Künstliche Intelligenz
&
Expertensysteme

Uwe Schmidt
FH Wedel

WS 1996/97

Inhaltsverzeichnis

Literatur

[Clocksin 90] Clocksin,Mellish:
Programmieren in Prolog,
Springer, 1990, DM 48,–
ISBN 3-540-16384-0

[Frühwirth 93] Frühwirth,T. e.a.:
Constraint Logic Programming
An Informal Introduction ECRC Technical
Report 93 5

[Ginsberg 93] Ginsberg,Matthew L.:
Essentials of Artificial Intelligence
Morgan Kaufmann Publishers, San Mateo, 1993
ISBN 1-55860-221-6

[GI] *KI: Künstliche Intelligenz*
Zeitschrift des Fachbereichs 1 der Gesellschaft
für Informatik

[Goldberg 89] Goldberg, D.E.:
Genetic Algorithms in Search Optimization &
Machine Learning
Addison–Wesley, 1989
ISBN 0-201-15767-5

[Karbach 90] Karbach,Werner, Linster,Marc:
Wissensaquisition für Expertensysteme:
Techniken, Modelle und Softwarewerkzeuge
Hanser, 1990
ISBN 3-446-15979-7

[O'Keefe 91] O'Keefe, R.:
The Craft of Prolog

The MIT Press, Cambridge, Mass., DM 97,–
ISBN 0-262-15039-5

[Puppe 88] Puppe,Frank:
Einführung in Expertensysteme,
Studienreihe Informatik, Springer, 1988, 48,-
DM

[Rich 88] Rich, Elaine:
KI-Einführung & Anwendungen,
McGraw-Hill, 1988, (deutsche Übersetzung)
ISBN 3-89028-031-5 (original 1983)

[Rich 91] Rich, Elaine:
Artificial Intelligence, 2nd ed
McGraw–Hill, 1991, DM 63,–

[Schnupp 87] Schnupp, Peter und Nguyen, C.T.:
Expertensystem-Praktikum
Springer, 1987
ISBN 3-540-17528-8

[Schöneburg 94] Schöneburg, E., Heinzmann, F., Feddersen, S.:
Genetische Algorithmen und
Evolutionsstrategien
Eine Einführung in Theorie und Praxis der
simulierten Evolution
Addion–Wesley, 1994
ISBN 3-89319-493-2

[Steele 90] Steele, G.L.:
Common LISP: The Language
2nd ed., Digital Press, 1990

[Stoyan 88] Stoyan, Herbert:

Programmiermethoden der künstlichen Intelligenz,
Band 1, Studienreihe Informatik, Springer, 1988

[Stoyan] Stoyan, H.,Görz,G.:
LISP - Eine Einführung in die Programmierung

[Tanimoto 90] Tanimoto,S.:
KI: die Grundlagen
Oldenbourg, München, 1990

[Weizenbaum 78] J. Weizenbaum:
Die Macht der Computer und die Ohnmacht der Vernunft,
Suhrkamp Taschenbuch Wissenschaft 274, 1978

[Winston 87] Winston, Patrick H.:
Künstliche Intelligenz
Addison–Wesley, 1987
ISBN 3-925118-60-8

[wwwLogProg] World Wide Web:
http://www.comlab.ox.ac.uk/archive/logic-prog.html

[SWI Prolog] SWI Prolog System, für Linux und Windows
Original: Anonymus FTP: von `swi.psy.uva.nl` unter `pub/SWI-Prolog` oder von `ftp.th-darmstadt.de` unter `pub/programming/languages/prolog`

[XSB Prolog] XSB Prolog System, Compiler for Linux, Sun, Sparc, NeXT
Original: Anonymus FTP: von `cs.sunysb.edu` unter `pub/XSB`

[clp(FD)] clp(FD), Constraint Logic Programming over finite domains
Original: Anonymus FTP: von `ftp.inria.fr` unter `/INRIA/Projects/ChLoE/LOGIC_PROGRAMMING/clp_fd`

1 Einführung

1.1 Was ist künstliche Intelligenz?

besser : Wissensverarbeitung
knowledge engineering

ungefähre Definition : KI

KI ist die Forschung darüber, wie man Computer Dinge ausführen lassen kann, die zur Zeit noch von Menschen besser beherrscht werden.

kurzlebige Definition

$\hookrightarrow$ Vermeidung der Begriffe künstlich und Intelligenz

$\hookrightarrow$ experimentelles Ergebnis der KI :
Fortschritt geht langsamer, als erwartet

erste Problemstellungen Spiele (Schach,...), Theorembeweise (Logik, Mathematik)

1963 erstes Schachprogramm, das seine eigenen Erfahrungen nutzte

1963 *The Logic Theorist*
erster Theorembeweiser für Logikaufgaben

Hoffnung schnelles Suchen und Ausprobieren im Rechner ermöglicht intelligentes Verhalten

kombinatorische Explosion bildet prinzipielle Schranke

1. Ansatz *GPS (General Problem Solver)*
Newell, 1963 :

symbolische Manipulation logischer Ausdrücke

keine Speicherung von Wissen aus einem speziellen Problembereich

$\hookrightarrow$ Sackgasse

Fortschritt Handhabung und Speicherung vom Weltwissen

$\hookrightarrow$ *neue Aufgabenstellungen*

- Wahrnehmung, Sehen, Sprechen
- Verständnis natürlicher Sprache
- Spezialgebiete
  Medizin $\rightarrow$ Diagnose
  Chemische Analyse

Wahrnehmung Mustererkennung
Bilderkennung

Verstehen natürlicher Sprache
extrem schwierig

- Vokabular,Grammatik *Syntax*
- Gegenstand (Gebiet) *Semantik*

Expertensysteme Experten
(Personen, die *schwierige* Dinge tun)
simulieren & ersetzen

1.2 KI-Gebiete (heute)

↪ Spiele

↪ Theorembeweiser

↪ Allgemeiner Problemlöser (GPS)

↪ Wahrnehmung, Verstehen, Sprache

↪ Verstehen natürlicher Sprache

↪ Lösen von Expertenproblemen

- symbolische Mathematik
- medizinische Diagnose
- Chemische Analyse
- technische Konstruktion

Pionierarbeiten

1966 *ELIZA*
Weizenbaum, Psychoanalyse

1965– *MACSYMA*
MIT, Symbolische Formeltransformation

1972 *MYCIN*
Stanford, Diagnose von Infektionskrankheiten

1965 *DENDRAL*
Stanford, chemische Analyse

1978 *R1*
CMU, Konfiguration von DEC VAX-Rechnern

1.3 Physikalische Symbolsysteme

Frage *Was ist ein KI-System ?*

Hypothese *Physikalische Symbolsystemhypothese*
Kern der KI-Forschung *[Newell,1976]*

Physikalisches Symbolsystem

↪ Menge von *Symbolen*

↪ *Ausdrücke* über den Symbolen

↪ Menge von *Ausdrücken*

↪ Menge von *Prozessen* zur

- Erzeugung
- Transformation
- Reproduktion
- Löschen

von Ausdrücken

↪ Maschine zur *Steuerung* der Prozesse

Hypothese Ein physikalisches *Symbolsystem* besitzt *hinreichende* und *notwendige* Mittel für eine allgemeine *intelligente* Tätigkeit

Denken ist reduzierbar auf Manipulation von Zeichenketten

Beweis durch Experimente

Hilfsmittel Computer zur Simulation physikalischer Symbolsysteme

1.4 Was ist ein KI-Verfahren?

Eigenschaften von KI-Verfahren
charakteristische Merkmale

Wissen Intelligentes Verhalten erfordert *Wissen !*

↪ Wissen hat (für die maschinelle Verarbeitung) unangenehme Eigenschaften

- ↪ umfangreich
- ↪ schwer zu charakterisieren
- ↪ es ändert sich dauernd

Darstellung von Wissen

- ↪ Erfassung von Verallgemeinerungen *(Abstraktion)*
- ↪ sonst :
  Speicherplatz $\nearrow$
  Änderungsaufwand $\nearrow$
- ↪ Es muß von Leuten verstanden werden, die es liefern (externe Darstellung)
- ↪ änderungsfreundlich
  Fehlerkorrektur, Änderung des Weltbildes
- ↪ Verwendung auch bei Unvollständigkeit
- ↪ Einengung, Spezialisierung, Konkretisierung

KI-PROBLEM $\Longleftrightarrow$ KI-VERFAHREN

ein KI-Problem braucht nicht mit einem KI-Verfahren gelöst zu werden ! *(einfache Spiele)*

auch nicht KI-Probleme können mit KI-Verfahren gelößt werden ! *(parsen)*

Tic-Tac-Toe 3 BEISPIELPROGRAMME

- ↪ Komplexität nimmt zu
- ↪ Grad der Verallgemeinerung nimmt zu
- ↪ Grad der expliziten Wissensdarstellung nimmt zu
- ↪ Erweiterbarkeit nimmt zu

Beispiel: Tic-Tac-Toe
Programm 1

Datenstrukturen

```
type wert =   (frei, X, 0);
var  Brett :  array [1..9] of Wert;
var  Zugtab:  array [Wert
                    ,Wert
                    ,Wert
                    ,Wert
                    ,Wert              9 ×
                    ,Wert
                    ,Wert
                    ,Wert
                    ,Wert]  of  1..9;
```

Algorithmus

```
var  zug : 1..9;
     zug := Zugtab[Brett[1], ... , Brett[9]];
```

Eigenschaften

- ↑ Schnell
- ↑ kann optimales Tic-Tac-Toe spielen
- ↓ viel Speicher $3^9 = 19.683$ Elemente
- ↓ wer initialisiert die Tabelle ?
- ↓ wer initialisiert die Tabelle richtig ? Korrektheit
- ↓ keine Erweiterungsmöglichkeiten
  3-dim, 4 × 4

Tic-Tac-Toe : Programm 2

Datenstrukturen

```
const frei   = 2 ;
      X      = 3 ;
      O      = 5 ;
type  wert   = frei ..O
var   Brett  :array [1..9]  of  Wert;
      Zug    : 1..9;
```

Algorithmus

Prozeduren

`function make2 : 1..9;`

versucht 2 Markierungen in einer Reihe zu bilden, erst Mitte, dann Seiten, nicht Ecken

`function poswin (p: X..O) : 0..9;`

→ 0 : Spieler kann mit nächsten Zug nicht gewinnen

→ i : Spieler p gewinnt mit Zug auf Feld i

`procedure go ( n : 1..9);`

$$\text{Brett}[n] := \begin{cases} O : \text{odd zug} \\ X : \neg\ \text{odd zug} \end{cases}$$

Zug := Zug+1;

TIC-TAC-TOE : PROGRAMM 2 (CONT.)

ALGORITHMUS

X	Zug=1	→	go(1)		
O	Zug=2	→	Brett[5]	= frei	→ go(5)
				≠	→ go(1)
X	Zug=3	→	Brett[9]	= frei	→ go(9)
		→		≠	→ go(3)
O	Zug=4	→	poswin(X)	≠ 0	→ go(poswin(X)) blockieren
				=	→ go(make2)
X	Zug=5	→	poswin(X)	≠ 0	→ go(poswin(X)) gewinnen
			poswin(O)	≠ 0	→ go(poswin(O)) blockieren
			Brett[7]	= frei	→ go(7) Ecken bes.
				≠	→ go(3)
O	Zug=6	→	poswin(O)	≠	→ go(poswin(O))
			poswin(X)	≠	→ go(poswin(X))
				=	→ go(make2)
XOX	Zug=7,8,9	→	poswin(wir)	≠	→ go(poswin(wir))
			poswin(gegner)	≠	→ go(poswin(gegner))
				=	→ go(?)

TIC-TAC-TOE : PROGRAMM 2

Wertung

↓	Effizienzverlust gegenüber Programm 1
↑	platzsparender als Programm 1
↑	Strategie kann verstanden werden
↓	Strategie fest geplant
↓	Fehler bei der Programmierung
	↪ schlecht gespielt
↓	nicht erweiterbar
	↪ 3-dim, 4×4

☞ *! immer noch kein KI-Programm !*

TIC-TAC-TOE : PROGRAMM 3

Daten-strukturen

`var Brett : array [1..9] of Wert;`

Liste von Brettpositionen, die sich aus dem nächsten Zug ergeben können und ein *Schätzwert* für die *Wahrscheinlichkeit*, daß der Zug zum *Sieg* führt.

`var wahr : array [1..9] of -1.0 .. 1.0;`

Wahrschein-lichkeit

Wahrscheinlichkeit, daß Zug i zum Sieg führt

1.0 ⇒ gewinnt sicher

-1.0 ⇒ verliert sicher

Algorithmus

a) Betrachte die möglichen nächsten Züge,
b) entscheide welcher der Beste ist,
c) mache Zug,
d) gib die Bewertung der aktuellen Position weiter
   1) Gewinnzug i :
      Wahr[i] = 1.0
   2) sonst :
      betrachte alle Züge des Gegners,
      nimm an, daß der Gegner den für dich schlechtesten Zug macht
   3) nimm den Zug mit der höchsten Wahrscheinlichkeit

TIC-TAC-TOE : PROGRAMM 3

Wertung

Minimax–Verfahren

Baumsuche mit dem Versuch, die *Wahrscheinlichkeit* des Sieges zu *maximieren*

↓ komplizierter Algorithmus

↪ Umgangssprache

↓ sehr zeitintensiv (besonders bei vollständiger Suche)

↪ 9! ZÜGE

↑ FLEXIBEL : Erweiterung auf 3-D oder 4 x 4 möglich

↑ FLEXIBEL : Algorithmus ist anwendbar auf andere Spiele, bei denen Programm 1 und Programm 2 nicht anwendbar sind

↑ VERBESSERUNG : Spieleigenschaften

↪ Teilmengen durchsuchen

↑ VERBESSERUNG : Suche in der Tiefe beschränken

☞ Programm 3 ist ein *KI-Verfahren*

↓ ineffizient (auch schon) bei kleinen Problemen

↑ allgemein einsetzbar

2. Beispiel : Mustererkennung

Erweiterung Strukturinformation ausnutzen

Problem Identifikation von Buchstaben aus einer Matrix mit *0*-en und *1*-en

Eingabe var muster : array $[1 \,..\, max_1$, $1 \,..\, max_2]$ of boolean;

Mustererkennung : Programm 1

Datenstrukturen Kodierung des Alphabets, indem nur einzelne Zeilen gespeichert werden, und zwar so, daß diese Zeilen eindeutig die Buchstaben bestimmen.

Algorithmus

1. selektiere die "Schlüsselzeilen" aus Eingabematrix
2. suche diese Zeilen in der Kodiertabelle

Bewertung

↑ feste Mustermengen,

↑ festes Eingabeformat

↓ Fehleranfällig : *1 Punkt falsch*

↪ *völlig falsches Ergebnis*

↓ änderungs- und erweiterungsunfreundlich

Mustererkennung : Programm 2

Datenstrukturen

Eingabe wie Programm 1

Zeichen gegeben ein $N > 1$, Einteilung der Matrix in N Gebiete, in diesen Gebieten werden die Anzahlen der *1*-en gezählt und in einem Vektor gespeichert

```
type Vektor  =  array [1..N]      of 0 .. max;
var  Muster  :  array ['A'..'Z']  of vektor;
```

Algorithmus

1. wandle die Eingabematrix in einen Vektor um
2. berechne den *"Abstand"* des Eingabevektors von allen Mustervektoren
3. bestimme das Muster mit dem minimalen Abstand

Bewertung

↑ viel besser als Programm 1 *(immer eine Lösung)*

↑ kleine Änderungen, Fehler in der Eingabe werden richtig erkannt

↓ Aufwand höher als in Programm 1

↪ *"Abstand" zu allen Zeichen berechnen*

↓ keine Trennung von *signifikanter* und *nicht signifikanter* Abweichung

↓ Größenunterschiede nicht wirksam behandelt

Mustererkennung : Programm 3

Datenstrukturen

bekannte Muster kodiert als boolsche Ausdrücke aus elementaren Zeicheneinheiten und Eigenschaften

G :

$$\mathit{arc}(a,b) \wedge \mathit{above}(a,b) \wedge \mathit{lines}(b,c)$$
$$\wedge\ \mathit{left}(c,b)$$
$$\wedge\ (\mathit{line}(b,d) \Rightarrow \mathit{above}(b,d))$$
$$\wedge\ (\mathit{line}(a,e) \Rightarrow \mathit{above}(e,a))$$

Algorithmus

1. Suche Bögen und Strecken und erstelle Eingabeausdruck
2. Vergleiche diesen Ausdruck mit den Musterausdrücken und nehme den am besten passenden

Bewertung

$\uparrow$ flexibel

$\uparrow$ leicht erweiterbar

$\uparrow$ keine absoluten Größenangaben

$\uparrow$ Trennung :

signifikant $\longleftrightarrow$ *nicht signifikant*

$\downarrow$ Umsetzung problematisch:

Bitmatrix $\rightarrow$ *Linien, Bögen*

$\downarrow$ Fehler : gespiegeltes G wird auch erkannt

Schlussfolgerungen

Tic-Tac-Toe & Mustererkennung

Programme 1,2 *direkte Verfahren*

Programm 3 *KI–Verfahren*

Techniken für KI–Verfahren

Suche

- Liefert Weg zur Lösung von Problemen, bei denen kein direkter Ansatz verfügbar ist
- Liefert *Rahmen* zur Einbettung eines Verfahrens

Wissen

- Liefert Lösung unter Ausnutzung der *Struktur* der Objekte

  $\hookrightarrow$ Programm 3 : Muster

Abstraktion

- liefert *Trennung* zwischen *wichtiger* und *nicht relevanter* Information
- liefert *Reduktion* der Informationsmenge

☞ Programme mit diesen Eigenschaften sind in der Lage auch große Probleme zu bearbeiten, bei denen *Direktverfahren scheitern*

☞ Immer noch keine Definition eines *KI-Verfahrens* aber eine Vorstellung

1.5 Erfolgskriterien

allgemein wie erkennen wir, ob wir bei der Lösung eines Problems erfolgreich waren?

in der KI Woran erkennen wir, daß wir eine *intelligente Maschine* gebaut haben?

Turing-Test eine Person befragt eine weitere Person und eine Testmaschine

Ziel Der Fragesteller soll herausfinden wer wer ist.

Die Maschine kann den Fragesteller in die Irre führen

gewinnt Maschine ⇒ **intelligente Maschine**

Wissensmenge in der Maschine entscheidend

Es kann noch lange dauern, bis eine Maschine den Turing-Test gewinnt !

(wenn überhaupt)

Spezialisierung Maschinen können höhere Leistungen auf eingeschränkten Gebieten erbringen

- Schach
- Dendral (chemische Analyse)
  - ↪ Ergebnisse als Originalarbeiten

Leistungsfähigkeit schwer zu messen

1.6 Zusammenfassung

KI-Probleme sehr

- *unterschiedlich*
- *schwierig*
- *interessant*

1. Schritt Erfolgskriterien für eine Lösung definieren

2. Schritt Lösung des Problems versuchen

Ein gutes KI-System muß viel Wissen enthalten

↪ Wissensauswahl schwierig
↪ Regeln zur Wissensauswahl hinzufügen
↪ mehr Wissen
↪ Wissensauswahl schwieriger
↪ ....

Implementierungssprachen

LISP *Mc Cathy, 1962* :

- ↪ Funktionales Programmieren
- ↪ Daten : Listen
- ↪ Funktionen : λ–Kalkül
- ↪ viele Dialekte, Erweiterungen, ... (USA)

PROLOG PROgramming in LOGic

- ↪ Horn-Klauseln
- ↪ Teilmenge des *Prädikatenkalküls* (Europa, Japan)

wesentlich Programme können sich dynamisch verändern

- ↪ LISP: *eval*
- ↪ Prolog: *asserta,assertz*
- ↪ zur Modellierung des Lernens

2 KI–Programmierung

2.1 Zentrale Datenstruktur: Graphen

<u>Graphen als semantische Bereiche</u>

Problemraum Zustandsraum (Graph) der successive aufgebaut wird bis eine Lösung gefunden wird

gegeben eine Datentyp $Knoten$ für die Knoten

Typ–defintionen

.0 $Graph = Knoten \xrightarrow[\text{m}]{} Nachfolger$

.1 $Nachfolger = Knoten$ -set

.2 $Knoten = \ldots$

Beispiel $Knoten = \{a, b, c, d\}$

Graph

$$g_7 = \{\ a \mapsto \{b, c\},\ b \mapsto \{d\},\ c \mapsto \{d\},\ d \mapsto \{c, a\}\}$$

Vorbereich (Knotenmenge) dom $g_7 = \{a, b, c, d\}$

Kommentar Ein Graph g wird modelliert als eine Menge dom g von Knoten, jedem Element $a \in$ dom g wird eine Menge von Knoten zugeordnet, zu denen eine Kante von a führt.

<u>Wasserkrugproblem modelliert mit semantischen Bereichen</u>

Datentypen

.0 $Graph = Knoten \xrightarrow[\text{m}]{} Nachfolger$

.1 $Nachfolger = Knoten$ -set

.2 $Knoten = Krug_3 \times Krug_4$

.3 $Krug_4 = 0..4$

.4 $Krug_3 = 0..3$

Knotenmenge

$n_0 = (0, 0)$ $n_1 = (4, 0)$ $n_2 = (0, 2)$

$n_3 = (4, 3)$ $n_4 = (1, 3)$ $n_5 = (3, 0)$

Konstruktion des Suchgraphen

.1 $g_0 = \{n_0 \mapsto \{\,\}\}$

.2

$$g_1 = \{\ n_0 \mapsto \{n_1, n_2\},\ n_1 \mapsto \{\,\},\ n_2 \mapsto \{\,\}\ \}$$

.3

$$g_2 = \{\ n_0 \mapsto \{n_1, n_2\},\ n_1 \mapsto \{n_0, n_3, n_4\},\ n_2 \mapsto \{n_0, n_3, n_5\},\ n_3 \mapsto \{\,\},\ n_4 \mapsto \{\,\},\ n_5 \mapsto \{\,\}\ \}$$

Markierte Graphen (Graphen mit Kosten)

Markierung Attributierung der

$\hookrightarrow$ *Knoten*

$\hookrightarrow$ *Kanten*

$\hookrightarrow$ Wegkosten, Transformationsregeln

$\hookrightarrow$ Datentypen für *Knoten* und *Kanten* erweitern

$$
\begin{array}{lll}
.0 & MarkierterGraph & = Graph \times Marken \\
.1 & Graph & = Knoten \xrightarrow[\text{m}]{} Nachfolger \\
.2 & Nachfolger & = Knoten \xrightarrow[\text{m}]{} KantenAttribut \\
.3 & Marken & = Knoten \xrightarrow[\text{m}]{} KnotenAttribut \\
.4 & Knoten & = \ldots \\
.5 & KnotenAttribut & = \ldots \\
.6 & KantenAttribut & = \ldots
\end{array}
$$

$\hookrightarrow$ Sollen die Kanten attributiert werden, so wird die Nachfolgermenge

$Knoten$ -set

erweitert zu

$Knoten \xrightarrow[\text{m}]{} KantenAttribut.$

$\hookrightarrow$ Sollen die Knoten attributiert werden, so kann dies mit einer zusätzlichen Abbildung *Marken* modelliert werden, ein markierter Graph wird dann als Paar dargestellt.

