farben (`3color.lp`)

```
1  $ gringo graph.lp 3color.lp | clasp 0
2  clasp version 3.3.4
3  Reading from stdin
4  Solving...
5  Answer: 1
6  color(2,g) color(1,b) color(3,g) color(4,r) color(5,b) color(6,r)
7  Answer: 2
8  color(1,r) color(2,g) color(3,g) color(4,b) color(5,r) color(6,b)
9  Answer: 3
10 color(1,r) color(2,b) color(3,b) color(4,g) color(5,r) color(6,g)
11 Answer: 4
12 color(1,g) color(2,b) color(3,b) color(4,r) color(5,g) color(6,r)
13 Answer: 5
14 color(2,r) color(1,b) color(3,r) color(4,g) color(5,b) color(6,g)
15 Answer: 6
16 color(2,r) color(1,g) color(3,r) color(4,b) color(5,g) color(6,b)
17 SATISFIABLE
18
19 Models       : 6
20 Calls        : 1
21 Time         : 0.000s (Solving: 0.00s 1st Model: 0.00s Unsat:
       0.00s)
22 CPU Time     : 0.000s
```

Listing 4.3: Das Ergebnis von `graph.lp` und `3color.lp`

Das Programm `3color.lp` in Listing 4.2 löst das Graphfärbungsproblem für drei Farben. Die Fakten in Zeile 1 definieren die verfügbaren Farben. Die Kardinalitätsregel in Zeile 3 legt fest, dass jeder Knoten genau eine Farbe haben muss. Die Integritätsbedingung in Zeile 4 schließt alle Lösungen aus, bei denen benachbarte Knoten die gleiche Farbe haben. Die Zeilen 6 und 7 legen fest, dass nur die Farbzuordnung angezeigt wird.

Der Befehl `gringo graph.lp 3color.lp | clasp 0` berechnet alle Lösungen des Graphfärbungsproblems für den Graph aus Listing 4.1. Dabei wird die Ausgabe des Grounders *gringo* an den Solver *clasp* weitergeleitet. Die Ausgabe von *clasp* ist in Listing 4.3 abgedruckt. Es ist möglich die Ausgabe auf `n` Lösungen zu beschränken, indem man die `0` durch `n` ersetzt. Der Befehl `clingo graph.lp 3color.lp 0` ist eine Abkürzung für `gringo graph.lp 3color.lp | clasp 0`.

Da ein Problem häufig mehrere Instanzen hat, ist es sinnvoll, die Instanz (`graph.lp`) von ihrer Klasse (`3color.lp`) zu trennen.

5 ANSWER SET PREFERENCES

In Kapitel 3 wurde vorgeschlagen, eine Korrektur so zu wählen, dass sie durch möglichst wenige Veränderungen aus der ursprünglichen Wissensbasis hervorgeht. Das ist ein Optimierungsproblem. Um ASP zur Berechnung der Korrektur einer Wissensbasis zu verwenden, ist es also nötig, ein Optimierungsproblem in ASP zu modellieren und zu lösen. Dieses Kapitel beschäftigt sich mit einer Spracherweiterung, die genau das möglich macht. Sie basiert auf einer Präferenzrelation, welche die stabilen Modelle eines Programms partiell ordnet. Es gibt dann stabile Modelle die maximal bezüglich dieser Ordnung sind. Sie heißen präferierte stabile Modelle.

Dieses Kapitel verwendet Ideen aus [BE99; Bre+15].

5.1 PRÄFERENZRELATION

Eine Präferenzrelation definiert eine partielle Ordnung auf dem Ergebnis[1] eines Programms:

Definition 5.1. Sei P ein Programm über die Atome aus A. Eine partielle Ordnung $<$ auf 2^A heißt *Präferenzrelation*. *Präferenzrelation*

Man beachte, dass eine partielle Ordnung $<$ auf 2^A einer partiellen Ordnung auf einer Teilmenge $\mathbb{M} \subseteq 2^A$ entspricht, wenn man jedes $(M_1, M_2) \in <$ mit $M_1 \subsetneq \mathbb{M}$ oder $M_2 \subsetneq \mathbb{M}$ ignoriert. Desweiteren ist jede partielle Ordnung auf $\mathbb{M}$ eine partielle Ordnung auf 2^A. Es ist daher gerechtfertigt, die Menge 2^A als Grundmenge einer Präferenzrelation zu verwenden, obwohl das Ergebnis eines Programms meistens nur eine Teilmenge davon ist.

Ein präferenzielles Programm ist ein Programm mit einer passenden Präferenzrelation:

Definition 5.2. Sei P ein Programm über die Atome aus A und $<$ eine Präferenzrelation auf 2^A. Das Tupel $(P, <)$ heißt *präferenzielles Programm*. *präferenzielles Programm*

Das Ergebnis eines präferenzielles Programms ist eine Menge von präferierten stabilen Modellen:

Definition 5.3. Sei $(P, <)$ ein präferenzielles Programms. Eine Menge $M \in A$ heißt *präferiertes stabiles Modell* von $(P, <)$, wenn M ein stabiles Modell von P und maximal bezüglich $<$ ist. *präferiertes stabiles Modell*

1 [BE99] schlägt abweichend von [Bre+15] vor, eine partielle Ordnung auf den Regeln des Programms zu definieren und auf das Ergebnis zu übertragen.

Die Berechnung eines oder aller präferierten stabilen Modelle ist in [Bre+15, S. 1471–1472] beschrieben.

Die folgenden Beispiele stellen verschiedene Möglichkeiten vor, eine Präferenzrelation zu definieren. Der Parameter $\mathbb{M} \subseteq 2^A$ beschränkt dabei die Präferenzrelation auf eine Teilmenge der Modelle und ermöglicht so weiteren Einfluss auf ihre Bedeutung. Das erste Beispiel macht die Mengeneigenschaften der Modelle nutzbar:

Beispiel 5.1. Für jedes $(M_1, M_2) \in \mathbb{M}^2$ sei

- $(M_1, M_2) \in \text{subset}(\mathbb{M}) \Leftrightarrow M_1 \subset M_2$
- $(M_1, M_2) \in \text{superset}(\mathbb{M}) \Leftrightarrow M_1 \supset M_2$
- $(M_1, M_2) \in \text{less}(\text{card})(\mathbb{M}) \Leftrightarrow |M_1| < |M_2|$
- $(M_1, M_2) \in \text{more}(\text{card})(\mathbb{M}) \Leftrightarrow |M_1| > |M_2|$

Jede Zeile definiert eine Präferenzrelation: $\text{subset}(\mathbb{M})$ bevorzugt Teilmengen von anderen Modellen, $\text{superset}(\mathbb{M})$ bevorzugt Obermengen. $\text{less}(\text{card})(\mathbb{M})$ bevorzugt Modelle mit wenigen, $\text{more}(\text{card})(\mathbb{M})$ mit vielen Atomen aus $\mathbb{M}$.

Ordnet man jedem Atom a ein Gewicht $w(a)$ zu, so kann man eine Präferenzrelation definieren, die die Summe der Gewichte in den Modellen vergleicht:

Beispiel 5.2. Sei $w : \bigcup_{M \in \mathbb{M}} M \to \mathbb{Z}$. Für jedes $(M_1, M_2) \in \mathbb{M}^2$ sei

- $(M_1, M_2) \in \text{less}(\text{weight})(\mathbb{M}, w) \Leftrightarrow \sum_{a \in M_1} w(a) < \sum_{a \in M_2} w(a)$
- $(M_1, M_2) \in \text{more}(\text{weight})(\mathbb{M}, w) \Leftrightarrow \sum_{a \in M_1} w(a) > \sum_{a \in M_2} w(a)$

Es ist möglich, eine Präferenzrelation aus einer oder mehreren anderen Präferenzrelationen abzuleiten, wie das folgende Beispiel zeigt:

Beispiel 5.3. Sei $\mathbb{R} = \{R_1, \ldots, R_n\} \subseteq \{R \mid R \subseteq \mathbb{M}^2\}$ eine Menge von Präferenzrelationen und $w : \mathbb{R} \to \mathbb{Z}$. Für jedes $(M_1, M_2) \in \mathbb{M}^2$ sei

- $(M_1, M_2) \in \text{neg}(R_1) \Leftrightarrow (M_2, M_1) \in R_1$
- $(M_1, M_2) \in \text{and}(\mathbb{R}) \Leftrightarrow$ Für alle $R \in \mathbb{R}$ ist $(M_1, M_2) \in R$.
- $(M_1, M_2) \in \text{lexico}(\mathbb{R}, w) \Leftrightarrow$ Es gibt ein $R \in \mathbb{R}$ mit $(M_1, M_2) \in R$ und es gibt ein $z \in \mathbb{Z}$, sodass für alle $R \in \mathbb{R}$ mit $w(R) > z$ gilt: $(M_1, M_2) \notin R$ und $(M_2, M_1) \notin R$.

Die Präferenzrelation $\text{neg}(R)$ invertiert die Präferenzrelation R. Zum Beispiel ist $\text{neg}(\text{subset}(\mathbb{M})) = \text{superset}(\mathbb{M})$. $\text{and}(\mathbb{R})$ konjugiert die Präferenzrelationen in $\mathbb{R}$. $\text{lexico}(\mathbb{R}, w)$ bildet die lexikografische Ordnung der Präferenzrelationen $R \in \mathbb{R}$, jeweils gewichtet nach $w(R)$.

```
1  { col(1..4) }.
2
3  1 { color(X,C) : col(C) } 1 :- node(X).
4  :- edge(X,Y), color(X,C), color(Y,C).
5
6  #preference(p, less(weight)) { C :: col(C) }.
7  #optimize(p).
8
9  #show.
10 #show col/1.
11 #show color/2.
```

Listing 5.1: Das minimierende Graphfärbungsproblem (`ncolor.lp`)

5.2 IMPLEMENTIERUNG: ASPRIN

Diese Arbeit verwendet das Framework *asprin* in Version 3.0.2 aus der Potassco, um präferenzielle Programme zu lösen. Dieser Abschnitt enthält eine kurze Einführung. Details werden im Handbuch [Geb+17, S. 86–105] beschrieben.

