Torsten Schaub
Einführung in die künstliche Intelligenz

Mitschrift von Christoph Stöpel

Wintersemester 2002/2003

Inhaltsverzeichnis

1 Prolog Einführung **5**

1.1 Syntax 5

1.2 Repräsentation von Wissen 6

1.3 Listen, Deklaration und Operationen 7

1.4 Erweiterungen des Grund Prolog 7

2 Repräsentations- und Entscheidungssysteme **11**

2.1 Überblick 11

2.2 Semantik und Interpretation 11

2.3 Modelle und logische Konsequenzen 12

2.4 Beweise, Gültigkeit und Vollständigkeit 13

2.4.1 Bottom-Up Beweisprozedur 13

2.4.2 Top-Down Beweisprozedur 14

2.5 Entscheidung mit Variablen 15

3 Suchen **17**

3.1 Blinde Suchstrategien 18

3.2 Heuristische Suche 19

3.3 Constraint Satisfaction Problems 21

3.3.1 Generieren und Testen 22

3.3.2 Backtracking Algorithmus 22

3.3.3 Konsistenz Algorithmen und Domain-Splitting 22

3.3.4 Hill Climbing 24

4 Lernen **25**

4.1 Entscheidungsbäume und ID3 26

4.2 Neuronale Netze 28

5 Jenseits definiten Wissens **29**

5.1 Einzigartige Namen (Unique Names Assumption) 29

5.2 Complete Knowledge Assumption (CKA) 30

5.3 Negation as Failure (NAF) 31

5.4 Integritätsbedingungen, Horn-Klauseln 32

6 Handlungsplanung **33**

6.1 Überblick 33

6.2 STRIPS Repräsentation 34

6.3 Situationskalkül 35

A Beispieldateien **37**

A.1 Prolog Einführung — Familie.pl 37

A.2 CSP — Send-More-Money Problem 38

Kapitel 1

Prolog Einführung

1.1 Syntax

Prolog ist eine logische Programmiersprache, die für zahlreiche Betriebssysteme frei verfügbar ist. Die bekanntesten Interpreter sind SWI-Prolog[1] und Eclipse-Prolog[2]. Wie bei allen logischen Programmiersprachen werden dem Prologsystem Fakten und (Schluß-)Regeln vorgegeben, so daß daraus Antworten abgeleitet werden können.

Variablen

- Variablen fangen mit einem Großbuchstaben an (X,List)
- anonyme Variablen fangen mit einem Unterstrich an (_member,_) und werden von Prolog nicht ausgegeben
- der Gültigkeitsbereich einer Variablen ist immer eine Klausel, d.h wenn X in zwei Klauseln vorkommt handelt es sich um zwei verschiedene Variablen

Klauseln

- sind Fakten (`weiblich(maria).`)
- sind Regeln (`vorfahre(X,Y):-elternteil(X,Z),vorfahre(Z,Y).`)
- Fakten und Regeln heißen *definite Klauseln*
- sind Ziele oder Anfragen (`?-vater(steffen,maria).`)

Ein logisches Programm ist eine Folge von definiten Klauseln.

Matching

Zwei Terme matchen (passen zueinander), wenn sie

- identisch sind
- die Variablen der beiden Terme so mit Objekten (Variablen, Atomen, anderen Objekten) instanziert werden knnen, daß nach der Ersetzung der Variablen durch diese Ersetzung identische Terme entstehen.

Beispiel 1.1.1 (Matching)

Term 1	*Term 2*	*Matchen*
`zeit(13,30).`	`zeit(13,X).`	*ja, X kann mit 13 unifiziert werden*
`zeit(14,30).`	`zeit(13,X).`	*nein, 13 kann nicht mit 14 unifiziert werden*
`zeit(13,30).`	`zeit(13,30,05).`	*nein, abweichende Stelligkeit*
`zeit(13,X,05).`	`zeit(Y,Z,05).`	*ja, ...*
`p(r(X,b),q(c,d).`	`p(r(a,b),Y).`	*ja, ...*

1.2 Repräsentation von Wissen

Fakten(Wissen) werden als Relationen in die Prolog Datenbasis aufgenommen. Das Faktum „Steffen ist Vater von Paul.“ wird durch die Syntax `vater(steffen,paul).` eingebunden. Konkrete Objekte (steffen, paul) heißen *Atome*.

Die Einführung von *Regeln* eröffnet dem Prolog-System, die Möglichkeit, aus gegebenen Fakten neues Wissen abzuleiten.

Beispiel 1.2.1 (Konjunktion)
X ist Vater von Y, wenn X Vater von Z ist und Z Vater von Y ist:
```
grossvater(X,Y):-vater(X,Z),vater(Z,Y).
```

Beispiel 1.2.2 (Disjunktion)
X ist Elternteil von Y, wenn X Vater Y ist oder X Mutter von Y ist:
```
elternteil(X,Y):-vater(X,Y).
elternteil(X,Y):-mutter(X,Y).
```

Beispiel 1.2.3 (Vergleich)
X ist verschieden von Y, wenn X ungleich Y ist
```
verschieden(X,Y):-X\==Y.
```

Wie beantwortet Prolog Anfragen?

- Prolog versucht alle Teilanfragen zu erfüllen, d.h. zu zeigen, daß die Anfrage logisch aus den Fakten und Regeln folgt
- Kommen Variablen in der Anfrage vor, versucht Prolog diese mit Objekten zu instanziieren und dann die instanziierte Anfrage zu erfüllen
- ist eine Anfrage nicht aus der Datenbasis herleitbar, kann Prolog sie nicht erfüllen

1.3 Listen, Deklaration und Operationen

- leere Liste: `[]`
- beliebige Liste mit Kopf x: `[x|Rest]` oder auch `.(x,Rest)` (Termnotation)
- in Termnotation ist `[]` Listenende: `.(1,.(2,.(3,[])))`
- Liste ansonsten `[1,2,3]`

Beispiel 1.3.1 (Liste durchlaufen)

```
liste([]).
liste([_|Rest]):-liste(Rest).
```

Beispiel 1.3.2 (Element einer Liste)

```
member(X,[X|Rest]). % X ist Element einer Liste, deren Kopf X ist
member(X,[_|Rest]):-member(X,Rest). % X ist Element einer Liste mit beliebigem Kopf, wenn X im Rumpf der Liste ist
```

Beispiel 1.3.3 (Liste L1 und L2 zu L3 konkatenieren)

```
app([],L,L). % leer + L = L
app([E,L1],L2,[E,L3]):-app(L1,L2,L3).
```

Beispiel 1.3.4 (Einfügen am Listenende)

```
tail_insert(E,[],[E]).
tail_insert(E,[H|R],[H,NR]):-tail_insert(E,R,NR).
```

1.4 Erweiterungen des Grund Prolog

Arithmetik

Syntax: X is Expression

- `is` unifiziert X und Expression
- Expression muß voll instanziert sein
- Expression wird dabei ausgewertet

Operationen:

+	Addition	-	Subtraktion
*	Multiplikation	\	Division
<	kleiner	>	großer
<=	kleiner gleich	>=	großer gleich
=:=	gleich	=/=	ungleich

Beispiel 1.4.1 (Arithmetik)
(1) `X is 3*4.`
(2) `3*4 is 4*3`
(3) `12 is X+4`

Klassifikation von Termen

number(X).	Ist X eine Zahl?	integer(X).	Ist X eine ganze Zahl?
float(X).	Ist X eine Fließkommazahl?		
atom(X).	Ist X ein Atom?	atomic(X)	Ist X ein Atom oder eine Zahl?
var(X).	Ist X eine Variable[1]?	nonvar(X)	Ist X keine Variable?
T1==T1	Sind T1 und T2 identisch?	T1\==T2	Sind T1 und T2 verschieden?
T1=T2	Sind T1 und T2 unifizierbar?		

Input/ Output

put(X).	schreibe ASCII Zeichen
get(X).	lese druckbares ASCII Zeichen und unifiziere es mit X
get0(X).	lese ASCII Zeichen und unifiziere es mit X
tab(N).	erzeugt N Leerzeichen
write(X).	schreibt Term X
read(X).	lese nächsten Term und unifiziere mit X

put, get, get0, write und read funktionieren nur einmal, sie werden beim Backtracing übergangen.

Cut!

Mit dem Cut werden Suchbäume für Effizienzgewinne beschnitten.

- Ein Cut schneidet alle Klauseln mit dem selben Prädikat aus dem Suchraum, die unter der aktuellen Klausel mit dem Cut stehen.
- Ein Cut schneidet alle alternativen Lösungen der Goals aus dem Suchraum, die in der Klausel links vom Cut stehen. D.h. für Goals links vom Cut wird höchstens eine Lösung betrachtet.
- Ein Cut beeinflußt nicht die Goals zu seiner Rechten.

Beispiel 1.4.2 (Maximales Element mit Cut bestimmen)

```
max(X,Y,Z) :- X>=Y, !, Z=X.
max(X,Y,Y).
```

Analyse und Synthese von Termen

Syntax:

```
functor(Term, FunctorName, Arity).
arg(N,Term,Argument).
Term = ..List.  % oder =..(Term,List)
```

Beispiel 1.4.3 (Termanalyse)

```
functor(f(a,b),f,2).            yes
functor(f(a,b),F,A).            F=f, A=2
functor(f(a,b),f,3)             no (more) solution.
arg(2,f(a,b,c),b))              yes
arg(2,f(a,f(a,b)),f(X,Y)).      X=a, Y=b
Term=..[likes,david,play].      Term=likes(david,play)
s([1,4,5,6])=..List             List=[s,[1,4,5,6]].
```

Kapitel 2

Repräsentations- und Entscheidungssysteme

2.1 Überblick

Ein Repräsentations- und Entscheidungssystem (Representation and Reasoning System - RRS) besteht aus:

- formaler Sprache: beschreibt gültige Sätze
- Semantik: beschreibt die Bedeutung der Symbole
- Entscheidungstheorie (Beweisprozedur): gibt an, wie ein Anwort produziert werden kann

Vereinfachende Festlegungen über initiale RRS:

- Das Wissen ist sinnvoll in Form von Objekten und Relationen zwischen Objekten beschrieben.
- Die Wissensbasis besteht nur aus entgültigen und positiven Aussagen.
- Die Umgebung ist statisch, d.h. Zeitverlauf u.ä. ist nicht zu beachten.
- Die Anzahl zu beschreibender Objekte ist endlich und kann eindeutig mit Namen identifiziert werden.