Markierte Graphen : Wasserkrugproblem

Markierung der Knoten : Wasserkosten

$$
\begin{array}{lll}
.0 & MarkierterGraph & = Graph \times Marken \\
.1 & Graph & = Knoten \xrightarrow[\text{m}]{} Nachfolger \\
.2 & Nachfolger & = \ldots \\
.3 & Marken & = Knoten \xrightarrow[\text{m}]{} Wasserkosten \\
.4 & Knoten & = \ldots \\
.5 & Wasserkosten & = \mathsf{N}_0
\end{array}
$$

Markierung der Kanten : Regelnummer

$$
\begin{array}{lll}
.0 & MarkierterGraph & = \ldots \\
.1 & Graph & = \ldots \\
.2 & Nachfolger & = Knoten \xrightarrow[\text{m}]{} Regelnummer \\
.3 & Knoten & = \ldots \\
.4 & Regelnummer & = 1 \,..\, 12
\end{array}
$$

2.2 Funktionale Programmierung

↪ wie Pascal oder C ohne Zuweisungen u. ohne Variablen (nur Parameter)

Typdefintion $f : X \times Y \longrightarrow Z$

↪ f ist eine Funktion mit 2 Parametern vom Typ X und Y und liefert einen Wert vom Typ Z als Resultat.

Funktionsrumpf $f(x,y) \triangleq \mathcal{EXPR}(x,y)$

Ausdrücke

.1	Konstanten	$1, 2, 3, \{\}, []$
.2	Parameter	x, y
.3	$op_1\ \mathcal{EXPR}$	$-x$
.4	$\mathcal{EXPR}_1\ op_2\ \mathcal{EXPR}_2$	$x + y$ $x \cup (y \cap z)$
.5	Bedingte Ausdrücke if $\mathcal{EXPR}_1$ then $\mathcal{EXPR}_2$ else $\mathcal{EXPR}_3$	 if $x \geq 0$ then x else $-x$
.6	Funktionsaufrufe (auch rekursiv)	 $f(\mathcal{EXPR}_1, \cdots, \mathcal{EXPR}_n)$
.7	als Parameter und Resultate auch *Funktionen* nicht nur Werte	

↪ *functions are first class members*

let –Ausdrücke für Zwischenergebnisse

Syntax let $id = \mathcal{EXPR}_1$ in $\mathcal{EXPR}_2$

Beispiel

.0 $f : \mathsf{N}_1 \longrightarrow \mathsf{N}_1$

.1 $f(x) \triangleq (x * x) + (x * x)$

.2 $f(x) \triangleq$

.3 let $x_2 = x * x$

.4 in $x_2 + x_2$

Nichtdeterministische Auswahl

Syntax let $id \in s$ in$EXPR$

Semantik ein beliebiges Element wird aus s ausgewählt und mit id benannt

Voraussetzung: s ist nicht leer

Beispiel

.0 $sum : \mathsf{N}_1$ -set $\longrightarrow \mathsf{N}_1$

.1 $sum(s) \triangleq$

.2 if $s = \{\,\}$

.3 then 0

.4 else

.5 let $x \in s$

.6 in $x + sum(s - \{x\})$

Konstruktion von Graphen

Datentypen (domains)

.0 $Graph = Knoten \xrightarrow[m]{} Nachfolger$

.1 $Nachfolger = Knoten$ -set

.2 $Knoten = \ldots$

Einfügen

.0 $einfügen :$
$Graph \times Knoten \times Knoten \longrightarrow Graph$

.1 $einfügen(g, n, n') \triangleq$

.2 let $g' =$ if $n' \in$ dom g

.3 then g

.4 else $g \cup \{n' \mapsto \{\,\}\}$ in

.5 let $nf = g'(n)$ in

.6 $g' + \{n \mapsto nf \cup \{n'\}\}$

.7 pre-$einfügen(g, n, n') \triangleq$

.8 $n \in$ dom g

Konstruktion : Graph mit Kosten

.0 $einfügen\_mit\_Kosten : MarkierterGraph \times Knoten \times Knoten \times Kosten \longrightarrow MarkierterGraph$

.1 $einfügen\_mit\_Kosten((g, m), n, n_1, k) \triangleq$

.2 let $g_1 = einfügen(g, n, n_1)$ in

.3 let $k_1 = m(n) + k$ in

.4 let $m_1 =$ if $n_1 \notin$ dom m

.5 then $m \cup \{n_1 \mapsto k_1\}$

.6 else $m + \{n_1 \mapsto$ min $\{m(n_1), k_1\}\}$

.7 in (g_1, m_1)

$\hookrightarrow$ Kante von n nach n_1 wird eingefügt (.2)

$\hookrightarrow$ Kosten für Weg nach n_1 werden aus Kosten von n berechnet (.3)

$\hookrightarrow$ ist n_1 ein neuer Knoten, werden die Wegekosten eingetragen (.5)

$\hookrightarrow$ sonst werden die minimalen Kosten berechnet (.6)

DIE FUNKTIONALE PROGRAMMIERSPRACHE: LISP

LISP — *LISt Programming*
erste funktionale Programmiersprache

Erfinder — McCarthey, 1962

Listen — (genauer Bäume) einzige Datenstruktur

.0 $LispValue = Pair \mid Atom$

.1 $Pair = LispValue \times LispValue$

.2 $Atom = \mathsf{true} \mid \mathsf{nil} \mid \mathsf{token}$

konkrete Syntax — LISP–Programme sind auch Listen
↪ LISP–Programme können sehr einfach verarbeitet (interpretiert, compiliert, transformiert, optimiert, …) und erzeugt werden

Sprach-definition — durch einen in LISP formulierten Interpretierer (auf 2 Seiten) *genial*

≈ 80 % aller KI-Programme sind in LISP(–Dialekten) geschrieben.

LISP ist eine *ungetypte* Sprache (es gibt nur einen universellen Datentyp)
↪ Unsicherheit
↪ hier: *getyptes* LISP mit Datentypen aus Software–Entwicklungs-Methoden

2.3 Logische Programmierung

Programmierung in einer Teilmenge des Prädikatenkalküls (Horn–Klauseln)

Programmierung mit Prädikaten

Programmierung mit Relationen

Syntax

Fakten

$$\underbrace{\mathsf{mensch}}_{(1)}(\underbrace{\mathsf{adam}}_{(2)}).$$

$$\underbrace{\mathsf{kind}}_{(1)}(\underbrace{\mathsf{kain}}_{(2)}, \underbrace{\mathsf{adam}}_{(2)}).$$

Regeln

$$\mathsf{mensch}(\underbrace{\mathsf{X}}_{(3)}) :- \mathsf{kind}(\underbrace{\mathsf{X}, \mathsf{Y}}_{(3)}), \mathsf{mensch}(\underbrace{\mathsf{Y}}_{(3)}).$$

(1) Prädikate (1– und 2–stellig)

(2) Werte

(3) Variablen

Variablen — ↪ frei
↪ gebunden

binden — zuordnen von Werten an Variable durch *matchen* (vergleichen)

Werte — ↪ einfach: *Symbole, Zahlen*
↪ zusammengesetzt: *Listen, Prädikate*

Stammbaum

/ ein paar Fakten */*

```
kind(kain,adam).
kind(kain,eva).
kind(abel,adam).
kind(abel,eva).
```

Ist Kain ein Kind von Eva?

?− *kind(kain, eva).*
yes

Wer sind die Eltern von Kain?

?− *kind(kain, X).*
X = adam ; */* weitere Lösungen? */*
X = eva ; */* weitere Lösungen? */*
no */* Reihenfolge der Lösungen wie in der Datenbasis */*

Wer sind die Kinder von Eva?

?− *kind(X, eva).*
X = kain ; */* weitere Lösungen? */*
X = abel ; */* weitere Lösungen? */*
no

⇒ *Regeln anwendbar*
a) *zum* Beweisen *von Aussagen*
b) *zum* Berechnen *von Lösungen*
anwendbar in "verschiedene Richtungen"

Listen

Listen $[e_1, \ldots, e_n]$ Liste mit n Elementen
[] leere Liste
[H|R] nicht leere Liste
H: Kopf der Liste
R: Rest der Liste

Test Ist ein Element X in einer Liste L enthalten?

1. L nicht leer und X = Kopf von L
2. L nicht leer und X im Rest von L enthalten

Prädikat member

1. member(X, L) :− L = [H|R], X = H.
2. member(X, L) :− L = [H|R],
   member(X, R).

Anfragen

↪ member(1, [3, 5, 7, 9]). Test
↪ no

↪ member(X, [3, 5, 7, 9]). Erzeugen
↪ X = 3; X = 5; X = 7; X = 9

↪ member(1, X). Erzeugen aller Listen die 1 enthalten
↪ X = [1]; X = [_, 1]; ...

LISTEN (CONT.)

Verknüpfen zweier Listen X und Y zu einer Liste Z:

1. wenn X leer ist, ist Y gleich Z.
2. wenn X nicht leer ist, verknüpfe den Rest von X mit Y zu einer Liste Z_1, und setze den Kopf von X vor diese neue Liste Z_1.

Prädikat append

1. $\mathsf{append(X,Y,Z) :- X = [], Y = Z.}$
2. $\mathsf{append(X,Y,Z) :- X = [H \mid R], append(R,Y,Z_1), Z = [H \mid Z_1].}$

Anfragen

↪ $\mathsf{append([1,2],[3,4],[1,2,3,4]).}$
↪ yes

↪ $\mathsf{append([1,2],[3,4],X).}$
↪ $\mathsf{X = [1,2,3,4]}$

↪ $\mathsf{append([1,2],X,[1,2,3,4]).}$
↪ $\mathsf{X = [3,4]}$

$\mathsf{append(X_1,X_2,[1,2,3,4]).}$
↪ $\mathsf{X_1 = [], X_2 = [1,2,3,4]}$
↪ $\mathsf{X_1 = [1], X_2 = [2,3,4]}$
↪ $\mathsf{X_1 = [1,2], X_2 = [3,4]}$
↪ $\mathsf{X_1 = [1,2,3], X_2 = [4]}$
↪ $\mathsf{X_1 = [1,2,3,4], X_2 = []}$

2.4 Funktionale Programmierung ⇔ Logische Programmierung

Funktion

$$f : D_1 \times \ldots \times D_n \longrightarrow R$$

$$f(X_1,\ldots,X_n) \triangleq \mathcal{EXPR}(X_1,\ldots,X_n)$$

$\Rightarrow$

Prädikat $n+1$–stelliges Prädikat

$$\underbrace{f_p(\underbrace{X_1,\ldots,X_n}_{Input},\ \underbrace{Res}_{Output})}_{Parameter}$$

Inputparameter	:	beim Aufruf gebunden
Outputparameter	:	beim Aufruf frei

Umsetzung von Bedingungen

funktional $f(X) \triangleq \mathsf{if}\ B(X)\ \mathsf{then}\ E_1(X)\ \mathsf{else}\ E_2(X)$

$\Rightarrow$

Prädikate

$\mathsf{f_p(X,R) :- B(X), R = E_1(X).}$

$\mathsf{f_p(X,R) :- not(B(X)), R = E_2(X).}$

3 Probleme & Problemräume

3.1 Definition: Problemraum

in Einführung	Beschreibung von Problemarten
	↪ Beispiele
hier	Untersuchung des Lösungswegs *(3 Schritte)*
.1	**Problemdefinition** *(präzise !)*
	↪ Anfangssituation
	↪ akzeptable Lösungen
.2	**Problemanalyse**
	wenige zentrale Funktionen entscheidend
.3	**Auswahl**
	eines geeigneten Lösungsverfahrens und Anpassung an das Problem
☞	**Problemstellung**
	Bewegung in einem Zustandsraum

<u>Problemraum : Beispiel : Schach</u>

Problem	spiele Schach
Problem–definition	*einfach :* 8 × 8 Schachbrett 2 × 16 Figuren Zugregeln
Anfangssituation	*klar*
akzeptable Lösung	Zugposition, in der der gegnerische König angegriffen wird *(Schach)* und keinen Ausweg mehr besitzt, um im nächsten Zug nicht geschlagen zu werden *(Matt)*
Zustandsraum	Positionen auf 8 × 8 Brett
Bewegung	Anwendung einer Regel
☞	Schach : Problemdefinition *einfach*
	↪ sehr strukturiertes Problem

ZUSTANDSRAUMDARSTELLUNG

Grundlage (fast) aller KI-Verfahren

Problemdef. formal :

Ausgangszustand transformieren in
Endzustand unter Anwendung erlaubter
Operationen

Lösung durch *Kombination* *bekannter Verfahren*
und *Suche*

im Beispiel erlaubte Züge : *bekannte Verfahren*
ausprobieren der Züge : *Suche*

Suche immer dann, wenn *kein direktes* Verfahren bekannt ist (VERGL. PARSEN)

$\hookrightarrow$ wesentliches Element von KI-Verfahren

☞ weiteres Beispiel zur Demonstration des Ansatzes *Wasserkrugproblem*

Suche & Transformation

WASSERKRUGPROBLEM

gegeben ein 4 Liter Krug, ein 3 Liter Krug (ohne Maß), eine Pumpe, ein Ausguß

Problem Wie kommen 2 Liter in den 4 Liter Krug

Zustandsraum $\{(x, y) \mid x \in 0..4, y \in 0..3\}$

Anfangszustand $(0, 0)$

Zielzustände $\{(2, y) \mid y \in 0..3\}$

Regeln : (x, y)

1.	$x < 4$	$\rightarrow$	$(4, y)$	eingießen
2.	$y < 3$	$\rightarrow$	$(x, 3)$	
3.	$x > 0$	$\rightarrow$	$(x - d, y)$	teilweise ausgießen
4.	$y > 0$	$\rightarrow$	$(x, y - d)$	
5.	$x > 0$	$\rightarrow$	$(0, y)$	ganz ausgießen
6.	$y > 0$	$\rightarrow$	$(x, 0)$	
7.	$x + y \geq 4 \wedge y > 0$	$\rightarrow$	$(4, x + y - 4)$	teilweise umfüllen
8.	$x + y \geq 3 \wedge x > 0$	$\rightarrow$	$(x + y - 3, 3)$	
9.	$x + y < 4 \wedge y > 0$	$\rightarrow$	$(x + y, 0)$	ganz umfüllen
10.	$x + y < 3 \wedge x > 0$	$\rightarrow$	$(0, x + y)$	

Wasserkrugproblem: Fragen

Güte der Lösung gesucht ist die *kürzeste* (oder billigste) *Lösung*

↪ Einfluß auf den Suchprozeß
(irgendein Pfad, ein spezieller Pfad)

Regelmenge Wie wird die Menge der erlaubten Regeln festgelegt ?

↪ BS : *Regeln 3 & 4 nicht nötig*

Redundanzen erlaubt ?

Entscheidung :

redundant ⟷ *notwendig*

SCHWIERIG

mögliche Kritik :
Lösung ist *einprogrammiert*
(Redundanz verboten)

allgemeingültig :
Regeln können für ein Problem redundant sein, für ein anderes notwendig

---

Folgerung aus den Beispielen

☞ *Schach* und *Wasserkrug* sehr *unterschiedlich*,

aber *Problemdefinition* 1. Schritt zur *Konstruktion* einer *Lösung*

☞ Entwurf einer *formalen* und *manipulierbaren* Beschreibung des Problems

im BS : leicht
allgemein : schwierig (natürliche Sprache, ...)

.1 Definition *Zustandsraum* & alle *Konfigurationen*

.2 Definition Menge von *Anfangszuständen*

.3 Definition Menge von *Zielzuständen*

.4 Definition Menge von *Regeln*
(Transformationen im Zustandsraum)

Achtung

☞ Welche nicht genannten Annahmen sind in der informellen Problembeschreibung vorhanden?

☞ Wie allgemein sollen die Regeln sein?

☞ Wieviele der für die Lösung des Problems erforderlichen Arbeiten sollen vorausberechnet und in den Regeln dargestellt werden?

3.2 Produktionssysteme

Strategie bei der Anwendung der Transformationen

Anfangszustand $\longleftrightarrow$ *Endzustand*

Kombination SUCHE + DIREKTE Ansätze (Berechnungen)
(Wasserkrug : ARITHMETIK)

<u>PRODUKTIONSSYSTEME</u>

Ziel Suchprozeß strukturieren und vereinfachen

Definition Ein *Produktionssystem* besteht aus

einer Menge von *Regeln* bestehend aus

- ↪ *linker Seite* : bestimmt die *Anwendbarkeit*
- ↪ *rechter Seite* : bestimmt die auszufürende *Aktion*

einer *Datenbasis*

- ↪ enthält alles *Wissen* zur Problemlösung
- ↪ *dauerhafte* und *temporäre* Information

Kontrollstrategie zur Bestimmung der

- ↪ *Reihenfolge*, in der die Regeln mit der Datenbasis verglichen werden
- ↪ *Auswahl*, wenn mehrere Regeln passen

<u>FORMALE SPRACHEN $\Longleftrightarrow$ PRODUKTIONSSYSTEME</u>

Alphabet
zu parsende Symbolfolge } $\Longleftrightarrow$ **Datenbasis** { **permanent** / **temporär** }

Grammatik $\Longleftrightarrow$ **Regelsystem**

Parser $\Longleftrightarrow$ **Kontrollstrategie**

Kontroll-strategie bei regulären Sprachen und deterministischen kontextfreien Sprachen

- ↪ deterministischer Parser ohne Suche
- ↪ kein KI–Verfahren

bei mehrdeutigen kontextfreien Sprachen und kontextsensitiven Sprachen (oder Chomsky L_0 Sprachen)

- ↪ Suche & Mehrdeutigkeit *backtracking*
- ↪ Problem für KI–Verfahren

Produktionssysteme in der KI

gute Eigenschaften

↪ Modellierung der datengetriebenen Natur intelligenter Tätigkeiten

↪ Datenbasisänderung ⇒Änderung des Systemverhaltens

↪ Erweiterung der Regelmenge ohne Modifikation der Steuerung

↪ führt oft zu Wechselwirkungen in den *(BS)* Regelmengen, z.B Zyklen *(beispiele)*

↪ *schlecht aber :*

↪ keine Modifikation des Codes nötig

Prolog ⟺ Produktionssysteme

Fakten ⟺ Datenbasis

mensch (adam)
kind (kain, adam)

Regeln ⟺ Regeln

mensch(X) :- kind(X,Y)
↖ *linke Seite* ↖ *rechte Seite*

Prolog–Interpretierer ⟺ Kontrollstrategie

Resolutionsverfahren

asserta, assertz ⟺ Änderung der Datenbasis
retract und Regelmenge

↪ Prolog–Programm & –Interpretierer

↪ spezielles Produktionssystem mit fester Kontrollstrategie

3.3 Kontrollstrategien für Produktionssysteme

Kernproblem Was macht man, wenn mehr als eine Regel paßt?

Anforderungen an eine Kontrollstrategie

↪ *Bewegung* erzeugen!
⇒*nicht :* in Zyklen laufen
(Wasserkrug : nicht von oben nach unten !)

↪ *Systematik* ⇒*nicht :* zufällig
besser als von oben nach unten, aber
wahrscheinlich zu viele *nutzlose* Schritte

↪ *Effizienz !* des Algorithmus
⇒*nicht :* der Kodierung

<u>SUCHSTRATEGIEN</u>

Breitensuche *breadth first search*

↪ exponentieller Platzbedarf

↪ exponentielle Laufzeit

Tiefensuche *deapth first search*

suche einen Pfad ab, bis eine

↪ Lösung gefunden ist oder

↪ bis maximale Tiefe erreicht worden ist
→ Zyklen ! (Prolog–Strategie)

☞ beides systematische Strategien

⇒*aber :* nicht unbedingt effizient

Effizienz *DAS Kriterium !*

<u>EFFIZIENZ</u>

Beispiel

Problem Traveling Salesman Problem

Ein Handlungsreisender hat eine Liste von Städten, die er genau einmal auf einer Rundreise besuchen muß. Es gibt von jeder Stadt zu jeder anderen eine direkte Verbindung.

gesucht der kürzeste Weg für eine Rundreise durch alle Städte

Lösung Baum aller möglichen Wege durchsuchen

Anzahl der Wege $(N-1)!$ für N Städte

Komplexität $O(N!)$

$\hookrightarrow$ kombinatorische Explosion

$\hookrightarrow$ nicht effizient lösbar

$\hookrightarrow$ nicht effektiv lösbar

3.4 Heuristische Suche

Ausweg aus kombinatorischer Explosion

Heuristik Verbesserung der Effizienz
Verlust der Vollständigkeit, Optimalität

$\hookrightarrow$ wie Reiseführer

$\hookrightarrow$ zeigen in interessante Richtungen

$\hookrightarrow$ führen manchmal in Sackgassen

Heuristik gute Lösung : Aufwand $< O(e^n)$
aber : nicht immer optimale Lösung

allgemeine Heuristiken :
nächster Nachbar (lokal optimal)

traveling Salesman

1. wähle beliebigen Anfangsort
2. wähle die nächste noch nicht besuchte Stadt
3. wiederhole vorigen Schritt so lange, bis alle Städte besucht sind

Komplexität $O(N^2)$ $(n-1)+(n-2)+\cdots+1$

Abschätzung maximale Verschlechterung gegenüber bester Lösung (*hier :* möglich, *allg. :* nicht)

Güte messen schwierig

unstrukturiertes Wissen

$\hookrightarrow$ kein mathematisches Modell

<u>Warum Heuristiken?</u>

☞ Effektivität, Effizienz

☞ nicht immer : optimale Lösung / oft : gute Lösung } gesucht

☞ schlechte Lösungen treten bei Heuristiken relativ unwahrscheinlich auf

KI–Verfahren ⇒ Suchverfahren

(genauer) ⇒ heuristische Suchverfahren

<u>neuer Definitionsversuch</u>

künstliche Intelligenz

Untersuchung von Verfahren zur Lösung exponentiell schwieriger Probleme in polynomischer Zeit durch Anwendung von Wissen über den Problembereich

Wissen ⇒ Heuristiken

3.5 Problemkennwerte

Ziel Klassifikation eines Problems

↪ Lösungsstrategie ableiten

Kennwerte

↪ Kann das Problem in *unabhängige Teilprobleme* zerlegt werden?