Die Eingabesprache von *asprin* ist eine Erweiterung der Eingabesprache von *gringo* aus Abschnitt 4.4. Die Regeln eines Programms werden daher wie bisher definiert.

Eine Direktive der Form `#preference(s,t) { e1; ...; en }` definiert eine Präferenzrelation. Der Term `s` bestimmt ihren Namen, der Term `t` ihren Typ. `e1` bis `en` sind sogenannte Präferenzelemente. Sie stellen weitere Informationen zur Verfügung, ähnlich wie es die Argumente $\mathbb{M}$, w, R_1 und $\mathbb{R}$ in den Beispielen 5.1, 5.2 und 5.3 tun. Die genaue Form der Präferenzelemente hängt vom Typ der Präferenzrelation ab.

Die *asprin library* stellt eine Reihe vordefinierter Typen zur Verfügung. Dazu zählt ein Typ für jede Präferenzrelation aus den Beispielen 5.1, 5.2 und 5.3.

Eine Direktive der Form `#optimize(s)` bestimmt den Namen `s` der Präferenzrelation, die für die Optimierung verwendet werden soll. Sie ist zwingend erforderlich, da es mehrere Präferenzrelationen geben kann. Es kann mehrere Präferenzrelationen geben, da Präferenzrelationen aus anderen Präferenzrelation zusammengesetzt werden können.

Das Programm `ncolor.lp` in Listing 5.1 löst das Graphfärbungsproblem mit einer minimalen Anzahl Farben. Die Auswahlregel in Zeile 1 legt fest, dass eine beliebige Teilmenge der Farben 1 bis 4 verwendet werden kann. Zeile 6 definiert eine Präferenzrelation, die die Summe der Farben minimiert. Dies hat den Effekt das weniger Farben, bei gleicher Anzahl kleinere Farben, bevorzugt werden[2]. Zeile 7 legt fest,

2 Man kann alternativ `#preference(p, less(cardinality)) { col(C) }.` und eine zusätzliche Regel `col(C-1) :- col(C), C > 2.` verwenden, um den selben Effekt zu erzielen. Dann tritt jedoch ein Phänomen in der Ausgabe nicht auf, das erklärt werden soll.

dass die Präferenzrelation aus Zeile 6 für die Optimierung verwendet wird. Zeile 10 wurde ergänzt, um die verwendeten Farben anzuzeigen. Die übrigen Zeilen erfüllen den selben Zweck wie in Listing 4.2.

Der Befehl `asprin graph.lp ncolor.lp 0` berechnet alle Lösungen des minimierenden Graphfärbungsproblems für den Graph aus Listing 4.1. Die Ausgabe ist in Listing 5.2 abgedruckt. Auf den ersten Blick sieht es so aus, als sei die Ausgabe falsch, da *asprin* standardmäßig stabile Modelle ausgibt, die als Zwischenergebnis bei der Berechnung dienen. Ein präferiertes stabiles Modell wird mit `OPTIMUM FOUND` gekennzeichnet. `OPTIMUM FOUND *` bedeutet, dass ein präferiertes stabiles Modell nicht strikt besser als ein vorheriges präferiertes stabiles Modell war. Die Ausgabe nicht präferierter stabiler Modelle kann mit `--quiet=1` unterdrückt werden.

```
$ asprin graph.lp ncolor.lp 0
asprin version 3.0.2
Reading from graph.lp ...
Solving...
Answer: 1
col(1) col(2) col(3) col(4) color(1,1) color(6,1) color(4,2)
    color(5,3) color(2,4) color(3,4)
Answer: 2
col(2) col(3) col(4) color(4,2) color(6,2) color(1,3) color(5,3)
    color(2,4) color(3,4)
Answer: 3
col(1) col(3) col(4) color(1,1) color(5,1) color(2,3) color(3,3)
    color(4,4) color(6,4)
Answer: 4
col(1) col(2) col(4) color(4,1) color(6,1) color(1,2) color(5,2)
    color(2,4) color(3,4)
Answer: 5
col(1) col(2) col(3) color(2,1) color(3,1) color(1,2) color(5,2)
    color(4,3) color(6,3)
OPTIMUM FOUND
Answer: 6
col(1) col(2) col(3) color(2,1) color(3,1) color(4,2) color(6,2)
    color(1,3) color(5,3)
OPTIMUM FOUND *
Answer: 7
col(1) col(2) col(3) color(4,1) color(6,1) color(1,2) color(5,2)
    color(2,3) color(3,3)
OPTIMUM FOUND *
Answer: 8
col(1) col(2) col(3) color(4,1) color(6,1) color(2,2) color(3,2)
    color(1,3) color(5,3)
OPTIMUM FOUND *
Answer: 9
col(1) col(2) col(3) color(1,1) color(5,1) color(4,2) color(6,2)
    color(2,3) color(3,3)
OPTIMUM FOUND *
Answer: 10
col(1) col(2) col(3) color(1,1) color(5,1) color(2,2) color(3,2)
    color(4,3) color(6,3)
OPTIMUM FOUND *

Models       : 10
  Optimum    : yes
  Optimal    : 6
Calls        : 8
Time         : 0.116s (Solving: 0.00s 1st Model: 0.00s Unsat:
    0.00s)
CPU Time     : 0.116s
```

Listing 5.2: Das Ergebnis von `graph.lp` und `ncolor.lp`

6 UMSETZUNG

Dieses Kapitel beschäftigt sich mit der Frage, wie man die theoretischen Verfahren aus Kapitel 3 praktisch in ASP umsetzen kann. Dazu soll ein Programm entwickelt werden, das eine Wissensbasis modelliert, Wissenslücken schließt, Widersprüche erkennt, alle Korrekturen berechnet und auf Basis einer Präferenzrelation eine Korrektur auswählt. Das Programm soll generisch sein, sich also auf verschiedene Anwendungen übertragen lassen. Es soll aus einer Instanz, die eine konkrete Wissensbasis modelliert, und einer Klasse, die das jeweilige Verfahren anwendet, bestehen.

Das Programm wird in diesem Kapitel Schritt für Schritt entwickelt. Abschnitt 6.1 beschäftigt sich mit der Modellierung der Wissensbasis, der Erkennung von Widersprüchen und der Vervollständigung von Wissenslücken. Abschnitt 6.2 erweitert das Programm um die Berechnung aller Korrekturen. Abschnitt 6.3 erweitert das Programm um eine Präferenzrelation, die eine Korrektur auswählt.

Die Übertragung des generischen Programms auf einen Anwendungsfall hat sich in der Praxis als sehr komplex und anfällig für Programmierfehler erwiesen. Die praktische Erfahrung zeigt, dass man trotzdem gute Ergebnisse erzielt, wenn man mit einem Programm startet, das Widersprüche in einer Wissensbasis erkennt, und anschließend die Umformungen aus diesem Kapitel sukzessive anwendet. Alle Programme in Kapitel 7 sind so entstanden.

6.1 ERKENNUNG UND VERVOLLSTÄNDIGUNG

Die Basis eines Programms, das Widersprüche eindeutig korrigiert und Wissenslücken schließt, ist ein Programm, das Widersprüche erkennt. Da ASP offensichtlich, wie jede andere logische Programmiersprache, Widersprüche von sich aus erkennt, ist eine Klasse nicht erforderlich[1]. Die nötige Instanz ist von der Modellierung der Wissensbasis abhängig. Diese ist wiederum vom konkreten Anwendungsfall abhängig. Es ist daher an dieser Stelle nur möglich, grobe Richtlinien für die Modellierung einer Wissensbasis anzugeben, auf denen die folgenden Kapitel aufbauen können: Ein Programm, das eine inkonsistente Wissensbasis modelliert, sollte kein stabiles Modell haben. Ein Programm, das eine konsistente, aber unvollständige Wissensbasis

Richtlinien

1 Die Klasse ist leer.

```
1 proposition(a;b).
2
3 proposition(c) :- proposition(a).
4 :- proposition(b), proposition(c).
```

Listing 6.1: `generic/detection/instance.lp`

```
1 $ clingo generic/detection/instance.lp 0
2 clingo version 5.3.0
3 Reading from instance.lp
4 Solving...
5 UNSATISFIABLE
6
7 Models       : 0
8 Calls        : 1
9 Time         : 0.001s (Solving: 0.00s 1st Model: 0.00s Unsat:
      0.00s)
10 CPU Time     : 0.001s
```

Listing 6.2: Das Ergebnis von `generic/detection/*`

modelliert, sollte ein stabiles Modell für jede totale Vervollständigung haben. Ein Programm, das eine konsistente und vollständige Wissensbasis modelliert, sollte genau ein stabiles Modell haben.

Beispielinstanz

Dieses Kapitel verwendet eine Beispielinstanz, die in Listing 6.1 angegeben ist. Das Prädikat `proposition(S)` modelliert, dass (es Grund zu der Annahme gibt, dass) die Aussage `S` wahr ist. Zeile 1 legt fest, dass die Aussagen `a` und `b` wahr sind. Zeile 3 legt fest, dass die Aussage `c` wahr ist, wenn `a` wahr ist. Zeile 4 legt fest, dass entweder `b` oder `c` wahr ist. Die zugehörige Wissensbasis ist offenbar inkonsistent.

Zeile 1 definiert Fakten. Fakten sind in der Praxis regelmäßig das Ergebnis einer Messung. Zeile 3 und 4 definiert Regeln. Sie werden in der Praxis meistens von Hand modelliert und dienen dazu, Messfehler zu erkennen. Die Regel in Zeile 3 dient der Vervollständigung. Die Regel in Zeile 4 definiert einen Widerspruch.