2.2 Semantik und Interpretation

Die *Semantik* beschreibt die Bedeutung von Sätzen einer Sprache.
Die *Interpretation* ist ein Tripel $I = (D, \phi, \pi)$ mit

- D, der nichtleeren Menge der betrachteten (realen) Objekte, wie z.B. Personen, Räume, die gewöhnlich nicht im Computer gespeichert werden können

- ϕ, eine Funktion, die jeder Konstanten ein Element aus D zuordnet
- π, eine Funktion, die einem Prädikat p auf n-Tupeln aus D ein Element aus $\{true, false\}$ zuordnet, hat p keine Argument ist $\pi(p)$ entweder true oder false

Die Klausel $h \leftarrow b_1 \wedge ... \wedge b_m$ ist falsch in Interpretation I, wenn h in I falsch ist und jedes b_i in I wahr ist, ansonsten ist sie wahr in Interpreation I.

Beispiel 2.2.1 (Interpretation)
Konstanten: Telefon, Stift, Phone
Prädikate: geräuschvoll (einstellig)
D = {✂, ☎, ✎}
φ(Telefon) = ☎, φ(Stift) = ✎, φ(Phone) = ☎
π(geräuschvoll) = {✂ → false, ☎ → true, ✎ → false}

2.3 Modelle und logische Konsequenzen

- Eine Wissenbasis KB ist wahr in Interpretation I, wenn **jede** Klausel der KB in I wahr ist.
- Ein **Modell** einer Menge von Regeln ist eine Interpretation in der alle Regeln wahr sind.
- Wenn KB eine Menge von Klauseln ist und g eine Konjunktion von Atomen (Fakten), ist g **logische Konsequenz** der Wissensbasis, falls g in jedem Modell von KB wahr ist. Man schreibt $KB \models g$.
- Kurz: $KB \models g$, falls es keine Interpretation, in der KB wahr und g falsch ist, gibt.

Beispiel 2.3.1 (Modell)

$$KB = \begin{cases} p \leftarrow q. \\ q. \\ r \leftarrow s. \end{cases}$$

Beispielhafte Interpretationen:

	$\pi(p)$	$\pi(q)$	$\pi(r)$	$\pi(s)$	
I_1	*wahr*	*wahr*	*wahr*	*wahr*	*ist ein Model von KB*
I_2	*falsch*	*falsch*	*falsch*	*falsch*	*kein Model von KB*
I_3	*wahr*	*wahr*	*falsch*	*falsch*	*ist ein Model von KB*
I_4	*wahr*	*wahr*	*wahr*	*falsch*	*ist ein Model von KB*
I_5	*wahr*	*wahr*	*falsch*	*wahr*	*kein Model von KB*

Atom g ist eine logische Konsequenz von KB falls:

1. g ist ein Fakt in KB, *oder*
2. es gibt eine Regel $g \leftarrow b_1 \wedge ... \wedge b_k$ in KB, so daß jedes b_i eine logische Konsequenz von KB ist.

2.4 Beweise, Gültigkeit und Vollständigkeit

- Ein *Beweis* ist eine mechanisch herleitbare Demonstration, daß eine Formel logisch aus einer Wissensbasis folgt.
- Bei einem gegebenen Beweisverfahren bedeutet $KB \vdash g$, daß g aus der Wissensbasis KB hergeleitet werden kann.
- Eine Beweisprozedur ist vernüftig, wenn: aus $KB \vdash g$ folgt $KB \models g$.
- Eine Beweisprozedur ist gültig, wenn: aus $KB \models g$ folgt $KB \vdash g$.

2.4.1 Bottom-Up Beweisprozedur

Wenn „$h \leftarrow b_1 \wedge ... \wedge b_m$" eine Klausel in der Wissensbasis ist (m kann auch 0 sein), und jedes b_i hergeleitet werden kann, dann kann h hergeleitet werden (*modus ponens*).

$KB \vdash g$, wenn $g \in C$ am Ende dieser Prozedur:

C:={}
repeat
  select Klausel „$h \leftarrow b_1 \wedge ... \wedge b_m$" in KB, so dass $b_i \in C$ für alle i und $h \notin C$
  $C := C \cup \{h\}$
until keine Klauseln mehr zu Auswahl

Beispiel 2.4.1 (Bottom-Up Beweis)

$a \leftarrow b \wedge c.$	$c \leftarrow e.$	$f \leftarrow j \wedge e.$
$a \leftarrow e \wedge f.$	$d \leftarrow k.$	$f \leftarrow c.$
$b \leftarrow f \wedge k.$	$e.$	$j \leftarrow c.$

Prozedur Werteverlauf:

i	*C*	*ausgewählte Regel(n)*
0	$\{\}$	–
1	$\{e\}$	$e.$
2	$\{e, c\}$	$c \leftarrow e.$
3	$\{e, c, f\}$	$f \leftarrow c.$, $j \leftarrow c.$
4	$\{e, c, f, j\}$	$j \leftarrow c.$, $a \leftarrow e \wedge f.$

Gültigkeit – Aus $KB \vdash g$ folgt $KB \models g$

Angenommen es gibt ein g, so daß $KB \vdash g$ und $KB \not\models g$. Sei h das erste zu C hinzugefügte Atom, das nicht in jedem Model von KB wahr ist und nehme an, daß es in Modell I von KB nicht wahr ist. Dann muß es eine Klausel der Form $h \leftarrow b_1 \wedge ... \wedge b_m$ in der Wissensbasis geben. Jedes b_i ist wahr in I. h ist falsch in I, also ist die Klausel in I falsch. Also ist I kein Modell von KB. *Widerspruch:* es gibt kein solches g.

Fixpunkt

Die Menge C am Ende des Bottom-Up Algorithmus wird *Fixpunkt* genannt. Sei I eine Interpretation in der jedes Elemnt des Fixpunktes wahr und jedes andere Atom falsch ist. I ist ein Modell von KB.
Beweis: Angenommen $h \leftarrow b_1 \wedge ... \wedge b_m$ der Wissensbasis ist falsch in I. Dann ist h falsch und

jedes b_i ist wahr in I. Also kann h zu C hinzugefügt werden. Widerspruch zu C ist Fixpunkt. I wird auch minimales Modell genannt.

Vollständigkeit — Aus $KB \models g$ folgt $KB \vdash g$

Angenommen $KB \models g$. Dann ist g in allen Modellen von KB wahr. Also ist g auch im minimalen Modell wahr. Also wird g vom Bottom-Up Algorithmus erzeugt. Daraus folgt $KB \vdash g$.

2.4.2 Top-Down Beweisprozedur

Idee: Suche rückwärts von einer Anfrage aus, um festzustellen, ob sie eine logische Konsequenz der Wissenbasis (KB) ist. Eine Antwort-Klausel hat die Form

$$yes \leftarrow a_1 \wedge a_2 \wedge a_2 \wedge ... \wedge a_m$$

Die Ersetzung (SLD Resolution) von Atom a_i mit

$$a_i \leftarrow b_1 \wedge ... \wedge b_p$$

führt zu der Anwortklausel

$$yes \leftarrow a_1 \wedge ... \wedge a_{i-1} \wedge b_1 \wedge ... \wedge b_p \wedge a_{i+1} \wedge ... \wedge a_m$$

Herleitung

Eine Antwort ist eine Antwortklausel mit $m = 0$, also die Antwortklausel $yes \leftarrow$.
Eine Herleitung der Anfrage „$?q_1 \wedge ... \wedge q_k$“ aus einer Wissensbasis ist eine Folge von Antwort-Klauseln $\gamma_0, \gamma_1, ..., \gamma_n$, so daß

- γ_0 ist die Antwort der Regel $yes \leftarrow q_1 \wedge ... \wedge q_k$
- γ_i wird gewonnen, indem γ_{i-1} mit einer Klausel aus KB aufgelöst wird und
- γ_n ist die Anwort.

Die folgende Prozedur löst die Anfrage $?q_1 \wedge ... \wedge q_k$:

ac := „$yes \leftarrow q_1 \wedge ... \wedge q_k$“
repeat
    select eine Konjunktion a_i aus dem Körper von ac
    choose eine Klausel C mit dem Kopf a_i aus KB
    ersetze a_i im Körper von ac mit dem Körper von C
until ac ist eine Anwort.

Nichtdeterministische Auswahl

Unkritischer Nicht-Determinismus: Wenn eine Auswahl nicht zu einer Lösung führt, gibt es keine Möglichkeit Alternativen auszuprobieren. In obiger Prozedur: `select`.
Kritischer Nicht-Determinismus: Wenn eine Auswahl nicht zu einer Lösung führt, könnten andere dies tun. Oben: `choose`.

