

Grundlagen der Künstlichen Intelligenz

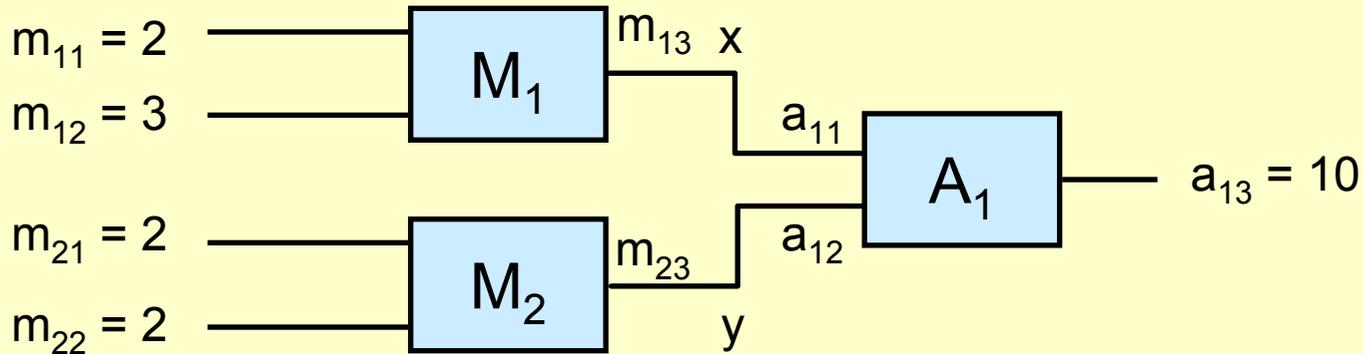
Sebastian Iwanowski
FH Wedel

Kap. 2:
Logische Grundlagen der KI

2.1: Das Prinzip der logischen Programmierung

Ein vereinfachtes Diagnose-Beispiel

Systemstruktur



Komponentenmodelle

- Multiplizierer M_i : $y_i = 1 \Rightarrow m_{i3} = m_{i1} * m_{i2}$; $y_i = 1 \vee y_i = 0$
- Addierer A_i : $x_i = 1 \Rightarrow a_{i3} = a_{i1} + a_{i2}$; $x_i = 1 \vee x_i = 0$

Frage:

In welcher Form braucht man die Constraints für das GDE-Prinzip ?

KI-Antwort:

Beschreibe das Diagnoseproblem mit einer aussagenlogischen Formel !

Wdh.: Aussagenlogik

Was ist eine Aussage ?

- **Eine Aussage ist ein beliebiges Objekt.**
- **In der Aussagenlogik sind Aussagen unteilbar.**
 - Wegen der Unteilbarkeit heißen Aussagen auch *Atome*
 - Da sie wegen der Unteilbarkeit sinnvollerweise mit Buchstaben abgekürzt werden, nennt man Aussagen auch *Literale*

Was ist ein Wahrheitswert ?

- **Ein Wahrheitswert ist ein Element aus einer zweielementigen Menge (z.B. dargestellt als $\{w, f\}$).**

Was macht die Aussagenlogik ?

- **Die Aussagenlogik beschäftigt sich mit Funktionen, die jeder Aussage einen Wahrheitswert zuordnen.**
 - Solche Funktionen heißen *binäre Funktionen*

Wdh.: Aussagenlogische Verknüpfungen

Einstelliger Operator:

- Negation (\neg)

Zweistellige Operatoren:

- Konjunktion (\wedge), Disjunktion (\vee), Implikation (\rightarrow), Äquivalenz (\leftrightarrow)

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
w	w	f	w	w	w	w
w	f	f	f	w	f	f
f	w	w	f	w	w	f
f	f	w	f	f	w	w

Syntaktische Ersetzungsregeln:

$$p \rightarrow q \Leftrightarrow \neg p \vee q$$

Ersetzen der Implikation durch \neg und \vee

$$p \leftrightarrow q \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$$

Ersetzen der Äquivalenz durch Implikationen

$$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$$

$$\neg\neg p \Leftrightarrow p$$

$$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$$

deMorgansche Regeln

Doppelte Negation

Folgerung aus den Ersetzungsregeln:

- Für die Darstellung jeder logischen Verknüpfung reichen die Operatoren \neg , \wedge und \vee aus !
- Man kann sogar auf einen dieser 3 Operatoren verzichten !
(auf welchen ?)

