Fakultät Informatik/Mathematik

Lehrstuhl für Künstliche Intelligenz / Theoretische Informatik

HOCHSCHULE FÜR
TECHNIK UND WIRTSCHAFT
DRESDEN
UNIVERSITY OF APPLIED SCIENCES

Vorlesungsmitschrift

KÜNSTLICHE INTELLIGENZ

Mitschrift von

Falk-Jonatan Strube

Vorlesung von

Prof. Dr. Boris Hollas

7. Juli 2017

INHALTSVERZEICHNIS

1 Prädikatenlogik **5**
1.1 Syntax der Prädikatenlogik 5
1.2 Semantik der Prädikatenlogik (informal) 6
1.2.1 Rechenregeln 7
1.2.2 Pränexform 8
1.2.3 Hornformeln 8

2 Prolog-Programmierung **10**
2.1 Syntax 10
2.1.1 Unterschied zwischen Prolog und imperativen Programmiersprachen . . 11
2.1.2 Existenzquantoren in Prolog durch Skolemisierung 11
2.1.3 Weiter Syntax 11
2.2 Auswertstrategie in Prolog 12
2.2.1 Tiefensuche 12
2.2.2 Breitensuche 13
2.2.3 Unifizierbarkeit 13
2.2.4 Backtracking 14
2.3 Darstellung von Relationen in Prolog 16
2.3.1 Transitive Hülle 16

3 Computeralgebra **18**
3.1 Arithmetik in Prolog 18
3.1.1 Vergleichsoperatoren 18
3.1.2 Anwendungen 19
3.1.2.1 Symbolisches Differenzieren 19
3.1.2.2 Vereinfachen von arithmetischen Ausdrücken 19
3.2 Listen 20
3.2.1 Suche (member) 20
3.2.2 Konkatenieren (append) 21

4 Sprachverarbeitung **22**
4.1 Genus-Kasus-Kongruenz 23
4.1.1 Wertigkeit von Verben 23

5 Problemlösen durch Suchen **25**
5.1 Uninformierte Suche 25
5.1.1 Breitensuche 25
5.1.2 Problem der Breiten- und Tiefensuche 26
5.1.3 Tiefensuche 26
5.1.4 Vorteile/Nachteile Breiten- und Tiefensuche 26
5.1.5 Iterative Tiefensuche 27
5.1.5.1 Anwendung: Planungsproblem 28
5.2 Informierte Suche (heuristische Suche) 29
5.2.1 Gierige Suche 29

5.2.2 A*-Suche 29
5.2.2.1 Implementierung durch Liste (ineffizient) 30
5.2.2.2 Implementierung durch Min-Heap 30
5.2.2.3 Vor-/Nachteile 30
5.2.3 IDA*-Suche 31
5.2.4 Spiele mit Gegner 34
5.2.4.1 Minimax 34
5.2.4.2 Alpha-Beta-Algorithmus 35

6 Bayes'sche Netze **38**
6.1 Formeln und Begriffe 38
6.2 Grundlagen 38
6.3 Semantik von Bayes-Netzen 39
6.4 Komplexität der Inferenz in Bayes'schen Netzen 40
6.4.1 Ausweg der NP-Vollständigkeit der exakten Inferenz 41

VORBEMERKUNGEN

INHALT

1. Prädikatenlogik
2. Prolog-Programmmierung
3. Logisches Schließen
4. Sprachverstehen
5. Problemlösen durch Suche
6. Bayes'sche Netze

PRÜFUNGSVORLEISTUNG

Alle 1-2 Wochen werden Aufgaben gestellt, die innerhalb von 2 Wochen gelöst und vorgeführt werden müssen. Mindestens 4 dieser Aufgaben müssen erfolgreich gelöst werden.

PRÜFUNG

- selbstbeschriebenes Blatt
- Taschenrechner

LITERATUR

- Ertl: Grundkurs Künstliche Intelligenz
- Russell/Norvig: Künstliche Intelligenz

1 PRÄDIKATENLOGIK

Vorlesung 20.03.2017

Problem: Aussagenlogik ist wenig mächtig.
Aussagen, die sich nicht formulieren lassen:

- „Alle Vögel können fliegen.“
  (Nur formulierbar, wenn alle Vögel einzeln als Atomformeln beschrieben werden. Diese Formel der Aussagenlogik wäre dann allerdings sehr lang.)
- „Wenn X eine Katze ist, dann ist X ein Haustier.“
  (In der Aussagenlogik sind keine Variablen möglich.)
- „Für jedes Land gibt es eine Hauptstadt.“

1.1 SYNTAX DER PRÄDIKATENLOGIK

1.1-1 Definition Sei V eine Menge von Variablen, K eine Menge von Konstante und F eine Menge von Funktionssymbolen.

- Dann sind alle Variablen V und Konstanten in K Terme.
- Wenn $t_1, \dots, t_n$ Terme sind und f ein n-stelliges Funktionssymbol ist, dann ist auch $f(t_1, \dots, t_n)$ ein Term.

1.1-2 Beispiel $V = \{x, y, z\},\ K = \{a, b, c\},\ F = \{+, *\}$. Terme sind $x, a, x + a, x * (x + 1)$.

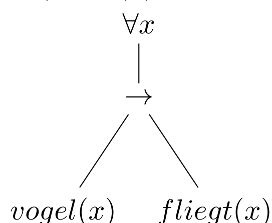
1.1-3 Definition Sei eine Menge von Prädikatensymbolen gegeben. Die Formeln der Prädikatenlogik sind induktiv definiert:

- Wenn $t_1, \dots, t_n$ Term und P ein Prädikatensymbol der Stelligkeit n ist,
  dann ist $P(t_1, \dots, t_n)$ eine Formel.
- Sind F, G Formeln, dann auch $F \wedge G,\ F \vee G,\ \neg F,\ (F)$.
- Wenn x eine Variable und F eine Formel ist, dann sind auch $\forall x\, F,\ \exists x\, F$ Formeln.

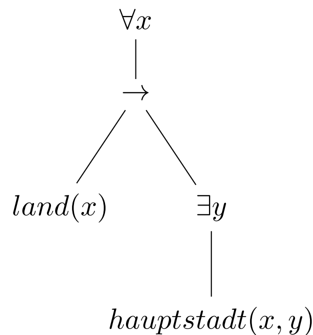
1.1-4 Bemerkung Wie in der Aussagenlogik definiert sich $\rightarrow,\ \leftrightarrow$.
Vorrang der Operatoren: $\neg;\ \forall,\ \exists;\ \wedge,\ \vee;\ \rightarrow,\ \leftrightarrow$

1.1-5 Beispiel

- $\forall x\,(vogel(x) \rightarrow fliegt(x))$



- $\forall x\, (katze(x) \rightarrow haustier(x))$
- $\forall x\, (land(x) \rightarrow \exists y\, hauptstadt(x, y))$



- $katz(feli)$ mit $feli \in K$

1.2 SEMANTIK DER PRÄDIKATENLOGIK (INFORMAL)

Wir betrachten zunächst eine (unwichtige) Menge U (Universum), die alle zu betrachtenden Objekte (Vögel, Katzen, Länder usw.) enthält. Davon betrachten wir Teilmengen, z.B. die Menge aller Vögel, Länder usw. sowie mehrstellige Relationen auf U (z.B. Hauptstadt, verheiratet).

Beispiel: $Hauptstadt = \{(Berlin,\ Deutschland), (Paris,\ Frankreich)\} \subseteq \underbrace{U \times U}_{oder:\ Stadt \times Land}$

Den Prädikatensymbolen müssen Prädikate (Relationen) zugeordnet werden.

Beispiel: Ordnen dem Prädikatensymbol $vogel$ das Prädikat „Menge aller Vögel" zu.

Dadurch ergibt sich der Wahrheitswert einer Formel:

- Die Formel $P(t_1, \ldots, t_n)$ ist wahr genau dann wenn $(t_1, \ldots, t_n) \in P$.
- Die Wahrheit von $F \wedge G,\ F \vee G,\ \neg F$ ist wie in der Aussagenlogik definiert.
- $\exists x\, F$ ist wahr genau dann wenn es ein x aus dem Universum gibt, mit dem F wahr ist.
- $\forall x\, F$ ist wahr genau dann wenn F für alle x aus dem Universum wahr ist.