↪ Können *Lösungsschritte ignoriert* oder *rückgängig* gemacht werden?

↪ Ist die *Gesamtheit* des Problems *vorhersehbar* oder kommen während der Lösung *Einflüsse von außen* dazu?

↪ Ist eine Lösung einfach als *gute Lösung* zu erkennen?

(kurzer, billiger Weg)

↪ Ist die interne *Wissensbasis konsistent* ?

($A \wedge \neg A$: alles kann bewiesen werden)

Ist die Gesamtheit voraussagbar?

8-puzzle *voraussagbar*

↪ bei jedem Zug ist genau vorhersehbar, was passiert

↪ die gesamte Zugfolge ist planbar

Kartenspiel *nicht voraussagbar*

(Skat, ...)

↪ nur mit einer gewissen Wahrscheinlichkeit voraussagbar

↪ Zugfolge nur mit einer gewissen Wahrscheinlichkeit planbar

↪ Nachberechnung nach einem Zug notwendig *(Planungsrevision)*

↪ aufwendig

sicheres Resultat ⟺ unsicheres Resultat

Planung

ohne ← Feedback → mit

Wechselwirkung wiederherstellbar ⇔ voraussagbar

Problem *nicht wiederherstellbar*

↪ *vollständige* Planung

↪ *nicht voraussagbar*

↪ keine Lösung! (Schach)

harte KI–Probleme : *nicht* wiederherstellbar
nicht voraussagbar

Ist eine gute Lösung absolut oder relativ (Wegabhängigkeit) ?

Beispiel	*Datenbankanfrage*
Daten	
.1	Markus war ein Mensch
.2	Markus war ein Pompejer
.3	Markus wurde im Jahre 40 geboren
.4	Alle Menschen sind sterblich
.5	Alle Pompejer starben beim Vulkanausbruch 79
.6	Kein Sterblicher lebt länger als 150 Jahre
.7	Jetzt ist 1996
Frage	Lebt Markus?
nur Antwort	interessant ↪ *alle* Lösungswege sind gleich gut
N.B.	*1) - 3), 7) Spezialwissen* *4) - 6) allgemeine Regeln* (mit Quantoren)

Ist Marcus tot?

ein Prolog Programm

/ die Fakten */*

```
mann(marcus).
pompejaner(marcus).
geboren(marcus,40).
```

/ die Regeln */*

```
sterblich(X) :-
        mensch(X).
gestorben(X,79) :-
        pompejaner(X).
tot(X,T2) :-
        sterblich(X),
        geboren(X,T0),
        nonvar(T2),nonvar(T0),
        T3 is T2 - T0,
        T3 > 150.
tot(X,T2) :-
        gestorben(X,T1),
        nonvar(T1),nonvar(T2),
        T2 > T1.
mensch(X) :-
        mann(X).
```

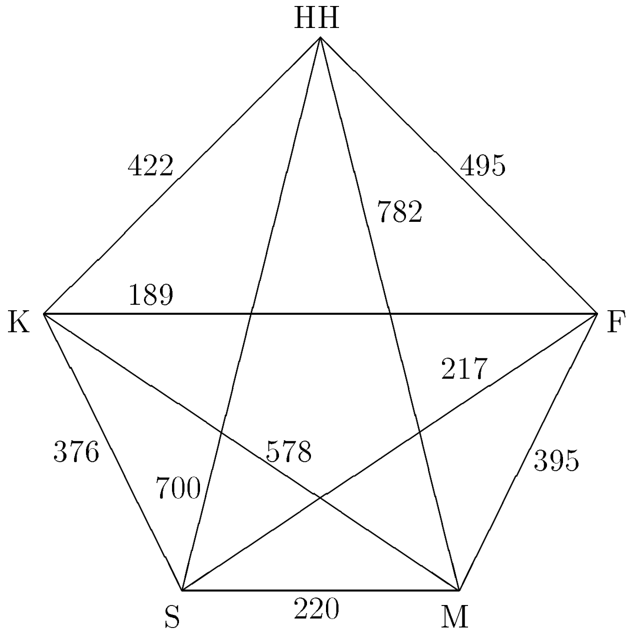
die nonvar-Prädikate sind zur Ausführung der Arithmetik notwendig

Anfrage : Wer ist jetzt alles tot?
trace

```
?- tot(X,1996).
call    :  tot(X,1996)
call    :    sterblich(X)
call    :      mensch(X)
call    :        mann(X)
return  :        mann(marcus)
return  :      mensch(marcus)
return  :    sterblich(marcus)
call    :    geboren(marcus,T0)
return  :    geboren(marcus,40)
eval    :    T3 is 1952
eval    :    1952>150
return  :  tot(marcus,1996)

X    = marcus ;                    /* weitere Lösungen? */
redo    :  tot(X,1996)
call    :    gestorben(X,T1)
call    :      pompejaner(X)
return  :      pompejaner(marcus)
return  :    gestorben(marcus,79)
eval    :    1996>79
return  :  tot(marcus,1996)

X    = marcus ;                    /* weitere Lösungen? */

no
```

⇒ *Markus ist nochmal 2 mal tot !!!*

Traveling Salesman



HH	:	Hamburg
K	:	Köln
F	:	Frankfurt
S	:	Stuttgart
M	:	München

WEGABHÄNGIGE Lösung !

↪ nur mit HEURISTIKEN
$\Rightarrow$ SUBOPTIMAL

Bestensuche $\longrightarrow$ *zufriedenstellende Suche*

Ist die Wissensbasis konsistent ?

Beispiel Mathematisches Axiomensystem (Gruppe, ...)

↪ Konsistenzbeweis
$\rightarrow$ *inkonsistent*
$\rightarrow$ *alles kann bewiesen werden*

Beispiel Vereinfachtes Modell der Realität

↪ Euklidische Geometrie auf der Erdkugel

☞ Alle Modelle der realen Welt enthalten Vereinfachungen

↪ Diognosesysteme, Expertensysteme für Rechtsanwälte,. . .

Welche Rolle spielt das Wissen?

Beispiel Schach : (fast) keine (endliches Spiel)

↪ gute Schachprogramme
$\rightarrow$ Eröffnungsbibliothek

Beispiel Expertensystem für Diagnose, Auskunftsystem

↪ Wissen wichtiger als Regeln

Erfordert die Aufgabe einen Dialog mit einer Person?

Solitär die Maschine erzeugt eine Antwort ohne Kommunikation und ohne Erklärung des Lösungswegs (Batch)

Dialog zwischen Mensch & Maschine

$\hookrightarrow$ *Hilfestellung* bei der Lösung

$\hookrightarrow$ *Erklärung* bei der Lösung

Beispiel Schach, Spiele — Solitär

Beispiel Expertensysteme, Diagnose — Dialog

Problemkennwerte

Zerlegbarkeit	:	Integration	$\rightarrow$ *ja* $\rightarrow$ $\left\{\begin{array}{l}\text{teile und}\\ \text{herrsche}\end{array}\right.$
		Bausteine	$\rightarrow$ *nein*
Lösungsschritte ignorieren	:	Theorembeweise	$\rightarrow$ *ja*
		8-puzzle	$\rightarrow$ *rückgängig machen*
		Schach	$\rightarrow$ *nein*
Voraussagbarkeit	:	8-puzzle	$\rightarrow$ *ja*
		Tic–Tac–Toe	$\rightarrow$ *ja*
		Kartenspiele	$\rightarrow$ *nein*
gute Lösung	:	Markus tot ?	$\rightarrow$ *wegunabhängig*
		Traveling Salesman	$\rightarrow$ *wegabhängig* (schwierig)
Konsistenz der Wissensbasis	:	Axiomensystem	$\stackrel{\neg}{\rightarrow}$ *alles kann bewiesen werden*
Rolle des Wissens	:	Schach, Spiele	$\rightarrow$ *klein*
		Expertensysteme	$\rightarrow$ *groß*
Dialog	:	Schach, Spiele	$\rightarrow$ *nein*
		Expertensysteme	$\rightarrow$ *ja*

3.6 Produktionssystemkennwerte

monotones Produktionssystem

Die Anwendung einer Regel verhindert nicht die Anwendung einer anderen Regel.

partiell kommutatives Produktionssystem

$X \stackrel{s}{\longmapsto} Y$

Transformation von X durch s nach Y:

Jede anwendbare Permutation der Regeln aus s ist zugelassen (unabhängig von der Reihenfolge).

kommutativ $\Longleftrightarrow$ **monoton & part. kommutativ**

PRODUKTIONSKENNWERTE $\Longleftrightarrow$ PROBLEMKENNWERTE

	monoton	**nicht monoton**
partiell kommutativ	Theorembeweise	Bausteine
nicht partiell kommutativ	chemische Synthese	Kartenspiele

kommutativ $\Longleftrightarrow$ ignorierbare Probleme

3.7 Zusammenfassung

☞ genaue Problemdefinition

↪ Problemraum

↪ Operatoren, Transformationen

↪ Anfangs- u. Endzustand

☞ Problemanalyse mit Blick auf

↪ Zerlegbarkeit

↪ Wiederherstellbarkeit

↪ Voraussagbarkeit

↪ Lösungsgüte

↪ Konsistenz der Wissensbasis

↪ Bedeutung des Wissens

↪ Dialogführung

☞ Produktionssystemklassen

↪ monoton

↪ partiell kommutativ

↪ kommutativ

4 Grundlegende Problemlösungsverfahren

Probleme in der KI kompliziert

↪ keine direkten Lösungsverfahren
↪ Suchverfahren
↪ Explosion der Komplexität
↪ heuristische Suche

☞ Varianten der heuristischen Suche

↪ unabhängig von speziellen Aufgabenbereichen
↪ Effektivität hängt stark vom Wissen ab
↪ Effizienz hängt stark vom Wissen ab, das verwertet wird

schwachen Verfahren kombinatorische Explosion ist immer noch möglich

↪ Kern der meisten KI–Systeme

Suchprozeß Durchlaufen eines gerichteten Grapßen mit

Knoten: Problemzustände

Kanten: Zustandsübergänge, Operationsanwendungen, Transformationen

4.1 Bäume und Graphen

<u>TRANSFORMATION : BAUM ⇒ GRAPH</u>

Knoten mit *gleichen Attributen* brauchen *nicht unterschieden* zu werden

↪ *Identifikation* der Knoten nur durch ihr *Attribut*

↪ Attributwertebereich ⇒ Knotenwertebereich

↪ Vereinfachung des Datenmodells

↪ Reduktion der Anzahl der Knoten

Konsequenz

↪ *explizite Konstruktion* des Suchgraphen

↪ Speicherplatzproblem

nicht mehr *implizite Konstruktion*

↪ Konstruktion nur der Knoten, die gerade untersucht werden
↪ Knoten als Aufrufe von Prozeduren
↪ einfache Speicherplatzverwaltung

Beispiel

- *recursiv decent* parsen
  - implizite Konstruktion des Syntaxbaums
- *Prolog Tiefensuche*
  - implizite Konstruktion des Suchbaumes

Bäume $\Longleftrightarrow$ Graphen

Problembäume

Eigenschaften

gut
- Leicht zu verwalten
- immer nur ein Vorgänger
- keine Zyklen $\Rightarrow$Terminierung
- implizite Darstellung durch Prozeduraufrufe

schlecht
- doppelte gleiche Knoten
- Mehrfachverarbeitung $\Rightarrow$nicht sehr effizient

Problemgraphen

Eigenschaften

gut
- keine Duplikate
- weniger Speicherplatz
- geringere Laufzeit

schlecht
- zusätzliche Buchführung
- mehrere Vorgänger
- keine implizite Darstellung mit Prozeduren (Coroutinen)

Transformation: Baum $\Rightarrow$Graph

Algorithmus

.1 Überprüfe ob ein neu zu erzeugender Knoten schon vorhanden ist

.2 Wenn nicht : Knoten eintragen

.3 Wenn ja : Kante eintragen

.4 wenn eine Kostenfunktion für den Weg berechnet wird, prüfe ob der neue Weg besser als frühere Wege ist und verringere die Kosten

Wichtige Punkte bei der Suche

Richtung in der die Suche durchzuführen ist

Topologie des Suchprozesses

↪ Baum
↪ zyklenfreier Graph
↪ allgemeiner Graph

Darstellung der Knoten

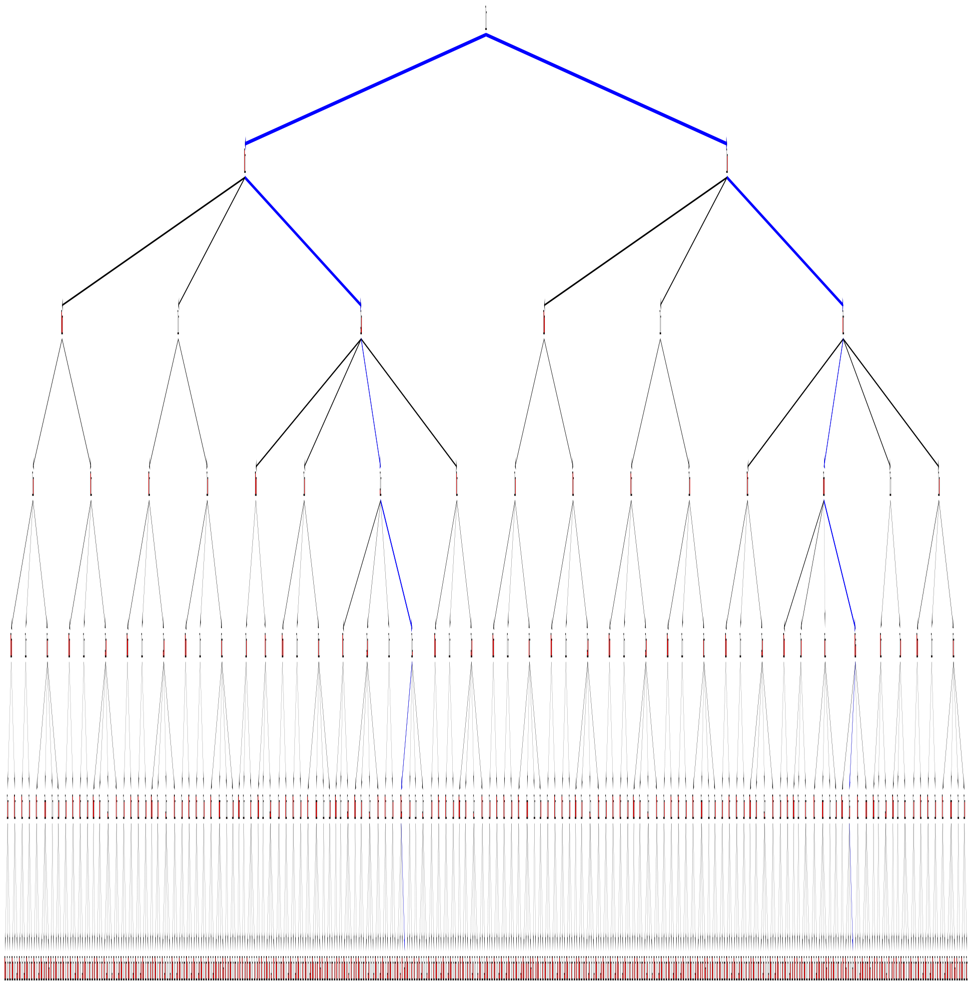
↪ Prozeduraufrufe
↪ Zeigerstruktur
↪ Attribute

Auswahl der anwendbaren Regeln

Heuristische Funktion zur Steuerung und Bewertung der Operationen

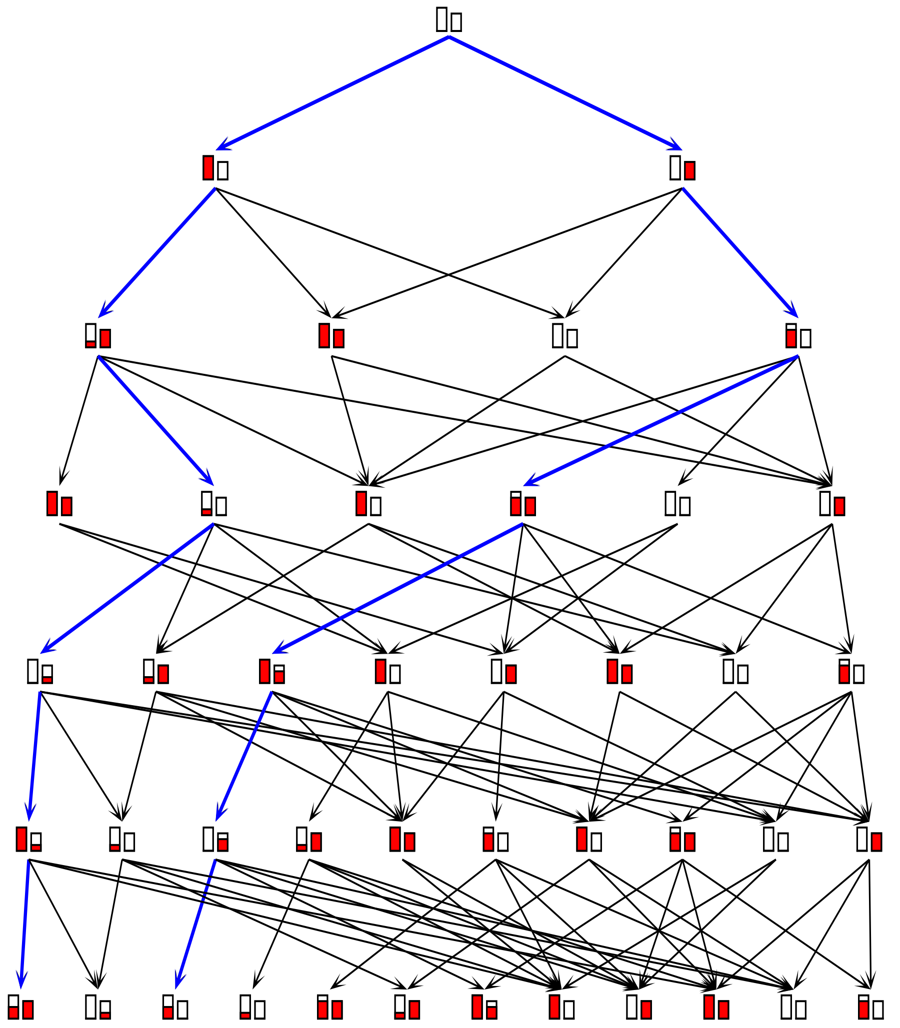
↪ nächste Abschnitte

Suchbaum: Wasserkrugproblem



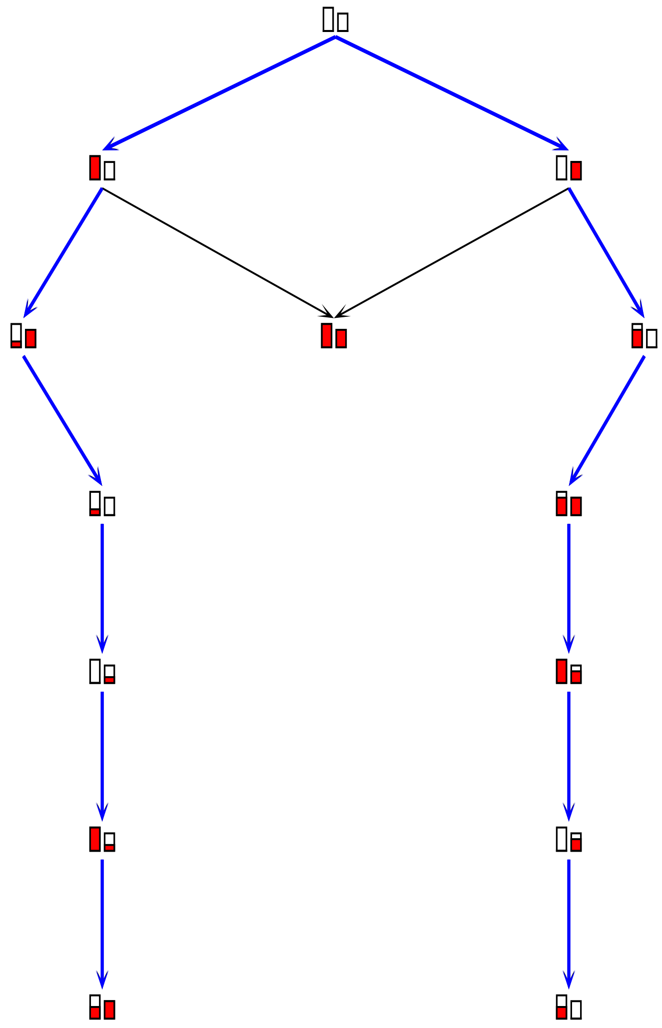
Suchbaum der Suchtiefe 6

Suchgraph: Wasserkrugproblem



Suchgraph der Suchtiefe 6 ohne Erkennung von Zyklen

Suchgraph: Wasserkrugproblem



Suchgraph der Suchtiefe 6 mit Erkennung von Zyklen

4.2 Suchstrategien

<u>VORWÄRTSSUCHE $\Longleftrightarrow$ RÜCKWÄRTSSUCHE</u>

vorwärts von den *Start*zuständen ausgehen

rückwärts von den *End*zuständen ausgehen

☞ beide Suchen sind grundsätzlich mit einem Produktionssystem möglich

vorwärts *linke* Seiten *matchen*

rückwärts *rechte* Seite *matchen*

Parsen

rückwärts *LR*-parsen
Programm $\rightarrow$ Startsymbol

vorwärts *LL*-parsen
Startsymbol $\rightarrow$ Programm

8-puzzle

symmetrisch vorwärts $\Longleftrightarrow$ rückwärts

beeinflußende Faktoren

- mehr Startzustände $\longrightarrow$ rückwärts
mehr Endzustände $\longrightarrow$ vorwärts
- Verzweigungsfaktor $\longrightarrow$ Richtung
- Erklärungsprozeß (notwendig ?)
$\hookrightarrow$ Richtung in die Benutzer denken

<u>VERZWEIGUNGSFAKTOR</u>

Theorembeweis

Startzustand kleine Menge von Axiomen

$\approx$ gleiche Mächtigkeit

Endzustand ein Theorem

Verzweigungs–faktor Start $\nearrow$ viele $\searrow$ Theoreme groß

Ziel $\leftarrow$ $\left\{\text{zurückführen auf Axiome}\right.$ klein

$\Rightarrow$ rückwärts schließen

Erklärungsprozess

	(Akzeptanz!)
MYCIN :	Diagnose Infektionskrankheiten
wenn	Nackensteife
und	hohes Fieber
und	Bewußtseinstrübung
dann	Verdacht auf Meningitis
Frage :	Warum soll ich Nackensteife prüfen?
Erklärung :	Es würde helfen zu bestimmen, ob Verdacht auf Meningitis besteht
$\Rightarrow$	vorwärts schließen

Bidirektionale Suche

Kombination	Vorwärtssuche und Rückwärtssuche
$\hookrightarrow$	ausgehend von einem Start- und einem Endzustand wird durch *gleichzeitige* Vorwärts- und Rückwärtssuche ein gemeinsamer *mittlerer* Zustand gesucht
Strategie	*Ähnlichkeiten* erhöhen
Gefahr	Die beiden Seiten können aneinander vorbeilaufen $\hookrightarrow$ doppelte Arbeit

4.3 Wissensrepräsentation

Beispiele
- ↪ Darstellung des Graphen wird schnell komplex
- ↪ Ablaufsteuerung wird komplex

Strukturierung

Knoten Wie wird ein *Knoten* dargestellt?