Beispiel 2.4.2 (Top-Down Prozedur: Wissensbasis KB)

$a \leftarrow b \wedge c.$	$a \leftarrow e \wedge f.$	$b \leftarrow j \wedge k.$
$c.$	$d \leftarrow k.$	$e.$
$f \leftarrow j \wedge e.$	$f \leftarrow c.$	$j \leftarrow c.$

Beispiel 2.4.3 (Top-Down Prozedur: Erfolgreiche Herleitung von $?a$)

$\gamma_0 : yes \leftarrow a$	$\gamma_3 : yes \leftarrow c$
$\gamma_1 : yes \leftarrow e \wedge f$	$\gamma_4 : yes \leftarrow e$
$\gamma_2 : yes \leftarrow f$	$\gamma_5 : yes \leftarrow$

Beispiel 2.4.4 (Top-Down Prozedur: Erfolglose Herleitung von $?a$)

$\gamma_0 : yes \leftarrow a$	$\gamma_3 : yes \leftarrow c \wedge k \wedge c$
$\gamma_1 : yes \leftarrow b \wedge c$	$\gamma_4 : yes \leftarrow e \wedge k \wedge c$
$\gamma_2 : yes \leftarrow f \wedge k \wedge c$	$\gamma_5 : yes \leftarrow k \wedge c$

2.5 Entscheidung mit Variablen

- Eine Instanz eines Atoms oder einer Klausel wird durch einheitliche Ersetzung der Variablen durch Terme gewonnen
- Eine *Ersetzung* ist eine endliche Menge der Form $\{V_1/t_1, ..., V_n/t_n\}$, wobei jedes V_i eine Variable und jedes t_i ein Term ist.
- Die Anwendung einer Substitution $\sigma = \{V_1/t_1, ..., V_n/t_n\}$ auf ein Atom oder eine Klausel e, geschrieben $e\sigma$, ist eine Instanz von e, in der jedes Vorkommen von V_i durch t_i ersetzt wurde.

Beispiel 2.5.1 (Einfache Ersetzungen)
$\sigma = \{X/A, Y/b, Z/C, D/e\}$
$p(A, b, C, D)\sigma = p(A, b, C, e)$
$p(X, Y, Z, e)\sigma = p(A, b, C, e)$

Unifikatoren

Die Ersetzung σ ist ein Unifikator von e_1 und e_2, wenn $e_1\sigma = e_2\sigma$. σ ist ein allgemeinster Unifikator (most general unifier - mgu) von e_1 und e_2, wenn σ Unifikator von e_1 und e_2 ist und wenn es eine weitere Substitution σ' gibt, die auch e_1 und e_2 unifiziert, so daß $e\sigma'$ eine Instanz von $e\sigma$ für alle Atome e ist. Wenn zwei Atome eine Unifikator haben, besitzen sie auch einen allgemeinsten Unifikator.

Beispiel 2.5.2 (Unifikatoren)
$p(A, b, C, D)$ und $p(X, Y, Z, e)$ haben die Unifikatoren:
$\sigma_1 = \{X/A, Y/b, Z/C, D/e\}$
$\sigma_2 = \{A/X, Y/b, C/Z, D/e\}$
$\sigma_3 = \{A/a, X/a, Y/b, C/c, Z/c, D/e\}$
$\sigma_4 = \{X/A, Y/b, Z/A, C/A, D/e\}$
σ_1 und σ_2 sind allgemeinste Unifikatoren (mgu).

Endliche Auflösung mit Variablen

Sei $yes(t_1, ..., t_k) \leftarrow a_1 \wedge a_2 \wedge ... \wedge a_m$ eine verallgemeinerte Antwortklausel, wobei $t_1, ..., t_k$ Terme und $a_1, ..., a_m$ Atome sind.
Die SLD-Resolution von a_i mit der Klausel $a \leftarrow b_1 \wedge ... \wedge b_p$, wobei a_i und a den allgemeinsten Unifikator σ haben ist: $(yes(t_1, ..., t_k) \leftarrow a_1 \wedge ... \wedge a_{i-1} \wedge b_1 \wedge ... \wedge b_p \wedge a_{i+1} \wedge ... \wedge a_m)\sigma$

Kapitel 3

Suchen

Oft hat man keinen Algorithmus zur Problemlösung gegeben, nur eine Umschreibung der Lösung — nach der Lösung muß gesucht werden. Suche ist eine Möglichkeit kritischen Nicht-Determinismus zu implementieren.

Suchgraphen

- Ein *Graph* besteht aus einer Menge N von Knoten und einer Menge A von geordneten Paaren von Knoten, genannt Kanten.
- Knoten n_2 ist ein Nachfolger von n_1, wenn es eine Kante von n_1 zu n_2 gibt, also wenn $(n_1, n_2) \in A$.
- Ein *Pfad* ist eine Folge von Knoten $(n_0, n_1, ..., n_k)$, so daß $(n_{i-1}, n_i) \in A$.
- Bei gebener Menge von Startknoten und Zielknoten ist eine Lösung ein Pfad von einem Startknoten zu einem Zielknoten.

Graphensuche

Gegeben seien ein Graph, Startknoten und Zielknoten. Ein generischer Suchalgorithmus durchsucht schrittweise Pfade von Startknoten aus und beachtet dabei eine Suchschranke von Pfaden (vom Startknoten aus), die schon durchsucht wurden. Bei fortschreitender Suche wird die Suchschranke auf unerkundete Knoten ausgedehnt, bis ein Zielknoten ereicht wird. Eine *Suchstrategie* wird definiert durch die Art, wie die Suchschranke erweitert wird.

Generischer Algorithmus zur Graphensuche

```
Input: ein Graph, eine Menge von Startknoten
       eine boolesche Funktion goal(n), die testet, ob n ein Zielknoten ist
Schranke := { s | s ist Startknoten};
while Schranke ist nicht leer:
      wähle und entferne Pfad (n_0, ..., n_k) aus Schranke;
      if goal(n_k) return (n_0, ..., n_k)
      für jeden Nachfolger n von n_k
            Füge (n_0, ..., n_k, n) zu Schranke hinzu
end while
```

3.1 Blinde Suchstrategien

Tiefensuche(Depth First Search)

- Tiefensuche verarbeitet die Suchschranke als einen Stack
- Es wird jeweils das als letztes zur Suchschranke hinzugefügte Element ausgewählt
- Wenn die Suchschranke $[p_1, p_2, ...]$ ist, wird p_1 ausgewählt und alle Pfade, die p_1 erweitern werden oben auf den Stack gelegt (vor p_2). Erst wenn alle Pfade von p_1 aus durchsucht worden sind, wird p_2 durchsucht.

Komplexität

Es ist nicht garantiert, daß Tiefensuche in unendlichen Graphen oder in Graphen mit Schleifen anhält. Der Speicherverbrauch ist linear im Vergleich zu den Knoten, die erkundet werden.

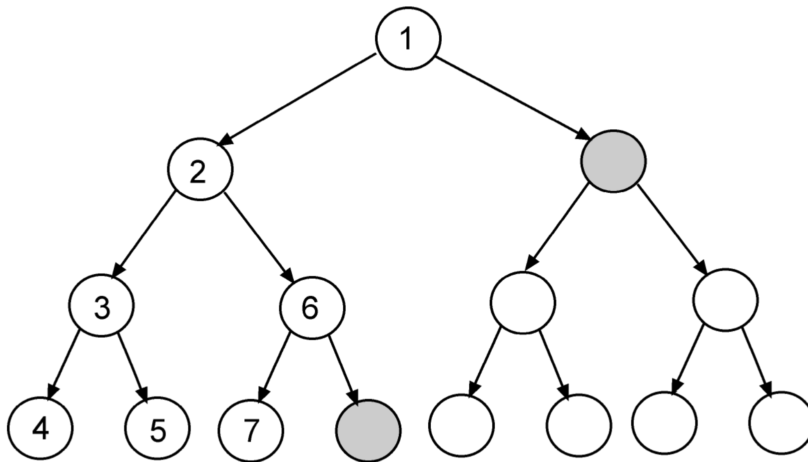

Abbildung 3.1: Strategie der Tiefensuche

Breitensuche (Breadth-first Search)

- Breitensuche verarbeitet die Suchschranke als eine Warteschlange (queue)
- Es wird jeweils das am nächsten zur Suchschranke liegende Element ausgewählt
- Wenn die Suchschranke $[p_1, p_2, ..., p_r]$ ist, wird p_1 ausgewählt und seine Nachfahren am Ende der Schlange (nach p_r) eingefügt. Dann wird p_2 ausgewählt.

Komplexität

Der Verzweigungsfaktor eines Knotens ist die Anzahl seiner Nachfahren, wenn der Verzweigungsfaktor für alle Knoten endlich ist, so findet Breitensuche garantiert eine Lösung, wenn eine existiert. Diese Lösung geht uber die wenigsten Knoten. Die Zeitkomplexität steigt exponentiell zur Pfadlänge n mit dem Verzweigungsfaktor b, nämlich b^n. Der Speicherbedarf steigt ebenso exponentiell zur Pfadlänge mit b^n.

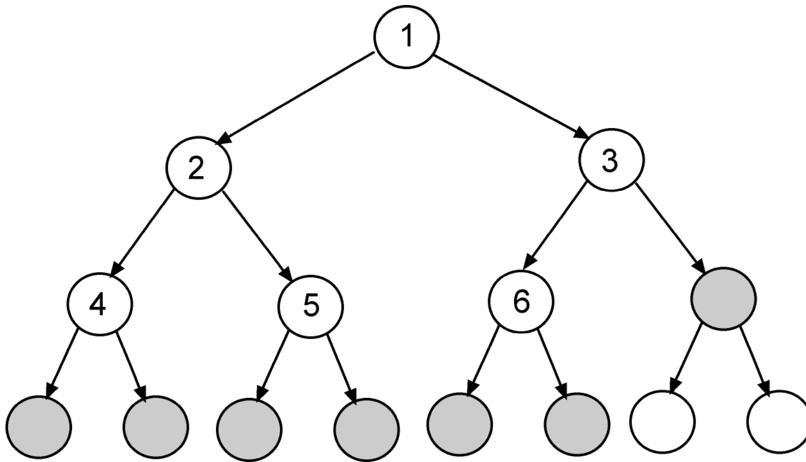

Abbildung 3.2: Strategie der Breitensuche

Iterative Tiefensuche (Iterative Deepening)

- Alle Suchstrategien, die das Halten garantieren benötigen exponentiell wachsenden Speicher, die Idee ist hier, alle Elemente der Suchschranke neu zu berechnen, anstatt sie zu speichern.
- Zuerst werden alle Pfade der Länge 0, dann der Länge 1, 2 usw. durchsucht, dazu wird die Tiefensuche in der Suchtiefe begrenzt.
- Wenn eine Lösung nicht in einem Graphen der Länge n gefunden wird, sucht Tiefensuche im Baum der Länge $n + 1$.