Listing 6.2 enthält die Ausgabe der Beispielinstanz. Das Programm hat wie erwartet kein stabiles Modell.

vereinfachende Annahmen

Um die eingangs genannten Schierigkeiten zu überwinden, wird in diesem Kapitel angenommen, dass bei der Modellierung einer Wissensbasis nur das Prädikat `proposition(S)` verwendet wurde. Dies vereinfacht die Definition der Klassen in den folgenden Abschnitten erheblich. Es handelt sich dabei um keine semantische Einschränkung, da jedes Prädikat `p(V1, ..., Vn)` in ASP durch `proposition(p(V1, ..., Vn))` ersetzt werden kann. Es wird darüber hinaus angenommen, dass alle Richtlinien eingehalten wurden.

```
1 proposition(a;b).
2
3 _proposition(c) :- _proposition(a).
4 :- _proposition(b), _proposition(c).
```

Listing 6.3: `generic/correction/instance.lp`

```
1 1 { believe(S); doubt(S) } 1 :- proposition(S).
2 _proposition(S) :- proposition(S), believe(S).
3
4 infer(S) :- _proposition(S), not proposition(S).
5
6 #preference(p, superset) { believe(S) }.
7 #optimize(p).
```

Listing 6.4: `generic/correction/objective.lp`

6.2 KORREKTUR

Ein Programm, das sich widersprechende Regeln enthält, hat kein stabiles Modell, unabhängig welche Regeln es sonst enthält. Es ist daher nicht möglich, die Beispielinstanz aus Listing 6.1 durch Hinzunahme einer Klasse zu korrigieren, ohne ihre Darstellung zu ändern.

Um einen Angriffspunkt für die Korrektur zu finden, teilt man das Prädikat `proposition(S)` wie folgt auf: `proposition(S)` beschreibt den Wahrheitsstatus von `S` vor der Korrektur. `_proposition(S)` beschreibt den Wahrheitsstatus von `S` nach der Korrektur. Diese Maßnahme entkoppelt bei richtigem Gebrauch alle Widersprüche, ohne sie zu verändern. Die Kopplung lässt sich durch die Regel `_proposition(S) :- proposition(S)` bei Bedarf wiederherstellen.

Widersprüche entkoppeln

Listing 6.3 enthält eine entkoppelte Version der Beispielinstanz. Sie verwendet `proposition(S)` für alle Fakten und `_proposition(S)` für alle Regeln. Diese Einteilung ist fast immer eine gute Wahl. Soll ein Fakt zwingend sein und nicht der Korrektur unterliegen, ist es jedoch hilfreich, von der Einteilung abzuweichen.

Die Klasse in Listing 6.4 ist geeignet, eine entkoppelte Instanz zu korrigieren. Sie berechnet ein stabiles Modell für jede Korrektur. Sie verwendet drei Hilfsprädikate, die von Definition 3.2 inspiriert sind: `believe(S)` ist genau dann wahr, wenn die Aussage `S` geglaubt wird. `doubt(S)` ist genau dann wahr, wenn die Aussage `S` bezweifelt wird. `infer(S)` ist genau dann wahr, wenn die Aussage `S` ergänzt wird.

Hilfsprädikate

Die Kardinalitätsregel in Zeile 1 legt fest, dass jede Aussage in der Wissensbasis entweder geglaubt oder bezweifelt wird. Sie erzeugt ein stabiles Modell für jede mögliche Einteilung der Regeln. Die Regel in Zeile 2 legt fest, dass eine Aussage, die geglaubt wird, in der Korrektur

```
1  $ asprin generic/correction/instance.lp generic/correction/
       objective.lp 0 --quiet=1
2  asprin version 3.0.2
3  Reading from instance.lp ...
4  Solving...
5  Answer: 1
6  Answer: 2
7  _proposition(b) believe(b) doubt(a) proposition(a) proposition(b)
8  OPTIMUM FOUND
9  Answer: 3
10 _proposition(a) _proposition(c) believe(a) doubt(b) proposition(a
       ) proposition(b) infer(c)
11 OPTIMUM FOUND
12
13 Models       : 3
14   Optimum    : yes
15   Optimal    : 2
16 Calls        : 8
17 Time         : 0.115s (Solving: 0.00s 1st Model: 0.00s Unsat:
       0.00s)
18 CPU Time     : 0.115s
```

Listing 6.5: Das Ergebnis von `generic/correction/*`

lose Kopplung

wahr ist. Sie erzeugt eine lose Kopplung zwischen den Prädikaten `proposition(S)` und `_proposition(S)`. Die lose Kopplung ermöglicht es den Regeln der Instanz, jedes stabile Modell zu verbieten, dass eine Inkonsistenz enthält. Die Regel in Zeile 4 modelliert die Bedeutung des Prädikats `infer(S)`. Sie ist optional und für die Korrektur irrelevant, aber für manche Anwendungsfälle nützlich. Die Direktive in Zeile 6 definiert eine Präferenzrelation, die ein stabiles Modell seiner Teilmenge vorzieht. Sie stellt sicher, dass möglichst viele Aussagen geglaubt werden. Zeile 7 legt fest, dass diese Präferenzrelation zur Optimierung verwendet wird.

Listing 6.5 enthält die Ausgabe der Beispielinstanz in Listing 6.3 und der Klasse in Listing 6.4. Das Programm berechnet zwei Korrekturen für die Wissensbasis. Die erste Korrektur in Zeile 7 glaubt `b` und bezweifelt `a`. Die zweite Korrektur in Zeile 10 glaubt `a`, bezweifelt `b` und ergänzt `c`.

6.3 AUSWAHL

Aus Abschnitt 3.3 ist bekannt, dass für eine Auswahl zusätzliche Informationen nötig sind. Aus diesem Grund ist es erforderlich, die Darstellung der Instanz erneut zu überarbeiten. Dazu wird das Prädikat

Prioritäten

`proposition(S)` um eine ganzzahlige Priorität `P` zu `proposition(P,S)`

```
priority(0..3).

proposition(
    0, a;
    1, b
).

_proposition(3, c) :- _proposition(_, a).
:- _proposition(_, b), _proposition(_, c).
```

Listing 6.6: `generic/choice/instance.lp`

```
1 { believe(P,S); doubt(P,S) } 1 :- proposition(P,S).
_proposition(P,S) :- proposition(P,S), believe(P,S).

infer(P,S) :- _proposition(P,S), not proposition(P,S).

#preference(p(P), superset) { believe(P,S) } : priority(P).
#preference(p, lexico) { -P :: **p(P) } : priority(P).
#optimize(p).
```

Listing 6.7: `generic/choice/objective.lp`

erweitert. Darüber hinaus wird ein Prädikat `priority(P)` ergänzt, dass den Wertebereich von `P` festlegt.

Das Programm in Listing 6.6 modelliert die Beispielinstanz mit geeigneten Prioritäten. Die Aussage `a` hat die höchste Priorität 0. Die Aussage `b` hat Priorität 1. Wird eine Aussage durch Vervollständigung, also durch die Regel in Zeile 8, hinzugefügt, hat sie die niedrigste Priorität 3. An allen anderen Stellen, insbesondere bei der Definition des Widerspruchs in Zeile 9, ist die Priorität egal. Dies wird durch die anonyme Variable `_` ausgedrückt, die bei jeder Verwendung einer neuen Variable entspricht.

Die Klasse in Listing 6.7 ist geeignet, eine Instanz dieser Art zu korrigieren und eine Korrektur auf Basis der Prioritäten auszuwählen. Die Zeilen 1 bis 4 erfüllen dabei den selben Zweck wie bei der Korrektur in Listing 6.4. Zeile 6 definiert eine Präferenzrelation `p(P)` zu jeder Priorität `P`. `p(P)` bevorzugt ein stabiles Modell, wenn es mehr Aussagen mit Priorität `P` glaubt. Ziel der Auswahl ist es, eine Korrektur zu finden, die möglichst viele Aussagen mit möglichst hohen Prioritäten glaubt. Dementsprechend werden die Präferenzrelationen `p(P)` in Zeile 7 so kombiniert, dass die Reihenfolge zweier stabiler Modelle im Zweifel von der Präferenzrelation mit der höchsten Priorität festlegt wird[2]. Dazu wird eine lexikografische Ordnung wie in Beispiel 5.3

2 Da eine Präferenzrelation eine partielle Ordnung ist, muss sie keine Reihenfolge vorgeben. In diesem Fall wird die Reihenfolge von einer Präferenzrelation mit niedrigerer Priorität festgelegt oder die stabilen Modelle bleiben ungeordnet.

```
1  $ asprin generic/choice/instance.lp generic/choice/objective.lp 0
       --quiet=1
2  asprin version 3.0.2
3  Reading from instance.lp ...
4  Solving...
5  Answer: 1
6  Answer: 2
7  Answer: 3
8  priority(0) priority(1) priority(2) priority(3) _proposition(0,a)
       _proposition(3,c) believe(0,a) doubt(1,b) proposition(0,a)
      proposition(1,b) infer(3,c)
9  OPTIMUM FOUND
10
11 Models       : 3
12   Optimum    : yes
13   Optimal    : 1
14 Calls        : 6
15 Time         : 0.128s (Solving: 0.00s 1st Model: 0.00s Unsat:
      0.00s)
16 CPU Time     : 0.127s
```

Listing 6.8: Das Ergebnis von `generic/choice/*`

verwendet, die eine Menge von Präferenzrelationen und ihre Gewichte als Eingabe erhält. Der Ausdruck `-P :: **p(P)` steht für das Tupel aus der Präferenzrelation `p(P)` und ihrem Gewicht `-P`. Zeile 8 legt fest, dass die kombinierte Präferenzrelation aus Zeile 7 zur Optimierung verwendet wird.

Listing 6.8 druckt die Ausgabe der Beispielinstanz in Listing 6.6 und der Klasse in Listing 6.7 ab. Das Programm berechnet genau eine Korrektur für die Wissensbasis. Die Korrektur glaubt `a`, bezweifelt `b` und ergänzt `c`.