Wasserkrug : Zahlenpaar

Objekte Wie werden die *Objekte kombiniert*?

Baum, Graph (mit Kosten)

↪ [Zustandsfolgen] Wie werden *Zustandsfolgen* dargestellt ?

implizit, explizit

Rahmenprogramm Wie wird der Suchprozeß organisiert

↪ fester Anfangszustand und Operationen an Kanten
- ↪ Rückschritt einfach
- ↪ momentane Zustandsbeschreibung schwierig (Schachpartie)

↪ Modifikation des Anfangszustandes und Protokollierung über Aufhebung der Op's
- ↪ Rückschritt schwierig
- ↪ momentane Zustandsbeschreibung einfach (Schachbrett)

Darstellung Prädikatenlogik, Objektorientierte Darstellung (Zusammenfassung : Daten u. Operationen)

4.4 Tiefensuche

Prolog–Suchstrategie
Tiefensuche
Regeln von oben nach unten
das Wasserkrug Problem: die Regeln

```
:-op(200,xfx,:).

rule(X:Y, 4:Y) :-
        X < 4,                    write('Krug4 fuellen'),nl.
rule(X:Y, X:3) :-
        Y < 3,                    write('Krug3 fuellen'),nl.
rule(X:Y, 0:Y) :-
        X > 0,                     write('Krug4 leeren'),nl.
rule(X:Y, X:0) :-
        Y > 0,                     write('Krug3 leeren'),nl.
rule(X:Y, 4:Z) :-
        X + Y >= 4,
        Z is X+Y-4, write('Krug3 teilweise --> Krug4'),nl.
rule(X:Y, Z:3) :-
        X + Y >= 3,
        Z is X+Y-3, write('Krug4 teilweise --> Krug3'),nl.
rule(X:Y, Z:0) :-
        Y > 0, X + Y < 4,
        Z is X + Y,      write('Krug3 ganz --> Krug4'),nl.
rule(X:Y, 0:Z) :-
        X > 0, X + Y < 3,
        Z is X + Y,      write('Krug4 ganz --> Krug3'),nl.
```

Die Steuerung der Zustandstransformationen
das Prädikat `trans` *sucht einen Weg*
vom Anfangszustand `S0` *zum Ziel* `Sn`
a) Erfolg

```
trans(S0,Sn)
   :- S0 = Sn, !.
```

b) eine Regel ausprobieren
⇒*neuer Zustand* `S1`
⇒*Prozess wiederholen*

```
trans(S0,Sn)
   :- rule(S0,S1),trans(S1,Sn).
```

das Ziel: leere Krüge ⇒ 2 Liter im Krug4

```
zweiliterendlos :-
        trans(0:0,2:_).
```

⇒ *Ziel* `zweiliter`*: Endlosschleife*
einfüllen ⇒ ausleeren ⇒ einfüllen ⇒ ...
⇒ *nie ein Ergebnis*

```
Krug4 fuellen   Krug3 fuellen   Krug4 leeren
Krug4 fuellen   Krug4 leeren
...
Krug4 fuellen   Krug4 leeren

Stack overflow.
Re-configure with Options if necessary.
```


Tiefensuche mit Beschränkung der Tiefe
Regeln unverändert
Prädikat `trans` *erhält Zähler für Rekursionstiefe*
und Wert für maximale Tiefe
a) Lösung gefunden

```
trans(S0,S0,_,_) :-
        write('  Loesung gefunden'),nl,
        !.
```

b) Suche zu tief

```
trans(_,_,Max,Max) :-
        write('  Sackgasse'),nl,
        !,
        fail.
```

c) weitersuchen mit erhöhter Tiefe

```
trans(S0,Sn,N,Max) :-
        N1 is N + 1,
        rule(S0, S1),
        trans(S1, Sn, N1, Max).
```

das Ziel: 0:0 ⇒2:x in Max Schritten

```
zweiliter(Max) :-
        trans(0:0, 2:_, 0, Max).
```

$Max \leq 6 \Rightarrow$ *keine Lösung*
$Max \geq 7 \Rightarrow$ *eine Lösung*
$Max > 7 \Rightarrow$ *Suche ineffizient*

Ziele `zweiliter(2)`,...,`zweiliter(6)`

```
no
```

Ziel `zweiliter(7)`

```
Krug4 fuellen
  Krug3 fuellen
    Krug4 leeren
      Krug4 fuellen
        Krug4 leeren
          Krug4 fuellen                         Sackgasse
          Krug3 leeren                          Sackgasse
          Krug4 teilweise --> Krug3             Sackgasse
          Krug3 ganz --> Krug4                  Sackgasse
        Krug3 leeren
          Krug3 fuellen                         Sackgasse
          ...
        Krug4 teilweise --> Krug3
          Krug4 leeren                          Sackgasse
      ...
      Krug3 leeren
        Krug4 fuellen
          Krug3 fuellen                         Sackgasse
...
          Krug3 teilweise --> Krug4             Sackgasse
          Krug4 teilweise --> Krug3      Loesung gefunden
yes
```

Ziele `zweiliter(8)`,...

```
...                                      sehr lange Suche
yes
```


Tiefensuche mit Erklärungskomponente
trace Ausgabe wird ersetzt durch Ausgabe des Wegs
Prädikat `trans` *wird um ein Argument (eine Liste)*
für den Weg erweitert
a) Lösung gefunden

```
trans(S0, S0, _, _, P) :-
        S0 = Sn,
        !,
        nl, writepath(P),nl.

writepath([S]) :- write(S).
writepath([S|T]) :-
        T = [_|_],
        writepath(T),write(' --> '),write(S).
```

b) Suche zu tief

```
trans(_,_,Max,Max,_) :-
        !, fail.
```

c) weitersuchen mit verlängertem Weg

```
trans(S0, Sn, N, Max, P0) :-
        N1 is N + 1,
        rule(S0, S1),
        trans(S1, Sn, N1, Max, [S1 | P0]).
```

das Ziel: der Weg besteht aus dem Startknoten

```
zweiliter(Max) :-
        trans(0:0, 2:_ ,0, Max, [0:0]).
```

Programmverhalten wie im vorigen Beispiel

Ziele `zweiliter(2)`,...,`zweiliter(6)`

```
no
```

Ziel `zweiliter(7)`

```
0:0
 --> 4:0
 --> 1:3
 --> 1:0
 --> 0:1
 --> 4:1
 --> 2:3

yes
```


Tiefensuche
zufällige Auswahl der Regeln
Die Auswahl einer Regel wird durch eine Zahl als zusätzliches 1. Argument gesteuert
⇒ `rule` *wird 3-stelliges Prädikat*

```
:-op(200,xfx,:).

rule(1, X:Y, 4:Y) :-
        X < 4.
rule(2, X:Y, X:3) :-
        Y < 3.
rule(3, X:Y, 0:Y) :-
        X > 0.
rule(4, X:Y, X:0) :-
        Y > 0.
rule(5, X:Y, 4:Z) :-
        X < 4, X + Y >= 4, Z is X + Y - 4.
rule(6, X:Y, Z:3) :-
        Y < 3, X + Y >= 3, Z is X + Y - 3.
rule(7, X:Y, Z:0) :-
        Y > 0, X + Y < 4, Z is X + Y.
rule(8, X:Y, 0:Z) :-
        X > 0, X + Y < 3, Z is X + Y.
```

Die Steuerung der Zustandstransformationen
a) Klausel für Lösung gefunden

```
trans(S0,S0) :-
     !, write('Loesung gefunden'),nl.
```

b) Klausel für Auswahl einer Regel und probieren mit Zufallszahlengenerator

```
trans(S0,Sn) :-
        choose(N),
        write('('),write(N),write(')'),
        rule(N,S0,S1),
        write(' --> '),write(S1),nl,
        trans(S1,Sn).
```

c) weitermachen wenn Regel nicht anwendbar war

```
trans(S0,Sn) :-
    trans(S0,Sn).
```

⇒ *manchmal Lösung*
manchmal Stackoverflow

⇒ *Suche nicht zielgerichtet*

1.Versuch: 129 Transformationen ⇒ stackoverflow

```
(1) --> 4:0
(7)(3) --> 0:0
(4)(1) --> 4:0
(7)(7)(8)(7)(4)(8)(5) --> 4:0
...
(7)(3) --> 0:0
(6)(5)(5)(5)(1) -->                          Stack overflow.
```

2.Versuch: 129 Transformationen ⇒ stackoverflow

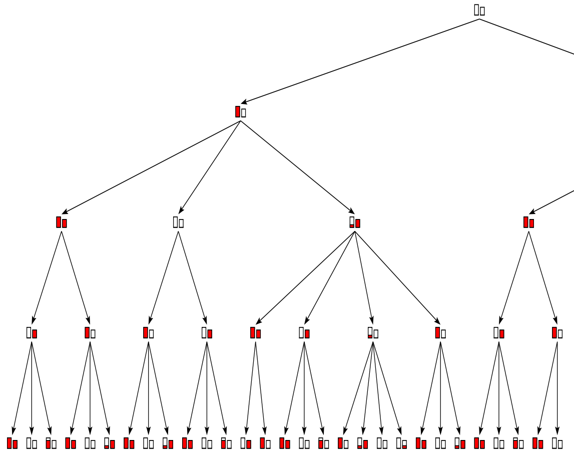
```
(6)(3)(1) --> 4:0
(1)(4)(3) --> 0:0
(7)(3)(5)(4)(6)(4)(1) --> 4:0
(2) --> 4:3
...
(3)(8)(4) --> 0:0
(4)(7)(4)(7)(4)(2) -->                       Stack overflow.
```

3.Versuch: 30 Transformationen ⇒ Lösung

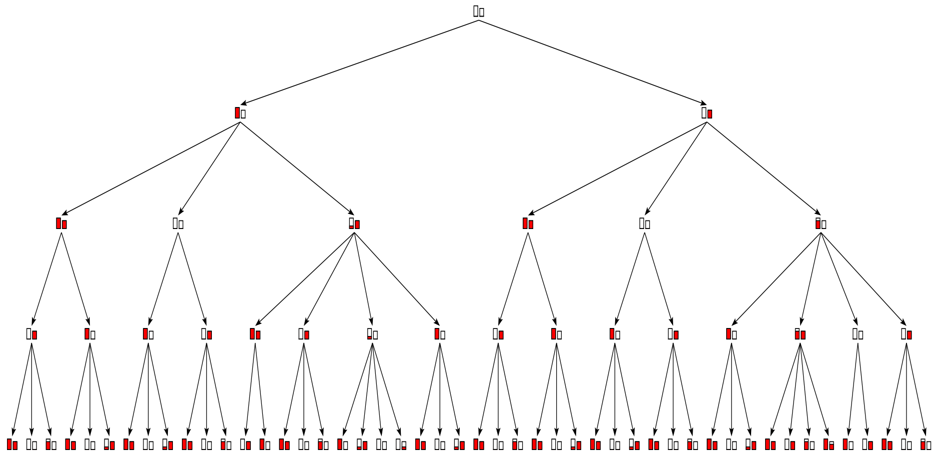
```
(3)(6)(1) --> 4:0
(1)(7)(1)(5) --> 4:0
(2) --> 4:3
(4) --> 4:0
(7)(6) --> 1:3
...
(5)(7) --> 1:0
(8) --> 0:1
(5)(1) --> 4:1
(6) --> 2:3
Loesung gefunden
```

4.Versuch: wieder Stackoverflow

Tiefensuche mit Beschränkung der Suchtiefe: Wasserkrugproblem



teilweise aufgebauter Suchbaum der Suchtiefe 4



vollständig aufgebauter Suchbaum der Suchtiefe 4

4.5 Breitensuche

Breitensuche in einem Baum:

a) *generiere aus einer Liste von Knoten alle Nachfolger*

b) *teste ob Lösungsknoten in der Liste enthalten ist*

Regeln wie in der zufälligen Tiefensuche

Regeln werden über ihr 1. Argument ausgewählt

die Prädikattypen

```
zweiliter(Max) :-
        solve([0:0], 2:_, 0, Max), !.

zweiliter(Max) :-
        write('Keine Loesung in '),
        write(Max),
        write(' Schritten moeglich'),nl.
```

das Prädikat `solve` *Argumente:*
1) eine Liste von Knoten
2) ein Zielknoten
3-4) Rekursionstiefe und Rekursionsbeschränkung
a) Erfolg

```
solve(Statelist,G,N,_) :-
        isin(G,Statelist),
        !,
        write('Loesung im '),write(N),write('. Schritt'),nl.
```

b) zu tief

```
solve(_,_,Max,Max) :-
        !, fail.
```

c) alle Nachfolger erzeugen und Prozess wiederholen

```
solve(Statelist,G,N,Max) :-
        gen(Statelist,Succ),
        N1 is N + 1,
        write(N1),write('. Schritt:'),nl,
        write(Succ),nl,
        solve(Succ,G,N1,Max).

isin(S,[H|_]) :- S = H,!.
isin(S,[_|T]) :- isin(S,T).
```

---

die Generierungsklauseln
a) `gen1`*: alle Nachfolger eines Knotens*
Schleife über alle anwendbaren Regeln

```
gen1(S,L) :-
        findall(S1,rule(S,S1),L).
```

Erweiterung von `gen1` *auf eine Liste*
a) Liste leer ⇒ Resultat leere Liste
b) Liste nicht leer ⇒ Kopf der Liste verarbeiten
⇒Rest der Liste verarbeiten
⇒zusammensetzen

```
gen([],[]).

gen([S|T],Succ) :-
        gen1(S,Succ1),
        gen(T,  Succ2),
        append(Succ1,Succ2,Succ).

append([],L,L).

append([H|R],L,[H|L1]) :-
        append(R,L,L1).
```

⇒ *generieren einfach*
⇒ *aber Knoten mehrfach in Liste*
⇒ *Suchbaum zu breit !!!*

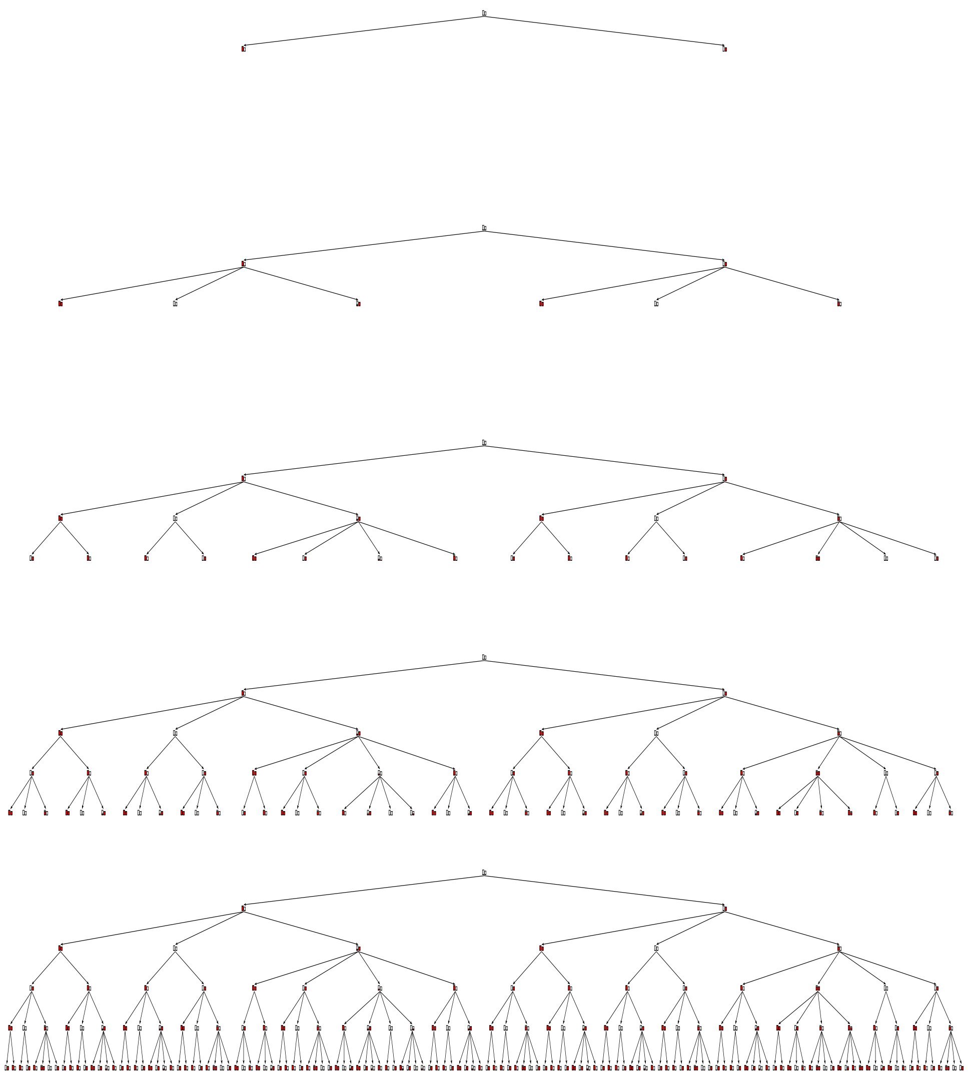
die Ziele: `zweiliter(2)`,...,`zweiliter(5)`

```
zweiliter(n)
1. Schritt:
   ...
2. Schritt:
   ...
n. Schritt:
   ...
Keine Loesung in n Schritten moeglich
```


das Ziel: `zweiliter(7)`

```
1. Schritt:                                          2 Knoten
[4:0,0:3]
2. Schritt:                                          8 Knoten
[4:3,0:0,4:0,1:3,4:3,0:0,0:3,3:0]
3. Schritt:                                         29 Knoten
[0:3,4:0,4:3,4:3,4:0,0:3,4:3,0:0,4:0,1:3,4:3,0:3,1:0,4:0,
 1:3,0:3,4:0,4:3,4:3,4:0,0:3,4:3,0:0,0:3,3:0,4:0,3:3,0:0,
 0:3]
4. Schritt:                                        113 Knoten
[4:3,0:0,0:3,3:0,4:3,0:0,4:0,1:3,0:3,4:0,4:3,4:3,0:3,4:0,
 4:3,4:3,4:3,0:0,4:0,1:3,4:3,0:0,0:3,3:0,0:3,4:0,4:3,4:3,
 4:0,0:3,4:3,0:0,4:0,1:3,4:3,0:3,1:0,4:0,1:3,0:3,4:0,4:3,
 4:3,4:3,0:0,0:3,3:0,4:0,1:3,0:0,0:1,4:3,0:0,4:0,1:3,4:3,
 0:3,1:0,4:0,1:3,4:3,0:0,0:3,3:0,4:3,0:0,4:0,1:3,0:3,4:0,
 4:3,4:3,0:3,4:0,4:3,4:3,4:3,0:0,4:0,1:3,4:3,0:0,0:3,3:0,
 0:3,4:0,4:3,4:3,4:0,0:3,4:3,0:0,0:3,3:0,4:0,3:3,0:0,0:3,
 4:3,0:0,4:0,1:3,4:3,0:3,3:0,4:2,3:3,4:0,0:3,4:3,0:0,0:3,
 3:0]
5. Schritt:                                        427 Knoten
[0:3,4:0,4:3,4:3,4:0,0:3,4:3,0:0, ... ,0:3]
```

Breitensuche in einem Baum:Wasserkrugproblem



Suchbäume der Suchtiefen 1,2,3,4 und 5

Breitensuche in einem Graph:
Generierungsprozess wie bei einem Baum
aber aus der Liste der Nachfolger werden die Duplikate gelöscht
⇒ `append` *wird modifiziert zu* `union`
Listen verknüpfen ⇒ Mengen vereinigen

```
gen([],[]).
gen([S|T],Succ) :-
        gen1(S,Succ1),
        gen(T,  Succ2),
        union(Succ1,Succ2,Succ).

union([],S,S).
union([H|T],S2,S) :-
        isin(H,S2),
        !,
        union(T,S2,S).
union([H|T],S2,[H|S]) :-
        union(T,S2,S).
```