1.2-1 Beispiel Sei $vogel = \{Amsel, Drossel, Fink, Star\}$
und $fliegt = vogel \cup \{Maikaefer, A380\}$

- $vogel(Amsel)$ ist wahr.
- $vogel(Maikaefer)$ ist falsch.
- $\exists x\, vogel(x)$ ist wahr.
- $\forall x\, vogel(x)$ ist falsch.
- $\forall x\, (vogel(x) \rightarrow fliegt(x))$ ist wahr.

1.2-2 Beispiel Sei $land = \{Deutschland, Frankreich\}$ und
$hauptstadt = \{(Berlin, Deutschland), (Paris, Frankreich)\}$.
Für jedes Land gibt es eine Hauptstadt:

$$\forall x\, (land(x) \rightarrow \exists y\, hauptstadt(y, x))$$

Vorlesung 27.03.2017

1.2-3 Beispiel Seien die Relationen $Katze$ (Menge aller Hauskatzen) sowie $Haustier$ (Menge aller Haustiere) gegeben.
Jede Katze ist ein Haustier:

$$\forall x\,(Katze(x) \to Haustier(x))$$

1.2-4 Beispiel Seien $Mensch,\ Mann,\ Frau$ die Menge aller Menschen, Männer und Frauen.

- Jeder Mensch ist Mann oder Frau:

  $$\forall x\,(Mensch(x) \to Mann(x) \lor Frau(x))$$

  Anmerkung: wichtige Unterscheidung:
  $Mensch(x) \to Mann(x) \lor Frau(x)$ liefert falsch zurück, wenn x bspw. ein Stuhl – das ist logisch richtig und
  $Mensch(x) \land (Mann(x) \lor Frau(x))$ sagt, dass jedes x entweder Mann oder Frau sein muss… ein Stuhl bspw. auch – das ist falsch!

- Jeder Mann und jede Frau ist ein Mensch:

  $$\forall x\,(Mann(x) \lor Frau(x) \to Mensch(x))$$

  Anmerkung: Obwohl im Satz „und“ steht, steht in der Prädikatenlogik das „oder“. Das „und“ ist in diesem Sinne gemeint:
  $\forall x\,(Mann(x) \to Mensch(x)) \land \forall x\,(Frau(x) \to Mensch(x))$

- Kein Mensch ist gleichzeitig Mann und Frau:

  $$\forall x\,(Mensch(x) \to \neg(Mann(x) \land Frau(x)))$$

1.2.1 RECHENREGELN

1.2-5 Definition

- $\neg\forall x\, F \equiv \exists x\, \neg F$
- $\neg\exists x\, F \equiv \forall x\, \neg F$
- Für $Q \in \{\forall, \exists\}$ und $\circ \in \{\land, \lor\}$ gilt:
  $(QxF) \circ G \equiv Qx\,(F \circ G)$, falls x in G nicht frei vorkommt.
- $\forall x\, F \land \forall x\, G \equiv \forall x(F \land G)$
- $\exists x\, F \lor \exists x\, G \equiv \exists x\,(F \lor G)$
- $\forall x\, \forall y\, F \equiv \forall y\, \forall x\, F$
- $\exists x\, \exists y\, F \equiv \exists y\, \exists x\, F$

1.2-6 Beispiel $(\exists x\, Katze(x)) \lor Vogel(y) \equiv \exists x\,(Katze(x) \lor Vogel(y))$

1.2.2 PRÄNEXFORM

1.2-7 Definition Eine Aussage F ist in BEREINIGTER PRÄNEXFORM, wenn

$$F = Q_1 y_1 \ldots Q_n y_n G$$

wobei $Q_i \in \{\forall, \exists\}$, G keine Quantoren enthält und $y_1, \ldots, y_n$ paarweise verschieden sind.

Für jede Aussage F gibt es eine äquivalente Formel im bereinigter Präexform.

1.2-8 Beispiel

$$\begin{aligned}
&\forall x\, (land(x) \rightarrow \exists y\, hauptstadt(y, x)) \\
\equiv\ &\forall x\, (\neg land(x) \vee \exists y\, hauptstadt(y, x)) \\
\equiv\ &\forall x\, (\exists y\, (\neg land(x) \vee hauptstadt(y, x))) \\
\equiv\ &\forall x\, \exists y\, (land(x) \rightarrow hauptstadt(y, x))
\end{aligned}$$

1.2-9 Beispiel Nicht jeder Vogel fliegt:

$$\begin{aligned}
&\neg \forall x\, (Vogel(x) \rightarrow fliegt(x)) \\
\equiv\ &\forall x\, (\neg Vogel(x) \vee fliegt(x)) \\
\equiv\ &\exists x\, \neg(\neg Vogel(x) \vee fliegt(x)) \\
\equiv\ &\exists x\, (Vogel(x) \wedge \neg fliegt(x))
\end{aligned}$$

(Nicht jeder Vogel fliegt. Durch Umformung erfährt man: Es muss einen Vogel geben, der nicht fliegt)
Achtung: Diese Formel sollte nicht wieder in die Implikation umgeformt werden, da sie sonst auch für bspw. einen Stuhl gültig wäre, was nicht gewollt ist: $\exists x(\neg Vogel(x) \rightarrow fliegt(x))$ (Stuhl ist kein Vogel, also fliegt er).

1.2.3 HORNFORMELN

1.2-10 Definition

- Ein LITERAL ist eine Atomformel oder eine negierte Atomformel. Eine nicht negierte Atomformel nennt man POSITIV (auch: Fakt), eine negierte NEGATIV.
- Eine Formel heißt HORNKLAUSEL, wenn sie eine $\vee$-Verknüpfung von Literalen ist, von denen höchstens eins positiv ist.
- Eine Hornformel ist eine $\wedge$-Verknüpfung von Hornklauseln.

1.2-11 Beispiel

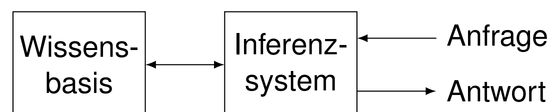
- Literale: $P(x, y),\ \neg P(x, y),\ \neg Q(z * x + y)$
- Hornklauseln: $\neg P(x, y) \vee Q(x),\ \neg P(x, y) \vee \neg Q(x),\ Q(x)$

1.2-12 Wichtiger Spezialfall Hornklauseln mit mindestens zwei Literalen und genau einem positiven Literal. Diese lassen sich als Folgerung darstellen, da

$$\begin{aligned} &\neg A_1 \vee \dots \vee A_{n-1} \vee A_n \\ \equiv\ &\neg(A_1 \wedge \dots \wedge A_{n-1}) \vee A_n \\ \equiv\ &A_1 \wedge \dots \wedge A_{n-1} \to A_n \end{aligned}$$

Mit Hilfe von Hornklauseln lassen sich Regeln formulieren, z.B. $\forall x\,(Katz(x) \to Haustier(x))$ (die Klammer lässt sich, wie im Spezialfall gesehen, umformen [dieses Mal von der Folgerung zurück), ist sie eine Hornformel]).
Wenn diese Regeln mit $\wedge$ verknüpft werden, können diese zusammen mit Fakten (einzelnes positives Literal) als Wissensbasis betrachtet werden.
Die Wissenbasis ist dann eine Hornformel.

1.2-13 Beispiel
$katze(feli) \wedge \forall x\,(katze(x) \to haustier(x))$ ist eine Hornformel, die als einfache Wissensbasis betrachtet werden kann. Daraus lässt sich die Aussage dass Feli ein Haustier ist schließen, obwohl es nicht explizit dort steht. Das ist die Aufgabe eines Expertensystems.
Grundlegender Aufbau eines Expertensystems:



2 PROLOG-PROGRAMMIERUNG

Prolog: Programming in Logic
Prolog-Programme sind im Wesentlichen Hornformeln, bei denen alle Variablen allquantisiert sind. Jede Klausel ist dabei eine Zeile eines Prolog-Programmes.