7

BEISPIELE

Dieses Kapitel stellt zwei Beispiele vor, die die generischen Programme aus Kapitel 6 auf je ein konkretes Problem übertragen. Auf diese Weise soll gezeigt werden, dass die Verfahren aus dieser Arbeit in der Praxis sinnvolle Ergebnisse erzielen. Abschnitt 7.1 führt die Buchwelt ein. Abschnitt 7.2 beschäftigt sich mit der Korrektur inkonsistenter Sudokus.

7.1 BUCHWELT

Regeln

Die Buchwelt modelliert eine stark vereinfachte Bibliothek. Es gibt Bücher und Regale. Jedes Buch gehört in ein bestimmtes Regal. Ein Buch ist entweder verfügbar oder verliehen. Falls ein Buch verfügbar ist, so ist es im richtigen Regal verfügbar. Das richtige Regal ist das Regal, zu dem das Buch gehört. Bücher und Regale existieren in der Buchwelt, wenn überhaupt, genau ein Mal. Es gibt sie nicht doppelt. Ein Buch gehört zu genau einem Regal. Die Eindeutigkeit soll die Buchwelt vereinfachen. Sie unterscheidet die Buchwelt von den meisten Bibliotheken.

Eindeutigkeit

Robotik

Man kann sich die Buchwelt als ein sehr einfaches Szenario aus der Robotik vorstellen. Ein Bibliothekar katalogisiert die Bücher am Computer und legt ihren Standort fest. Ein Ausleihsystem, das am Ausgang installiert ist, verwaltet, welche Bücher verliehen wurden. Ein mobiler Roboter fährt durch die Bibliothek und prüft, wo sich die Bücher tatsächlich befinden. Das Gesamtsystem verwendet Informationen aus drei verschiedenen Quellen. Die Informationen sind im Allgemeinen nicht fehlerfrei. Möglicherweise ordnet der Bibliothekar ein Buch falsch zu, das Ausleihsystem erkennt ein falsches Buch oder der mobile Roboter ist falsch lokalisiert. Solche oder ähnliche Fehler erzeugen regelmäßig Widersprüche, die im Datensatz oder der echten Welt korrigiert werden müssen.

Parallelen zur Blockwelt

Der Name der Buchwelt ist nicht zufällig gewählt. Er soll an die Blockwelt [RN10, S. 370–372] erinnern. Die Blockwelt ist eine bekannte Domäne für propositionale Planungsalgorithmen. Sie ist zugleich einfach genug, um schnell erklärt zu werden und umfangreich genug, um komplizierte Phänomene zu erfassen. Die Buchwelt strebt dasselbe für Algorithmen zur Korrektur inkonsistenter Wissensbasen an.

Prädikat	Bedeutung
`book(B)`	Buch `B` existiert.
`shelf(S)`	Regal `S` existiert.
`belongs(B, S)`	Buch `B` gehört in Regal `S`.
`is(B, lent)`	Buch `B` ist verliehen.
`is(B, available(S))`	Buch `B` ist in Regal `S` verfügbar.

Tabelle 7.1: Prädikate und ihre Bedeutung in der Buchwelt

```
1  book(
2      hitchhiker;
3      martian;
4      hobbit
5  ).
6
7  shelf(
8      scifi;
9      fantasy
10 ).
11
12 belongs(
13     hitchhiker, scifi;
14     martian,    scifi;
15     hobbit,     fantasy
16 ).
17
18 is(
19     hitchhiker, lent;
20     martian,    available(scifi);
21     hobbit,     available(fantasy)
22 ).
```

Listing 7.1: `bookworld/simple/instances/consistent.lp`

7.1.1 *Instanzen*

Prädikate

Um eine Instanz der Buchwelt zu modellieren, sind vier[1] Prädikate notwendig. Sie werden gemeinsam mit ihrer Bedeutung in Tabelle 7.1 beschrieben. Dieser Abschnitt stellt fünf ähnliche Instanzen der Buchwelt vor. Die erste Instanz ist konsistent und vollständig, die anderen Instanzen enthalten Fehler unterschiedlicher Art.

consistent.lp

Listing 7.1 zeigt die konsistente und vollständige Instanz `consistent.lp`. Sie besteht aus drei bekannten Büchern, die gemäß ihres Genres auf zwei Regale aufgeteilt sind. Ein Buch wurde verliehen. Zwei Bücher sind im richtigen Regal verfügbar. Die Zeilen 1 bis 17 sind für alle

1 Es handelt sich bei `is(B, lent)` und `is(B, available(S))` nur um ein Prädikat `is(B, X)`, dessen zweites Argument zwei Ausprägungen `X = lent` und `X = available(S)` hat.

```
18 is(
19                                      % missing information
20     martian,    available(scifi);
21     hobbit,     available(fantasy)
22 ).
```

Listing 7.2: bookworld/simple/instances/incomplete.lp

```
18 is(
19     hitchhiker, available(fantasy);     % invalid information,
           with no valid alternative
20     martian,    available(scifi);
21     hobbit,     available(fantasy)
22 ).
```

Listing 7.3: bookworld/simple/instances/invalid.lp

Instanzen in diesem Abschnitt identisch und werden daher bei den folgenden Instanzen nicht erneut abgedruckt.

Listing 7.2 zeigt die unvollständige Instanz `incomplete.lp`. Der Standort des Buches `hitchhiker` ist unbekannt. Bei der Vervollständigung muss diese Information aus den vorhandenen Regeln hergeleitet werden. Die Regeln erlauben es, dass das Buch entweder verliehen oder im Regal `scifi` verfügbar ist. *incomplete.lp*

Listing 7.3 zeigt die inkonsistente Instanz `invalid.lp`. Laut Zeile 13 gehört das Buch `hitchhiker` zum Regal `scifi`. Laut Zeile 19 ist das Buch jedoch im Regal `fantasy` verfügbar. Betrachtet man Informationen über die Zuordnung eines Buches zu einem Regal als zuverlässiger als Informationen über den tatsächlichen Standort, so muss die Information in Zeile 19 bezweifelt werden. Die Instanz ist nach dieser Korrektur unvollständig. Sie entspricht dann der Instanz `incomplete.lp`. *invalid.lp*

Listing 7.4 zeigt die konsistente Instanz `invalid_duplicate.lp`. Laut Zeile 19 ist das Buch `hitchhiker` verliehen. Laut Zeile 20 ist das Buch im Regal `fantasy` verfügbar. Laut Zeile 13 gehört das Buch zum Regal `scifi`. Betrachtet man Informationen über die Zuordnung eines Buches zu einem Regal als zuverlässiger als Informationen über den *invalid_duplicate.lp*

```
18 is(
19     hitchhiker, lent;
20     hitchhiker, available(fantasy);     % invalid information,
           with valid alternative
21     martian,    available(scifi);
22     hobbit,     available(fantasy)
23 ).
```

Listing 7.4: bookworld/simple/instances/invalid_duplicate.lp

```
18 is(
19     hitchhiker, lent;
20     hitchhiker, available(scifi);       % dublicate information,
           each individually valid
21     martian,    available(scifi);
22     hobbit,     available(fantasy)
23 ).
```

Listing 7.5: `bookworld/simple/instances/valid_duplicate.lp`

```
1 :- belongs(B,S), not book(B).
2 :- belongs(B,S), not shelf(S).
3 1 { belongs(B,S) : shelf(S) } 1 :- book(B).
4
5 :- is(B,X), not book(B).
6 :- is(B,X), X != lent, X != available(S) : shelf(S).
7 1 { is(B, available(S)) : shelf(S); is(B, lent) } 1 :- book(B).
8
9 :- is(B, available(S)), not belongs(B,S).
```

Listing 7.6: `bookworld/simple/objectives/complete.lp`

tatsächlichen Standort, so muss die Information in Zeile 20 bezweifelt werden. Die Instanz ist nach dieser Korrektur vollständig, da die Information in Zeile 19 eine gültige Alternative zu der Information in Zeile 20 darstellt.

valid_duplicate.lp

Listing 7.5 zeigt die inkonsistente Instanz `valid_duplicate.lp`. Laut Zeile 19 ist das Buch `hitchhiker` verliehen. Laut Zeile 20 ist es jedoch im richtigen Regal verfügbar. Die Korrektur dieser Instanz ist nicht eindeutig, selbst wenn man Informationen über die Zuordnung eines Buches zu einem Regal als zuverlässiger betrachtet als Informationen über den tatsächlichen Standort. Die Alternativen sind gleichwertig, falls es keine weiteren Präferenzen gibt.

7.1.2 *Klassen*

complete.lp

Die Klasse `complete.lp` in Listing 7.6 modelliert die Regeln der Buchwelt und ist geeignet, Widersprüche zu erkennen und Wissenslücken zu schließen. Sie erfüllt die Richtlinien aus Abschnitt 6.1. Zeile 1 legt fest, dass nur Bücher in Regale gehören. Zeile 2 legt fest, dass Bücher nur in Regale gehören. Zeile 3 legt fest, dass jedes Buch in genau ein Regal gehört. Zeile 5 legt fest, dass nur Bücher in Regalen vorhanden oder verliehen sein können. Zeile 6 legt fest, dass Bücher nur in Regalen vorhanden oder verliehen sein können. Zeile 7 legt fest, dass jedes Buch entweder in genau einem Regal verfügbar oder verliehen

```
1  :- belongs(B,S), not book(B).
2  :- belongs(B,S), not shelf(S).
3  1 { belongs(B,S) : shelf(S) } 1 :- book(B).
4
5  :- _is(B,X), not book(B).
6  :- _is(B,X), X != lent, X != available(S) : shelf(S).
7  1 { _is(B, available(S)) : shelf(S); _is(B, lent) } 1 :- book(B).
8
9  :- _is(B, available(S)), not belongs(B,S).
10
11 1 { believe(H); doubt(H) } 1 :- H = is(B,X), is(B,X).
12 infer(H) :- H = is(B,X), _is(B,X), not is(B,X).
13 _is(B,X) :- is(B,X), believe(is(B,X)).
14
15 #preference(p, superset) { believe(H) }.
16 #optimize(p).
17
18 #show doubt/1.
19 #show infer/1.
```

Listing 7.7: `bookworld/simple/objectives/correct.lp`

ist. Zeile 9 legt fest, dass ein Buch im richtigen Regal verfügbar ist, falls es überhaupt verfügbar ist.