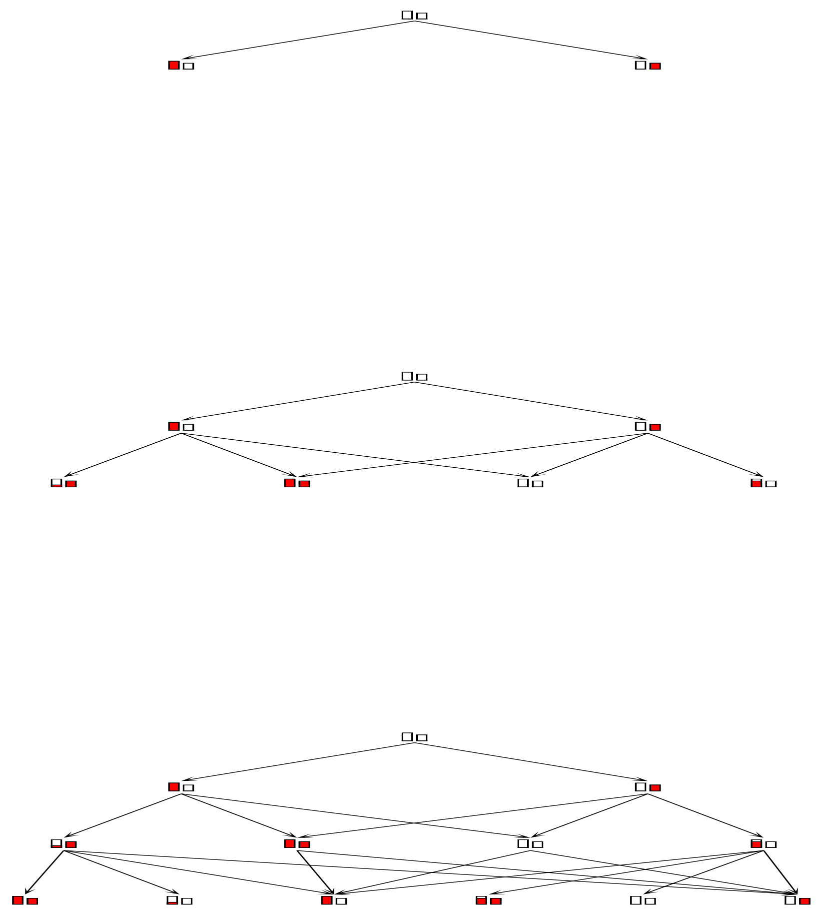
⇒ `union` *ist aufwendiger als* `append`
zusätzlicher `isin`*-Test*
⇒ *aber die Breite des Suchbaumes (genauer -graphen) wird reduziert*
⇒ *Effizienzgewinn !!!*
⇒ *aber Zyklen werden nicht berücksichtigt*

das Ziel `zweiliter(7)`

```
1. Schritt:                                   2 Knoten
[4:0,0:3]

2. Schritt:                                   6 Knoten
[4:0,1:3,4:3,0:0,0:3,3:0]

3. Schritt:                                   8 Knoten
[1:0,1:3,4:3,3:0,4:0,3:3,0:0,0:3]

4. Schritt:                                  10 Knoten
[0:1,1:0,1:3,4:2,3:3,4:0,4:3,0:0,0:3,3:0]

5. Schritt:                                  12 Knoten
[4:1,0:1,1:0,0:2,4:2,1:3,4:3,3:0,4:0,3:3,0:0,0:3]

6. Schritt:                                  14 Knoten
[2:3,4:1,0:1,2:0,0:2,1:0,1:3,4:2,3:3,4:0,4:3,0:0,0:3,3:0]

Loesung im 6. Schritt
```

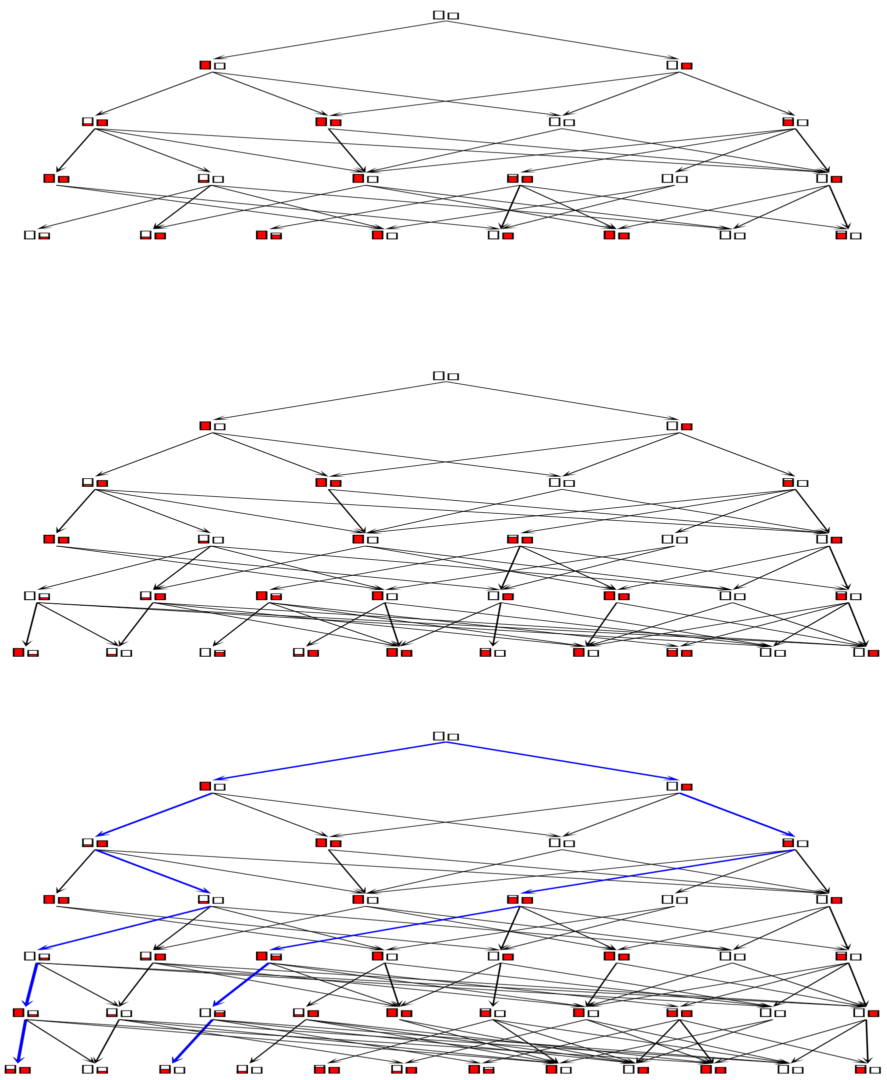
2 Lösungen !!!

Breitensuche in einem Graphen: Wasserkrugproblem



Suchgraphen der Suchtiefen 1,2 und 3

Breitensuche in einem Graphen (cont): Wasserkrugproblem



Suchgraphen der Suchtiefen 4,5 und 6

Breitensuche in einem Graph mit Elimination von Zyklen
Generierungsprozess wie oben
zusätzlich wird die Menge der schon besuchten Knoten bei der Generierung herausgefiltert
⇒ `solve` *wird erweitert um ein Argument für die schon besuchten Knoten*
`minus` *ist die Mengensubtraktion*

```
solve(Statelist,G,N,_,_) :-
       isin(G,Statelist),
       !,
       write('Loesung im '),write(N),write('. Schritt'),nl.

solve(_,_,Max,Max,_) :-
       !, fail.

solve(Statelist,G,N,Max,Visited) :-
       gen(Statelist,Succ),
       union(Visited,Statelist,Newvisited),
       minus(Succ,Newvisited,Newsucc),
       N1 is N + 1,
       write(N1),write('. Schritt:'),nl,
       write(Newsucc),nl,
       solve(Newsucc,G,N1,Max,Newvisited).
```

⇒ `solve` *ist aufwendiger als vorher*
⇒ *aber die Breite des Suchbaumes (genauer -graphen) wird nochmals reduziert*
⇒ *Effizienzgewinn !!!*

das Ziel `zweiliter(7)`

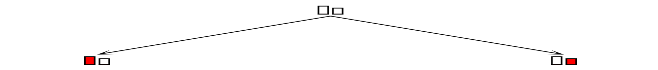
```
1. Schritt:                                       2 Knoten
[4:0,0:3]

2. Schritt:                                       3 Knoten
[1:3,4:3,3:0]

3. Schritt:                                       2 Knoten
[1:0,3:3]

4. Schritt:                                       2 Knoten
[0:1,4:2]

5. Schritt:                                       2 Knoten
[4:1,0:2]

6. Schritt:                                       2 Knoten
[2:3,2:0]

Loesung im 6. Schritt
```

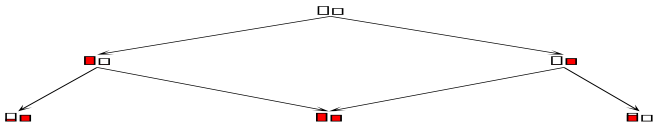
2 Lösungen !!!

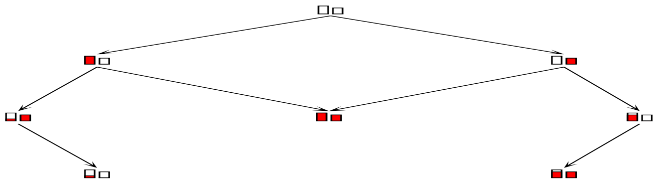
Breitensuche mit Erklärungskomponente
alle Zustände werden durch Listen dargestellt
Kopf der Liste: Knoten
Rest der Liste: Weg vom Startknoten zum Knoten

```
1. Schritt:
[[4:0,0:0]
,[0:3,0:0]]
2. Schritt:
[[1:3,4:0,0:0]
,[4:3,0:3,0:0]
,[3:0,0:3,0:0]]
3. Schritt:
[[1:0,1:3,4:0,0:0]
,[3:3,3:0,0:3,0:0]]
4. Schritt:
[[0:1,1:0,1:3,4:0,0:0]
,[4:2,3:3,3:0,0:3,0:0]]
5. Schritt:
[[4:1,0:1,1:0,1:3,4:0,0:0]
,[0:2,4:2,3:3,3:0,0:3,0:0]]
6. Schritt:
[[2:3,4:1,0:1,1:0,1:3,4:0,0:0]
,[2:0,0:2,4:2,3:3,3:0,0:3,0:0]]

Loesung(en) im 6. Schritt
Loesung:
(0,0) --> (4,0) --> (1,3) --> (1,0) --> (0,1) --> (4,1)
      --> (2,3)
Loesung:
(0,0) --> (0,3) --> (3,0) --> (3,3) --> (4,2) --> (0,2)
      --> (2,0)
```

Breitensuche in einem Graphen: Wasserkrugproblem

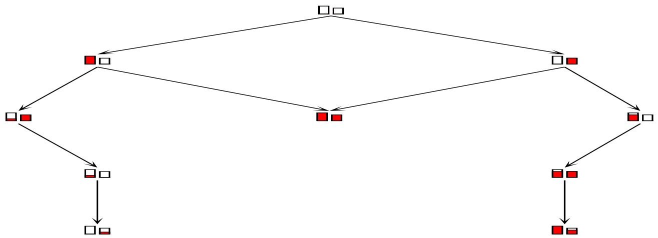


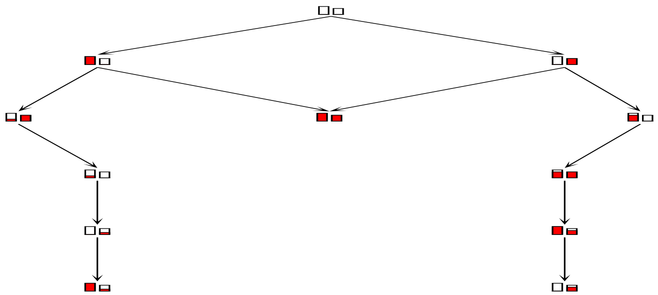


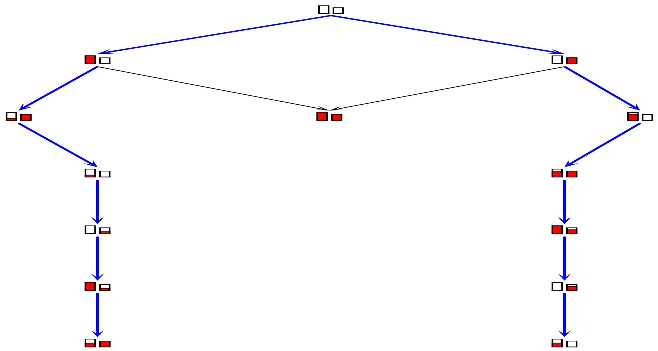


zyklenfreie Suchgraphen der Suchtiefen 1,2 und 3

Breitensuche in einem Graphen (cont): Wasserkrugproblem







zyklenfreie Suchgraphen der Suchtiefen 4,5 und 6

4.6 Der A^*–Algorithmus

DER A^*–ALGORITHMUS

A^*–Algorithmus Arbeitsweise

Prinzip

- Suche in einem gerichteten Graphen
- Zyklen werden erkannt und gelöscht

Idee die Menge der Knoten eines Graphen wird partitioniert in zwei Klassen

- die schon besuchten und analysierten Knoten
- die noch zu analysierenden, die offenen Knoten.

Alle offenen Knoten werden bewertet. Diese Bewertung setzt sich aus zwei Kostenfaktoren zusammen

- den Kosten für den Weg vom Startknoten bis zum Knoten
- eine Abschätzung für die Güte des Knotens, d.h. für die Kosten des Weges vom Knoten zu einem Lösungsknoten

Die zweie Komponente ist eine Schätzung. Sie wird durch eine heuristische Funktion berechnet Das Wissen über das Problem fließt in die heuristische Funktion ein.

Suchprozess

- der beste offene Knoten wird berechnet
- bildet dieser Knoten eine Lösung, hat man eine Lösung des Problems gefunden
- von diesem Knoten werden alle Nachfolger erzeugt
- der Knoten selbst wird aus der Menge der offenen Knoten entfernt und in die Menge der besuchten Knoten eingefügt.
- alle Nachfolgerknoten die schon in der Menge der besuchten Knoten enthalten sind werden nicht weiter behandelt
- dieser Erzeugungsprozess wird fortgesetzt bis eine Lösung gefunden wurde oder bis die Menge der offenen Knoten leer ist, in diesem Fall gibt es keine Lösung

Datenstrukturen

- die schon besuchten Knoten werden in einer Menge (Liste) gesammelt.
- Die offenen Knoten werden in einer Menge (Liste) gesammelt. Hier wird zusätzlich der Pfad vom Startknoten zum Knoten in einer Liste gespeichert,
  der Kopf der Liste repräsentiert den Knoten,
  der Rest den Pfad zum Knoten.

  Für die offenen Knoten wird zusätzlich die Bewertung der Knoten gespeichert

der A–Algorithmus am Beispiel des 8-puzzle*
der Problemraum des 8-puzzle
als Prolog–Datenstrukturen
ein Zustand

```
<i> + [zahl1, ... , '-', ..., zahl9]
                     ^
                     i-te Stelle
```

i	Die Position des freien Feldes, ein Wert zwischen 1 und 9
zahl	Die Spielsteine dargestellt als Werte zwischen 1 und 9. ein '-' repräsentiert das leere Feld
Brett	Die Spielstellungen dargestellt als eine Liste mit 9 Elementen.
Zustand	Ein Zustand dargestellt als Paar aus Position des freien Feldes und des Felds selbst. Die Position des freien Feldes ist redundant, beschleunigt aber das Ziehen

---

die 4 Transformationsregeln

```
rule('nach rechts', I+B, J+B1) :-
        0 =\= I mod 3, J is I + 1,          ziehe(B,I,J,B1).

rule('nach links',I+B,J+B1) :-
        1 =\= I mod 3, J is I - 1,          ziehe(B,I,J,B1).

rule('nach oben',I+B,J+B1) :-
        I > 3, J is I - 3,                  ziehe(B,I,J,B1).

rule('nach unten',I+B,J+B1) :-
        I < 7, J is I + 3,                  ziehe(B,I,J,B1).
```

das Hilfsprädikat `move`
`sel` *und* `chg` *implementieren Feldzugriff und Überschreiben einer Feldkomponente*

```
sel([Z | _], 1, Z) :- !.
sel([_ | R], I, Z) :- I1 is I -1, sel(R,I1,Z).

chg([_ | R], 1, Z, [Z | R])    :- !.
chg([Z1|R], I, Z, [Z1|R1]) :- I1 is I-1, chg(R,I1,Z,R1).

ziehe(B, I, J, B1) :- sel(B, J, Z),
        chg(B, I, Z, B0), chg(B0, J, '-', B1).

move(s(I,B),J,s(J,B1))
   :- sel(B,J,Z),chg(B,I,Z,B0),chg(B0,J,9,B1).
```

die Zielzustände als Prädikat

```
solution(9+[1,2,3,4,5,6,7,8,'-']).
```

---

die problemunabhängigen Datenstrukturen des A-Algorithmus*

Path	der Weg vom Startknoten zum momentanen Knoten, der Knoten selbst bildet den Kopf der Liste, der Pfad bildet die Erklärungskomponente
State_set	eine Menge von Knoten für die schon besuchten Knoten
Path_cost	die Wegekosten
Heuristic	die geschätzten Kosten für den Weg zum Ziel
Cost	die kombinierten Kosten
Attr_state	die Darstellung eines offenen Knotens als Paar aus Pfad und Kosten
States	die Menge aller offenen Knoten

die Funktionen für den Suchprozess

```
solve(+OpenStateList, +VisiteStateSet, +PathLen, +MaxLen,
                                                   -Solution)

test_solution(+OpenStateList, +VisitedStateSet,
            +BestAttrState,  +PathLen, +MaxLen, -Solution)

del_open_node (+OpenStateList, +BestAttrState,
                                              -OpenStateList)
add_open_nodes(+OpenStateList, +StateSet, +BestAttrState,
                                              -OpenStateList)
ins_open_nodes(+OpenStateList, +BestAttrState,
                                              -OpenStateList)
gen1(+OpenState, -StateSet)
```

solve hat als Parameter die Menge der offenen Knoten und die Menge der schon analysierten Knoten,
hier außerdem einen Zähler für die Generierungsschritte und eine obere Grenze zm Abbrechen der Suche.
solve testet ob die Suche erfolglos abzubrechen ist anderenfalls wird der beste Knoten aus der Menge der offenen Knoten gewählt und **test_solution** gerufen

test_solution hat die gleichen Parameter wie **solve**, zusätzlich aber noch den momentan besten Knoten.
Ist dieser eine Lösung, wird die Suche erfolgreich beendet, anderenfalls werden mit **gen1** neue Nachfolgerknoten erzeugt, und mit Prädikaten für Mengenoperationen die Menge der offenen und besuchten Knoten aktualisiert und der Suchprozess durch einen Aufruf von **solve** fortgesetzt

die Funktion `solve`
a) keine offenen Knoten mehr

```
solve([], _, N, _, empty(N)) :-
        !.
```

b) maximale Zahl der Generierungsschritte erreicht

```
solve(_, _, Max, Max, limit(Max)) :-
        !.
```

c) besten Knoten suchen

```
solve(Open, Visited, N, Max, Solution) :-
        sel_best_node(Open, Best),
        test_solution(Open, Visited, Best, N, Max, Solution).
```


die Funktion `test_solution`
a) Erfolg

```
test_solution(Open, _, Best, N, _, found(N, P, Open)) :-
        Best = P + _,
        P = S :-> _,
        solution(S),
        !.
```

b) nächste Generierungsschritt

```
test_solution(Open, Visited, Best, N, Max, Solution) :-
        Best = P + _,
        gen1(P, Succ),
        union1(P, Visited, Newvisited),
        minus(Succ, Newvisited, Newsucc),
        del_open_node(Open, Best, Open1),
        add_open_nodes(Open1, Newsucc, Best, Newopen),
        N1 is N + 1,
        solve(Newopen, Newvisited, N1, Max, Solution).
```

neue Knoten bewerten
und die Menge der offenen Knoten erweitern
Pfadkosten : `pathcost`
Nähe zu einer Lösung : `guess`

```
add_open_nodes(Open, [], _, Open).

add_open_nodes(Open, [S | R], Node, Newopen) :-
        add_open_nodes(Open, R ,Node, Newopen1),
        add_open_node(Newopen1, S ,Node, Newopen ).

add_open_node(Open, S1, Best, Newopen) :-
        Best = S + (G0 + _),
        pathcost(S, S1, G1),
        G is G0 + G1,
        guess(S1, H),
        ins_open_node(Open, S1 + (G + H), Newopen).
```


Testläufe
Ausgangsposition `[2,3,-,1,4,5,7,8,6]`
a) Breitensuche
Testausgabe

```
(0) Bester Knoten: [2,3,-,1,4,5,7,8,6] Kosten: (0,0)
(1) Bester Knoten: [2,3,5,1,4,-,7,8,6] Kosten: (1,0)
(2) Bester Knoten: [2,-,3,1,4,5,7,8,6] Kosten: (1,0)
(3) Bester Knoten: [2,3,5,1,4,6,7,8,-] Kosten: (2,0)
    ...
(6) Bester Knoten: [-,2,3,1,4,5,7,8,6] Kosten: (2,0)
(7) Bester Knoten: [2,3,5,1,4,6,7,-,8] Kosten: (3,0)
    ...
(14)Bester Knoten: [1,2,3,-,4,5,7,8,6] Kosten: (3,0)
(15)Bester Knoten: [2,3,5,1,-,6,7,4,8] Kosten: (4,0)
    ...
(30)Bester Knoten: [1,2,3,4,-,5,7,8,6] Kosten: (4,0)
(31)Bester Knoten: [2,-,5,1,3,6,7,4,8] Kosten: (5,0)
    ...
(50)Bester Knoten: [1,2,3,4,5,-,7,8,6] Kosten: (5,0)
(51)Bester Knoten: [-,2,5,1,3,6,7,4,8] Kosten: (6,0)
    ...
(88)Bester Knoten: [1,2,3,4,5,6,7,8,-] Kosten: (6,0)
```