2.0-1 Beispiel

```
katze(reni).
haustier(X):-katze(X).
```

Dieses Prolog-Programm stellt die Hornformel $katze(reni) \wedge \forall x\,(katze(x) \rightarrow haustier(x))$ dar.
Anfrage an den Prolog-Interpreter:

```
?-haustier(reni).
true.
```

2.1 SYNTAX

Prädikat: Wort in Kleinbuchstaben.
Konstante: Wort in Kleinbuchstaben.
(Bestimmung ob Prädikat oder Konstante erfolgt über die Position des Wortes: Konstanten treten immer in der Klammer von Prädikaten auf)
Variable: Wort, das mit einem Großbuchstaben beginnt.
„ $\leftarrow$ “: „ `:-` “ (wird impliziert von).
„ $\wedge$ “: „ `,` “.
Jede Klausel wird mit „ `.` “ abgeschlossen. Alle Klauseln sind implizit mit $\wedge$ verknüpft.

$\vee$-VERKNÜPFUNG

Vorlesung 03.04.2017

Die Formel $A \vee B \rightarrow C$ ist wegen $A \vee B \rightarrow C \equiv (\neg A \wedge \neg B) \vee C$ keine Hornklausel. Da jedoch $(\neg A \wedge \neg B) \vee C \equiv (\neg A \vee C) \wedge (\neg B \vee C) \equiv (A \rightarrow C) \wedge (B \rightarrow C)$, lässt sich $A \vee B \rightarrow C$ als Hornformel darstellen.
Daher gibt es in Prolog den $\vee$-Operator „ `;` “.

2.1-1 Beispiel Die Formel $\forall x\,(katze(x) \vee kater(x) \rightarrow tier(x))$ kann in Prolog als Hornformel dargestellt werden durch:

```
tier(X) :- katze(X).
tier(X) :- kater(X).
```

Die skann mit dem $\vee$-Operator abgekürzt werden durch `tier(X):- katze(X); kater(X).`
Die Variable `X` ist implizit allquantisiert (alle Variablen sind in Prolog allquantifiziert).

2.1.1 UNTERSCHIED ZWISCHEN PROLOG UND IMPERATIVEN PROGRAMMIERSPRACHEN

Eine Variable kann sowohl Eingabe als auch Ausgabe sein. In einer imperativen Programmiersprache lässt sich nachbilden, dass eine Variable entweder Eingabe oder Ausgabe ist.
Prolog:

```
katze(reni).
katze(mimi).
kater(momo).
tier(X) :- katze(X) ; kater(X).
```

Nachbildung in Java:

```
Boolean katze(String x) {
  return x.equals("reni") || x.equals("mimi");
}

Boolean katze(String x) {
  return x.equals("momo");
}

Boolean tier(String x) {
  return katze(x) || kater(x);
}
```

Damit können wir die Anfrage für `tier("momo")` stellen, aber keine Anfrage `tier(X)` wie in Prolog, die alle `X` mit `tier(X)` wahr liefert.

2.1.2 EXISTENZQUANTOREN IN PROLOG DURCH SKOLEMISIERUNG

In Prolog sind alle vorkommenden Variablen allquantisiert. Daher können existenzquantisierte Variablen nicht unmittelbar dargestellt werden. Existenzquantoren können jedoch durch Skolemisierung eliminiert werden. Einfacher Spezialfall: Einführung einer Skolemkonstante.
Eine Formel der Form

$$\exists x \; P(X)$$

kann erfüllbarkeitsäquivalent umgeformt werden zu

$$P(a)$$

wobei a eine noch nicht verwendete Konstante ist (Skolemkonstante).
So wird die Abfrage $\exists x \; katze(X)$ beispielsweise durch `katze(reni)` umgesetzt.

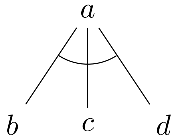
2.1.3 WEITER SYNTAX

`test(X,_)`	_ ist anonyme Variable / Platzhalter, falls Variable im Kontext nicht relevant ist.
`not(X=Y)` oder `X\=Y`	Prüfung, ob `X` ungleich (im Sinne von „nicht unifizierbar“) `Y`.
`\+ (...)`	Aussage ... wird negiert (ähnlich wie die Prüfung `not(...)`).

2.2 AUSWERTSTRATEGIE IN PROLOG

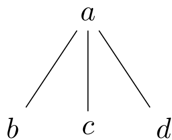
Die Struktur eines Prolog-Programmes lässt sich durch einen Und-Oder-Baum darstellen:

Und-Verknüpfung: `a(X) :- b(X) , c(X) , d(X).`



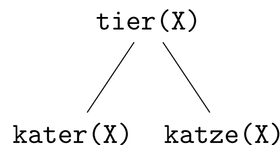
Für die Anfrage `a(X)` werden die Zeilziele `b(X)`, `c(X)`, `d(X)` erzeugt, die alle wahr sein müssen.

Oder-Verknüfung: `a(X) :- b(X) ; c(X) ; d(X).`

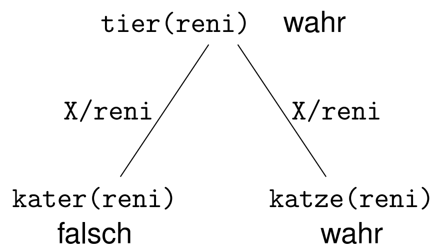


Hier muss eines der Teilziele wahr sein.

2.2-1 Beispiel Programm von oben:



Bei der Anfrage `tier(X)` wird die Variable `X` ersetzt durch die Konstante `reni` und die Variablen auf tieferen Ebenen werden entsprechend ersetzt.
Schreibweise: `X/reni` (`X` wird unifiziert mit `reni`)



Allgemein kann ein Und-Oder-Baum aus beliebig vielen Und- oder Oder-Verknüpfungen bestehen.
Dieser Baum muss systematisch durchsucht werden, um einen Wahrheitswert für die Wurzel zu bestimmen.
Mögliche Strategien: Tiefensuche, Breitensuche.

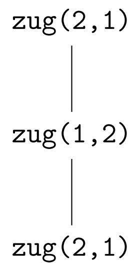
2.2.1 TIEFENSUCHE

Problem bei der Tiefensuche:

2.2-2 Beispiel

```
zug(X,Y) :- zug (Y,X).
zug(1,2).
```

Anfrage `zug(2,1)`:

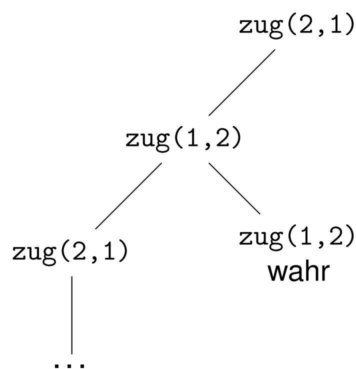


Der Und-Oder-Baum ist unendlich und eine Tiefensuche, die den ersten Zweig (links) verfolgt, findet keine Lösung.
Das gleiche Problem tritt bei einer entsprechenden Implementierung in Java auf:

```
Boolean zug(int x, int y) {
  return zug(y,x) || (x==1 && y==2);
}
```

2.2.2 BREITENSUCHE

Die Breitensuche würde hier eine Lösung finden:



Das heißt, die Breitensuche liefert eine Lösung, die Tiefensuche unter Umständen nicht.

Lösung: Die Abbruchbedingung der Rekursion muss die erste Klausel sein damit die Rekursion nicht endlos läuft bzw. die Tiefensuche die Abzweigung nimmt, die zu einer Lösung führt:

```
zug(1,2).
zug(X,Y) :- zug(Y,X).
```

2.2.3 UNIFIZIERBARKEIT

Vorlesung 10.04.2017

2.2-3 Definition Zwei Atome P, Q heißen UNIFIZIERBAR, wenn es eine Ersetzung der in P, Q vorkommenden Variablen gibt, so dass damit $P \equiv Q$.