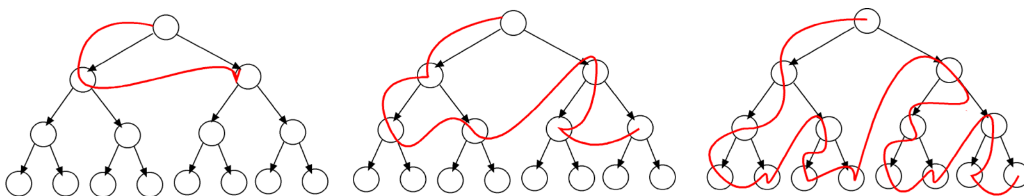

Abbildung 3.3: Strategie der Iterativen Tiefensuche

3.2 Heuristische Suche

- Idee: Das Ziel bei der Auswahl der Pfade nicht ignorieren, oft gibt es „grobes“ Wissen, das verwendet werden kann, um das Ziel zu finden: *Heuristik*
- $h(n)$ ist eine Schätzung der Kosten des kürzesten Pfades vom Knoten n zum Ziel
- $h(n)$ verwendet nur schnell berechnenbare Infomationen über einen Knoten und kann auf Pfade erweitert werden $h((n_0, ..., n_k)) = h(n_k)$
- $h(n)$ ist eine untere Schranke, es gibt also keinen Pfad von n zum Ziel, der kürzer als $h(n)$ ist

Best-first Search (Greedy Search — Gierige Suche)

- Idee: Wähle immer den Pfad dessen Ende nach der heuristischen Funktion dem Ziel am nächsten ist.
- Best-first Search selektiert also immer den Knoten mit kleinstem h-Wert der Suchschranke.
- Die Suchschranke wird als Prioritäts-Warteschlange geordnet durch h behandelt.

Komplexität

Best-first Search benötigt exponentiellen Speicher in Pfadlänge und ist anfällig für Endlosschleifen und kann damit kein Ergebnis garantieren, selbst wenn eine Lösung existiert. Es wird nicht immer der kürzeste Weg gefunden.

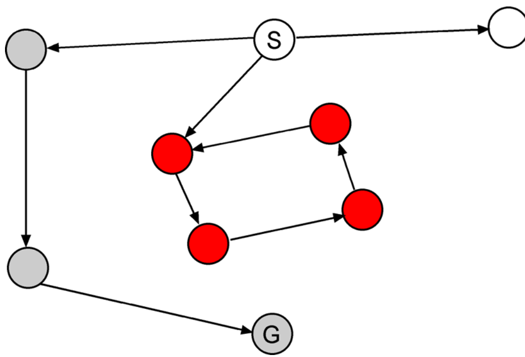

Abbildung 3.4: Greedy-Search in der Falle

Heuristische Tiefensuche

- Ist eine Möglichkeit heuristisches Wissen bei der Tiefensuche anzuwenden.
- Idee: Bevor die Nachfahren eines Knotens auf den Stack der Suchschranke gelegt werden, werden diese nach h sortiert.
- Es wird lokal ein Unterbaum zur Suche ausgewählt, aber immer noch Tiefensuche ausgeführt. Alle Pfade vom obersten Knoten des Stacks werden vor den anderen erkundet.
- Der Speicherbedarf ist linear, es wird garantiert eine Lösung gefunden (wenn sie existiert).

Der A* Algorithmus

- A* berücksichtigt sowohl die Kosten des bisher gegangenen Pfades als auch die heuristischen Werte.
- $cost(p)$ sind die Kosten des Pfades p und $h(p)$ schätzt die Kosten von p aus bis zum Ziel.
- $f(p) = cost(p) + h(p)$ schätzt die gesamten Kosten vom Anfangsknoten zum Zielknoten über den Pfad p.
- Die Suchschranke wird als Prioritätswarteschlage sortiert nach $f(n)$ gespeichert.

- Es wird immer der Knoten der Suchschranke ausgewählt, der der Schätzung nach die wenigsten Kosten vom Start zum Zielknoten über diesen Knoten verursacht.

Warum ist A* optimal?

Wenn es eine Lösung gibt, findet A* immer die optimale Lösung falls

- der Verzweigungsfaktor ist endlich
- jeder Schritt Kosten größer Null verursacht (es gibt ein $\epsilon > 0$, so daß alle Kosten größer als ϵ sind) und
- $h(n)$ ist eine untere Schranke für die Länge des kürzesten Pfades von n zu einem Zielknoten

Beweis:
Kann es einen kürzeren Pfad zum Ziel geben, falls ein Pfad p zum Ziel aus der Suchschranke ausgewählt wird?
Angenommen Pfad p' ist in der Suchschranke. Weil p vor p' ausgewählt wurde und $h(p) = 0$ gilt $cost(p) \leq cost(p') + h(p')$. Weil h eine untere Schranke ist, gilt $cost(p') + h(p') \leq cost(p'')$ für jeden Pfad p'' zu einem Ziel, der p' erweitert. Es folgt $cost(p) \leq cost(p'')$ für jeden anderen Pfad zu einem Ziel.

Zusammenfassung Suchstrategien

Strategie	Auswahl aus Suchschranke	Hält?	Speicherbedarf
Tiefensuche	letzter hinzugefügter Knoten	Nein	linar
Breitensuche	erster hinzugefügter Knoten	Ja	exponentiell
heuristische Tiefensuche	lokales Minimum nach $h(n)$	Nein	liniear
Best-First	Minimum nach $h(n)$	Nein	exponentiell
A*	Minimum von $f(n) = cost(n) + h(n)$	Ja	exponentiell

3.3 Constraint Satisfaction Problems

Gegeben seien eine Menge von Variablen mit einem Wertebereich (Domain). Gesucht ist eine mögliche Belegung der Variablen mit Werten, so daß bestimmte Bedingungen erfüllt sind (hard constraints) und die Kosten gering sind (Optimierung, soft constraints). Ein solches Problem wird Randbedingungsproblem oder Contraint Satisfaction Problem (CSP) gennannt.

Wichtig bei diesen Problemen ist aber nicht der Pfad zur Lösung, sondern nur die Lösung an sich. Im Vergleich zur herkömmlichen Suche gibt es keine vordefinierten Startknoten und die Probleme sind so groß (z.B. an Variablen), daß systematische Suche aus Platzgründen nicht möglich ist.

Ein CSP ist also charakterisiert durch:

- eine Menge von Variablen $V_1, V_2, ..., V_n$
- jede Variable verfügt über einen Wertebereich D_{V_i}
- es gibt Relationen (Randbedingungen) zwichen Wertebereichen von Variablen, die mögliche Belegungen eingrenzen

- die Lösung eines CSP ist ein n-Tupel mit Werten für Variabeln, so daß die Randbedingungen erfüllt sind

Beispiel 3.3.1 (Zeitplanung)

Variablen:	*A...E stehen für die Startzeitpunkte von verschiedenen Aktivitäten*			
Wertebereiche:	$D_A = D_B = D_C = D_D = D_E = \{1,2,3,4\}$			
Constraints:	$A \neq B$	$A = D$	$C < D$	$E < C$
	$B \neq C$	$B \neq 3$	$E < A$	$E < C$
	$B \neq D$	$C \neq 2$	$E < B$	

3.3.1 Generieren und Testen

Generiere jede mögliche Kombination $D = D_{V_1} \times D_{V_2} \times ... \times D_{V_n}$ und teste, ob jede davon den Bedingungen entspricht. Diese Methode ist immer exponentiell.

Am Beispiel:

$$\begin{aligned} \text{D } &= D_A \times D_B \times D_C \times D_D \times D_E \\ &= \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \\ &= \{(1,1,1,1,1),(1,1,1,1,2),(1,1,1,1,3),...,(4,4,4,4,4)\} \end{aligned}$$

3.3.2 Backtracking Algorithmus

Durchsuche $D = D_{V_1} \times D_{V_2} \times ... \times D_{V_n}$ systematisch, indem Variablen nacheinander belegt werden, aber jede Bedingung überprüft wird, sobald alle seine Variablen belegt sind. Jede partielle Lösung, die eine Bedingung nicht erfüllt wird sofort verworfen (Beschneidung des Suchbaumes).

Am Beispiel:

Es werden belegt: $A = 1$ und $B = 1$. Jetzt werden alle Bedingungen getestet, die A und B enthalten. Wegen $A \neq B$ wird diese Lösung verworfen und nicht $C...E$ belegt und getestet, sondern B anders belegt.

3.3.3 Konsistenz Algorithmen und Domain-Splitting

Domain Konsistenz

Bevor irgendwelche Variablen belegt werden, wird getestet, ob alle Werte des Wertebereich verwendet werden können und nicht schon vorher durch einzelne Bedingungen ausgeschlossen sind.

Am Beispiel:

$D_B = \{1,2,3,4\}$ ist nicht konsistent für $B = 3$, denn es verletzt die Bedingung $B \neq 3$. D_B kann also für die folgenden Betrachtungen auf $\{1,2,4\}$ reduziert werden.

Kantenkonsistenz

- Ein Constraint-Graph hat Knoten für jede Variable mit ihrem Wertebereich und jede Bedingung $P(X,Y)$ wird durch Kanten (X,Y) und (Y,X) dargestellt.
- Eine Kante (X,Y) ist konsistent, wenn für es jedes $X \in D_X$ ein $Y \in D_Y$ gibt, so daß $P(X,Y)$ erfüllt ist. Ein Graph ist konsistent, wenn alle seine Kanten konsistent sind.
- Falls eine Kante (X,Y) nicht konsistent ist, werden alle Werte von X in D_X gelöscht, für die es keinen Wert in D_Y gibt.