Führt man die Klasse `complete.lp` gemeinsam mit einer Instanzen aus Abschnitt 7.1.1 aus, erhält man das folgendes Ergebnis: `consistent.lp` hat ein stabiles Modell. `incomplete.lp` hat zwei stabile Modelle, jeweils eines für jede Vervollständigung. Die Instanzen `invalid.lp`, `invalid_duplicate.lp` und `valid_duplicate.lp` haben kein stabiles Modell.

correct.lp

Die Klasse `correct.lp` in Listing 7.7 kann Widersprüche in einer Instanz korrigieren und Wissenslücken schließen. Die Korrektur beschränkt sich jedoch auf das Prädikat `is/2`, um das Beispiel einfach zu halten. Widersprüche im Prädikat `belongs/2` werden nur erkannt[2]. Die Zeilen 1 bis 9 entsprechen den Regeln der Buchwelt aus Listing 7.6. Die Zeilen 11 bis 16 übertragen das bekannte Korrekturschema aus Listing 6.4 auf das Prädikat `is/2`. Die Zeilen 18 und 19 legen fest, dass nur Aussagen angezeigt werden, die bezweifelt oder ergänzt werden. Dies entfernt irrelevante Informationen aus der Ausgabe.

Führt man die Klasse `correct.lp` gemeinsam mit einer Instanz aus Abschnitt 7.1.1 aus, erhält man die in Tabelle 7.2 zusammengefassten Ergebnisse. `consistent.lp` lässt sich eindeutig und ohne Revision korrigieren. Dies ist laut Satz 3.3 zu erwarten. `invalid_duplicate.lp` lässt sich eindeutig korrigieren, da von zwei Alternativen nur eine konsistent ist. Alle anderen Instanzen haben zwei Korrekturen.

2 Das Beispiel geht davon aus, dass der Bibliothekar kein Buch falsch katalogisiert hat.

Instanz `.lp`	stabile Modelle	
`consistent`	1	
`incomplete`	1	`infer(is(hitchhiker,lent))`
	2	`infer(is(hitchhiker,available(scifi)))`
`invalid`	1	`doubt(is(hitchhiker,available(fantasy)))`
		`infer(is(hitchhiker,lent))`
	2	`doubt(is(hitchhiker,available(fantasy)))`
		`infer(is(hitchhiker,available(scifi)))`
`invalid_duplicate`	1	`doubt(is(hitchhiker,available(fantasy)))`
`valid_duplicate`	1	`doubt(is(hitchhiker,lent))`
	2	`doubt(is(hitchhiker,available(scifi)))`

Tabelle 7.2: Die Ergebnisse von `bookworld/simple/objectives/correct.lp`

```
book(
    hitchhiker;
    martian;
    hobbit
).

shelf(
    scifi;
    fantasy
).

belongs(
    0, hitchhiker, scifi;
    0, martian,    scifi;
    0, hobbit,     fantasy
).

is(
    1, hitchhiker, lent;
    2, martian,    available(scifi);
    3, hobbit,     available(fantasy)
).
```

Listing 7.8: `bookworld/prio/instances/consistent.lp`

Instanz `.lp`	stabile Modelle
`consistent`	`1`
`incomplete`	`1 infer(8,is(hitchhiker,lent))`
	`2 infer(8,is(hitchhiker,available(scifi)))`
`invalid`	`1 doubt(1,is(hitchhiker,available(fantasy)))`
	`infer(8,is(hitchhiker,lent))`
	`2 doubt(1,is(hitchhiker,available(fantasy)))`
	`infer(8,is(hitchhiker,available(scifi)))`
`invalid_duplicate`	`1 doubt(2,is(hitchhiker,available(fantasy)))`
`valid_duplicate`	`1 doubt(2,is(hitchhiker,available(scifi)))`

Tabelle 7.3: Die Ergebnisse von `bookworld/prio/objectives/choose.lp`

Um eine Korrektur wie in Abschnitt 6.3 auszuwählen, ist es notwendig, die vorhandenen Aussagen über die Buchwelt zu ordnen. Dazu werden die Prädikate `belongs(B,S)` und `is(B,X)` um eine ganzzahlige Priorität `P` zu `belongs(P,B,S)` und `is(P,B,X)` erweitert[3] und die Instanzen aus Abschnitt 7.1.1 angepasst. Listing 7.8 zeigt die Anpassung für die Instanz `consistent.lp` exemplarisch[4]. Die Katalogisierung des Bibliothekars erhält die höchste Priorität 0. Alle anderen Aussagen erhalten eine Priorität, die mit der Zeilennummer steigt.

choose.lp

Die Klasse `choose.lp` in Listing 7.9 kann Widersprüche in einer Instanz korrigieren, Wissenslücken schließen und eine Korrektur auf Basis der Prioritäten auswählen. Zeile 1 legt den Wertebereich der Prioritäten fest. Die Zeilen 4 bis 12 entsprechen den Regeln der Buchwelt aus Listing 7.6. Die Zeilen 2 und 14 bis 18 ermöglichen es dabei, Prädikate ohne Priorität zu verwenden. Dazu legen die Zeilen 14 und 17 fest, dass ein Prädikat ohne Priorität gültig ist, wenn das Prädikat mit einer beliebigen Priorität gültig ist. Die Zeilen 15 und 18 legen fest, dass ein Prädikat mit der in Zeile 2 festgelegten Priorität gültig ist, wenn es nur ohne Priorität gültig ist. Auf diese Weise legt Zeile 2 fest, welche Priorität eine Aussage hat, die ergänzt wurde. Die Zeilen 20 bis 30 übertragen das bekannte Korrektur- und Auswahlschema aus Listing 6.7 auf die Prädikate `belongs/3` und `is/3`. Die Zeilen 32 und 33 beschränken die Ausgabe auf Aussagen, die bezweifelt oder ergänzt wurden.

Tabelle 7.3 fasst die Ergebnisse der Klasse `choose.lp` für alle priorisierten Instanzen zusammen. Vergleicht man diese Ergebnisse mit den Ergebnissen der Klasse `correct.lp` in Tabelle 7.2 erhält man den

3 Da die Katalogisierung des Bibliothekars eine höhere Priorität als der tatsächliche Standort eines Buches erhalten kann, ist es nun möglich, das Prädikat `belongs(P,B,S)` in die Korrektur einzubinden, ohne die Anzahl der ausgewählten Korrekturen zu stark zu erhöhen.

4 Es wird darauf verzichtet, die Anpassung der anderen Instanzen explizit anzugeben. Das vorhandene Beispiel sollte genügen, um das Prinzip deutlich zu machen.

```
priority(0..8).
priority_autocomplete(8).

:- _belongs(B,S), not book(B).
:- _belongs(B,S), not shelf(S).
1 { _belongs(B,S) : shelf(S) } 1 :- book(B).

:- _is(B,X), not book(B).
:- _is(B,X), X != lent, X != available(S) : shelf(S).
1 { _is(B, available(S)) : shelf(S); _is(B, lent)} 1 :- book(B).

:- _is(B, available(S)), not _belongs(B,S).

_belongs(B,S) :- _belongs(_,B,S).
_belongs(PP,B,S) :- _belongs(B,S), { _belongs(P,B,S) : priority(P
    ), not priority_autocomplete(P) } 0, priority_autocomplete(PP
    ).

_is(B,X) :- _is(_,B,X).
_is(PP,B,S) :- _is(B,S), { _is(P,B,S) : priority(P), not
    priority_autocomplete(P) } 0, priority_autocomplete(PP).

1 { believe(P,H); doubt(P,H) } 1 :- H = belongs(B,S), belongs(P,B
    ,S).
infer(P,H) :- H = belongs(B,S), _belongs(P,B,S), not belongs(P,B,
    S).
_belongs(P,B,S) :- belongs(P,B,S), believe(P, belongs(B,S)).

1 { believe(P,H); doubt(P,H) } 1 :- H = is(B,X), is(P,B,X).
infer(P,H) :- H = is(B,X), _is(P,B,X), not is(P,B,X).
_is(P,B,X) :- is(P,B,X), believe(P, is(B,X)).

#preference(p(P), superset) { believe(P,H) } : priority(P).
#preference(p, lexico) { -P :: **p(P) } : priority(P).
#optimize(p).

#show doubt/2.
#show infer/2.
```

Listing 7.9: bookworld/prio/objectives/choose.lp

4	2	9	1	6	7	8	3	5
7	5	3	8	9	2	6	1	4
8	6	1	5	4	3	9	2	7
2	1	8	7	5	6	3	4	9
3	7	6	9	1	4	5	8	2
5	9	4	3	2	8	7	6	1
9	8	5	2	3	1	4	7	6
6	3	2	4	7	5	1	9	8
1	4	7	6	8	9	2	5	3

Abbildung 7.1: Sudoku `A_3_37_1` mit Lösung

Eindruck, als hätte nur im Fall der Instanz `valid_duplicate.lp` eine Auswahl stattgefunden. Dies ist nicht der Fall. Da das Prädikat `belongs/3` in diesem Beispiel der Korrektur unterliegt, ergibt sich in vielen Fällen die Möglichkeit, die Katalogisierung des Bibliothekars infrage zu stellen. Dies wurde jedoch stets bei der Auswahl verworfen, da die Katalogisierung in diesem Beispiel die höchste Priorität hat. Trotz der Auswahl ist die Korrektur der Instanzen `incomplete.lp` und `invalid.lp` nicht eindeutig. Das liegt daran, dass bei diesen Instanzen nach der Korrektur eine Vervollständigung notwendig ist, die Aussagen ergänzt. Ergänzte Aussagen haben die selbe Prioriät. Eine Auswahl kann daher nicht stattfinden.