2.2-4 Beispiel Mit $a,\ b,\ c\ldots$ Konstanten, $x,\ y,\ z\ldots$ Variablen.

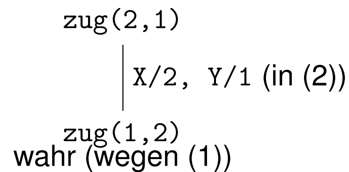
- $P(x,a),\ P(a,a)$ sind unifizierbar durch x/a.
- $Q(a,x,y),\ Q(z,b,c)$ sind unifizierbar durch $z/a,\ x/b,\ y/c$.
- $P(x,x)\ P(a,b)$ sind nicht unifizierbar.

→ Auswertstrategie von Prolog:
Für ein Ziel P wird das Prolog-Programm von oben nach unten durchsucht, bis eine linke Seite einer Klausel (das, was hinter der Implikation steht bzw. in Prolog vor `:-`) mit P unifiziert. Indem die rechte Seite der Klausel ebenso ersetzt wird, werden neue Teilziele erzeugt.

2.2-5 Beispiel

```
zug(1,2).              % (1)
zug(X,Y):- zug(Y,X).   % (2)
```

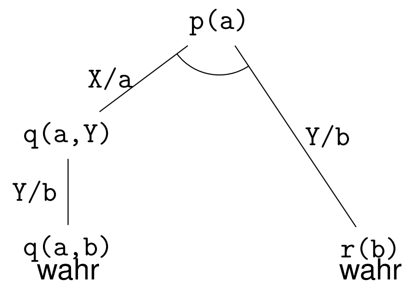
Für die Anfrage `zug(2,1)` wird folgender Suchbaum erzeugt:



Wenn beispielsweise

```
q(X,Y):- (a,b).
r(Y):- (b).
p(X):- q(X,Y) , r(Y).
```

dann:



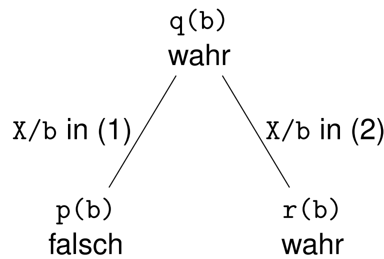
2.2.4 BACKTRACKING

Wenn das neue Teilziel `false` liefert, wird ein Backtracking ausgeführt und eine andere Alternative gesucht.

2.2-6 Beispiel

```
q(X):- p(X).   % (1)
q(X):- r(X).   % (2)
p(a).
r(b).
```

Anfrage: `q(b)`
Suchbaum:

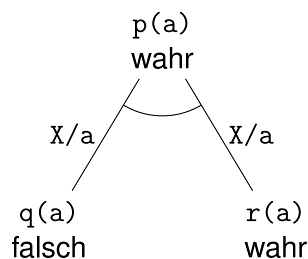


Bei einer Und-Verknüpfung im Suchbaum wird zuerst der linke Teilbaum durchsucht, dann die rechten Teilbäume, wobei jeweils die gleichen Ersetzungen von Variablen vorgenommen werden.

2.2-7 Beispiel

```
p(X):- q(X), r(X).
```

Anfrage: `p(X)` Suchbaum:

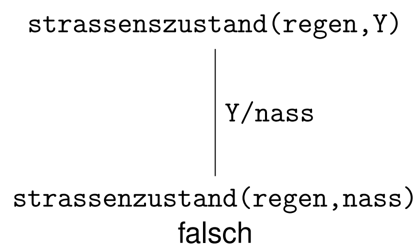


Die Auswertstrategie von oben nach unten lässt sich nutzen, um ein `if-else` zu implementieren.

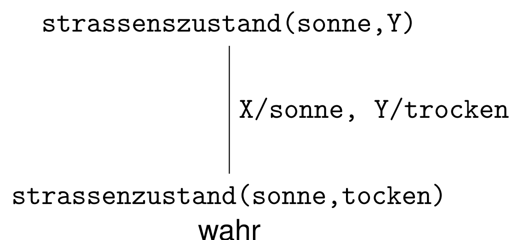
2.2-8 Beispiel

```
strassenzustand(regen,nass).
strassenzustand(X,trocken).
```

Anfrage 1:



Anfrage 2:



Um zu verhindern, dass auch `strassenzustand(regen, trocken)` zu wahr auswertet, kann die zweite Zeile zu `strassenzustand(X,trocken):- X \== regen` geändert werden.

2.3 DARSTELLUNG VON RELATIONEN IN PROLOG

Eine endliche Relation kann durch Aufzählung dargestellt werden. Reflexive, transitive und symmetrische Relationen werden durch entsprechende Regeln dargestellt.

- Wenn `p(X,Y)` eine Relation ist, kann deren symmetrische Hülle dargestellt werden durch `q(X,Y):- p(X,Y); p(Y,X).`
- Die reflexive Hülle von `p` wird erzeugt, indem die Regel `p(X,X)` hinzugefügt wird.
- Naiver Versuch Transitivität:

```
p(X,Y).
q(X,Z):- p(X,Z); p(X,Y), p(Y,Z).\\
% Achtung: Funktioniert nur für Wege der Länge 2!
```

  $\rightarrow$ Berechnung der transitiven Hülle einer Relation.

2.3.1 TRANSITIVE HÜLLE

Sei `weg/2` eine binäre Relation, von der wir die transitive Hülle berechnen wollen. Es gilt:
Es gibt einen Weg von X nach Y genau dann wenn:

- $X = Y$ oder
- es gibt einen Knoten z und einen Weg von X nach Z und von Z nach Y.

Wir müssen daher die reflexive und transitive Hülle von `weg` berechnen.
Als Formel:

$$\forall X \; weg(X,X) \wedge$$
$$\forall X \; \forall Y \; ((\exists Z \; weg(X,Z) \wedge weg(Z,Y)) \rightarrow weg(X,Y))$$

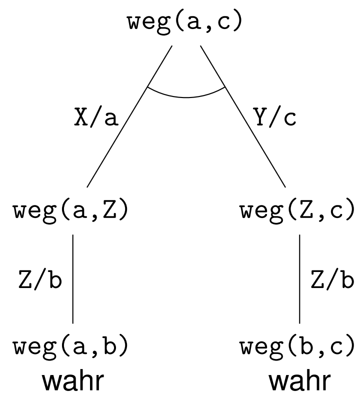
Um den Existenzquantor zu beseitigen formen wir die zweite Klausel um:

$$\begin{aligned} &\forall X \; \forall Y \; (\forall Z \; \neg weg(X,Z) \vee \neg weg(Z,Y) \vee weg(X,Y)) \\ \equiv &\forall X \; \forall Y \; \forall Z \; (\neg weg(X,Z) \vee \neg weg(Z,Y) \vee weg(X,Y)) \\ \equiv &\forall X \; \forall Y \; \forall Z \; (weg(X,Z) \wedge weg(Z,Y) \rightarrow weg(X,Y)) \end{aligned}$$

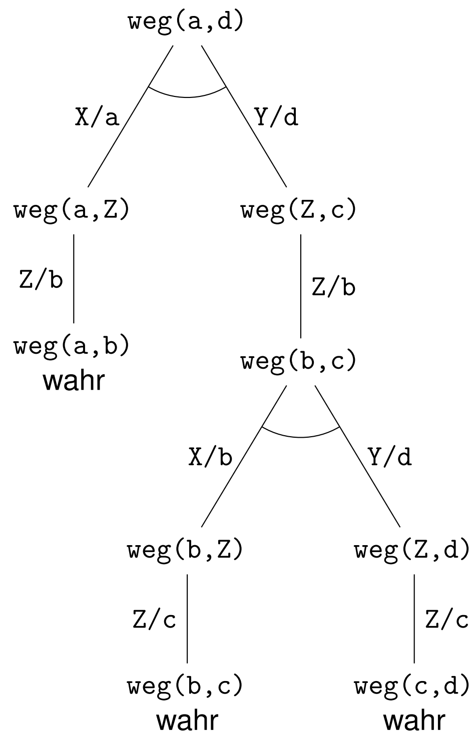
In Prolog bspw.:

```
weg(a,b).
weg(b,c).
weg(c,d).
weg(X,X).
weg(X,Y):- weg(X,Z), weg(Z,Y).
```

Anfrage 1:



Anfrage 2:



3 COMPUTERALGEBRA

3.1 ARITHMETIK IN PROLOG

Vorlesung
24.04.2017

Arithmetische Ausdrücke werden mit dem Operator `is` ausgewertet. Dabei muss die rechte Seite instantiiert sein.