Der Algorithmus

Input: - Menge von Variablen x,y
- Wertebereiche D_x, D_y für jede Variable x,y
- einstellige Bedingungen P_x für Variablen x
- zweistellige Bedingungen $P_{x,y} f\ddot{u}r x, y$
Output - konsistente Wertebereiche für jede Variable

for each variable A **do**
  $D_A = \{x \in D_A | P_A(x)\}$
$TDA = \{(x,y),(y,x) | P_{x,y}$ ist Bedingung für $x,y\}$
repeat
  select $(A,B) \in TDA$
  $TDA = TDA - (A,B)$
  $ND_A = D_A - \{x \in D_A | \exists y \in D_y(P_{x,y}(x,y))\}$
  if $ND_A \neq D_A$ **then**
    $TDA = TDA + \{(z,x) | P_{z,x}$ ist Bedingung für $z,x\}$
    $D_A = ND_A$
until $TDA = \emptyset$

Beispiel 3.3.2 (Arc-Consistency)
Domains: $A = B = \{1,2,3\}$, *Constraints:* $A < B$.

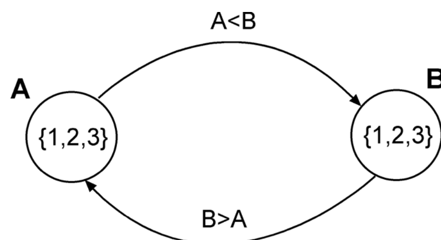

Abbildung 3.5: Contraint-Netz

Es gibt im Wertebereich von B keinen Wert, so daß mit $A = 3$ *die Bedingung* $A < B$ *erfüllt werden kann, ebenso kann* $B > A$ *mit* $B = 1$ *nicht erfüllt werden, so daß diese Werte überflüssig sind und sich als kantenkonsistente Domains ergeben:* $A = \{1,2\}$ *und* $B = \{2,3\}$.

Domain Splitting

Domain Splitting (Aufteilung von Wertebereichen) wird angewendet, wenn allein mit Konsistenzverfahren noch keine Lösung gefunden wird.

- es gibt noch Domains mit mehr als einem Element ⇒ eine davon auswählen
- diese aufteilen und für jeden Teil davon eine Lösung suchen
- es müssen nur Kanten neu besucht werden, die durch die Aufteilung beeinflußt wurden
- oft ist es günstig, eine Domain genau zu halbieren

An Beispiel 3.3.2: Arc-Consistency ergab die Domains $A = \{1, 2\}$ und $B = \{2, 3\}$. Dies ist noch keine Lösung, da nicht alle Domains einelementig sind. Also wird zum Beispiel die Domain von A in zwei Teile $A_1 = \{1\}$ und $A_2 = \{2\}$ aufgeteilt. Nun wird versucht das Problem mit den Domains (A_1, B) und (A_2, B) per Arc-Consistency zu lösen. Für A_1 ergibt sich keine Veränderung im Constraint Netz, so daß auch B aufgeteilt werden muß. Aus A_2 folgt direkt eine Lösung, denn für $B = 2$ gibt es kein A, so daß $B > A$ erfüllt wird.

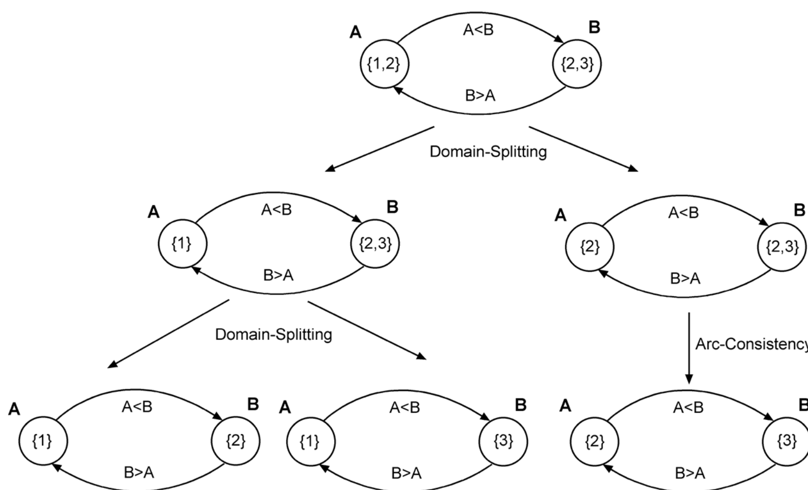

Abbildung 3.6: Lösung eines CSP ermitteln

3.3.4 Hill Climbing

Viele Suchräume sind zu groß für eine systematische Suche, daher wird in der Praxis oft Hill Climbing für Optimierungsprobleme verwendet. Dabei gibt man sich zufrieden, wenn möglichst viele, aber nicht zwangsläufig alle Constraints erfüllt sind.

- jede Variablenbelegung habe einen heuristischen Wert (beispielsweise die Anzahl der erfüllten Bedingungen)
- zu Beginn wähle eine zufällige Belegung für jede Variable
- nun wähle einen Nachbarn (eine Variation der aktuellen Belegung) mit besserem heuristischem Wert
- Probleme ergeben sich bei lokalen Maxima, die vom Algorithmus nicht mehr verlassen werden, da er nicht heuristisch schlechter gelegene Punkte besucht

Kapitel 4

Lernen

Lernen ist die Fähigkeit basierend auf Erfahrung Verhaltensweisen zu verbessern.

- der Umfang an Verhaltensweisen wird erweitert: man kan mehr tun
- die Genauigkeit wird verbessert: man kann etwas besser tun
- die Geschwindigkeit steigt: man kann etwas schneller tun

Im Zusammenhang mit Künstlicher Intelligenz bedeutet Erfahrung meist die Vorgabe von Trainingsdaten, aus denen ein Muster (eine Klasseneinteilung) gewonnen werden kann, um Vorhersagen zu treffen. Die Ungenauigkeit mit der die Klasseneinteilung erstellt und festgelegt wird, ob neue Daten in die eine oder andere Klasse gehören, wird *bias* genannt.

Beispiel 4.0.3 (Quelle: Machine Learning, Tom M. Mitchell 1997)
Im nun folgenden Beispiel werden Trainingsdaten vorgegeben, aus denen für neue Wertebelegungen eine Vorhersage zum Tennisspielen getroffen werden soll.

Tag	*Aussichten*	*Temperatur*	*Luftfeuchtigkeit*	*Wind*	*Tennis spielen*
1	*sonnig*	*heiß*	*hoch*	*schwach*	*nein*
2	*sonnig*	*heiß*	*hoch*	*stark*	*nein*
3	*bedeckt*	*heiß*	*hoch*	*schwach*	*ja*
4	*Regen*	*mild*	*hoch*	*schwach*	*ja*
5	*Regen*	*kühl*	*normal*	*schwach*	*ja*
6	*Regen*	*kühl*	*normal*	*stark*	*nein*
7	*bedeckt*	*kühl*	*normal*	*stark*	*ja*
8	*sonnig*	*mild*	*hoch*	*schwach*	*nein*
9	*sonnig*	*kühl*	*normal*	*schwach*	*ja*
10	*Regen*	*mild*	*normal*	*schwach*	*ja*
11	*sonnig*	*mild*	*normal*	*stark*	*ja*
12	*bedeckt*	*mild*	*hoch*	*stark*	*ja*
13	*bedeckt*	*heiß*	*normal*	*schwach*	*ja*
14	*Regen*	*mild*	*hoch*	*stark*	*nein*

4.1 Entscheidungsbäume und ID3

Ein Entscheidungsbaum ist ein Baum, mit

- innere Knoten sind mit Attributen beschriftet (*z.B. Wind*)
- die Kanten ausgehend von einem mit Attribut A beschrifteten Knoten, sind mit aller möglichen Werten von A beschriftet (*schwach, stark*)
- die Blätter des Baumes sind eine Klasseneinteilung (*ja, nein*)

Algorithmus: Entscheidungsbaum erstellen

- Wähle Attribut A, das „beste Entscheidungsattribut" für den nächsten Knoten
- Markiere den Knoten mit Attribut A
- Für jeden Wert von A, erstelle einen neuen Knoten unter A
- Sortiere die Beispiele zu den Attibutwerten der neuen Knoten
- Wenn die Beispiele eindeutig klassifiziert sind, dann STOP, sonst iteriere über die neuen Knoten

```
% dtlearn(+Goal,+Examples,+Attributs,-DecisionTree)
dtlearn(Goal, Examples, Attributes, Values) ←
        all_examples_agree(Goal,Examples,Values).
dtlearn(Goal, Examples, Attributes, if(Cond, YesTree, NoTree))←
        examples_disagree(Goal, Examples)∧
        select_split(Goal, Examples, Attributes, Cond, RemainAtts)∧
        split(Examples, Cond, Yes, No)∧
        dtlearn(Goal, Yes, RemainAtts, YesTree)∧
        dtlearn(Goal, No, RemainAtts, NoTree).
```

Am Beispiel

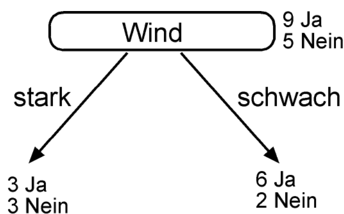

Abbildung 4.1: Teilbaum für Attribut Wind als Wurzel

In der Praxis können Attribute mehr als einen Wert annehmen, das erhöht die Komplexität der Bäume. Bisher ist nicht definiert, welche Attribute zur Aufteilung ausgewählt werden, die Bäume können beliebig groß werden. Ein idealer Entscheidungsbaum könnte aber der mit der geringsten Tiefe oder der mit den wenigsten Knoten sein.

Der ID3-Algorithmus

Ein Begriff aus der Codierungstheorie, die *Entropie* wird nun auch hier verwendet. Daher eine kurze Übersicht über die Berechnung von Informationsgehalt (Werte des Zielattributs auf Ja und Nein eingeschränkt).