7.2 SUDOKU

In diesem Abschnitt soll ein Problem betrachtet werden, das kombinatorisch komplex ist. Auf diese Weise soll die Leistungsfähigkeit des Korrekturverfahrens untersucht werden. Das kombinatorische Logikrätsel Sudoku ist für diese Untersuchung gut geeignet, da es einfache, klar definierte Regeln hat und die Berechnung einer Lösung NP-vollständig in der Größe des Spielfeldes ist [YS03].

Regeln

Ein Sudoku der Ordnung n besteht aus $n \times n$ Blöcken, die in $n \times n$ Felder unterteilt sind. Jedes Feld dieser $n^2 \times n^2$-Matrix kann eine Zahl zwischen 1 und n^2 enthalten. Einige Felder sind bereits ausgefüllt; sie enthalten eine Vorgabe. Ziel des Spiels ist es, alle übrigen Felder so auszufüllen, dass keine Zahl in einer Zeile, Spalte oder einem Block mehrfach vorkommt. Abbildung 7.1 zeigt ein Sudoku der Ordnung 3. Es macht 37 Vorgaben und hat eine eindeutige Lösung[5].

5 Es wird in dieser Arbeit nicht gefordert, dass ein Sudoku immer eine eindeutige Lösung hat.

```
1 dim(3).
2 fill(1,2,2). fill(1,5,6). fill(1,9,5). fill(2,1,7). fill(2,3,3).
3 fill(2,5,9). fill(2,6,2). fill(3,1,8). fill(3,5,4). fill(3,6,3).
4 fill(3,7,9). fill(3,8,2). fill(3,9,7). fill(4,4,7). fill(4,5,5).
5 fill(5,6,4). fill(5,8,8). fill(5,9,2). fill(6,1,5). fill(6,2,9).
6 fill(6,3,4). fill(6,4,3). fill(6,5,2). fill(7,1,9). fill(7,2,8).
7 fill(7,3,5). fill(7,4,2). fill(7,6,1). fill(7,7,4). fill(7,9,6).
8 fill(8,2,3). fill(8,4,4). fill(8,6,5). fill(8,7,1). fill(8,9,8).
9 fill(9,2,4). fill(9,6,9).
```

Listing 7.10: `sudoku/instances/lp/A_3_37_1.lp`

Ein Sudoku, das keine Lösung hat, wird in diesem Abschnitt als inkonsistent bezeichnet. Es wird gezeigt, dass es mit den Verfahren aus dieser Arbeit möglich ist, die Vorgabe eines inkonsistenten Sudokus sinnvoll zu korrigieren.

7.2.1 *Instanzen*

Prädikate

Um die Vorgabe eines Sudokus in ASP zu modellieren, sind zwei Prädikate notwendig. Das Prädikat `dim(N)` gibt die Ordnung `N` des Sudokus an und muss genau einmal verwendet werden. Das Prädikat `fill(I,J,V)` legt fest, dass das Feld in Zeile `I` und Spalte `J` des Sudokus den Wert `V` enthält. Das Programm in Listing 7.10 modelliert alle

A_3_37_1

37 Vorgaben des Sudokus `A_3_37_1` aus Abbildung 7.1. Es wird in Abschnitt 7.2.2 als Instanz verwendet.

Im Folgenden werden vier weitere Sudokus in Matrixdarstellung vorgestellt, die ebenfalls als Instanzen dienen sollen. Es wird darauf verzichtet, die Modellierung in ASP anzugeben, da sie leicht hergeleitet werden kann und keinen Mehrwert für den menschlichen Leser hat.

A_3_36_2

Das Sudoku `A_3_36_2` in Abbildung 7.2 entsteht aus `A_3_37_1`, wenn man die Vorgabe in Zeile 1 Spalte 9 entfernt. Das Sudoku ist nicht eindeutig lösbar. Es hat zwei Lösungen.

A_3_38_0e

Das Sudoku `A_3_38_0e` in Abbildung 7.3 entsteht aus `A_3_37_1`, wenn man in Zeile 1 Spalte 1 den Wert 2 als Vorgabe hinzufügt. Das Sudoku ist inkonsistent, hat also keine Lösung. Die Inkonsistenz ist offensichtlich, da sie direkt aus der Vorgabe abgelesen werden kann. Zeile 1 Spalte 2 gibt ebenfalls den Wert 2 vor.

A_3_38_0h

Das Sudoku `A_3_38_0h` in Abbildung 7.4 entsteht aus `A_3_37_1`, wenn man in Zeile 1 Spalte 1 den Wert 1 als Vorgabe hinzufügt. Das Sudoku ist ebenfalls inkonsistent. Die Inkonsistenz ist jedoch versteckt. Es ist notwendig, das Sudoku zu lösen, um die Inkonsistenz zu erkennen.

A_3_39_0hx

Das Sudoku `A_3_39_0hx` in Abbildung 7.5 entsteht aus `A_3_38_0h`, wenn man in Zeile 9 Spalte 7 den Wert 3 als Vorgabe hinzufügt. Es ist ebenfalls versteckt inkonsistent, hat jedoch keine eindeutige Korrektur.

	2			6				x
7		3		9	2			
8				4	3	9	2	7
			7	5				
					4		8	2
5	9	4	3	2				
9	8	5	2		1	4		6
	3		4		5	1		8
	4				9			

Abbildung 7.2: Sudoku `A_3_36_2`

2	2			6				5
7		3		9	2			
8				4	3	9	2	7
			7	5				
					4		8	2
5	9	4	3	2				
9	8	5	2		1	4		6
	3		4		5	1		8
	4				9			

Abbildung 7.3: Sudoku `A_3_38_0e`

1	2			6				5
7		3		9	2			
8				4	3	9	2	7
			7	5				
					4		8	2
5	9	4	3	2				
9	8	5	2		1	4		6
	3		4		5	1		8
	4				9			

Abbildung 7.4: Sudoku `A_3_38_0h`

1	2			6				5
7		3		9	2			
8				4	3	9	2	7
			7	5				
					4		8	2
5	9	4	3	2				
9	8	5	2		1	4		6
	3		4		5	1		8
	4				9	3		

Abbildung 7.5: Sudoku `A_3_39_0hx`

Die Eigenschaften eines Sudokus können in dieser Arbeit auch aus dem Namen abgelesen werden. Das `A` in `A_3_39_0hx` dient als zusätzlicher Bezeichner. Die `3` steht für die Ordnung. Die `39` steht für die Anzahl der Vorgaben. Die `0` steht für die Anzahl der Lösungen. Das `h` bedeutet, dass die Inkonsistenz versteckt (engl. *hidden*) ist. Im Gegensatz dazu bedeutet ein `e`, dass die Inkonsistenz offensichtlich (engl. *evident*) ist. Das `x` steht für ein inkonsistentes Sudoku, dass keine eindeutige Korrektur hat.

Bedeutung des Namens

7.2.2 *Klassen*

Bevor man ein Sudoku korrigieren kann, benötigt man eine Modellierung der Regeln in ASP. Eine Modellierung der Regeln ist geeignet, Widersprüche zu erkennen und Wissenslücken zu schließen. Sie kann also ein konsistentes Sudoku lösen.

Die Klasse `complete.lp` in Listing 7.11 modelliert die Regeln des Sudokus. Dazu werden in Zeile 1 bis 3 drei Hilfsprädikate definiert. Das Prädikat `val(N)` legt fest, welche Zahlen ein Feld enthalten kann. Es ist genau dann wahr, wenn `N` zwischen 1 und n^2 liegt. Genau in

`complete.lp` Hilfsprädikate

```
1  val(1..N*N) :- dim(N).
2  pos(I,J) :- val(I), val(J).
3  box(I,II) :- val(I), val(II), (I-1) / N = (II-1) / N, dim(N), I
       != II.
4
5  :- fill(I,J,V), fill(II,J,V), I != II.
6  :- fill(I,J,V), fill(I,JJ,V), J != JJ.
7  :- fill(I,J,V), fill(II,JJ,V), box(I,II), box(J,JJ).
8
9  1 { fill(I,J,V) : val(V) } 1 :- pos(I,J).
10
11 #show dim/1.
12 #show fill/3.
```

Listing 7.11: `sudoku/objectives/complete.lp`

diesem Fall gibt es auch eine Zeile `N` und eine Spalte `N`. Das Prädikat `pos(I,J)` legt fest, welche Positionen es in der Matrix gibt. Es ist genau dann wahr, wenn es eine Zeile `I` und eine Spalte `J` gibt. Das Prädikat `box(I,II)` überprüft in einer Dimension, ob zwei Felder zu einem Block gehören. Ein Feld in Zeile `I` und Spalte `J` und ein Feld in Zeile `II` und Spalte `JJ` gehören genau dann zu einem Block, wenn `box(I,II)` und `box(J,JJ)` wahr sind. Um die übrigen Regeln zu vereinfachen, ist das Prädikat `box(I,II)` jedoch irreflexiv, `box(I,I)` ist also für jedes `I` falsch. Die Zeilen 5 bis 7 modellieren die Konsistenzregeln des Sudokus. Zeile 5 legt fest, dass eine Zahl in einer Zeile nicht mehrfach vorkommen kann. Zeile 6 legt fest, dass eine Zahl in einer Spalte nicht mehrfach vorkommen kann. Zeile 7 legt fest, dass eine Zahl in einem Block nicht mehrfach vorkommen kann. Die Regel in Zeile 9 legt fest, dass jedes Feld genau eine Zahl enthalten muss. Sie definiert, wann ein Sudoku vollständig gelöst ist[6]. Die Regeln in Zeile 11 und 12 unterdrücken die Ausgabe der Hilfsprädikate.