3.1-1 Beispiel

```
?- X is 3+4.
X=7.
?- 10 is 3+4.
false.
?- 2 is 1+X
Fehler!!!
f(X,Y):- Y=3*X+1  % Falsch! = ist Unifikationsoperator, keine
    Funktionszuweisung
f(1,Y).
Y=3*1+1.
```

3.1.1 VERGLEICHSOPERATOREN

Operator	Bedeutung	Beispiel
`==`	identisch	`p(X) == p(X)`
`\==`	nicht identisch	`p(X) \== p(Y)`
`=`	unifizierbar	`p(X) = p(Y)`
`\=`	nicht unifizierbar	`p(X) \= q(X)`
`=:=`	arithmetisch gleich	`2 =:= 1+1`
`=\=`	arithmetisch ungleich	`3 =\= 1+1`

In Prädikaten können Operatorsymbole wie $+, -, *, /, \hat{}$ verwendet werden, um arithmetische Ausdrücke zu zerlegen. Prolog berücksichtigt dabei die Priorität der Operatoren.

3.1-2 Beispiel

```
?- x + 3*y + x*2 = A + B + C.
A=x, B=3*y, C=x*2.
?- 2*x + 3*y = A + B * C.
A=2*x, B=3, C=y
?- 2*x + y + z = A + B % nicht eindeutig!
```

3.1.2 ANWENDUNGEN

3.1.2.1 SYMBOLISCHES DIFFERENZIEREN

Ziel: Wir wollen ein Prädikat `diff/3` (3-Stelliger Operator) definieren, mit dem Polynome symbolisch differenziert werden können.

3.1-3 Beispiel

```
diff(3*x^2+5x-2 , x , D).
D = 6*x+5.
```

```
d(X,X,1).
d(C,X,0) :- atomic(C), C\==X.

?- d(x,x,D).
D = 1.
?- d(c,x,D).
D = 0.
?- d(x+y, x+y, D).
D = 1.

% weitere Ableitungsregel (Ableitung negativer Funktion):
d(-F,X,-DF) :- d(F,X,DF).

?- d(-x,x,D).
D = -1.

% weitere Ableitungsregel (Ableitung mit führender Konstante):
d(C*F,X,C*DF) :- d(C,X,0), d(F,X,DF).

?- d(3*x,x,D).
D = 3*1.

% weitere Ableitungsregel (Kettenregel):
d(F+G,X,DF+DG) :- d(F,X,DF), d(G,X,DG).
d(F-G,X,DF-DG) :- d(F,X,DF), d(G,X,DG).

?- d(3*x+5,x,D).
D = 3*1+0.

% weiter Ableitungsregel (einfache Ableitung):
d(F^N,X,N*F^M):- number(N), M is N-1, d(F,X,DF).
```

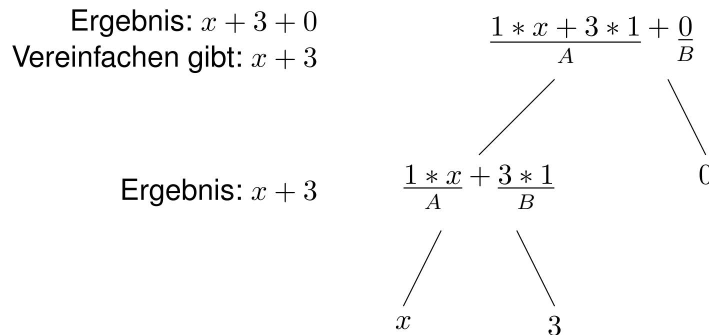
3.1.2.2 VEREINFACHEN VON ARITHMETISCHEN AUSDRÜCKEN

Idee: Wir definieren elementare Vereinfachungsregeln und vereinfachen rekursiv.
Elementare Vereinfachungsregeln:

```
s0(1*X,X).
s0(X*1,X).
s0(0+X,X).
```

```
s0(X+0,X).
%...
s0(X,X).  % Lösung zu folgendem Problem
```

Problem: Wenn ein Term bereits vereinfacht ist, liefert `s0 false`. Zum Beispiel `s0(x,X)`.
Lösung: Regel `s0(X,X).` am Ende hinzufügen (wenn keine der Regeln greift, ist Term schon vereinfacht).
Um Ausdrücke zu vereinfachen, die aus mehreren Termen bestehen, wenden wir die Vereinfachung rekursiv auf jeden Term an und vereinfachen das Ergebnis. Zum Beispiel: Ausdruck $1 * x + 3 * 1 + 0$



3.2 LISTEN

Vorlesung 08.05.2017

Listen sind induktiv definiert:

- `[]` ist die leere Liste
- Wenn `H` ein Element und T eine Liste sind, dann ist `[H|T]` eine Liste die aus dem Kopf `H` und der restlichen Liste `T` besteht.

Alternativ kann eine Liste in der Form `[x1, ..., xn]` aufgeschrieben werden.

3.2-1 Beispiel Listen sind:

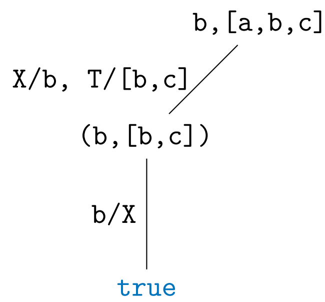
- `[a,b,c]`
- `[a|[b,c]]`
- `[a,b,c,1,2,[u,v]]`

3.2.1 SUCHE (MEMBER)

Als Beispiel für ein Prädikat auf einer Liste definieren wir das member-Prädikat:

```
member(X, [X|_]). % Element ist im Kopf, _ kann auch die leere
    Liste sein.
member(X, [_|T]):- member (X,T).  % Element in der Tail-Liste
    suchen.
```

3.2-2 Beispiel Suchbaum für die Anfrage `member(b,[a,b,c])`:



3.2.2 KONKATENIEREN (APPEND)

Weiteres Listenelement: `append/3`
`append(L1, L2, L3)` ist wahr genau dann, wenn die Verkettung der Listen L1, L2 und L3 gleich ist.

3.2-3 Beispiel

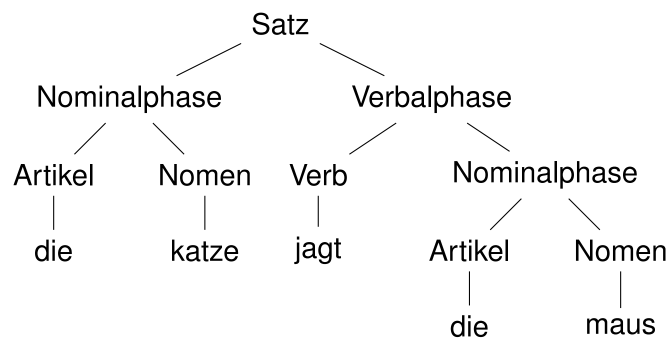
- `append([a,b,c], [d,e,f], L)` liefert `L=[a,b,c,d,e,f]`.
- `append([a,b,c], L, [a,b,c,d,e,f])` liefert `L=[d,e,f]`.
- `append(L1, L2, [a,b])` liefert:
  `L1=[], L2=[a,b];`
  `L1=[a], L2=[b];`
  `L1=[a,b], L2=[].`

4 SPRACHVERARBEITUNG

Sprache lässt sich durch Grammatiken beschreiben und analysieren. Einfache Grammatik für einen kleinen Teil der deutschen Sprache:

- Satz $\rightarrow$ Nominalphrase Verbalphrase
- Nominalphrase $\rightarrow$ Artikel Nomen
- Artikel $\rightarrow$ die | eine
- Nomen $\rightarrow$ katze | maus
- Verbalphrase $\rightarrow$ Verb Nominalphrase
- Verb $\rightarrow$ jagt | sieht

Damit lässt sich ableiten:



Zur Darstellung von Strings verwenden wir Listen, zum Beispiel `[die, katze, schläft]`. Achtung: Wörter müssen klein geschrieben oder in `"..."` eingeschlossen werden, zum Beispiel `["die", "Katze"]`.
Zur Darstellung der Grammatik verwenden wir einen Recursive Descent Parser (siehe Vorlesung TI). Idee dazu:
Die Grammatik-Prädikate haben die Form `p(In, Rest)`. Dabei ist `In` die Eingabe. Das Prädikat `p` versucht einen Präfix der Eingabe entsprechend der Grammatikregeln, die `p` implementiert, zu verarbeiten. Der Rest der Eingabe, den `p` nicht verarbeitet hat, wird an `Rest` gebunden und ggf. als Eingabe an ein weiteres Grammatikprädikat weitergeleitet.