- S ist die Menge der Trainingsbeispiele
- $p_\oplus$ ist der Anteil der positiven Beispiele in S
- $p_\ominus$ ist der Anteil der negativen Beispiele in S
- Die Entropie ist ein Maß für die Verteilung der Werte in S

$$\text{Entropie}(S) = -p_\oplus \log_2 p_\oplus - p_\ominus \log_2 p_\ominus$$

Da nun ein Maß für den Informationsgehalt zur Verfügung steht, läßt sich das „beste Entscheidungsattribut“ mathematisch bestimmen. Dabei wird neben der Entropie des Testkandidaten auch der Informationsgewinn möglicher Nachfolger berücksichtigt - *Information Gain*. Derjenige Knoten mit dem höchsten Wert ist der „Beste“ und wird als Knoten selektiert.
Sei S eine Menge Trainingsdaten und A das Attribut, dessen Informationsgehalt ermittelt werden soll, dann ist

$$\text{Gain}(S, A) = \text{Entropie}(S) - \sum_{v \in Werte(A)} \frac{|S_v|}{|S|} \text{Entropie}(S_v)$$

Beispiel 4.1.1 (Bestimmung des Wurzelknotens mittels Information Gain)

Luftfeuchte → $9\oplus, 5\ominus$, $Entropie(Luftfeuchte) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0{,}940$
→ $7\times$ *Hoch, davon* $3\oplus, 4\ominus$
$Entropie(Hoch) = -\frac{3}{7}\log_2\frac{3}{7} - \frac{4}{7}\log_2\frac{4}{7} = 0{,}985$
→ $7\times$ *Normal, davon* $6\oplus, 1\ominus$
$Entropie(Normal) = -\frac{6}{7}\log_2\frac{6}{7} - \frac{1}{7}\log_2\frac{1}{7} = 0{,}592$
→ ***Gain(S,Luftfeuchte)***$= 0{,}940 - \frac{7}{14}\cdot 0{,}985 - \frac{7}{14}\cdot 0{,}592$ ***= 0,151***

Wind → $9\oplus, 5\ominus$, $Entropie(Wind) = 0{,}940$
→ $8\times$ *Schwach, davon* $6\oplus, 2\ominus$
$Entropie(Schwach) = -\frac{6}{8}\log_2\frac{6}{8} - \frac{2}{8}\log_2\frac{2}{8} = 0{,}811$
→ $6\times$ *Stark davon* $3\oplus, 3\ominus$
$Entropie(Stark) = -\frac{3}{6}\log_2\frac{3}{6} - \frac{3}{6}\log_2\frac{3}{6} = 1$
→ ***Gain(S,Wind)***$= 0{,}940 - \frac{8}{14}\cdot 0{,}811 - \frac{6}{14}\cdot 1$ ***= 0,048***

Aussichten → $9\oplus, 5\ominus$, $Entropie(Aussichten) = 0{,}940$
→ $5\times$ *Sonnig, davon* $2\oplus, 3\ominus$, $Entropie(Sonnig) = 0{,}971$
→ $4\times$ *Bedeckt, davon* $4\oplus, 0\ominus$, $Entropie(Bedeckt) = 0$
→ $5\times$ *Regen, davon* $3\oplus, 2\ominus$, $Entropie(Regen) = 0{,}971$
→ ***Gain(S,Aussichten)=0,246***

Temperatur → $9\oplus, 5\ominus$, $Entropie(Temperatur) = 0{,}940$
→ $4\times$ *Heiß, davon* $2\oplus, 2\ominus$, $Entropie(Heiß) = 1$
→ $6\times$ *Mild, davon* $4\oplus, 2\ominus$, $Entropie(Mild) = 0{,}918$
→ $4\times$ *Kühl, davon* $3\oplus, 1\ominus$, $Entropie(Kühl) = 0{,}811$
→ ***Gain(S,Temperatur)=0,029***

Beispiel 4.1.2 (Forsetzung, Ermittlung weiterer Knoten)
Im oberen Beispiel wurde ***Aussichten*** *mit dem Wert 0,246 als Wurzelknoten ausgewählt. Von diesem gehen 3 Kanten mit den Beschriftungen sonnig, bedeckt und Regen aus. Wie werden nun die Unterbäume für diese Kanten erzeugt?*

1. ***bedeckt****: wählt man aus den Trainingsdaten S nur jene mit dem Attributwert bedeckt, erhält man die Trainingsdaten* $S_{bedeckt} = \{T_3, T_7, T_{12}, T_{13}\}$*, diese besitzen alle den Wert „Ja“ für Tennisspiele, sind also eindeutig klassifiziert. Diese Kante endet somit im Blatt „Ja“.*

2. ***sonnig****: Die Menge Trainingsdaten mit dem Attributwert sonnig ist* $S_{sonnig} = \{T_1, T_2, T_8, T_9, T_{11}\}$*, davon 2⊕, 3⊖,* $Entropie(sonnig) = 0,971$*. Nun werden für die restlichen Attribute die Informationsgehalte berechnet:*
   $Gain(S_{sonnig}, Luftfeuchtigkeit) = 0,971 - (3/5)0,0 - (2/5)0,0 = 0,971$
   $Gain(S_{sonnig}, Temperatur) = 0,971 - (2/5)0,0 - (2/5)1,0 - (1/5)0,0 = 0,571$
   $Gain(S_{sonnig}, Wind) = 0,971 - (2/5)1,0 - (3/5)0,918 = 0,020$
   und man sieht leicht, dass nun Luftfeuchtigkeit als nächster Knoten gewählt wird. Die Klassifizierung von S_{sonnig} *in hoch und normal ergibt im nächsten Schritt die Blätter „Ja“ und „Nein“.*

3. ***Regen****: ähnlich wie sonnig.*

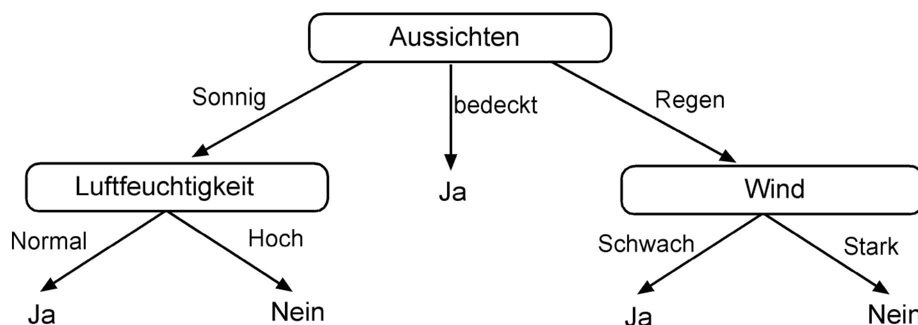

Abbildung 4.2: mittels Information Gain erzeugter Baum

4.2 Neuronale Netze

Neuronale Netze sind von den Neuronen des Gehirns und Ihren Verbindungen inspiriert. Künstliche Neuronen (Units) haben Eingaben und eine Ausgabe, diese kann mit der Eingabe anderer Units verbunden werden. Die Ausgabe einer Unit ist eine parametrische nicht-lineare Funktion seiner Eingaben. Lernen geschieht durch die Anpassung der Parameter um die Trainingsdaten zu erfüllen.

[...] *kein Vorlesungsschwerpunkt*

Kapitel 5

Jenseits definiten Wissens

5.1 Einzigartige Namen (Unique Names Assumption)

Gleichheit

Manchmal beschreiben zwei Terme dasselbe Individuum, z.B. $4 \cdot 4 = 11 + 5$. Term t_1 ist gleich Term t_2 ($t_1 = t_2$), wenn in der Interpration I t_1 und t_2 dasselbe beschreiben. Doch wie erkennt eine Maschine Gleichheit unabhängig von der Interpretation?

1. *Gleichheitsaxiome* beschreiben Gleichheit durch gewönhliche Prädikate
   - $X = X.$
   - $X = Y \leftarrow Y = X.$
   - $X = Z \leftarrow X = Y \wedge Y = Z.$
   - Für jedes n-stellige Funktionssymbol f gibt es eine Regel der Form $f(X_1, ..., X_n) = f(X_1, ..., X_n) \leftarrow X_1 = Y_1 \wedge ... \wedge X_n = Y_n$
   - Für jedes n-stellige Prädikatensymbol p gibt es eine Regel der Form $p(X_1, ..., X_n) \leftarrow p(Y_1, ..., Y_n) \wedge X_1 = Y_1 \wedge ... \wedge X_n = Y_n$
2. *Gleichheitsentscheidung je nach Problem*
   - Paramodulation: wenn $t_1 = t_2$ bekannt ist, kann jedes Vorkommen von t_1 durch t_2 ersetzt werden. Gleichheit wird als Ersetzungsregel benutzt.
   - Man kann eine kanonische Repräsentation für jedes Individuum auswählen und jedes andere Repräsentation durch diese ersetzen.

Unique Names Assumption (UNA)

Die Konvention, daß verschiedene Bezeichnungen auch verschiedene Individuen beschschreiben, ist die Unique Names Assumption. Für jedes unterscheidbare Paar t_1 und t_2 nimmt man $t_1 \neq t_2$ an.

Beispiel 5.1.1 (UNA)
Es muss nicht für jedes Paar Kurse festgestellt werden $mathe302 \neq psych302$, ...

Beispiel 5.1.2 (UNA versagt)
Manchmal ist die Annahme der Einzigartigkeit von Namen unangemessen, so ist $3 + 7 \neq 2 \cdot 5$ *falsch.*

Ungleichheitsaxiome und UNA

- $c \neq c'$ für alle unterscheidbaren Konstanten c und c'.
- $f(X_1, ..., X_n) \neq g(Y_1, ..., Y_m)$ für alle unterscheidbaren Symbole f und g.
- $f(X_1, ..., X_n) \neq f(Y_1, ..., Y_m) \leftarrow X_i \neq Y_i$ für jedes i mit $1 \leq i \leq n$.
- $f(X_1, ..., X_n) \neq c$ für jedes Funktionssymbol f und Konstante c.
- $t \neq X$ für jeden Term t in dem X vorkommt.