Führt man die Klasse `complete.lp` gemeinsam mit einer Instanz aus Abschnitt 7.2.1 aus, erhält man das erwartete Ergebnis. Das Sudoku `A_3_37_1` hat eine Lösung. Das Sudoku `A_3_36_2` hat zwei Lösungen. Die anderen Sudokus haben keine Lösung.

correct.lp

Die Klasse `correct.lp` in Listing 7.12 kann inkonsistente Sudokus korrigieren und vervollständigen. Die Zeilen 1 bis 9 entsprechen den Regeln aus Listing 7.11, die zur Lösung eines Sudokus nötig sind. Die Zeilen 11 bis 16 übertragen das bekannte Korrekturschema aus Listing 6.4 auf das Prädikat `fill/2`. Dabei wird in Zeile 15 abweichend vom Schema eine Präferenzrelation vom Typ `more(cardinality)` statt vom Typ `superset` verwendet, um die Anzahl möglicher Korrekturen

6 Man könnte meinen, dass die Regel in Zeile 9 nicht nötig ist, um einen Widerspruch zu erkennen. Das ist nicht ganz richtig. Ohne die Regeln in Zeile 9 kann das Programm offensichtliche Inkonsistenzen erkennen. Mit der Regel in Zeile 9 kann es offensichtliche und versteckte Inkonsistenzen erkennen.

```
1  val(1..N*N) :- dim(N).
2  pos(I,J) :- val(I), val(J).
3  box(I,II) :- val(I), val(II), (I-1) / N = (II-1) / N, dim(N), I
       != II.
4
5  :- _fill(I,J,V), _fill(II,J,V), I != II.
6  :- _fill(I,J,V), _fill(I,JJ,V), J != JJ.
7  :- _fill(I,J,V), _fill(II,JJ,V), box(I,II), box(J,JJ).
8
9  1 { _fill(I,J,V) : val(V) } 1 :- pos(I,J).
10
11 1 { believe(S); doubt(S) } 1 :- S = fill(I,J,V), fill(I,J,V).
12 infer(S) :- S = fill(I,J,V), _fill(I,J,V), not fill(I,J,V).
13 _fill(I,J,V) :- fill(I,J,V), believe(fill(I,J,V)).
14
15 #preference(p, more(cardinality)) { believe(S) }.
16 #optimize(p).
17
18 #show dim/1.
19 #show doubt/1.
20 #show infer/1.
```

Listing 7.12: `sudoku/objectives/correct.lp`

deutlich zu begrenzen. `more(cardinality)` wählt aus allen Korrekturen jene aus, die möglichst viele Vorgaben machen[7]. Die Zeilen 19 und 20 legen fest, dass nur Aussagen angezeigt werden, die bezweifelt oder ergänzt werden.

Führt man die Klasse `correct.lp` gemeinsam mit einer Instanz aus Abschnitt 7.2.1 aus, erhält man die in Tabelle 7.4 zusammengefassten Ergebnisse. Das Programm löst die Sudokus `A_3_37_1` und `A_3_36_2` wie bisher. Es identifiziert die Fehler, die absichtlich in `A_3_38_0e` und `A_3_38_0h` eingebaut wurden, korrigiert sie und berechnet eine Lösung, die der Lösung von `A_3_37_1` entspricht. Das Programm ist nicht in der Lage, den Fehler in `A_3_38_0hx` eindeutig zu identifizieren. Es erkennt ihn als dritte von drei Möglichkeiten.

Statistiken

Führt man *asprin* mit der Kommandozeilenoption `--stats` aus, erhält man eine Reihe von Statistiken, die während des Lösungsprozesses erhoben werden. Tabelle 7.5 fasst die Statistiken der Klasse `correct.lp` für die Instanzen in Abschnitt 7.2.1 zusammen. Zeile 1 zeigt die Statistiken der Klasse `bookworld/simple/objectives/correct.lp` für die Instanz `bookworld/simple/instances/invalid.lp` zum Vergleich. Die Zeilen in Grau zeigen die Statistiken der Klasse `complete.lp`, ebenfalls zum Vergleich.

7 In der Notation aus Beispiel 5.1 gilt: $(M_1, M_2) \in \text{superset}(\mathbb{M}) \Rightarrow M_1 \supset M_2 \Rightarrow |M_1| > |M_2| \Rightarrow (M_1, M_2) \in \text{more}(\text{card})(\mathbb{M})$.

Instanz	stabile Modelle
A_3_37_1	1 infer(fill(1,1,4)) ... infer(fill(9,9,3))
A_3_36_2	1 infer(fill(1,1,4)) ... infer(fill(9,9,5))
	2 infer(fill(1,1,4)) ... infer(fill(9,9,3))
A_3_38_0e	1 doubt(fill(1,1,2))
	infer(fill(1,1,4)) ... infer(fill(9,9,3))
A_3_38_0h	1 doubt(fill(1,1,1))
	infer(fill(1,1,4)) ... infer(fill(9,9,3))
A_3_39_0hx	1 doubt(fill(1,1,1)) doubt(fill(5,9,2))
	infer(fill(1,1,4)) ... inter(fill(5,1,6)) ...
	infer(fill(5,9,3)) ... infer(fill(9,9,2))
	2 doubt(fill(1,1,1)) doubt(fill(5,9,2))
	infer(fill(1,1,4)) ... inter(fill(5,1,2)) ...
	infer(fill(5,9,3)) ... infer(fill(9,9,2))
	3 doubt(fill(1,1,1)) doubt(fill(9,7,3))
	infer(fill(1,1,4)) ... infer(fill(9,9,3))

Tabelle 7.4: Die Ergebnisse von `sudoku/objectives/correct.lp`

Instanz	Time	Choices	Conflicts	Rules	Atoms	Bodies
Buchwelt	0,112	9	2	170	129	79
	0,002	0	1	50	39	21
A_3_37_1	0,242	2455	790	18437	3578	9155
	0,017	0	0	6093	1130	2859
A_3_36_2	0,323	2893	863	19100	4246	9691
	0,015	1	0	6335	1131	3004
A_3_38_0e	0,262	3482	1311	18377	3602	9175
	0,012	0	1	2420	0	1078
A_3_38_0h	0,238	3693	1228	18099	3324	9007
	0,014	0	1	5756	0	2666
A_3_39_0hx	0,327	4227	1862	18728	3948	9421
	0,013	0	1	5604	0	2575

Tabelle 7.5: Die Statistiken von `sudoku/objectives/correct.lp`

Bedeutung der Einträge

Die Spalte „Time“ misst die Laufzeit des Programms in Sekunden. „Choices“ zählt, wie oft der Wahrheitsstatus eines Atoms gewählt werden musste. „Conflicts“ gibt die Anzahl der Widersprüche an, die während des Lösungsprozesses entstanden sind. Die Spalten „Rules“, „Atoms“ und „Bodies“ geben die Anzahl der Regeln, Atome und Rümpfe in der Grundinstanz des Programms an.

Laufzeit

Vergleicht man die Laufzeiten, fällt zunächst auf, dass die Berechnung einer Korrektur deutlich mehr Zeit als die Berechnung einer Vervollständigung benötigt. Dieses Ergebnis war zu erwarten. Es ist jedoch sehr überraschend, dass die Korrektur eines inkonsistenten Sudokus weniger als eine halbe Sekunde dauert, da es rund $6,671 * 10^{21}$ konsistente Sudokus gibt [FJ05]. Ein Algorithmus, der das Konzept aus Kapitel 3 naiv implementiert, müsste alle konsistenten Sudokus berechnen und das ähnlichste auswählen.

Betrachtet man die Anzahl der „Choices“ und „Conflicts“ fällt auf, dass es fast nichts zu tun gibt, um die gewählten Instanzen zu vervollständigen oder ihre Inkonsistenz zu beweisen. Der Algorithmus musste genau einmal eine Auswahl treffen, wenn die Instanz mehr als eine Vervollständigung hat. Er hat genau einmal einen Konflikt erkannt, wenn die Instanz inkonsistent ist. Diese Beobachtung ist ein Indiz dafür, dass die gewählten Instanzen nicht sehr kompliziert sind. Daher stellt sich die Frage, wie eine komplizierte Instanz aussieht und was dies für die Korrektur bedeutet. Der Autor vermutet, dass eine komplizierte Instanz sehr viele Vervollständigungen hat oder sehr viele Inkonsistenzen[8] enthält.

Vergleich mit der Buchwelt

Die Korrektur eines Sudokus benötigt in dieser Auswertung im Durchschnitt rund 109-mal mehr Regeln, 29-mal mehr Atome und 117-mal mehr Rümpfe, aber nur doppelt bis dreimal mehr Zeit als die Korrektur der Buchwelt. Diese Beobachtung ist erstaunlich, da die Verhältnisse der Regeln, Atome und Rümpfe bei der Vervollständigung ungefähr den Verhältnissen bei der Korrektur entsprechen, die Vervollständigung eines Sudokus im Durchschnitt aber 35-mal mehr Zeit als die Vervollständigung der Buchwelt benötigt. Das lässt vermuten, dass die Laufzeit der Korrektur eine relativ große Konstante enthält[9]. Es ist daher auf Basis der vorhandenen Daten unmöglich zu sagen, wie schnell eine Korrektur in der Praxis tatsächlich ist, wenn man die Größe der Eingabe variiert. Vermutlich lässt sich diese Frage beantworten, wenn man Sudokus vergleicht, deren Ordnungen verschieden, aber deutlich größer als 3 sind.

8 Die Anzahl der Inkonsistenzen sollte sich natürlich nur auf die Komplexität der Korrektur auswirken. Der Nachweis, dass die Instanz inkonsistent ist, sollte weiterhin vergleichsweise einfach sein.

9 Tatsächlich gibt es Unterschiede bei der Implementierung: Die Vervollständigung verwendet nur das C++-Programm *clingo*. Die Korrektur benötigt das Python-Programm *asprin*, das eine Schnittstelle zu *clingo* verwendet.