Lösung

```
     [2,3,-,1,4,5,7,8,6]
 --> [2,-,3,1,4,5,7,8,6] --> [-,2,3,1,4,5,7,8,6]
 --> [1,2,3,-,4,5,7,8,6] --> [1,2,3,4,-,5,7,8,6]
 --> [1,2,3,4,5,-,7,8,6] --> [1,2,3,4,5,6,7,8,-]
Loesung in 88 Schritten gefunden
noch offene Knoten:      62
Laenge des Loesungswegs:  6
```

Heuristik: möglichst viele richtige Steine
Weglänge nicht wichtig
Testausgabe

```
(0)Bester Knoten: [2,3,-,1,4,5,7,8,6] Kosten: (0,0)
(1)Bester Knoten: [2,-,3,1,4,5,7,8,6] Kosten: (0,6)
(2)Bester Knoten: [-,2,3,1,4,5,7,8,6] Kosten: (0,5)
(3)Bester Knoten: [1,2,3,-,4,5,7,8,6] Kosten: (0,4)
(4)Bester Knoten: [1,2,3,4,-,5,7,8,6] Kosten: (0,3)
(5)Bester Knoten: [1,2,3,4,5,-,7,8,6] Kosten: (0,2)
(6)Bester Knoten: [1,2,3,4,5,6,7,8,-] Kosten: (0,0)
```

Lösung

```
    [2,3,-,1,4,5,7,8,6]
--> [2,-,3,1,4,5,7,8,6] --> [-,2,3,1,4,5,7,8,6]
--> [1,2,3,-,4,5,7,8,6] --> [1,2,3,4,-,5,7,8,6]
--> [1,2,3,4,5,-,7,8,6] --> [1,2,3,4,5,6,7,8,-]

Loesung in 6 Schritten gefunden
noch offene Knoten:      7
Laenge des Loesungswegs:  6
```


Heuristik: Anzahl der notwendigen Züge bis zum Ziel
Weglänge nicht wichtig
Testausgabe

```
(0) Bester Knoten: [2,3,-,1,4,5,7,8,6] Kosten: (0,0)
(1) Bester Knoten: [2,3,5,1,4,-,7,8,6] Kosten: (0,8)
(2) Bester Knoten: [2,3,5,1,4,6,7,8,-] Kosten: (0,6)
(3) Bester Knoten: [2,-,3,1,4,5,7,8,6] Kosten: (0,8)
(4) Bester Knoten: [2,3,5,1,4,6,7,-,8] Kosten: (0,8)
(5) Bester Knoten: [2,4,3,1,-,5,7,8,6] Kosten: (0,8)
(6) Bester Knoten: [2,4,3,1,5,-,7,8,6] Kosten: (0,6)
(7) Bester Knoten: [2,4,3,1,5,6,7,8,-] Kosten: (0,4)
(8) Bester Knoten: [2,4,3,1,5,6,7,-,8] Kosten: (0,6)
(9) Bester Knoten: [-,2,3,1,4,5,7,8,6] Kosten: (0,8)
(10)Bester Knoten: [1,2,3,-,4,5,7,8,6] Kosten: (0,6)
(11)Bester Knoten: [1,2,3,4,-,5,7,8,6] Kosten: (0,4)
(12)Bester Knoten: [1,2,3,4,5,-,7,8,6] Kosten: (0,2)
(13)Bester Knoten: [1,2,3,4,5,6,7,8,-] Kosten: (0,0)
```

Lösung

```
    [2,3,-,1,4,5,7,8,6]
--> [2,-,3,1,4,5,7,8,6] --> [-,2,3,1,4,5,7,8,6]
--> [1,2,3,-,4,5,7,8,6] --> [1,2,3,4,-,5,7,8,6]
--> [1,2,3,4,5,-,7,8,6] --> [1,2,3,4,5,6,7,8,-]

Loesung in 13 Schritten gefunden
noch offene Knoten:      13
Laenge des Loesungswegs:  6
```

Testläufe
Ausgangsposition [-,8,7,6,5,4,3,2,1]
Heuristik: Anzahl richtiger Steine

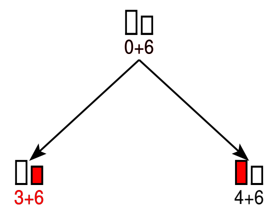
```
(0) Bester Knoten: [-,8,7,6,5,4,3,2,1] Kosten: (0,0)
(1) Bester Knoten: [6,8,7,-,5,4,3,2,1] Kosten: (0,8)
(2) Bester Knoten: [8,-,7,6,5,4,3,2,1] Kosten: (0,8)
    ...
    richtungsloses Suchen mit Kosten 8 und 7
    dann sehr zielgerichtet zur Loesung
    ...
(41)Bester Knoten: [2,3,6,1,5,-,4,7,8] Kosten: (0,8)
(42)Bester Knoten: [2,3,-,1,5,6,4,7,8] Kosten: (0,7)
(43)Bester Knoten: [2,-,3,1,5,6,4,7,8] Kosten: (0,6)
(44)Bester Knoten: [-,2,3,1,5,6,4,7,8] Kosten: (0,5)
(45)Bester Knoten: [1,2,3,-,5,6,4,7,8] Kosten: (0,4)
(46)Bester Knoten: [1,2,3,4,5,6,-,7,8] Kosten: (0,3)
(47)Bester Knoten: [1,2,3,4,5,6,7,-,8] Kosten: (0,2)
(48)Bester Knoten: [1,2,3,4,5,6,7,8,-] Kosten: (0,0)
```

Lösung

```
     [-,8,7,6,5,4,3,2,1]
 --> [6,8,7,-,5,4,3,2,1] --> ...
      ...
 --> [1,2,3,4,5,6,-,7,8] --> [1,2,3,4,5,6,7,-,8]
 --> [1,2,3,4,5,6,7,8,-]

Loesung in 48 Schritten gefunden
noch offene Knoten:      26
Laenge des Loesungswegs: 28
```

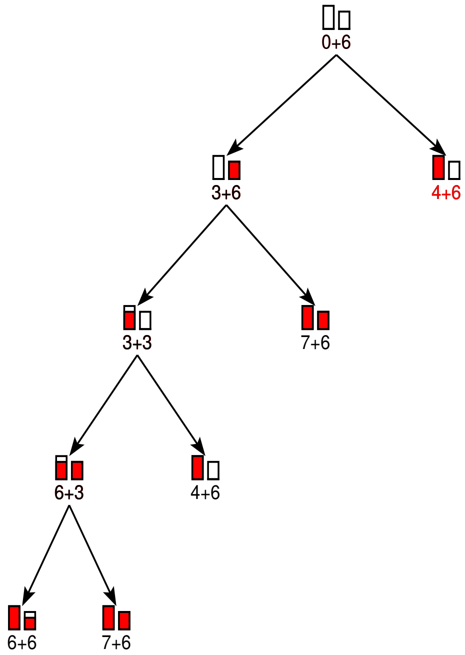
alle anderen Strategien führen zu keiner Lösung (max 200 Schritte) oder Suchbaum zu breit (heap overflow)

A*: Wasserkrugproblem



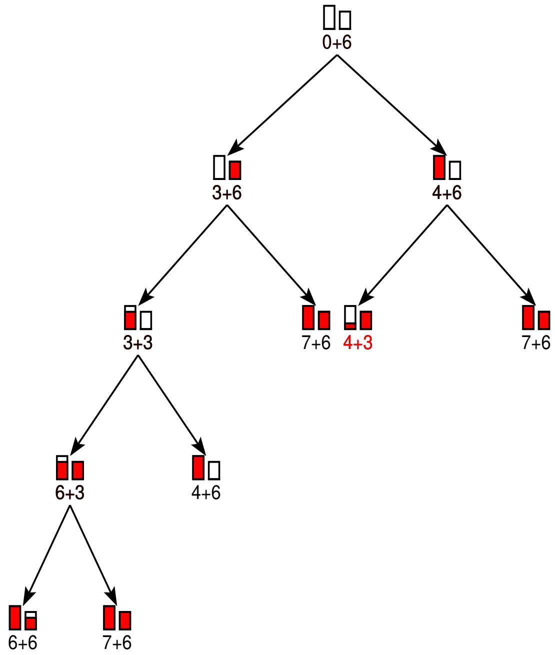
Heuristik: Wasserkosten + Nähe zum Ziel, 1. Knoten

A*: Wasserkrugproblem



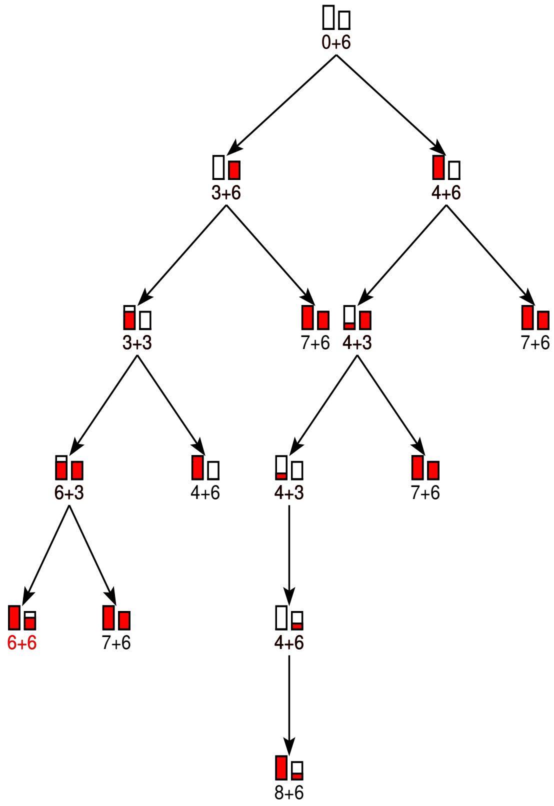
Heuristik: Wasserkosten + Nähe zum Ziel, 4. Knoten

A*: Wasserkrugproblem



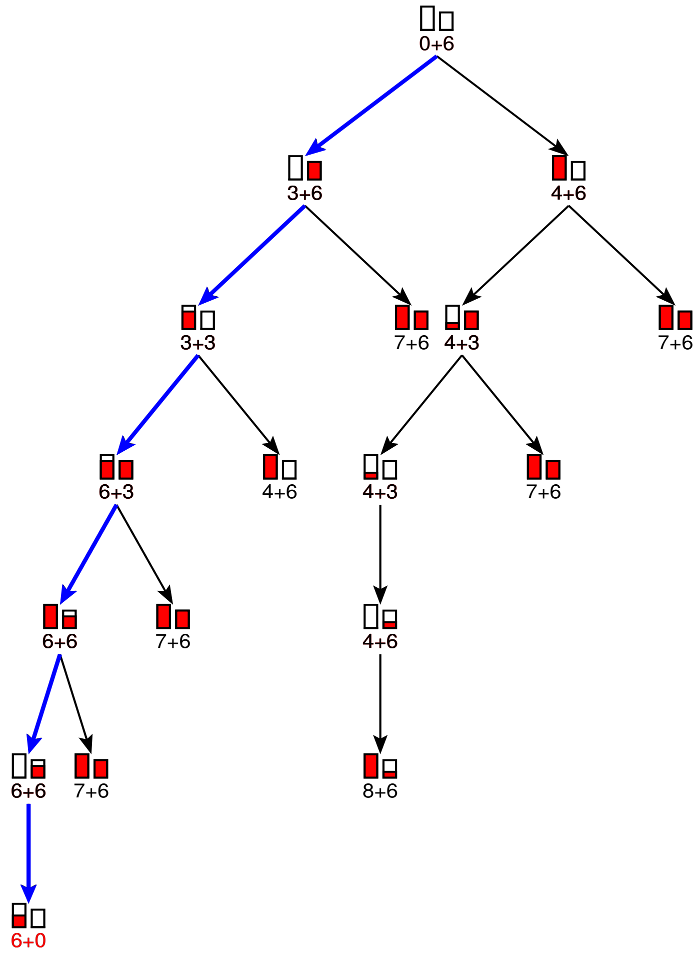
Heuristik: Wasserkosten + Nähe zum Ziel, 5. Knoten

A*: Wasserkrugproblem



Heuristik: Wasserkosten + Nähe zum Ziel, 8. Knoten

A*: Wasserkrugproblem



Heuristik: Wasserkosten + Nähe zum Ziel, 10. Knoten

ANALYSE : A*-ALGORITHMUS

$\hookrightarrow$	versuche, im Mittel schnell zu sein
$\hookrightarrow$	versuche, eine ziemlich gute Lösung zu finden
Qualität	hängt ab von der Schätzfunktion h'
h' **gut**	$\Rightarrow$ A* effizient
h' **überschätzt nicht**	$\Rightarrow$ optimale Lösung
im Mittel	Abschätzung über $h' - h$ $\hookrightarrow$ genaue Kosten
Problem	mathem. Analyse schwierig $\hookrightarrow$ experimentelle Analyse

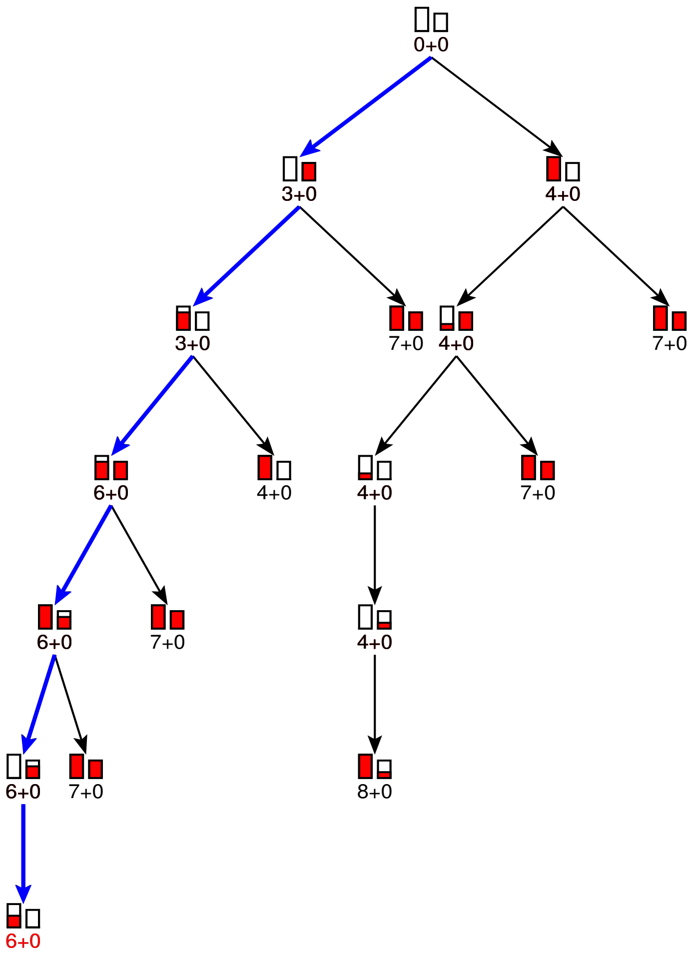
A*–ALGORITHMUS : EIGENSCHAFTEN

Bewertungsfunktion $f' = (h', g)$

Einfluß von g**:**

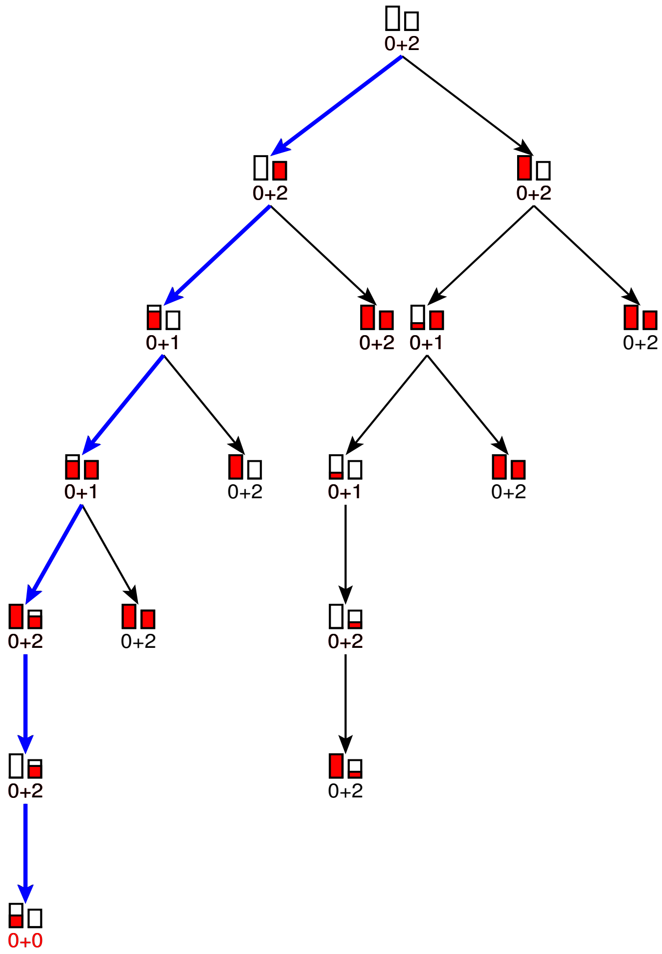
$g > 0$	Pfadkosten werden berücksichtigt $\hookrightarrow$ Pfadsuche
$g = 0$	nur die Lösung interessiert
$g < 0$	lange Pfade und Schleifen gut !(?) $\hookrightarrow$ nicht terminieren
g **konstant** > 0	kürzester Weg als Lösung
$g\neg$ **konstant**	Gewichtung der Schritte
Einfluß von h'	Schätzwert für den noch zurückzulegenden Wert bis zum Ziel
h' **perfekt**	sofortige Konvergenz
$h' = 0$	Suche wird von g gesteuert $\hookrightarrow$ Breitensuche *branch u. bound* $\hookrightarrow$ Bestensuche
h' **überschätzt** h **nicht**	optimaler Pfad zum Ziel

A*: Wasserkrugproblem



Suchbaum mit Heuristik: Wasserkosten minimieren
gewichtete Breitensuche

A*: Wasserkrugproblem



Suchbaum mit Heuristik: Nähe zum Ziel
Bestensuche

4.7 Analyse der Suchalgorithmen

<u>Bewertung von Zuständen: heuristische Funktion</u>

allgemeine Heuristiken Weglänge im Graphen, Prioritäten in Regeln, ...

spezielle Heuristiken problemspezifische Informationsausnutzung

Schach „vernünftige Züge“

Implementierung

$\hookrightarrow$ Spezialfälle in den Regeln

$\hookrightarrow$ heuristische Funktion, Bewertung der Zustände

Beispiele

Schach Figurenvorteil

8-puzzle # richtige Plättchen

travelling salesman bisher zurückgelegte Strecke

Wasserkrug Abstand zur Lösung

$\hookrightarrow$ viele Varianten und Beispiele

Frage Wie gut sind die Suchverfahren ?

erschöpfende Suche Baum der Tiefe D und Verzweigungsfaktor $F > 1$

F^D Knoten in Schicht D

$\hookrightarrow F\frac{F^D-1}{F-1} \approx F^D$ Knoten in Baum

$\hookrightarrow$ exponentielles Wachstum
$\hookrightarrow$ praktisch nicht anwendbar

Verbesserungen Klassen

$\hookrightarrow$ allgemeine Verfahren, die eine genausogute Lösung finden wie die *erschöpfende Suche* aber schneller sind

$\hookrightarrow$ Verfahren, die im *schlechtesten Fall* genauso viel Zeit wie die *erschöpfende Suche* brauchen, mit hoher Wahrscheinlichkeit aber schneller sind

$\hookrightarrow$ Verfahren, die suboptimale Lösungen in beschränkter Zeit finden.

(Lösungsqualität?)

Anpassung an ein Produktionensystem

Problem	Wie findet man die Menge der anwendbaren Regeln aus der Gesamtmenge der Regeln?
lineare Suche	bei großen Regelmengen hoffnungslos ineffizient $O(n)$
Sortierung auf der Regelmenge	binäre Suche $O(\log n)$
Hashfunktion auf der Regelmenge	Indizierung fast $O(1)$
	bei allgemeinen Regeln (mit Variablen)
	$\hookrightarrow$ Überprüfen der Anwendbarkeit schwierig
	$\hookrightarrow$ „matchen“, Unifikation
Regeln mit Variablen	Beispiel: Prolog
$\hookrightarrow$	extensive Suche, ob eine Regel anwendbar ist
	$\hookrightarrow$ welche Regel zuerst anwenden?
	$\hookrightarrow$ Effizienz

Beispiel: Stammbaum

die Fakten

```
sohn(anna,jochen).
sohn(jochen,karl).
sohn(karl,christian).
sohn(karl,thomas).
tochter(jochen,susi).
tochter(susi,julia).
```

die regeln

```
enkel(X,Z) :- sohn(X,Y),sohn(Y,Z).
enkel(X,Z) :- tochter(X,Y),sohn(Y,Z).

enkelin(X,Z) :- sohn(X,Y),tochter(Y,Z).
enkelin(X,Z) :- tochter(X,Y),tochter(Y,Z).
```

Ziel: Wer ist Jochens Enkel?