5.2 Complete Knowledge Assumption (CKA)

Bei der Complete Knowledge Assumption (CKA, Annahme von vollständigem Wissen) wird davon ausgegangen, daß in einer Wissensbasis alle wahren Fakten aufgeführt sind. Jeder nicht enthaltene Fakt ist falsch.

Clark's Completion

- Gegeben seien die Regeln $a \leftarrow b_1$, ..., $a \leftarrow b_n$
- Dies ist äquivalent zu $a \leftarrow b_1 \vee ... \vee b_n$
- Nach CKA muß, wenn a wahr ist, eines der b_i wahr sein, also $a \rightarrow b_1 \vee ... \vee b_n$
- Clark's Completion ergibt die Ableitbarkeit von Fakten in beide Richtungen $a \leftrightarrow b_1 \vee ... \vee b_n$

Beispiel 5.2.1 (Clark's Completion)
Bei gegebenen Klauseln

$a \leftarrow b \wedge not\ c$	$d \leftarrow e$	$g \leftarrow d \wedge not\ c$
$a \leftarrow d$	$d \leftarrow f$	$c \leftarrow h$
$b \leftarrow e$	$f \leftarrow$	

ergibt Clark's Completion

$a \leftrightarrow (b \wedge not\ c) \vee d$	$d \leftrightarrow e \vee f$	$e \leftrightarrow false$
$b \leftrightarrow e$	$f \leftrightarrow true$	$h \leftrightarrow false$
$c \leftrightarrow h$	$g \leftrightarrow d \wedge not\ c$	

Clark Normal Form

Die *Clark Normal Form* der Klausel $p(t_1, ..., t_k) \leftarrow B$
ist die Klausel

$$p(V_1, ..., V_k) \leftarrow \exists W_1 ... \exists W_m : V_1 = t_1 \wedge ... \wedge V_k = t_k \wedge B$$

wobei $V_1, .., V_k$ verschiedene Variablen sind, die nicht in der Originalklausel auftreten und $W_1, ..., W_k$ die Originalvariablen der Klausel sind. Bei mehreren Klauseln eines Prädikates wird die Normalform mit den gleichen neuen Variablen für alle Klauseln und danach Clark's Completion angewendet.

Beispiel 5.2.2 (Clark's Completion eines Prädikats)
$live(outside).$
$live(Y) \leftarrow connected_to(Y, Z) \wedge live(Z).$
$\Rightarrow\ live(A) \leftrightarrow \exists Y \exists Z : A = outside \vee (A = Y \wedge connected_to(Y, Z) \wedge live(Z))$

5.3 Negation as Failure (NAF)

Negationen im Körper von Klauseln ($\sim p$) werden unter der Complete Knowledge Assumption als nicht herleitbar aus der Wissensbasis und damit als falsch interpretiert.

Buttom-Up NAF Beweisprozedur

$C := \{\}$
repeat
  either select $(h \leftarrow b_1 \wedge ... \wedge b_m) \in KB$ so daß $b_i \in C$ für alle i und $h \notin C$;
    $C := C \cup \{h\}$
  or select h so daß
    für jede Regel $(h \leftarrow b_1 \wedge ... \wedge b_m) \in KB$ gilt
      für einige b_i ist $\sim b_i \in C$
      or für einige $b_i =\sim g$ ist $g \in C$
    $C := C \cup \{\sim h\}$
until keine Auswahl mehr möglich

Beispiel 5.3.1 (Bottom Up NAF)

$p \leftarrow q \wedge \sim r$	$p \leftarrow s$	$q \leftarrow\sim s$
$r \leftarrow\sim t$	$t \leftarrow$	$s \leftarrow w$

Prozedur Werteverlauf:

i	*C*	*ausgewählte Regel*	*Kommentar*
0	$\{\}$		
1	$\{t\}$	$t \leftarrow$	*Fakt, Zweig 1*
2	$\{t, \sim r\}$	$r \leftarrow\sim t$	*alle Regeln mit r im Kopf*
3	$\{t, \sim r, \sim w\}$	$\varnothing$	*alle Regeln mit w im Kopf*
4	$\{t, \sim r, \sim w, \sim s\}$	$s \leftarrow w$	
5	$\{t, \sim r, \sim w, \sim s, \sim q\}$	$q \leftarrow\sim t$	

Top-Down NAF

Wenn der Beweis von a fehlschlägt, kann daraus $\sim a$ geschlossen werden. Seien für a die Regeln $a \leftarrow b_1$, ..., $a \leftarrow b_n$ gegeben. Wenn jeder Körper b_i fehlschlägt, ist a nicht beweisbar.

5.4 Integritätsbedingungen, Horn-Klauseln

Klauseln, die $false$ implizieren werden Integritätbedingungen (Integrity Constraints) genannt. Das erlaubt, Schlüsse aus Widersprüchen zu ziehen. Eine Integritätsbedingung ist eine Regel der Form $false \leftarrow a_1 \wedge ... \wedge a_k$, in der a_i die Atome sind und $false$ ein spezielles Atom, das in allen Interpretationen falsch ist. Eine Horn Klausel ist entweder eine definite Klausel oder eine Integritätsbedingung. Aus Horn-Klauseln können Negationen abgeleitet werden.

Negationen herleiten

Eine Negation von a, geschrieben $\neg a$, ist eine Formel, die wahr in Interpretation I ist, falls a in I falsch ist. Die Negation ist falsch in Interpretation I, falls a in I wahr ist.

Beispiel 5.4.1 (Negative Schlüsse)

$$KB = \begin{Bmatrix} false \leftarrow a \wedge b. \\ a \leftarrow c. \\ b \leftarrow c. \end{Bmatrix} \qquad KB \models \neg c.$$

Beispiel 5.4.2 (Disjunktive Schlüsse)

$$KB = \begin{Bmatrix} false \leftarrow a \wedge b. \\ a \leftarrow c. \\ b \leftarrow d. \end{Bmatrix} \qquad KB \models \neg c \vee \neg d.$$

Konflikte in Horn Klauseln

Ein *Assumable* ist ein Atom, dessen Wert in einer (disjunktiven) Antwort auch falsch sein darf. Ein Konflikt in einer Wissensbasis KB ist eine Menge von Assumables, die KB falsch werden lassen. Ein minimaler Konflikt ist eine echte Teilmenge von denen keine Teilmenge ein Konflikt ist.

Beispiel 5.4.3 (Konflikte) $\{c, d, e, f, g, h\}$ *sind Assumables*

$$KB = \begin{Bmatrix} false \leftarrow a \wedge b. \\ a \leftarrow c. \\ b \leftarrow d. \\ b \leftarrow e. \end{Bmatrix}$$

- $\{c, d\}$ *ist ein Konflikt*
- $\{c, e\}$ *ist ein Konflikt*
- $\{c, d, e, h\}$ *ist ein Konflikt*

Auf die Darstellung des Bottum-Up Conflict Finding Algorithmus wird an dieser Stelle verzichtet, da dieser zwar kurz aber doch recht aufwendig in der Anwendung ist. [Kein Klausurthema!] Bei Interesse kann er der Literatur entnommen werden.

Kapitel 6

Handlungsplanung

6.1 Überblick

Agenten/ Roboter handeln im Verlauf der Zeit. Bei gegebenem Ziel ist es nützlich, wenn darüber nachgedacht wird, welche Aktionen in Zukunft ausgeführt werden müssen, um zu entscheiden wie jetzt zu agieren ist (Handlungsplanung).

Zeitrepräsentation

Zeit und Zeitverläufe sind in Computern schwer zu modellieren, einige unterschiedliche Ansätze werden nun erläutert.

- *Diskrete Zeit* wird als Sprung von einem Zeitpunkt zum nächsten modelliert
- *Kontinuierliche Zeit* ähnlich diskreter Modellierung mit beliebig Dichten Abständen zwischen Zeitpunkten
- *Ereignisbasierte Zeit*, Abstände zwischen Zeitpunkten müssen nicht konstant sein, sondern werden an interessierenden Ereignissen ausgerichtet
- *Zustandsbasierte Zeit* wird nicht explizit angenommen, sondern Aktionen als Zustandsübergänge interpretiert

Zeit und Relationen

- Statische Relationen sind zeitunabhängig
- Dynamische Relationen hängen von der Zeit ab und sind entweder
  - Hergeleitete Relationen (können über andere Relationen zu verschiedenen Zeitpunkten hergeleitet werden) oder
  - Primitive Relationen (Wahrheitswert der Relationen kann durch Betrachtung vorausgegangener Zeit bestimmt werden)

Aktionsmodellierung

- *Individuen*, z.B. Räume, Türen, Schlüssel und der Roboter
- *Aktionen*, z.B. Bewegen, etwas Aufnehmen, Aufschließen
- *Relationen*, z.B. Positionen von Gegenständen und dem Roboter, Zustände

Beispiel 6.1.1 (Relationen)

`at(Obj,Loc)`	*Objekt „Obj“ befindet sich an Ort „Loc“*
`carrying(Rob, Obj)`	*Roboter trägt zum aktuellen Zeitpunkt Objekt*
`opens(Key, Door)`	*Schlüssel öffnet eine Tür*

Beispiel 6.1.2 (Aktionen)

`move(Rob, From, To)`	*Agent bewegt vom Punkt „From“ zu einem anderen*
	Roboter muß sich am Ausgangspunkt der Bewegung befinden
`pickup(Rob, Obj)`	*Agent nimmt Gegenstand auf*
	muß sich am Ort des Objektes befinden
`unlock(Rob,Door)`	*Roboter schließt eine Tür auf*
	muß Schlüssel haben und vor der Tür stehen

6.2 STRIPS Repräsentation

Die STRIPS Repräsentation für Handlungsplanung verfolgt eine zustandsbasierte Sichtweise der Zeit. Bei gegebenem Zustand und einer Aktion ermittelt STRIPS (Stanford Research Institute Problem Solver), ob die Aktion im gegenwärtigen Zustand ausgeführt werden kann und welche Relationen im resultierenden Zustand wahr sind.
Prädikate werden entweder als primitiv oder hergeleitet angenommen und dementsprechend wird davon ausgegangen, daß die meisten Prädikate von einer einzelnen Aktion nicht beeinflußt werden (außer bei expliziter Erwähnung).