8 AUSBLICK

Zusammenfassung

Diese Arbeit beschreibt das Konzept für ein theoretisches Verfahren, das Widersprüche korrigieren kann. Es basiert auf der Hoffnung, dass man stets eine konsistente Wissensbasis findet, die der zu korrigierenden Wissensbasis ähnlich ist. Das Verfahren hat viele erwünschte Eigenschaften, die theoretisch bewiesen wurden: Es ist relativ unabhängig vom Inkonsistenzbegriff, der sich nicht eindeutig definieren lässt. Es minimiert die Menge der bezweifelten und ergänzten Aussagen und maximiert die Menge der geglaubten Aussagen. Es verändert eine konsistente Wissensbasis nicht. Es ergänzt keine Aussagen, wenn die verwendete Logik monoton ist. Es garantiert die Existenz einer Korrektur für jede endliche Wissensbasis. Mit etwas Mühe lässt sich sogar Eindeutigkeit und Vollständigkeit herstellen.

Diese Arbeit erklärt im Detail, wie man eine Korrektur praktisch in ASP umsetzen kann. Die Arbeit beschreibt eine Umformung, die ein Programm, das Widersprüche erkennt und Wissenslücken schließt, in ein Programm übersetzt, das Widersprüche korrigiert und eine Korrektur auswählt. Sie stellt zwei Beispiele vor, die die Umsetzung in ASP nutzen. Die Buchwelt motiviert, dass Korrekturen in der Robotik sinnvoll sind. Sie ist einfach, aber umfangreich genug, um bei der Korrektur komplexe Phänomene zu erzeugen. Die Korrektur inkonsistenter Sudokus stellt die Leistungsfähigkeit des Verfahrens auf die Probe. Es ist möglich, Fehler, die zuvor mit Absicht in ein Sudoku eingebaut wurden, eindeutig und schnell zu identifizieren.

Offene Fragen

Einige Fragen mussten in dieser Arbeit offen bleiben: Hat jede unendliche Wissensbasis eine Korrektur? Wie gut sind die Korrekturen? Würde ein Mensch ähnlich korrigieren? Wie steht es um die Laufzeit- und Speicherkomplexität der theoretischen Verfahren? Welche Auswirkungen hat dies auf die praktische Umsetzung in ASP? Die Umformung eines ASP-Programms, das Widersprüche erkennt und Wissenlücken schließt, in ein Programm, das Widersprüche korrigiert und eine Korrektur auswählt, ist sehr aufwändig und anfällig für Programmierfehler. Ist es möglich eine Spracherweiterung zu entwickeln, die diese Aufgabe übernimmt?

Fazit

Falsche Informationen treten überall auf. Der Mensch lügt oder irrt sich. Sensoren machen Messfehler. Gespeicherte Informationen sind häufig veraltet. Das erzeugt Widersprüche. Auf der anderen Seite steht ein enormer Bedarf an zuverlässigen Informationen. Viele Algorithmen sind darauf angewiesen. Es ist erstaunlich, dass es überhaupt möglich ist, ein abstrakten Verfahren zu entwickeln, das Widersprüche

korrigieren kann. Das ist eine große Chance, um Roboter zu bauen, die noch besser mit ihrer Umgebung interagieren.

ABKÜRZUNGEN

ASP Answer Set Programming

Potassco Potsdam Answer Set Solving Collection

ABBILDUNGEN

TABELLEN

LISTINGS

LITERATUR

[BE99] Gerhard Brewka und Thomas Eiter. „Preferred answer sets for extended logic programs“. In: *Artificial Intelligence* 109.1–2 (1999), S. 297–356. DOI: 10.1016/S0004-3702(99)00015-6 (zitiert auf S. 31).

[BET11] Gerhard Brewka, Thomas Eiter und Mirosław Truszczyński. „Answer Set Programming at a Glance“. In: *Communications of the ACM* 54.12 (2011), S. 92–103. DOI: 10.1145/2043174.2043195 (zitiert auf S. 17).

[BG02] Salem Benferhat und Laurent Garcia. „Handling Locally Stratified Inconsistent Knowledge Bases“. In: *Studia Logica* 70.1 (2002), S. 77–104. DOI: 10.1023/A:1014606325783 (zitiert auf S. 2).

[BHS05] Leopoldo Bertossi, Anthony Hunter und Torsten Schaub, Hrsg. *Introduction to Inconsistency Tolerance*. Springer, 2005. DOI: 10.1007/978-3-540-30597-2_1 (zitiert auf S. 1, 5).

[Bre+15] Gerhard Brewka u. a. „asprin: Customizing Answer Set Preferences without a Headache“. In: *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI-2015. Hrsg. von Blai Bonet und Sven Koenig. AAAI Press, 2015, S. 1467–1474 (zitiert auf S. 31–32).

[Bre89] Gerhard Brewka. „Preferred Subtheories: An Extended Logical Framework for Default Reasoning“. In: *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*. IJCAI-89. Bd. 2. Morgan Kaufmann Publishers, 1989, S. 1043–1048 (zitiert auf S. 2).

[DS03] James P. Delgrande und Torsten Schaub. „A consistency-based approach for belief change“. In: *Artificial Intelligence* 151.1–2 (2003), S. 1–41. DOI: 10.1016/S0004-3702(03)00111-5 (zitiert auf S. 2).

[DS07] James P. Delgrande und Torsten Schaub. „A consistency-based framework for merging knowledge bases“. In: *Journal of Applied Logic* 5.3 (2007), S. 459–477. DOI: 10.1016/j.jal.2006.03.005 (zitiert auf S. 2).

[Ert16] Wolfgang Ertel. *Grundkurs Künstliche Intelligenz. Eine praxisorientierte Einführung*. 4. Aufl. Springer Vieweg, 2016. DOI: 10.1007/978-3-658-13549-2 (zitiert auf S. 5).

[FJ05] Bertram Felgenhauer und Frazer Jarvis. *Enumerating possible Sudoku grids*. 2005. URL: http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf (besucht am 09. 12. 2018) (zitiert auf S. 57).

[Fre98] Michael Freund. „Preferential reasoning in the perspective of Poole default logic“. In: *Artificial Intelligence* 98.1–2 (1998), S. 209–235. DOI: 10.1016/S0004-3702(97)00053-2 (zitiert auf S. 2).

[Geb+12] Martin Gebser u. a. *Answer Set Solving in Practice*. Hrsg. von Ronald J. Brachman, William W. Cohen und Thomas G. Dietterich. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012. DOI: 10.2200/S00457ED1V01Y201211AIM019 (zitiert auf S. 17, 21–25).

[Geb+17] Martin Gebser u. a. *Potassco User Guide*. Version 2.1.0. 5. Okt. 2017. URL: https://github.com/potassco/guide/releases/download/v2.1.0/guide.pdf (besucht am 19. 10. 2018) (zitiert auf S. 27, 33).

[GL88] Michael Gelfond und Vladimir Lifschitz. „The Stable Model Sematics for Logic Programming“. In: *Proceedings of the Fifth International Conference and Symposium of Logic Programming*. ICLP'88. Hrsg. von Robert Kowalski und Kenneth A. Bowen. MIT Press, 1988, S. 1070–1080 (zitiert auf S. 18).

[Kan84] Immanuel Kant. „Beantwortung der Frage: Was ist Aufklärung?“ In: *Berlinische Monatsschrift* 12.1 (1784), S. 481–494 (zitiert auf S. 1).

[Ker10] Gabriele Kern-Isberner. *Commonsense Reasoning*. 2010. URL: https://ls1-www.cs.tu-dortmund.de/images/courses/ie/sose17/cr/Skript/cr_script.pdf (besucht am 19. 10. 2018) (zitiert auf S. 4).

[KL92] Michael Kifer und Eliezer L. Lozinskii. „A Logic for Reasoning with Inconsistency“. In: *Journal of Automated Reasoning* 9.2 (1992), S. 179–215. DOI: 10.1007/BF00245460 (zitiert auf S. 2).

[MM76] Harry McGurk und John MacDonald. „Hearing Lips and seeing voices“. In: *Nature* 264.1 (1976), S. 746–748. DOI: 10.1038/264746a0 (zitiert auf S. 1).

[Poo88] David Poole. „A Logical Framework for Default Reasoning“. In: *Artificial Intelligence* 36.1 (1988), S. 27–47. DOI: 10.1016/0004-3702(88)90077-X (zitiert auf S. 2).

[Res64] Nicholas Rescher. *Hypothetical Reasoning*. Hrsg. von L. E. J. Brouwer, E. W. Beth und A. Heyting. Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Company, 1964 (zitiert auf S. 1, 9, 14).

[RN10] Stuart Russel und Peter Norvig. *Artificial Intelligence. A Modern Approach*. 3. Aufl. Pearson, 2010 (zitiert auf S. 10, 43).

[Schoo] Uwe Schöning. *Logik für Informatiker*. 5. Aufl. Spektrum Akademischer Verlag, 2000 (zitiert auf S. 4, 20).

[Wal11] Christian Wallmann. „Theorie der Konsequenzoperationen und Grundbegriffe der Logik“. In: *Kriterion – Journal of Philosophy* 25.1 (2011), S. 64–77 (zitiert auf S. 3–4, 6–7).

[YS03] Takayuki Yato und Takahiro Seta. „Complexity and Completeness of Finding Another Solution and Its Application to Puzzles“. In: *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A.5 (2003), S. 1052–1060 (zitiert auf S. 51).

[Zie17] Martin Ziegler. *Mathematische Logik*. Hrsg. von Martin Brokate u. a. 2. Aufl. Mathematik Kompakt. Birkhäuser, 2017. DOI: 10.1007/978-3-319-44180-1 (zitiert auf S. 15).

ERKLÄRUNG

Ich versichere, dass ich die eingereichte Bachelorarbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Osnabrück, 18. Dezember 2018

Tobias Stolzmann