4.0-1 Beispiel

```
% Recursive Descent Parser für L = {a^n b^n | n >= 0  }
% Anfrage: s([a,a,a,b,b,b], []).

s(In, Rest) :-  % S -> aSb
  match(a, In, R1),
  s(R1, R2),
  match(b, R2, Rest).
s(L, L).  % S -> epsilon

match(X, [X|Rest], Rest).
```

Beispiele:

```
?- s([a,a,a,b,b,b], []).
true.
?- s([a,a,a,b,b,b], R).
R = [b].
```

4.0-2 Beispiel

```
satz(In, Rest) :- nominal_phrase(In, R), verbal_phrase(R, Rest).
nominal_phrase(In, Rest) :- artikel(In, R), nomen(R, Rest).
verbal_phrase(In, Rest) :- verb(In, R), nominal_phrase(R, Rest).
artikel(In, Rest) :- match(eine, In, Rest); match(die, In, Rest).
verb(In, Rest) :- match(jagt, In, Rest); match(sieht, In, Rest).
nomen(In, Rest) :- match(katze, In, Rest); match(kater, In, Rest);
    match(maus, In, Rest).

match(X, [X|Rest], Rest).
```

4.1 GENUS-KASUS-KONGRUENZ

In der vorherigen Katze-Maus Grammatik können grammatikalisch falsche Sätze erzeugt werden, zum Beispiel `[die, kater, jagt, die maus]`.
Lösung: In der Nominalphrase müssen Artikel und Nomen in Genus und Kasus übereinstimmen. Wir führen Parameter `Genus` und `Kasus` in `nominal_phrase` ein, die an `artikel` und `nomen` weitergeleitet werden.

4.1.1 WERTIGKEIT VON VERBEN

Vorlesung 15.05.2017

Zwei wichtige Klassen von Verben sind

- intransitives Verben: Diese führen kein Objekt nach sich (z.B. laufen, schlafen)
- transitive Verben: Diese benötigen ein Objekt (z.B. (etwas) sehen, (etwas) fangen).

Das Verb bestimmt den Kasus des zugehörigen Objektes, z.B. jagen, fangen, sehen benötigen ein Akkusativobjekt (das Subjekt sieht wen oder was?).

4.1-1 Beispiel

```
satz(In, Rest) :-
  nominal_phrase(_, nom, In, R), % Subjekt (Nominativ, wer oder was
     ?)
  verbal_phrase(R, Rest).

nominal_phrase(Genus, Kasus, In, Rest) :-
  artikel(Genus, Kasus, In, R), nomen(Genus, R, Rest).

verbal_phrase(In, Rest) :-
  verb_intransitiv(In, Rest);
  verb_transitiv(In, R), nominal_phrase(_, akk, R, Rest). % Objekt
     (Akkusativ, wen oder was?)
```

```
artikel(m, nom, In, Rest) :- match(A, In, Rest), member(A, [ein,
    der]).
artikel(f, nom, In, Rest) :- match(A, In, Rest), member(A, [eine,
    die]).
artikel(m, akk, In, Rest) :- match(A, In, Rest), member(A, [einen,
    den]).
artikel(f, akk, In, Rest) :- match(A, In, Rest), member(A, [eine,
    die]).

verb_intransitiv(In, Rest) :- match(V, In, Rest), member(V, [jagt,
    schläft, rennt]).
verb_transitiv(In, Rest) :- match(V, In, Rest), member(V, [jagt,
    sieht]).

nomen(m, In, Rest) :- match(kater, In, Rest).
nomen(f, In, Rest) :- match(V, In, Rest), member(V, [katze, maus]).

match(X, [X|Rest], Rest).
```

5 PROBLEMLÖSEN DURCH SUCHEN

Viele Probleme der KI lassen sich auf eine systematische Suche in einem Wurzelbaum reduzieren.
Problem: Riesige Anzahl von Knoten in typischen Suchbäumen.

5.0-1 Beispiel Schachspiel, ca. 30 Möglichkeiten pro Halbzug (Zug einer Farbe). Bei 50 Halbzügen enthält der Suchbaum:

$$\sum_{d=0}^{50} 30^d = \frac{30^{51}-1}{30-1} \approx 7,4 \cdot 10^{73}\,\mathrm{Knoten}$$

Bei $10\,000$ Computern, die $10^9\,\mathrm{Knoten/s}$ erzeugen und durchsuchen können, würde das durchsuchen so lange dauern:

$$\frac{7,4 \cdot 10^{73}}{10\,000 \cdot 10^9}\,\mathrm{s} = 2,3 \cdot 10^{53}\,\mathrm{Jahre}$$

5.1 UNINFORMIERTE SUCHE

Bereits bekannt: Breiten- und Tiefensuche. Zur Implementierung in Prolog benötigen wir:

- `findall(X, P, L)`: Sucht alle `X`, für die `P` wahr ist und erzeugt daras die Liste `L`.
- `not(P)`: Ist wahr genau dann, wenn Prolog `P` nicht beweisen kann („Negation by failure“).

5.1.1 BREITENSUCHE

```
% Adjazenzrelation des ungerichteten Graphen (nicht effizient)
adj(X,Y) :- adj0(X,Y); adj0(Y,X).
adj0(X,Y) :- member((X,Y), [(1,2), (2,4), (2,5), (3,4), (3,6),
    (4,5)]).

goal(6).

% Breitensuche
bfs([H|T], Discovered) :-
  goal(H);
  findall(Node, (adj(H, Node), not(member(Node, Discovered))),
     NewNeighbors),
  append(T, NewNeighbors, Queue),
  append(Discovered, NewNeighbors, Dc),
  write('Queue: '), writeln(Queue), % zur Illustration
  bfs(Queue, Dc).
```

Starten der Breitensuche beim Knoten 1:

```
?- bfs([1], [1]).
Queue: [2]
Queue: [4,5]
Queue: [5,3]
Queue: [3]
Queue: [6]
true .
```

5.1.2 PROBLEM DER BREITEN- UND TIEFENSUCHE

Wenn alle bereits besuchten Knoten gespeichert werden: Exponentielle Laufzeit und exponentieller Speicherplatzbedarf (für die Discoverd-Liste) in der Tiefe des Baumes.