?– *enkel(jochen, X).*

$\hookrightarrow$ *Reihenfolge der Belegung*
$\hookrightarrow$ *Anzahl anwendbarer Regeln*
$\hookrightarrow$ *Art der Belegung bei der Anwendung einer Regel*

4.8 Problemreduzierung

UND-ODER-Graph (-Baum)

Idee Problem (Operation) zerlegen in Teilprobleme

$\hookrightarrow$ nicht nur ein Nachfolger pro Zustandsübergang, sondern eine Menge von Nachfolgern (Teilziele), die alle gelöst werden müssen

$\uparrow$ Probleme werden kleiner

$\downarrow$ nicht ein Nachfolger, sondern eine Menge von Nachfolgern muß bei jedem Schritt verarbeitet werden

Ziel ähnlicher Algorithmus wie A^*

A^* nicht dirket übertragbar

4.9 Beschränkungserfüllung

Problemstellung ausgehend von einem allgemeinen Zustand werden schrittweise Beschränkungen ausgewertet und der Zustand konkretisiert

Beispiel Buchstabenrätsel

Wahrnehmung :
Erkennen von Linienzügen als geometrische Objekte

Sichtweisen

.1 Aufteilung des Problemraums
$\hookrightarrow$ Raum für Beschränkungsliste
$\hookrightarrow$ Raum für Suche

Kombination allgemeine Suchverfahren (A^*)
$\Updownarrow$
Suchverfahren in den Beschränkungen
(nicht alle Nachfolger generieren)

.2 Bergsteigen

Zustandsraum	$\longrightarrow$	alle möglichen Lösungen
Klettern	$\longrightarrow$	Lösungsmenge verkleinern
Ziel	$\longrightarrow$	Lösungsmenge
	$\longrightarrow$	1 Zielmenge $\neq \{\,\}$

Beschränkungserfüllung

Algorithmus

.1 Wähle nicht entwickelten Knoten des Suchgraphen

.2 Wende Einschränkungsregeln an, um alle möglichen Einschränkungen zu erzeugen

.3 Widerspruch in Einschränkungen
$\hookrightarrow$ Sackgasse

.4 Einschränkungen $\Longrightarrow$ Lösung
$\hookrightarrow$ Erfolg

.5 kein Widerspruch, keine Lösung
$\hookrightarrow$ neue Teillösung
$\hookrightarrow$ **(1)**

ungenau
- Problemraum
- Knotenauswahl
- Darstellung der Einschränkungen

Buchstabenrätsel

Beobachtungen

Verzweigungsfaktor Effizienz
erst geringe Verzweigung,
dann breite Verzweigung

Suche hier Tiefensuche
$\hookrightarrow$ Effizienz

Vorgehen

.1 Ausnutzung gleicher Ziffern

.2 Ausnutzung ungleicher Ziffern (vorne)

.3 $>$–Beziehungen, Ungleichungen

.4 Symbolmanipulation, Arithmetik

.5 ausprobieren

Expertenwissen über Addition (Übertrag,. . .)
und Multiplikation (gerade, ungerade,. . .)

4.10 Genetische Algorithmen

Optimierungsstrategie übertragen aus biologischer Evolution

Phänotyp	reale Ausprägung eines Individuums
Geontyp	Zeichenkette *(Cromosom)* als Bauplan
Entwicklung	durch
Kreuzung	neue Individuen aus Kombination existierender Individuen erzeugen
Auslese	*survival of the fittest*
Mutation	spontane zufällige genetische Veränderung

<u>OPTIMIERUNGSPROBLEM</u>

Suchraum Problemraum
aller *realen* Ausprägungen
aller Phänotypen P

wird abgebildet in

Suchraum für Cromosomen
für Zeichenketten fester Länge über einem Alphabet
für Genotypen G

Alphabet

- $\{0,1\}$ Bitstring
- $\{A,T,G,L\}$ in Natur 4 Zeichen
- allgemein problemspezifisch wählbar

Bewertungs-funktion für die Individuen

die zu maximiernde Funktion h
ordnet jedem Individuum eine *Fitness* (Stärke, Güte) zu.

$\hookrightarrow$ $h : P \longrightarrow \mathsf{R}$

SUCHPROZESS

Suche durch Simulation der Evolution
in einer Bevölkerung (Menge von Individuen)
wird über Generationen gekreuzt und selektiert

Start eine zufällig gewählte Menge von Individuen
eine Population
diese werden mit der Funktion h bewertet

Prozess

.1 kreuzen

↪ neue Individuen

.2 bewerten der neuen Individuen

.3 selektieren

↪ gute überleben
↪ schlechte sterben

KREUZEN

kreuzen züchten

.1 wähle 2 Individuen (Cromosomen) aus der Population

.2 wähle eine Position in den Cromosomen

.3 breche die Cromosomen an dieser Stelle auf und füge die vertauschten Teile zu zwei neuen Cromosomen zusammen

.4 füge die 2 neu entstandenen Cromosomen zur Population dazu

erzeugen neuer Individuen aus vorhandenen Individuen

Hoffnung durch kreuzen 2-er guter Individuen entsteht ein sehr gutes Individuum

- Wie werden die Cromosomen ausgewählt?
- Wie wird die Position ausgewählt?
- Entsprechen die neuen Cromosomen überhaupt einem Zustand im Problemraum?

Auswahstrategien: kreuzen

Auswahlstrategien

.1 die (beiden) besten werden gekreuzt

lokale Maxima

↪ *Inzucht*

.2 Roulette: die Auswahlwahrscheinlichkeit ist proportional zur Güte eines Individuums

↪ alle Individuen werden berücksichtigt

↪ *breite* Suche

wenn Anfangspopulation ungünstig gewählt, immer noch Gefahr lokaler Maxima

↪ Mutation

Kreuzungsstrategien

.1 eine Position zufällig wählen

↪ wirksam, wenn *gute* Teilstücke zusammenhängen

.2 2 Positionen zufällig wählen

↪ gute Teilstücke brauchen nicht zusammenzuhängen

.3 zufällige Auswahl der einzelnen Gene

Problem neue Kombination hat in der Realität keine Entsprechung

↪ *problemspezifisches* Kreuzen
eigene Kreuzungsoperatoren

↪ nachträgliche Korrektur

zufällige Suche

<u>Selektionsstrategien</u>

.1	die schlechtesten Individuen sterben
	Vielfalt wird zu schnell verringert Suche beschränkt sich auf einen Teilraum lokale Maxima
.2	Roulette: Wahrscheinlichkeit des Überlebens proportional zur Güte
↪	Vielfalt bleibt besser erhalten
	auch die momentan beste Lösung kann gelöscht werden
↪	Sonderbehandlung der besten Lösung
	Selektion verringert die Vielfalt

<u>Mutation</u>

Problem	Vielfalt zu gering
↪	lokale Maxima
Sprünge	im Suchprozess erforderlich
Mutation	
.1	ein Cromosom zufällig auswählen
.2	ein Gen zufällig auswählen
.3	den Wert (Allel) zufällig ändern
↪	Vielfalt in der Population wird durch Mutation erhöht
↪	*breitere* Suche

<u>Parametrisierung des Suchprozesses</u>

 # Individuen in der Population

zu groß ⇒ ineffizient, zu breit
zu klein ⇒ Vielfalt zu gering

 # Kreuzungen pro Generation

zu klein ⇒ zu wenig Fortschritt
zu groß ⇒ nicht zielgerichtet genug

Mutationswahrscheinlichkeit

zu groß ⇒ zufällige Suche
zu klein ⇒ Vielfalt zu gering

↪ experimentieren

↪ Problemspezifische Kreuzungs– und Mutationsoperatoren

- ↪ mehr problemspezifisches Wissen im Suchprozess

↪ stabiles Suchverfahren

↪ breit gefächertes Einsatzgebiet

↪ anwendbar auch bei unstrukturierten und komplexen Suchräumen

4.11 Zusammenfassung Problemlösungsverfahren

- Vorwärts-, Rückwärtssuche — *8-puzzle, Integral*
- Problembäume, -Graphen — *Wasserkrug*
- Anpassung der Regelmengen — *Wasserkrug, Stammbaum*
- Heuristische Suche — *trav. salesman, tic-tac-toe, Schach*
- Schwache Verfahren
  - ↪ generiere u. teste Tiefensuche
  - ↪ Bergsteigen
  - ↪ Breitensuche — *"branch and bound"*
  - ↪ Bestensuche — *ODER-Graphen, A^**
  - ↪ Problemreduzierung — *UND-ODER-Graphen AO^**
  - ↪ Beschränkungserfüllung constraint satisfaction — *Zahlenpuzzle*
  - ↪ Genetische Algorithmen — *parallele Suche survival of the fittest*

KI-Probleme lösen

Vorgehen bei der Lösung eines KI-Problems

Problem-Definition genau
↪ Problemraum, Anfangs-, End-Zustand

Problemanlyse (7 Punkte) Kapitel 2

- ↪ Zerlegbarkeit
- ↪ Wiederherstellbarkeit
- ↪ Vorhersehbarkeit
- ↪ gute Lösung
- ↪ konsistentes Wissen
- ↪ Wissensmenge
- ↪ Dialog

Problem-lösungs-verfahren Auswahl

- ↪ Wie wird der zu entwickelnde Knoten ausgewählt?
- ↪ Wie wir der anzuwendende Operator ausgewählt?
- ↪ optimale Lösung gefordert?
- ↪ mehrfache Betrachtung eines Knotens?
- ↪ Speicherung der Zustandsbeschreibung?
- ↪ spezielle Suchpfade möglich?

5 Spiele

Warum ? Fazination, eine Maschine spielen zu lassen

praktisch gut strukturierte Probleme
Erfolg leicht zu messen

↪ keine große Wissensmenge (naiv)
direkte Suche

Schach

Verzweigungsfaktor	$\approx$	35
Weglänge	$\approx$	50
Positionen	$\approx$	35^{100} zu testen

↪ Erzeuge und Prüfe

$\left.\begin{array}{l}\text{geschicktes Erzeugen}\\ \text{geschicktes Prüfen}\end{array}\right\}$ durch Wissens–ausnutzung

↪ akzeptable Lösung in akzeptabler Zeit

heuristische Bewertungsfunktion

↪ Spielen fällt in die im vorigen Kapitel beschriebene Klasse von Problemen

Anwendung von Problemlösungsverfahren ?!

↪ A*-Algorithmus

5.1 MINIMAX - Suchverfahren

Einpersonenspiele
Puzzle ↪ A*-Algorithmus

Zweipersonen–spiele Wechsel : gut $\longleftrightarrow$ schlecht

↪ A* nicht geeignet

↪ immer wechselnde Annahmen in den einzelnen Ebenen

Verfahren MINIMAX

↪ Tiefensuchverfahren mit beschränkter Tiefe

Bewertungsfunktion

$f : Pfad \longrightarrow -max \,..\, +max$

$+max$ Spieler gewinnt mit Sicherheit

$-max$ Gegner gewinnt mit Sicherheit

MINIMAX - Algorithmus

Funktionen

$gen(pos)$ generiert die Menge der Züge, die von Pos aus gemacht werden können

$statisch(pos, tiefe)$ Auswertungsfunktion für die Güte von

pos	hohe Werte	$\rightarrow$	gut
	kleine Werte	$\rightarrow$	schlecht
$tiefe$	ungerade	$\rightarrow$	maximieren
	gerade	$\rightarrow$	minimieren

- Wann wird die Rekursion angehalten?
- Wann wird die Bewertung aufgerufen?

berücksichtigen

$\hookrightarrow$ Hat eine Seite gewonnen?

$\hookrightarrow$ Wieviele Schichten?

$\hookrightarrow$ Wie gut ist der momentane Pfad?

$\hookrightarrow$ Wieviel Zeit bleibt noch?

$\hookrightarrow$ Wie stabil ist die Konfiguration?

$tief\_genug$ ist ein Prädikat, das die Rekursionstiefe steuert und die Punkte berücksichtigt

Resultat Paar : Bewertung eines Pfades
Pfad selber

MINIMAX - Algorithmus (cont.)

Bereiche

.0 $N = Knoten$

.1 $Pfad = N^*$

.2 $Wert = -max \, .. \, +max$

Hilfsfunktionen

.3 $tief\_genug$: $\mathsf{N}_0 \times Zustandsinfo \longrightarrow \mathsf{B}$

.4 $statisch$: $N \times \mathsf{N}_0 \longrightarrow Wert$

.5 $gen\_nachfolger : N \longrightarrow N$ -set

Hauptfunktionen

.6 $minimax$: $N \times \mathsf{N}_0 \longrightarrow Pfad \times Wert$

.7 $maxi$: N -set $\times$ $Pfad \times Wert \times \mathsf{N}_0 \longrightarrow Pfad \times Wert$

$maxi$ sucht besten Nachfolger aus Nachfolgermenge

$minimax$ steuert Erzeugung und Abbruch
Resultat ist ein Paar aus Pfad und Bewertung

MINIMAX - ALGORITHMUS (CONT.)

$minimax : N \times \mathsf{N}_0 \longrightarrow Pfad \times Wert$
$minimax(n, t) \triangleq$
$\quad \mathsf{if}\ tief\_genug(t)$
$\quad \mathsf{then}\ ([\,], statisch(n, t))$
$\quad \mathsf{else\ let}\ nfs = gen\_nachfolger(n)\ \mathsf{in}$
$\qquad\quad \mathsf{if}\ nfs = \{\,\}$
$\qquad\quad \mathsf{then}\ ([\,], statisch(n, t))$
$\qquad\quad \mathsf{else}\ maxi(nfs, [\,], -max, t)$

$maxi : N\text{-}\mathsf{set} \times Pfad \times Wert \times \mathsf{N}_0 \longrightarrow Pfad \times Wert$
$maxi(nfs, pf, w, t) \triangleq$
$\quad \mathsf{if}\ nfs = \{\,\}$
$\quad \mathsf{then}\ (pf, w)$
$\quad \mathsf{else\ let}\ nf \in nfs\ \mathsf{in}$
$\qquad \mathsf{let}\ (pf_1, w_1) = minimax(nf, t+1)\ \mathsf{in}$
$\qquad \mathsf{let}\ nw_1 = -w_1\ \mathsf{in}$
$\qquad \mathsf{let}\ (pf_2, w_2) = \mathsf{if}\ nw_1 > w$
$\qquad\qquad\quad \mathsf{then}\ ([nf] \frown pf_1, nw)$
$\qquad\qquad\quad \mathsf{else}\ (pf, w)\ \mathsf{in}$
$\qquad maxi(nfs - \{nt\}, pf_2, w_2, t)$

5.2 Alpha–Beta–Beschneidung

Minimax	Tiefensuchverfahren
$\hookrightarrow$	Erschöpfende Suche bis zur Tiefe N ineffizient
Ziel	Effizienzsteigerung durch frühzeitigen Abbruch schlechter Pfade
wie	*nächster Nachbar*, Verzweigungs- und Markierungsstrategie anwenden mit 2 Grenzen arbeiten
$\hookrightarrow$	α–β–Beschneidung
α–**Knoten**	*maximierende* Knoten
β–**Knoten**	*minimierende* Knoten

α–β–Beschneidung : Algorithmus

$minimax\_\alpha\_\beta$:

$$N \times \mathsf{N}_0 \times Wert \times Wert \longrightarrow Pfad \times Wert$$

$$minimax\_\alpha\_\beta(n, t, \overbrace{nu, we}^{\substack{\alpha-\beta \\ Grenzen}}) \triangleq$$

if $tief\_genug(t)$

then $([\,], statisch(n, t))$

else

let $nfs = gen\_nachfolger(n)$ in

if $nfs = \{\,\}$

then $([\,], statisch(n, t))$

else $maxi\_\alpha\_\beta(nfs, [\,], nu, we, t)$

α–β–Beschneidung : Algorithmus

$maxi\_\alpha\_\beta$:

$$N\text{ -set} \times Pfad \times Wert \times Wert \times \mathsf{N}_0 \longrightarrow Pfad \times Wert$$

$maxi\_\alpha\_\beta(nfs, pf, nu, we, t) \triangleq$

if $nfs = \{\,\}$

then (pf, we)

else

let $nf \in nfs$ in

let $(pf_1, w_1) = minimax\_\alpha\_\beta(nf, \overbrace{-we, -nu}^{vertauschen}, t + 1)$ in

let $nw = -w_1$ in

let $(pf_2, w_2) =$ if $nw > we$

then $([nf] \frown pf_1, nw)$

else (pf, we) in

$$\underbrace{\overbrace{\text{if } we >= w_2}\ \text{then } (pf_2, w_2)\ \text{else } maxi\_\alpha\_\beta(nfs - \{nf\}, pf_2, nu, w_2, t)}_{Beschneidung}$$

<u>ANALYSE MINIMAX-VERFAHREN</u>

tief_genug	Rekursion erst abbrechen, wenn Ruhe eingekehrt ist; keine großen Schwankungen
Sekundärsuche	Den ausgewählten Zug noch ein paar Züge weiterrechnen (Fallgruben)
generiere_ nachfolger	Problemspezifisches Wissen ausnutzen Auswahl einschränken (Eröffnungsbibliothek)
Horizonteffekt	unausweichlich schlechte Pfade können so verlängert werden, daß der „Absturz“ hinter dem Horizont verschwindet
nicht risikofreudig	Annahme : Gegner macht immer schlechtesten Zug ↪ Defensivverfahren schlecht bei Verlustsituationen

5.3 Zusammenfassung : Spiele

Erweiterung	Suchverfahren ↪ 2–Personen–Spiele
MINIMAX –	Verfahren (Tiefensuche)
α–β– **Beschneidung**	Effizienzsteigerung gegenüber MINIMAX
Variationen	Verbesserung der Lösungsqualität

6 Wissensdarstellung mit Prädikatenlogik

bisher allgemeine Verfahren zur Bearbeitung von Wissen mit Hilfe von Suchalgorithmen

keine Berücksichtigung wie das Wissen dargestellt ist

Ziel leistungsfähige Folgerungsmechanismen durch geschickte Wissensdarstellung

Ebenen der Wissensdarstellung

.1 Fakten in der Realität

.2 interne Darstellung der Fakten in einer Maschine

.3 externe Darstellung der Fakten zur Kommunikation mit dem Anwender

Transformationen
.1 $\longrightarrow$ **.2** formalisieren
.2 $\longrightarrow$ **.1** interpretieren
.2 $\longrightarrow$ **.2** schließen
.2 $\longrightarrow$ **.3** erklären
.3 $\longrightarrow$ **.2** natürliche Sprache erkennen

6.1 Aussagenlogik

Aussagen werden in *2* Klassen eingeteilt
wahre Aussagen
falsche Aussagen

Aussageform Aussage bei der einzelne Fakten durch Variable ersetzt werden

äquivalente Aussageformen Aussageformen, denen immer der gleiche Wahrheitswert zugewiesen wird

logische Verknüpfungen von Aussageformen durch Operatoren
$\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \ldots$

1–stellig : 4 Operatoren
2–stellig : 16 Operatoren

Sätze der Aussagenlogik allgemeingültige Aussageformen
(Aussageformen die immer wahr sind)

Sätze der Aussagenlogik

(Aussagen, die bei allen Belegungen wahr sind)

.1	$A \lor \neg A$	Satz vom ausgeschlossenen Dritten
.2	$\neg(A \land \neg A)$	Satz vom Widerspruch
.3	$\neg(\neg A) \Leftrightarrow A$	doppelte Verneinung
.4	$\neg(A \land B) \Leftrightarrow \neg A \lor \neg B$	} de Morgan
.5	$\neg(A \lor B) \Leftrightarrow \neg A \land \neg B$	
.6	$A \Rightarrow B \Leftrightarrow \neg B \Rightarrow \neg A$	Kontraposition
.7	$((A \Rightarrow B) \land A) \Rightarrow B$	modus ponens ABTRENNUNGSREGEL
.8	$((A \Rightarrow B) \land \neg B) \Rightarrow \neg A$	modus tollens
.9	$(A \Rightarrow B) \land (B \Rightarrow C) \Rightarrow (A \Rightarrow C)$	modus barbara KETTENSCHLUSS
.10	...	alle Axiome u.a.
.11	$A \land (B \lor C) \Leftrightarrow (A \land B) \lor (A \land C)$	} Distributivgesetze
.12	$A \lor (B \land C) \Leftrightarrow (A \lor B) \land (A \lor C)$	

Umsetzung : Umgangssprache $\longleftrightarrow$ Formelsprache

Umgangssprache	*Formelsprache*
Es ist *regnerisch*	*regnerisch*
Es ist *sonnig*	*sonnig*
Es ist *windig*	*windig*
Wenn es *regnerisch* ist, ist es nicht *sonnig*	*regnerisch* $\Rightarrow$ $\neg$*sonnig*
Sokrates ist ein *Mensch*	*Mensch–Sokrates*
Plato ist ein *Mensch*	*Mensch–Plato*

IN AUSSAGENLOGIK NUR BOOLESCHE VARIABLEN
$\hookrightarrow$ Aufzählen aller Menschen notwendig

Wie Umsetzung „Alle *Menschen* sind *sterblich*“ ?