Wdh.: Aussagenlogische Formeln

- Eine aussagenlogische **Formel** ist eine Verknüpfung von endlich vielen Literalen mit aussagenlogischen Operatoren.
 - Die Literale sind als Variable aufzufassen, die genau einen von 2 Werten annehmen können
 - Eine **Belegung einer Formel** ist eine Zuweisung von Wahrheitswerten an die Literale derart, dass dieselben Literale immer denselben Wahrheitswert erhalten.
 - Eine Formel heißt **erfüllbar**, wenn es eine Belegung gibt derart, dass die Formel wahr ist.
 - Das Erfüllbarkeitsproblem ist in der Aussagenlogik immer lösbar, da man alle (endlich vielen) Belegungsmöglichkeiten der Variablen nur nacheinander auszuprobieren braucht.
 - Leider dauert das Lösen durch Ausprobieren sehr lange (exponentiell in der Anzahl der Variablen). Es ist bis heute kein effizienterer Algorithmus bekannt.
- Das Problem ist NP-vollständig !***

Aussagenlogik als Beschreibungssprache

Nach dieser Wiederholung wissen wir, wie das Diagnoseproblem als aussagenlogisches Problem formuliert werden kann:

Warum sollten wir das Diagnoseproblem als aussagenlogisches Problem formulieren ?

- Wegen der allgemeingültigen und international einheitlichen Ausdrucksmöglichkeit
- Um das Diagnoseproblem mit einem allgemeinen aussagenlogischen Inferenzmechanismus zu lösen

Das ist der klassische KI-Ansatz zur Realisierung wissensbasierter Systeme !

Bsp. für die mangelnde Ausdruckskraft der Aussagenlogik

In der Aussagenlogik haben alle Formeln nur zwei mögliche Werte für die Variablen.

Formeln mit Variablen, die einen unendlichen Wertebereich haben, sind also nicht enthalten.

- Anm.: Für solche Formeln versagt die Methode des Ausprobierens aller Möglichkeiten.

Bsp.: Finde $x, y \in \mathbf{R}$ mit folgenden Eigenschaften:

$$(2 < x < 4) \wedge (0 < y < 6) \wedge (x + y > 7) \wedge (x \cdot y < 10)$$

Kommen solche Probleme bei wissensbasierten Systemen überhaupt vor ?

Ja, sehr oft !

Wdh.: Prädikatenlogik

Die Prädikatenlogik (1. Stufe) erweitert die Aussagenlogik um folgende Elemente:

- **Prädikate**
 - Aussagen, die von Variablen abhängen
(wenn es von k Variablen abhängt,
dann heißt das Prädikat k -stellig)
- **Variable**
 - entsprechen den Literalen der Aussagenlogik,
können aber beliebig viele Werte annehmen
- **Funktionen**
 - eindeutige Zuordnungen, die von Variablen abhängen
(wenn sie von k Variablen abhängt,
dann heißt die Funktion k -stellig)
 - 0-stellige Funktionen sind Konstante
- **Quantoren**
 - Existenzquantor (\exists) und Allquantor (\forall)
 - Quantoren werden nur auf Variablen angewendet (sonst nicht 1. Stufe)

Wdh.: Prädikatenlogische Formeln

- Eine prädikatenlogische **Formel** ist eine Verknüpfung von endlich vielen Variablen, Funktionen und Prädikaten mit aussagenlogischen Operatoren oder Quantoren. Die Quantoren dürfen sich bei Formeln **1. Stufe** nur auf Variable beziehen.

Bsp.: Formel $\varphi = \forall x (R(y, z) \wedge \exists y (\neg P(y, x) \vee R(y, z)))$

Grüne Vorkommen von y und z sind **frei**.

Rote Vorkommen von x , y und z sind **gebunden**.

Geschlossene Formeln:

Formeln, die keine freien Variablen enthalten.

Offene Formeln:

Formeln, die keine gebundenen Variablen enthalten.

Wdh.: Prädikatenlogische Formeln

- Eine **Belegung einer Formel** ist eine Zuweisung von *Werten aus festgelegten Definitionsbereichen an die freien Variablen* derart, dass dieselben Variablen immer denselben Wert erhalten.
- Eine Formel heißt **erfüllbar**, wenn es eine Belegung gibt derart, dass die Formel wahr ist.



- Das Erfüllbarkeitsproblem ist in der Prädikatenlogik **nicht entscheidbar**, d.h. kein Algorithmus kann jemals in der Lage sein, von jeder Formel zu entscheiden, ob sie erfüllbar ist oder nicht.

Das allgemeine Problem ist unlösbar !

Gibt es dennoch einen Ausweg ?

Ja, löse ein spezielleres Problem !

Logische Programmiersprachen

Prinzip:

Verwendung eines Algorithmus,
der jedes logisch formulierte Problem löst.

*Anm.: Problem muss zu einer
speziellen Klasse gehören !*

Vorteil:

Logisches Programmieren reduziert sich auf das Formulieren des Problems
in einer logischen Beschreibungssprache.