5.1.3 TIEFENSUCHE

Einfach: aus `append(T, NewNeighbors, Queue)` wird `append(NewNeighbors, T, Queue`.
Ohne expliziten Stack, Knoten auf aktuellen Pfad werden gespeichert (nicht alle Knoten wie bei Discovered → vermeidet exponentiellen Speicherplatzbedarf – funktioniert, da der zu durchsuchende Graph in der Regel ein Baum ist [Knoten könnten nur durch Schleifen doppelt besucht werden]):

```
% Adjazenzrelation des ungerichteten Graphen (nicht effizient)
adj(X,Y) :- adj0(X,Y); adj0(Y,X).
adj0(X,Y) :- member((X,Y), [(1,2), (2,4), (2,5), (3,4), (3,6),
    (4,5)]).

goal(6).

dfs3(Node, Path) :-
  goal(Node);
  adj(Node, NewNeighbor), not(member(NewNeighbor,Path)),
  write('Knoten: '), writeln(NewNeighbor), % zur Illustration
  dfs3(NewNeighbor, [NewNeighbor|Path]).

% Wie dfs3, gefundener ReturnPath wird zurückgegeben
dfs4(Node, Path, ReturnPath) :-
  goal(Node), reverse(Path, ReturnPath);
  adj(Node,NewNeighbor), not(member(NewNeighbor,Path)),
  dfs4(NewNeighbor, [NewNeighbor|Path], ReturnPath).
```

5.1.4 VORTEILE/NACHTEILE BREITEN- UND TIEFENSUCHE

Vorlesung 22.05.2017

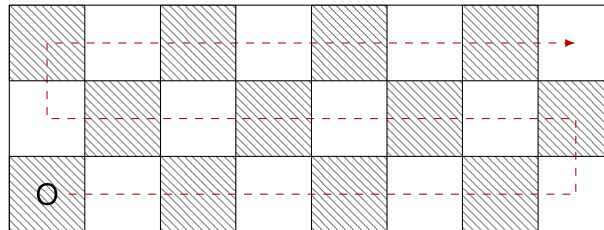
BREITENSUCHE

+ Liefert kürzesten Pfad, funktioniert auch für unendliche Graphen.
– Alle Knoten werden gespeichert.

TIEFENSUCHE

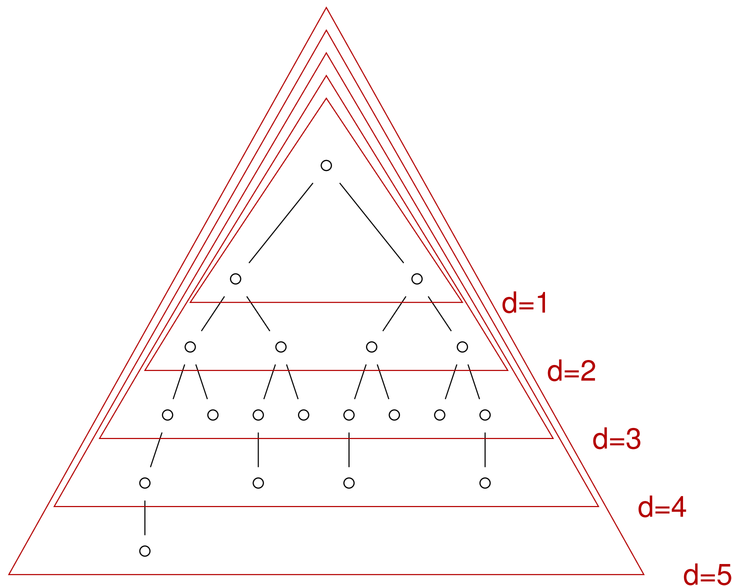
+ Nur die Knoten auf dem aktuellen Pfad werden gespeichert.

– Liefert nicht immer den kürzesten Pfad und funktioniert nicht für unendliche Graphen. Beispiel:



5.1.5 ITERATIVE TIEFENSUCHE

Wir verwenden eine Tiefensuche mit einer Tiefenschranke, die sukzessive erhöht wird, bis das Ziel gefunden ist.



Die iterative Tiefensuche besitzt damit alle Vorteile.
Zur Rechenzeit: Diese ist länger als bei der Breitensuche, da alle vorherigen Kanten nochmal besucht werden.
In einem Suchbaum mit Verzweigungsfaktor > 1 sind fast alle Knoten Blätter. Daher fällt auch die meiste Rechenzeit für das Durchsuchen der Blätter an. Durch eine genaue Rechnung lässt sich zeigen, dass die Laufzeit der iterativen Tiefensuche nur um einen kleinen Faktor höher ist als die der Tiefensuche.

```
% Node: aktueller Knoten
% Goal: Zielknoten
% Path: Liste der Knoten auf dem Pfad bis Node
% ReturnPath: Rückgabe, wenn ein Pfad zum Ziel gefunden wurde
dlDfs(Node, Goal, Path, DepthLimit, ReturnPath) :-
  Node = Goal, reverse(Path, ReturnPath);
  DepthLimit > 0,
  adj(Node,NewNeighbor), not(member(NewNeighbor,Path)),
```

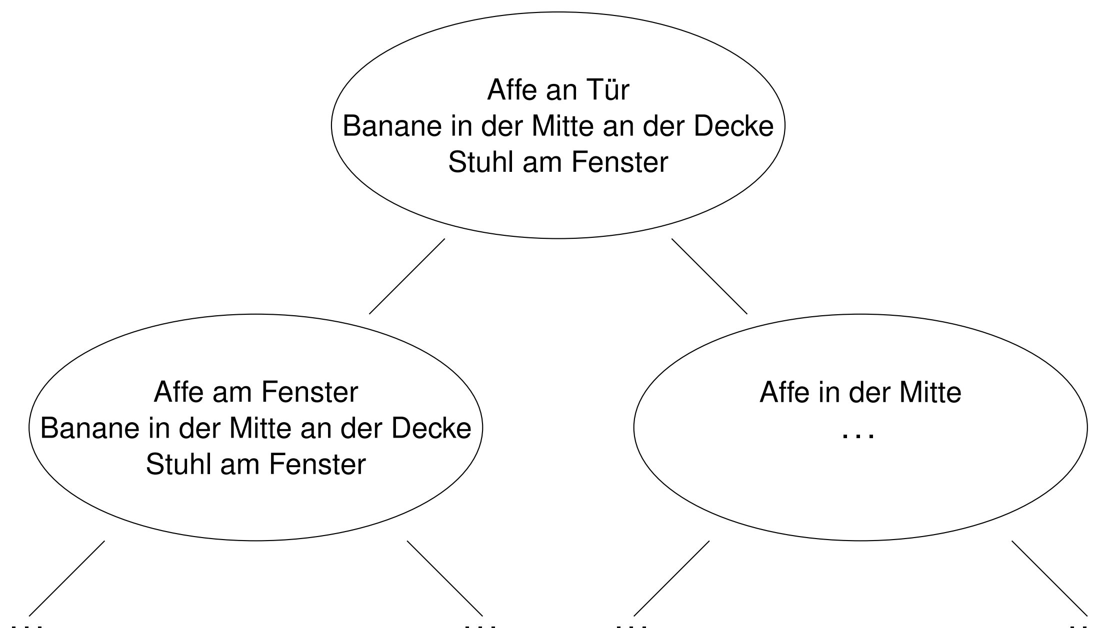
```
9   dlDfs(NewNeighbor, Goal, [NewNeighbor|Path], DepthLimit-1,
      ReturnPath).
10
11 idDfsLoop(Start, Goal, D, ReturnPath) :-
12   dlDfs(Start, Goal, [Start], D, ReturnPath);
13   % Wenn die Tiefensuche mit Schranke D nicht erfolgreich war, wird
       mit Schranke D+1 weitergesucht.
14   idDfsLoop(Start, Goal, D+1, ReturnPath).
15
16 idDfs(Start, Goal, ReturnPath) :- idDfsLoop(Start, Goal, 1,
     ReturnPath).
```

5.1.5.1 ANWENDUNG: PLANUNGSPROBLEM

Affe-Banane-Problem
Situation: Raum mit

- Affe an der Tür
- Banane an der Decke in der Mitte des Raumes
- Stuhl am Fenster