$\hookrightarrow$ Quantifizierung

$\hookrightarrow$ Prädikatenlogik

- $\hookrightarrow$ Variablen aus verschiedenen Bereichen
- $\hookrightarrow$ Prädikate (Relationen) über diese Variablen

Beispiel

$mensch(sokrates)$
$mensch(plato)$
$\forall\, x \bullet mensch(x) \Rightarrow sterblich(x)$

6.2 Prädikatenlogik

Subjekte Individuenmenge

Prädikate Namen von Relationen über der Individuenmenge

Quantoren „für alle" $\forall$
„es gibt" $\exists$

Beispiel $\forall x \bullet x > y \Rightarrow x + 1 > y$
$\exists x \bullet prim(x) \wedge 5 < x < y$

$\hookrightarrow$ hier wird eine universelle Individuenmenge angenommen, die nicht explizit notiert wird

$\hookrightarrow$ $\forall x \in M \bullet P(x) \Leftrightarrow \forall x \bullet x \in M \Rightarrow P(x)$

gebundene Variablen dürfen umbenannt werden;
x an Quantoren gebunden

freie Variable *ungebundene* Variable
y frei

Prädikatenkalkül 1.Stufe Quantoren nur über Individuenmenge, nicht über Prädikate!

vollständig alle Sätze der Prädikatenlogik sind ableitbar (Gödel)

unentscheidbar für einen gegebenen Ausdruck ist im Allgemeinen nicht entscheidbar, ob er ein Satz des Prädikatenkalküls ist (Church)

Sätze der Prädikatenlogik

.1 $\forall x \bullet P(x) \Rightarrow P(a)$ Spezialisierung

.2 $P(a) \Rightarrow \exists x \bullet P(x)$ Generalisierung

.3 $\neg(\forall x \bullet P(x)) \Leftrightarrow \exists x \bullet \neg P(x)$

$\neg(\exists x \bullet P(x)) \Leftrightarrow \forall x \bullet \neg P(x)$ } de Morgan

.4 $\forall x, y \bullet P(x, y) \Leftrightarrow \forall y, x \bullet P(x, y)$

$\exists x, y \bullet P(x, y) \Leftrightarrow \exists y, x \bullet P(x, y)$ } Vertauschen

.5 $\exists x \bullet \forall y \bullet P(x, y) \Rightarrow \forall y \bullet \exists x \bullet P(x, y)$

TRANSFORMATION : UMGANGSSPRACHE ⇒ PRÄDIKATENKALKÜL
BEISPIEL

1. Marcus war ein Mann
2. Marcus war ein Pompejer
3. Alle Pompejer waren Römer
4. Caesar war ein Herrscher
5. Alle Römer waren entweder Caesar ergeben oder haßten ihn
6. Jeder ist irgend jemanden ergeben
7. Menschen versuchen nur die Herrscher zu ermorden, denen sie nicht ergeben sind
8. Marcus versuchte Caesar zu ermorden

ERWEITERUNG DURCH BERECHENBARE FUNKTIONEN

berechenbare Prädikate

$$\left.\begin{array}{l} größer(1,0) \\ größer(2,1) \\ größer(3,2)\ \ldots \end{array}\right\} größer(x,y)$$

berechenbare Funktionen

$$größer(\underbrace{2+3}_{+Funktion},1)$$

Funktionen
⇕
Prädikate

$$2 = f(x,y) \Leftrightarrow f'(x,y,2)$$

berechenbare Funktionen und Prädikate

↪ Reduktion der Regelmenge — GUT

↪ nur Werte als Argumente
keine Variable — SCHLECHT

↪ Richtung der Regelanwendung wichtig
VERLUST AN FLEXIBLILTÄT

Schliessen im Prädikatenkalkül mit Funktionen

1. $mensch(marcus)$
2. $pompejer(marcus)$
3. $geboren(marcus, 40)$
4. $\forall\, x \bullet mensch(x) \Rightarrow sterblich(x)$
5. $\forall\, x \bullet pompejer(x) \Rightarrow gestorben(x, 79)$
6. $ausgebrochen(Vulkan, 79)$
7. $\forall\, x, t_1, t_2 \bullet$
   $sterblich(x) \wedge geboren(x, t_1) \wedge größer(t_2 - t_1, 150) \Rightarrow tot(x, t_2)$
8. jetzt $= 1996$
9. $\forall\, x, t \bullet lebt(x, t) \Rightarrow \neg tot(x, t)$
10. $\forall\, x, t_1, t_2 \bullet gestorben(x, t_1) \wedge größer(t_2, t_1) \Rightarrow tot(x, t_2)$

Klauseln

bisher

viele Operatoren in den Formeln $\wedge, \vee, \Rightarrow, \Leftrightarrow$

tief geschachtelte Formeln $((.\wedge.)\vee.)\Rightarrow .)$

$\exists$ und $\forall$ überall in den Formeln

gut

Korrenspondenz : Kalkül $\leftrightarrow$ natürliche Sprache

schlecht

viele äquivalente Formeln
mehrdeutig

☞ Transformation in Klauseln

$\hookrightarrow$ Standartform

$\hookrightarrow$ bis auf Reihenfolge eindeutig

konjunktive Normalform

aufgebaut aus

1. elementare Prädikate
2. $\neg$ von 1.
3. $\vee$ von 1. und 2.
4. $\wedge$ von 3.
5. $\exists$ Quantoren eleminiert
6. $\forall$ Quantoren nur vor der gesamten Formel

TRANSFORMATION IN NORMALFORM

.1 $\Rightarrow$, $\Leftrightarrow$ eliminieren, ersetzen durch $\neg$, $\wedge$, $\vee$

$a \Rightarrow b \longrightarrow \neg a \vee b$

.2 $\neg$ nach innen ziehen

$\neg\neg a \longrightarrow a$
$\neg(a \vee b) \longrightarrow \neg a \wedge \neg b$
$\neg \forall x \bullet P(x) \longrightarrow \exists x \bullet \neg P(x)$

.3 umbenennen

$\forall x \bullet P(x) \vee \forall x \bullet Q(x) \longrightarrow \forall x \bullet P(x) \vee \forall y \bullet Q(y)$

.4 alle Quantoren vor die Formel

$\forall x \bullet P(x) \vee \forall y \bullet Q(y) \longrightarrow \forall x \bullet \forall y \bullet P(x) \vee Q(y)$

.5 Beseitigung von $\exists$ Quantoren durch Skolemfunktionen

$\exists y \bullet Kanzler(y) \longrightarrow Kanzler(S1)$
$\forall x \bullet \exists y \bullet Vater(y,x) \longrightarrow \forall x \bullet Vater(S2(x),x)$

$S1$ Skolemkonstante „erzeugende“
$S2$ Skolemfuntion Funktionen

.6 $\forall$ Quantoren löschen (redundant)

.7 in konjunkive Normalform transformieren

$a \vee (b \wedge c) \longrightarrow (a \vee b) \wedge (a \vee c)$

jede Konjunktion : KLAUSEL
alle KLAUSELN müssen wahr sein

6.3 Resolution

RESOLUTION (AUFLÖSUNG) IN DER AUSSAGENLOGIK

gegeben $\mathcal{F}$ die Menge der Axiome (Wissen)
S die zu beweisende Aussage

Algorithmus

.1 Wandle alle Aussagen in Klauselform um

.2 Negiere S und wandle $\neg S$ in Klauselform um nehme $\neg S$ zu **.1** dazu

.3 Wiederhole folgende Schritte, bis Wiederspruch erzeugt wird oder kein Fortschritt gemacht wird

.a Wähle zwei Klauseln *(Stammklauseln)* so, daß ein Literal L in der einen Klausel negiert, in der anderen nicht negiert auftritt

.I $L \vee B$ und **.II** $\neg L \vee C$

.b Löse **.I** und **.II** auf zur

Resolvente $B \vee C$

.c Ist $B \vee C$ leer (**.I** L, **.II** $\neg L$)

$\hookrightarrow$ Widerspruch

$\hookrightarrow$ sonst : Resolvente zu $\mathcal{F}$ und wiederhole **.a**

Auflösung im Prädikatenkalkül

Vereinigungsalgorithmus (Unifikation)

Problem Variablen in den Prädikaten

Variablen müssen konsistent ersetzt werden

Beispiel

.1	$p(X,Y)$	$p(a,Z)$	$X \mapsto a$ $Y = Z$
.2	$p(X,X)$	$p(a,Z)$	$X \mapsto a$ $X = Z$ $Z \mapsto a$
.3	$p(X,X)$	$p(a,b)$	$X \mapsto a$ $X \mapsto b$ $a \neq b$ $\longrightarrow$ *Sackgasse*

konsistente Variablenersetzung in der Resolvente

.3.b im vorigen Algorithmus erweitern

Auflösung : Beispiel Marcus

Klauseln	Axiome
.1	$mensch(marcus)$
.2	$pompejer(marcus)$
.3	$\neg pompejer(X_1) \vee römer(X_1)$
.4	$herrscher(caesar)$
.5	$\neg römer(X_2) \vee ergeben(X_2, caesar) \vee hassen(X_2, caesar)$
.6	$\neg mensch(X_3) \vee \neg herscher(Y_1) \vee \neg mordversuch(X_3, Y_1) \vee \neg ergeben(X_3, Y_1)$
.7	$mordversuch(marcus, caesar)$
zu beweisen	$hassen(marcus, caesar)$

Resolution (Auflösung): Eigenschaften

korrekt — es können nur wahre Aussagen abgeleitet werden

vollständig — wenn eine Menge von Klauseln *inkonsistent* ist, dann ist es möglich, alleine mit der Auflösung die *leere Klausel* (Widerspruch) abzuleiten

$\hookrightarrow$ $A_1, ..., A_n$ *konsistent :*

B folgt aus $A_1, ..., A_n$

$\Updownarrow$

$A_1, ..., A_n, \neg B$ *inkonsistent*

$\hookrightarrow$ Konsistenznachweis von $A_1, ..., A_n$

Schrittweise: $A_1, ..., A_{i-1}, \neg A_i$ auflösen

Problem — Entscheidung über die Reihenfolge der Auflösung ! *hartes Problem*

ein Ansatz — Einschränkung an die Form der Klauseln

Hornklauseln — nur ein Literal darf nichtnegiert in den Klauseln auftreten : *Prolog*

Klauseln $\Longleftrightarrow$ Prolog–Regeln

Prolog

A	**:-**	B **,** C **,** .... **.**	*Regel*
A	**:-**	**.**	*Fakt*
	:-	**.**	*leereRegel*

Notation : **;** $\leftrightarrow \vee$ **,** $\leftrightarrow \wedge$ **:-** $\leftrightarrow \Leftarrow$

Klausel

$A \vee B \vee \ldots$	$\vee$	$\neg K \vee \neg L \vee \ldots$
$(A \vee B \vee \ldots)$	$\vee$	$(\neg K \vee \neg L \vee \ldots)$
$(A \vee B \vee \ldots)$	$\vee$	$\neg(K \wedge L \wedge \ldots)$
$(A \vee B \vee \ldots)$	$\Leftarrow$	$(K \wedge L \wedge \ldots)$
A **;** B **;** ...	**:-**	K **,** L **,** .... **.**

Fakten

	negiert
A	$\neg A$
$A \vee$ false	false $\vee \neg A$
$A \vee \neg$true	false $\Leftarrow A$
$A \Leftarrow$ true	**:-** A **.**
A **:-** **.**	

leere Klausel **:-** **.**

Einschränkung **;** auf der linken Seite nicht zugelassen

Zusammenfassung

Wissensdarstellung mit Prädikatenlogik

☞ Aussagenlogik

☞ Prädikatenkalkül 1. Stufe

☞ Schließen im Prädikatenkalkül

☞ Klauseln

☞ Resolution (Auflösung)

☞ Unifikation

7 Wissensdarstellung mit anderen Logikarten

Prädikatenkalkül *einfach*
brauchbar für exaktes Wissen
entweder $\leftrightarrow$ oder
"closed world assumption"
unbekanntes wird als falsch interpretiert
Prinzip des ausgeschlossenen Dritten: $a \vee \neg a$

monoton Wenn eine Aussage P aus einer Menge von Axiomen A ableitbar ist, dann auch von jeder Menge B, die von A impliziert wird.

$A \vdash P$ und $A \Rightarrow B : B \vdash P$

$\uparrow$ Bei Hinzufügen einer Aussage sind keine Prüfungen auf Inkonsistenzen zu vorhandenem Wissen erforderlich

$A \Rightarrow B : A \vee B \Rightarrow P$

Problem nicht alle Information kann im Prädikatenkalkül dargestellt werden

unvollständiges & fehlerbehaftetes Wissen ist die Realität

- Erwartungswerte ändern sich
- Ausnahmen von der Regel werden bekannt
- Eingabedaten werden korrigiert
- Begründungen ändern sich
- Zeitliche Änderung der Eingabedaten

NICHT EXAKTES WISSEN: BEISPIELE

Bewertung Heute ist es *sehr* heiß

Wahrscheinlichkeit Menschen mit blonden Augen haben *oft* blaue Augen

Mangel Wenn es keinen Beweis für das Gegenteil gibt, *nimm an*, daß jeder Erwachsene lesen kann

Heuristik Es ist *besser*, wenn man mehr Figuren auf dem Schachbrett hat, als der Gegner

Glaube Ich *glaube*, daß der HSV nicht absteigt

Überzeugungssysteme Menge von Überzeugungen

$\hookrightarrow$ unvollständig

$\hookrightarrow$ inkonsistent

$\hookrightarrow$ veränderbar

wie im Krimi alle leugnen die Tat,

gesucht die wahrscheinlichste Schlußfolgerung für die Tat

Suche nach der kleinsten Menge von Regeln, die einen Widerspruch erzeugen

7.1 Nichtmonotone Logik: Überzeugungssysteme

Mangelbeweisführung Ist X unbekannt, dann folgere Y

- $\hookrightarrow$ Wenn X nicht bewiesen werden kann, dann folgere Y
- $\hookrightarrow$ Wenn X nicht in einer gewissen Zeit bewiesen werden kann, dann folgere Y

Umgangssprache Wenn es für X keine Ausnahme gibt, dann folgere Y

Beispiel A ist nicht alergisch gegen Milcheiweiß
$\Rightarrow$ Milch ist gesund für A

unvollständige Information $\hookrightarrow$ Mangelbeweisführung

Änderungen in der Welt $\hookrightarrow$ Änderungen in der Wissensmenge wärend des Beweisens

vollständige Lösung $\hookrightarrow$ vorübergehende Annahme von Teillösungen

---

<u>TRUTH MAINTENANCE SYSTEMS (WAHRHEITSERHALTUNGSSYSTEM)</u>

☞ nicht monotones Schlußfolgerungssystem

Aussagen & : Regeln Knoten

befinden sich an jedem Punkt der Beweisführung in einem von zwei Zuständen: **G**ültig oder **U**ngültig

Knoten $\Rightarrow$ Rechtfertigungen

Unterstützungsliste $UL([\textit{gültige Knoten.}], [\textit{ungültige Knoten}])$

bedingter Beweis $BB(\textit{konsequenz Kn.}, [\textit{gültige Hypothesen}]$ $[\textit{ungültige Hypothesen}])$

Prämissen Knoten ohne Rechtfertigungen

Annahmen Knoten mit nicht leerer UL für ungültige Knoten

UL **gültig** alle *gültigen Knoten* im Moment gültig
alle *ungültigen Knoten* im Moment ungültig

BB **gültig** *Konsequenz Knoten* gültig $\Leftrightarrow$

gültige Hypothesen im Moment gültig
ungültige Hypothesen im Moment ungültig

7.2 3–wertige Logik

2–wertig

true ($\triangleq 1$) false ($\triangleq 0$)

keine Möglichkeit, *unbekannt* darzustellen "etwas zwischen true und false"

Ziel

Erweiterung der 2–wertigen Logik um einen 3. Wert *U*nbekannt, so daß die 2–wertige Logik erhalten bleibt, wenn mit *U* gearbeitet wird

$(0 < U < 1)$

Wahrheitstafeln

$\wedge$:

$\wedge$	F	U	T
F	F	F	F
U	F	U	U
T	F	U	T

$a \wedge b \triangleq min(a, b)$

$\vee$:

$\vee$	F	U	T
F	F	U	T
U	U	U	T
T	T	T	T

$a \vee b \triangleq max(a, b)$

7.3 Statistik & Wahrscheinlichkeitsschlüsse

2–wertige Logik Aussage wahr oder falsch $\{0, 1\}$

3–wertige Logik Aussage wahr, falsch oder unbekannt $\{0, \frac{1}{2}, 1\}$

probabilistisches Schließen Aussage mit einer Wahrscheinlichkeit wahr

$$0 \leq P(A) \leq 1$$

Ziel Schlußfolgerung mit der höchsten Wahrscheinlichkeit finden

Basis Schätzungen, empirische Untersuchungen, Statistiken, Wahrscheinlichkeiten

Anwendungsbereich Diagnostik, Mustererkennung, Kartenspiele

Unsicherheitsquellen Evidenz (offensichtlichkeit) der Symptome

Bewertung: Zuordnung Symptom $\hookrightarrow$ Diagnose unzulängliche Kontrollstrategien

Diagnose-Algorithmus

1. starte mit *a–priori* Wahrscheinlichkeiten
2. modifiziere für jedes Symptom die Wahrscheinlichkeit aller Diagnosen (Symptom$rightarrow$Diagnose Wahrscheinlichkeit)
3. selektiere wahrscheinlichste Diagnose

Wahrscheinlichkeiten

Ereignisraum Ω

Ereignis $e \subseteq \Omega$

Elementarereignis e einelementig

zusammengesetzte Ereignisse $e_1 \cap e_2, e_1 \cup e_2, \Omega - e_1$

sicheres Ereignis Ω

unmögliche Ereignis $\{\,\}$

Wahrscheinlichkeits-

maß $P : \Omega\text{ -set} \longrightarrow [0\ ..\ 1]$

$$P(\Omega) = 1 \qquad P(\{\,\}) = 0$$

$$P(\bigcup_i A_i) = \sum_i P(A_i)$$

$$A_i \cap A_j = \{\,\} \text{ für } i \neq j$$

bedingte Wahrscheinlichkeit $P(A|B) = \dfrac{P(B \cap A)}{P(A)}$

unabhängige Ereignisse $P(A \cap B) = P(A) \cdot P(B)$

oder

$$P(A|B) = P(A)$$

Satz von Bayes

gegeben $P(D)$ W. der Diagnose D

$P(S|D)$ W. der Symptome S bei D

gesucht $P(D|S)$ W. der Diagnose D bei S

Satz $P(D|S) = P(D)\dfrac{P(S|D)}{P(S)}$

$\hookrightarrow$ zurückführen von $P(D|S)$ auf $P(S|D)$

$\hookrightarrow$ zurückschließen von der Wirkung auf die Ursache

Erweiterung auf viele Symptome

$\hookrightarrow$ jede Symptomkombination muß bewertet werden
$\hookrightarrow$ kombinatorische Explosion

$\hookrightarrow$ vereinfachtes Modell: Symptome sind unabhängig

(in Realität falsch)

$\hookrightarrow$ Satz von Bayes

$$P(D|\bigcap_i S_i) = P(D)\frac{\prod_i P(S_i|D)}{P(\bigcap_i S_i)}$$

PROBLEME MIT DEM SATZ VON BAYES

$\hookrightarrow$	Die Symptome dürfen nur von der Diagnose abhängen und müssen untereinander unabhängig sein
$\hookrightarrow$	Vollständigkeit der Diagnosemenge
$\hookrightarrow$	Wechselseitiger Ausschluß der Diagnosen
$\hookrightarrow$	fehlerfreie und vollständige Statistiken
$\hookrightarrow$	Konstanz der Wahrscheinlichkeiten

Verbesserungen

$\hookrightarrow$	Symptomkombinationen $A \cap B \cap C \longrightarrow D$
$\hookrightarrow$	Partitionierung der Diagnosemengen
$\hookrightarrow$	Wahrscheinlichkeitsintervalle
$\hookrightarrow$	positive und negative Evidenz einer Diagnose
$\hookrightarrow$	detailierteres Wissen (Ausnahmen, ...)

7.4 fuzzy–logic

verschwommene Logik	zur Modellierung unscharfer Aussagen
Beispiel	Heute ist es *warm* Heute ist es *kalt* Heute ist es *sehr kalt*
gewöhnlich	Mengendefinition durch ein Prädikat $M = \{x \vert P(x)\}$
Klassengrenzen	unscharf
Mitglieds-funktion	$m : M \longrightarrow [0\ ..\ 1]$ $m(x) = 1$ sicher $m \in M$ $m(x) = 0$ sicher $m \notin M$
$\hookrightarrow$	Mengenzugehörigkeit mit Unsicherheit belastet
$\hookrightarrow$	Logikregeln wie gewöhnlich
$\hookrightarrow$	zur Modellierung von: *groß, alt, warm, viele, die meisten*

7.5 Zusammenfassung: Wissensdarstellung mit anderen Logikarten

.1 nichtmonotones Schließen

↪ truth maintenance systems, Überzeugungssysteme

.2 3–wertige Logik

↪ *unbekannt* modellieren

.3 probabilistisches Schließen

↪ mit Satz von Bayes

↪ mit Dempster–Sheffer–Verfahen (Verbesserung)

.4 fuzzy logic

↪ Modellierung verbaler Quantifizierung

8 Wissensdarstellung mit Frames

frames Rahmen, M. Minsky 1975

Definition Ein *frame*–System besteht aus einer Kollektion von Objekten,
jedes Objekt besitzt eine Menge von *slots* (Attribute),
diese können beliebige Werte aufnehmen

$\hookrightarrow$

$$\begin{aligned} FrameSys &= Object \xrightarrow[\mathrm{m}]{} Frame \\ Frame &= Slot \xrightarrow[\mathrm{m}]{} Value \\ Value &= \ldots \end{aligned}$$

Value bleibt uninterpretiert,
kann einfache Werte, *Slot*s und *Object*s referenzieren

$\hookrightarrow$ *frames* $\Longleftrightarrow$ *records*

slot–Menge dynamisch veränderbar

2.Modell *frames* mit einfacher Vererbung

Idee Gemeinsamkeiten zusammenfassen
default–Werte nur einmal speichern
Sonderfälle getrennt behandeln

$\hookrightarrow$ Hierarchie aufbauen (Baum)

$\hookrightarrow$ Vererbung

Lösung ohne Datenstruktur zu verändern

ein fest vorgegebener *slot*: IS_A
eine feste Interpretation des IS_A–*slots*

Unterschied zu OO–Programmiersprachen:
keine Trennung von Typen und Instanzen

3.Modell *frames* mit mehrfacher Vererbung

Idee Objekte haben viele verschiedene Facetten, unterschiedliche Sichtweisen, aber gemeinsame Bestandteile und Eigenschaften

$\hookrightarrow$ Objekte aus einfacheren Objekten zusammensetzen

$\hookrightarrow$ mehrfache Vererbung
(anstatt Benutzung $\hookrightarrow$ OOP)

Lösung *IS_A–slot* eine Menge (Liste) von Basisobjekten zuordnen

Problematik: *slots* mit gleichem Namen und unterschiedlichen Werten in verschiedenen Basisklassen

Lösung

.1 Mehrdeutigkeiten $\hookrightarrow$ Fehlersituation

.2 Überschreiben in abgeleiteten Klassen

.3 Prioritäten: Reihenfolge der Vererbung festlegen

4.Modell Erweiterung um Dämonen

Idee Objekte sollen aktive Bestandteile des Systems werden,
intelligentes, selbständiges Agieren implementieren

Dämon Ein Prozess, der (versteckt) immer bei einem Zugriff auf einen *slot* ausgeführt wird

$\hookrightarrow$ vor/nach dem Lesen
vor/nach dem Schreiben
nach dem Erzeugen
vor dem Löschen

$\hookrightarrow$

$$\begin{aligned} Slot &= Value \times Demons \\ Demons &= Access \xrightarrow[\mathrm{m}]{} DemonProcess \\ DemonProcess &= FrameSys \longrightarrow FrameSys \\ Access &= B\_READ \mid A\_READ \mid \ldots \end{aligned}$$

$\hookrightarrow$ Dämonen: Funktionen in Daten