Die STRIPS Repräsentation von Aktionen besteht aus

- *Preconditions* (Vorbedingungen), einer Liste von Atomen die erfüllt sein müssen, um die Aktion auszuführen
- *Delete List*, eine Liste von primitiven Relationen, die nach der Aktion nicht mehr erfüllt sind
- *Add List*, eine Liste primitiver Relationen, die durch Ausführung der Aktion wahr werden

Beispiel 6.2.1 (STRIPS Aktionsausführung)

$$\left\{\begin{array}{c} sitting_at(rob, o109). \\ sitting_at(parcel, storage). \\ sitting_at(key, mail). \end{array}\right\} \stackrel{move(rob,o109,storage)}{\longrightarrow}$$

$$\left\{\begin{array}{c} sitting_at(rob, storage). \\ sitting_at(parcel, storage). \\ sitting_at(key, mail). \end{array}\right\} \stackrel{pickup(rob,parcel)}{\longrightarrow}$$

$$\left\{\begin{array}{c} sitting_at(rob, storage). \\ carrying(rob, parcel). \\ sitting_at(key, mail). \end{array}\right\}$$

Beispiel 6.2.2 (STRIPS putdown(Ag, Obj))
Preconditions: `[carrying(Ag, Obj), at(Ag, Loc)]`
Delete List: `[carrying(Ag, Obj)]`
Add List: `[sitting_at(Obj, Loc)]`

6.3 Situationskalkül

Das Situationskalkül verfügt über dieselbe Beschreibungsmächtigkeit wie STRIPS und ist dazu in der Lage Aktionen wie „Heb alles auf“ zu modellieren.
Ebenso wie STRIPS verwendet das Situationskalkül eine zustandsbasierte Repräsentation, wobei Zustände als Terme angegeben werden. Eine Situation ist dementsprechend ein Term, der den Zustand beschreibt. Es gibt zwei Möglichkeiten Zustände anzugeben: *init* entspricht dem initialen Zustand und *do(A,S)* entspricht dem Zustand der erreicht wird, wenn in S die Aktion A ausgeführt wird (sofern dies möglich ist). Eine Situation merkt sich also (recht speicherintensiv) durch welche Aktionen sie erreicht wurde.

Beispiel 6.3.1 (Aktionsausführung)
init
→ *do(move(rob, o109, o103), init)*
→ *do(move(rob,o103,mail),do(move(rob, o109, o103), init))*
→ *do(pickup(rob,key),do(move(rob,o103,mail),do(move(rob, o109, o103), init)))*

Axiome für das Situationskalkül

Der Anfangszustand wird durch Axiome beschrieben, die *init* als Situationsparameter verwenden. Hergeleitete Relationen (vgl. 6.1) werden durch Klauseln mit einer Variablen als Situation angegeben. Dabei kann zu differenzierteren Betrachtung angegeben werden, wann eine Aktion möglich ist: $poss(putdown(Ag, Obj), S) \leftarrow carrying(Ag, Obj, S)$. Frame-Axiome können eingeführt werden, um zu definieren, welche Relationen stets unbeeinflußt bleiben.

Beispiel 6.3.2 (Axiome)
Tür aufschließen: $unlocked(Door, do(unlock(Ag, Door), S)) \leftarrow poss(unlock(Ag, Door), S)$.
Tür unverschließbar: $unlocked(Door, do(A, S)) \leftarrow unlocked(Door, S) \wedge poss(A, S)$.

Beispiel 6.3.3 (Roboter färbt alles in seiner Umgebung)
Allgemeine Axiome:

$poss(paint_everything(Robot, Color), S).$

$color(Obj, Color, do(paintevery_thing(Robot, Color), S)) \leftarrow$
$at(Robot, Loc, S) \wedge$
$at(Obj, Loc, S).$

Frame Axiom:

$color(Obj, Color, do(A, S)) \leftarrow$
$poss(A, S) \wedge$
$\neg paint_everything_action(A) \wedge$
$color(Obj, Color, S).$

Anhang A

Beispieldateien

A.1 Prolog Einführung — Familie.pl

```
maennlich(fritz).
maennlich(steffen).
maennlich(paul).
maennlich(josef).
maennlich(tom).

weiblich(sina).
weiblich(karin).
weiblich(lisa).
weiblich(maria).
weiblich(jerry).

vater(steffen,paul).
vater(steffen,lisa).
vater(fritz,karin).
vater(paul,maria).
vater(paul,josef).

mutter(sina,paul).
mutter(sina,lisa).
mutter(karin,maria).
mutter(karin,josef).
mutter(lisa,tom).
mutter(lisa,jerry).

elternteil(E,Kind) :- vater(E,Kind).
elternteil(E,Kind) :- mutter(E,Kind).

bruder(X,Y) :-
        maennlich(X),
        elternteil(E,X),
        elternteil(E,Y), X \== Y.
```

```
schwester(X,Y) :-
        weiblich(X),
        elternteil(E,X),
        elternteil(E,Y), X \== Y.

geschwister(X,Y):-schwester(X,Y).
geschwister(X,Y):-bruder(X,Y).

vorfahre(X,Y) :- elternteil(X,Y).
vorfahre(X,Y) :- elternteil(X,Z), vorfahre(Z,Y).

onkel(X,Y):- maennlich(X),geschwister(X,Z),elternteil(Z,Y).
onkel2(X,Y):- bruder(X,Z),elternteil(Z,Y).

tante(X,Y):- schwester(X,Z),elternteil(Z,Y).
```

A.2 CSP — Send-More-Money Problem

```
select(X, [X|R], R ).
select(X, [A|L], [A|R]) :- select(X, L, R).

carrier(X) :- member(X,[0,1]).

addition(D1,D2,D3,S,C) :-
    Sum is (D1 + D2 + D3),
    S is Sum mod 10,
    C is Sum // 10.

loesung([S,E,N,D,M,O,R,Y]) :-
    select(S,[0,1,2,3,4,5,6,7,8,9],RestS),
    select(E,RestS,RestE),
    select(N,RestE,RestN),
    select(D,RestN,RestD),
    select(M,RestD,RestM),
    select(O,RestM,RestO),
    select(R,RestO,RestR),
    select(Y,RestR,RestY),

    carrier(C1),
    carrier(C2),
    carrier(C3),

    M=1,
    S>0,

    addition(D,E, 0,Y,C1),
    addition(N,R,C1,E,C2),
    addition(E,O,C2,N,C3),
    addition(S,M,C3,O, M).
```

Literaturverzeichnis

[1] SWI-Prolog Homepage http://www.swi-prolog.org

[2] Eclipse-Prolog für wissenschaftliche Zwecke mit Lizenz
frei verfügbar unter http://www.icparc.ic.ac.uk/eclipse/

[3] Tom M. Mitchell: Machine Learning, McGraw Hill 1997

[4] Thomas Linke: Einführung Prolog, Universität Potsdam 2000

[5] Poole, Mackworth, Goebel: Computational Intelligence: A Logical Approach, Oxford University Press 1997

[6] Russel, Stuart und Norvig, Peter: Artificial Intelligence: A Modern Approach, Prentice Hall 1995

[7] Nilsson, Nils: Artificial Intelligence: A New Synthesis, Morgan Kaufmann 1998

[8] Dachsbacher: Schnellster im Ziel: Der A* Algorithmus, PC Magazin 08/01

Index

A* Algorithmus, 20
Aktionsmodellierung, 34
Arc-Consistency, 23

Best-first Search, 20
Beweis, 13
Beweisprozedur, 13
Bias, 25
Bottom-Up Beweisprozedur
  NAF, 31
  Wissenbasis, 13
Breitensuche, 18

Clark Normal Form, 30
Clark's Completion, 30
Complete Knowledge Assumption (CKA), 30
Constraint Satisfaction Problems (CSP), 21
CSP
  Backtracking, 22
  Konsistenz, 22
  Wertebereich, 21

Domain Konsistenz, 22
Domain Splitting, 24

Entropie, 27
Entscheidungsbäume, 26
Entscheidungssysteme, 11
Ersetzung von Variablen, 15

Fixpunkt, 13
Frame Axiom, 35

Gleichheit, 29
Gleichheitsaxiome, 29
Graph, 17
Graphensuche, 17

Handlungsplanung, 33
Herleitung
  Antwort, 14
Heuristische Suche, 19
Heuristische Tiefensuche, 20
Hill Climbing, 24
Horn Klausel, 32

ID3-Algorithmus, 27
Information Gain, 27
Integrity Constraints, 32
Iterative Tiefensuche, 19

Kantenkonsistenz, 23
Konflikte, 32

Lernen, 25
Logische Konsequenz, 12

Minimales Modell, 13
Modell, 12

Negation as Failure (NAF), 31
Neuronale Netze, 28

Pfad, 17
Prolog
  Arithmetik, 7
  Atom, 6
  Cut, 8
  Erweiterungen, 7
  Fakten, 6
  Input/ Output, 8
  Klassifikation von Termen, 8
  Klauseln, 5
  Listen, 7
  Matching, 6
  Operationen, 7
  Regel, 6
  Termanalyse, 8
  Variablen, 5

Randbedingungsprobleme, 21
RRS, 11

Semantik, 11
Situationskalkül, 35
SLD Resolution, 14
Startknoten, 17
STRIPS Repräsentation, 34

Suchen, 17
Suchgraphen, 17
Suchstrategie, 17

Tiefensuche, 18
Top-Town Beweisprozedur, 14

Ungleichheit, 30
Unifikator, 15
Unique Names Assumption, 29

Zeitrepräsentation, 33
Zielknoten, 17