- Damit ist der Algorithmenentwurf automatisiert.



- Die Lösung ist nicht auf das spezielle Problem angepasst, kann also sehr ineffizient sein.

Hilfstechnik zur Beschleunigung des Verfahrens:



→ Resolution

$$(p \vee q) \wedge (r \vee \neg q) \Rightarrow (p \vee r)$$

Resolventenbildung



- Im ungünstigen Fall bringt das gegenüber Ausprobieren keine Verbesserung.

Logische Programmiersprachen

Aufgabe für den Interpreter:

Eigentliches Ziel: Konstruktionsaufgabe

*erst recht nicht lösbar für
allgemeine Formeln*

Gegeben eine Menge \mathcal{F} von logischen Formeln. Bestimme alle Formeln F , die aus \mathcal{F} folgen.

Hilfsziel: Verifikationsaufgabe

nicht lösbar für allgemeine Formeln

Gegeben eine Menge \mathcal{F} von logischen Formeln und eine (neue) logische Formel F .
Bestimme, ob F aus \mathcal{F} folgt.

Äquivalente Formulierungen zur Verifikationsaufgabe:

- 1) Gegeben eine Menge \mathcal{F} von logischen Formeln und eine (neue) logische Formel F . Bestimme, ob die Formelmenge $\{\neg F\} \cup \mathcal{F}$ widersprüchlich ist.
- 2) Gegeben eine Menge \mathcal{F} von logischen Formeln. Bestimme, ob sie widersprüchlich ist.

entspricht Erfüllbarkeitsproblem: nicht lösbar für allgemeine Formeln

Möglichkeit zur Vereinfachung des Problems:

Schränke die Klasse der zulässigen Formeln ein!

Logische Programmiersprachen

Technik 1: Widerspruchsfindung mittels *Resolution*

Aufgabe:

Gegeben eine Menge \mathcal{F} von prädikatenlogischen Formeln. Berechne, ob sie widersprüchlich ist.

Methode:

Äquivalente Formelumformungen: Ziel ist es, die Konstante \perp herzuleiten.

Resolutionsprinzip:

Generierung einer neuen Formel als Folgerung aus 2 gegebenen Formeln

Prinzip: Finde Literal c , der in den Formeln $a \vee c$ und $b \vee \neg c$ vorkommt.

Dann kann c **eliminiert** werden: $(a \vee c) \wedge (b \vee \neg c) \rightarrow (a \vee b)$

Die neue Formel heißt **Resolvente** der alten Formeln.

Durch eine solche Eliminierung können einzelne Literale isoliert werden:

Bsp.: $(a \vee c) \wedge \neg c \rightarrow a$ Interpretation: a muss in der Formelsammlung gelten.

Wenn auf diese Weise auch die Negation isoliert wird, ergibt sich ein Widerspruch:

Widerspruch!

Bsp.: $(\neg a \vee d) \wedge \neg d \rightarrow \neg a$ Interpretation: $\neg a$ muss in der Formelsammlung gelten.

Logische Programmiersprachen

Technik 2: Reduktion der Termvielfalt mittels *Unifikation*

Beispiel:

$$\Phi = \{\neg P(x, f(y)), P(z, f(g(z)))\}$$

Frage: Wieso kann diese Formelmenge widersprüchlich sein ?

Antwort: Weil die beiden Atome $P(x, f(y))$ und $P(z, f(g(z)))$ durch geschickte Wahl von z und g identifiziert werden können.

Das kann man mit Resolution alleine nicht herausfinden!

Eine logische Programmiersprache braucht daher *Unifikation*:

Ersetzung der Variablen durch Terme, so dass beide Atome gleich werden.

Logische Programmiersprachen

Technik 2: Reduktion der Termvielfalt mittels *Unifikation*

Substitution:

Die Ersetzung $[x/t]$ angewendet auf φ bezeichnet diejenige Formel, die aus φ entsteht, wenn alle **freien** Vorkommen in φ von x durch Term t ersetzt werden.

Analog wird die Ersetzung $[x_1/t_1, x_2/t_2, \dots, x_n/t_n]$ gebildet.

Bezeichnung: $\sigma = [x_1/t_1, x_2/t_2, \dots, x_n/t_n]$ heißt **Substitution**.
 $\sigma \varphi$ ist die **Anwendung von** σ auf φ

Beispiel: Formel: $\varphi = P(f(x), y)$
Substitution: $\sigma = [x/z, y/f(z)]$
Anwendung: $\sigma \varphi = P(f(z), f(z))$

Definition:

Eine Substitution σ heißt **Unifikator** für die Formeln α_1 und α_2 , wenn gilt: $\sigma\alpha_1 = \sigma\alpha_2$.

Beispiel:

Unifikation der Atome $Q(f(x), v, b)$ und $Q(f(a), g(u), y)$
durch die Substitution $\sigma = [x/a, v/g(u), y/b]$

Logische Programmiersprachen

Technik 2: Reduktion der Termvielfalt mittels *Unifikation*

Satz (Existenz):

Für je zwei Ausdrücke gibt es, bis auf Variablenumbenennung, entweder einen eindeutigen allgemeinsten Unifikator oder die beiden Ausdrücke sind nicht unifizierbar.

Satz (Berechenbarkeit):

Es gibt einen Algorithmus, der für zwei beliebige Ausdrücke entweder die Nichtunifizierbarkeit beweist oder den allgemeinsten Unifikator berechnet.

Verfahren: *ziemlich einfach !*

Wiederhole bis die Ausdrücke gleich sind oder die Nichtunifizierbarkeit gezeigt ist:

Wenn die Prädikate verschieden sind

oder die Anzahl der Parameter gleicher Prädikate verschieden ist

→ **nicht unifizierbar**

Anderenfalls

Wähle eine Variable x im ersten oder zweiten Ausdruck

und einen an der äquivalenten Stelle stehenden Term t im jeweils anderen Ausdruck, der x nicht enthält.

Wenn das nicht möglich ist → **nicht unifizierbar**

Anderenfalls ersetze x in beiden Ausdrücken durch t .

Logische Programmiersprachen

Technik 2: Reduktion der Termvielfalt mittels *Unifikation*

Übungsbeispiele für Unifikation:

$P(x)$ und $Q(y)$

$P(x, y)$ und $P(z)$

$P(x, y)$ und $P(a, f(a))$

$P(x, y)$ und $P(f(z), g(z))$

$P(x, f(x, x), z, f(z, z))$ und $P(f(a, a), y, f(y, y), u)$

$P(x, f(y))$ und $P(z, f(g(z)))$

$P(x, x)$ und $P(f(y), f(g(z)))$

$P(x, f(x))$ und $P(y, y)$

$P(x, a)$ und $P(b, x)$

Die logische Programmiersprache PROLOG

Prinzip von PROLOG:

PROLOG versucht, mittels wiederholter und verschachtelter Anwendung von **Resolution** und **Unifikation** zu einer gegebenen Formelmenge einen Widerspruch zu finden.

Satz (Widerspruchsvollständigkeit):

Falls die Formelmenge widersprüchlich ist, kann man den Widerspruch immer finden.

Was fehlt ?

Satz (Folgerung der Folgerbarkeit aus der Widerspruchsaufdeckung):

Wer zu jeder Formelmenge jeden Widerspruch aufdecken kann, kann zu jeder Formelmenge und zu jeder neuen **daraus folgenden** Formel beweisen, dass die neue Formel aus der alten Formelmenge folgt.

Was fehlt hier ?

Die logische Programmiersprache PROLOG

Wie macht man aus PROLOG eine vollständige Programmiersprache ?

Durch Beschränkung der Eingabe !

PROLOG akzeptiert nur Mengen von Formeln der Form:

$$p \wedge q \wedge \dots \wedge r \rightarrow x$$

Regeln (Hornklauseln)

In der Voraussetzung darf nur eine Konjunktion von positiven Literalen stehen.

Satz (Vollständigkeit der Resolution auf Hornklauseln):



Für jede Menge von Hornklauseln und eine neue Hornklausel kann Prolog nach endlicher Zeit entscheiden, ob die neue Hornklausel aus der alten Menge folgt **oder nicht**



Anmerkung: „Endliche Zeit“ kann „sehr lange“ heißen !

Die logische Programmiersprache PROLOG

Wie erkennt man, ob eine Formelmenge nur aus Formeln besteht, die äquivalent zu Hornklauseln sind ?

Die Formelmenge sei in KNF gegeben, d.h. als eine Konjunktion von Disjunktionen.

Die einzelnen Klauseln (Disjunktionen) seien die einzelnen Formeln.

Eine einzelne Formel ist nach Definition eine Hornklausel, wenn sie äquivalent zu einer Regel ist, deren Voraussetzungen alle durch einen positiven Literal repräsentiert sind.

Eine Hornklausel sieht also immer so aus:

$\neg p \vee \neg q \vee \dots \vee \neg r \vee x$ *Maximal ein Literal ist positiv.*

Anmerkung:

Natürlich kann man eine beliebige Aussage auch durch einen negativen Literal repräsentieren.

Um die Hornkauseleigenschaft zu erhalten, muss dieser Literal dann aber in allen Voraussetzungen von Regeln, in denen er vorkommt, in negativer Form auftauchen.

Daher können wir uns ohne Beschränkung der Allgemeinheit auf positive Literale beschränken.