Ziel: Affe soll die Banane greifen.
Regeln: Affe kann laufen, den Stuhl verschieben, auf den Stuhl steigen, die Banane greifen, wenn er unter der Banane auf dem Stuhl steht.
Zugehöriger Graph, der das Suchproblem darstellt: Knoten sind Situationen, Kanten entsprechen den anwendbaren Regeln.




```
1 :- [idDfs]. % entspricht include
2
3 ort(X) :- member(X, [tuer, mitte, fenster]).
4 % Bedeutung der Listen: [Affe, Banane, Stuhl, Affe auf Stuhl]
5 adj0([A1,B,S,f], [A2,B,S,f]) :- ort(A1), ort(A2).   % laufen
```

```
6  adj0([A1,B,A1,f], [A2,B,A2,f]) :- ort(A1), ort(A2).   % Stuhl
      schieben
7  adj0([A,B,A,f], [A,B,A,t]).   % auf Stuhl steigen
8  goal([A,A,_,t]).    % Banane greifen
9
10 adj(X,Y) :- adj0(X,Y); adj0(Y,X).
11
12 solution(Path) :- Start = [tuer, mitte, fenster, f], goal(Goal),
      idDfs(Start,Goal,Path).
```

5.2 INFORMIERTE SUCHE (HEURISTISCHE SUCHE)

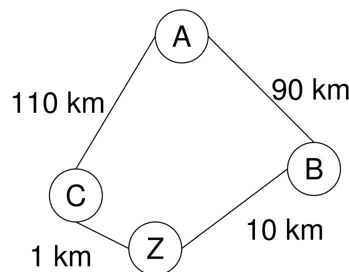
Vorlesung 29.05.2017

Ziel: Information über das Suchproblem nutzen, um gute Pfade zuerst zu verfolgen. Dabei wird eine Bewertungsfunktion für die Knoten verwendet.
Die heuristische Suche verwendet eine heuristische Bewertungsfunktion $f : V \to \mathbb{R}_0^+$. Für den Zielknoten v gilt $f(v) = 0$. Die Knoten mit der niedrigsten Bewertung werden zuerst verfolgt.

5.2.1 GIERIGE SUCHE

Verwendet in jedem Schritt den Knoten, der dem Ziel am nächsten liegt.

5.2-1 Beispiel Suche nach kürzestem Weg zu einem Ort $f(v) =$ Luftlinienentfernung zum Ziel



Die gierige Suche verfolgt den Weg A–C–Z (111 km). Dabei ist A–B–Z (100 km) kürzer.

5.2.2 A*-SUCHE

Die gierige Suche berücksichtigt nicht die Kosten, die bis zum Knoten v bereits entstanden sind. Wir führen daher eine Funktion g ein, die die Kosten vom Startknoten bis v angibt (also der bereits zurückgelegte Weg) und eine Funktion h, die die (verbleibenden) Kosten bis zum Ziel schätzt.
Daher definieren wir die heuristische Bewertungsfunktion f durch:

$$f(v) = g(v) + h(v)$$

Der A*-Algorithmus verwendet eine Bewertungsfunktion f, die die Summe der Kosten bis v und die geschätzten Kosten bis zum Ziel sind. Dabei muss h zulässig sein.

5.2-2 Definition Eine heuristische Kostenschätzfunktion h heißt ZULÄSSIG, wenn h die Kosten bis zum Ziel nie überschätzt.

5.2-3 Beispiel

- Die Luftlinienentfernung bis zum Ziel ist eine zulässige Kostenschätzfunktion.
- 0 (aber nicht nützlich, $\rightarrow$ gierige Suche).

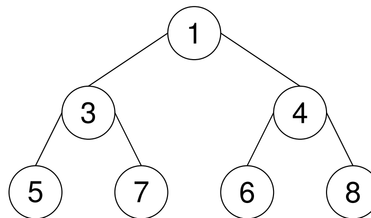
5.2.2.1 IMPLEMENTIERUNG DURCH LISTE (INEFFIZIENT)

Für jeden aktuellen Konten u werden die noch unbesuchten Nachbarn v bestimmt und diese entsprechend ihrer f-Werte sortiert in eine Liste der nach zu besuchenden Knoten eingefügt. Wenn die Liste dabei in jedem Schritt neu sortiert wird, entsteht jeweils ein Aufwand von $O(n \log n)$.

5.2.2.2 IMPLEMENTIERUNG DURCH MIN-HEAP

Eine effiziente Implementierung verwendet einen MIN-HEAP.
Ein Min-Heap ist ein Binärbaum, in dem jeder Knoten, der kein Blatt ist, einen kleineren oder maximal gleichen Wert besitzt als alle Nachfolger.

5.2-4 Beispiel



Alle wichtigen Heap-Operationen (`readMin, add`) können in der Zeit $O(\log n)$ ausgeführt werden. Eine effiziente A*-Suche verwendet einen Min-Heap, um die Knoten entsprechend ihrer f-werte zu verwalten. Dadurch fällt in jedem Schritt nur noch ein zusätzlicher Aufwand von $O(\log n)$ an.

Die A*-Suche ist optimal, d.h., sie findet einen kürzesten Weg.

5.2.2.3 VOR-/NACHTEILE

+ Bei guter Heuristik wird das Ziel oft wesentlich schneller gefunden als mit einer uninformierten Suche.
– Hoher Speicherbedarf, weil im worst-Case alle Knoten gespeichert werden müssen.
– (wie bei Breitensuche:) es ist nicht einfach möglich den gefundenen Pfad zurückzugeben.

```
1 % Heuristische Bewertungsfunktion
2 % [H|T]: Pfad zum aktuellen Knoten H
3 f([H|T],Y) :- g([H|T],Y1), h(H,Y2), Y is Y1+Y2.
4 g(L,Y) :- length(L,Y1), Y is Y1-1.
5 h(X,Y) :- Y is 0. % Ändern! Hier muß eine Kostenschätzfunktion
    angegeben werden.
6
7 % A*-Suche
8 % [H|T]: Sortierte Liste der noch zu untersuchenden Knoten
```

```
% Closed: Liste der Knoten, die bereits untersucht wurden
% Die Knoten müssen hier Pfade zum aktuellen, eigentlichen Knoten
    sein, damit f dessen Bewertung berechnen kann.
hs([H|T], Closed) :-
  goal(H);
  Cl = [H|Closed],
  findall(Node, (adj(H, Node), not(member(Node, Cl))), NewNeighbors
     ),
  append(T, NewNeighbors, Queue),
  fsort(Queue, SortedQ),
  hs(SortedQ, Cl).

% Sortieren der Liste gemäß der Werte von f
fsort(List, Sorted) :-
        map_list_to_pairs(f, List, Pairs),
        keysort(Pairs, SortedPairs),
        pairs_values(SortedPairs, Sorted).
```

5.2.3 IDA*-SUCHE

Vorlesung 12.06.2017

Die IDA*-Suche kombiniert die Vorteile der A*-Suche mit denen der iterativen Tiefensuche. Anstelle der Tiefenschranke der iterativen Tiefensuche verwenden wir eine Schranke für die Werte der Bewertungsfunktion. Der Speicherbedarf der IDA*-Suche entspricht dem der Tiefensuche.

```
% f: Heuristische Bewertungsfunktion
% h: Heuristische Kostenschätzfunktion
% [Head|Tail]: Aktueller Pfad, Head: aktueller Knoten
f([Head|Tail], F) :-
  length(Tail, G),
  h(Head, H),
  F is G+H.

% Node: aktueller Knoten
% Goal: Zielknoten
% Path: Liste der Knoten auf dem Pfad bis Node
% ReturnPath: Rückgabe, wenn ein Pfad zum Ziel gefunden wurde
% adj: Adjazenz eines Knotens
flDfs(Node, Goal, Path, FLimit, ReturnPath) :-
  Node == Goal, reverse(Path, ReturnPath);
  adj(Node,NewNeighbor), not(member(NewNeighbor,Path)),
  f([NewNeighbor|Path], F), F =< FLimit,
  flDfs(NewNeighbor, Goal, [NewNeighbor|Path], FLimit, ReturnPath).

idasLoop(Start, Goal, FLimit, ReturnPath) :-
  flDfs(Start, Goal, [Start], FLimit, ReturnPath);
  % Schranke für f wird um kleinstmögliche Schrittweite erhöht (
     einfach zu programmieren). Sie wird also erhöht, wenn Ziel
     noch nicht gefunden wurde, um den Radius zu erhöhen.
  idasLoop(Start, Goal, FLimit+1, ReturnPath).

idas(Start, Goal, ReturnPath) :-
```

```
% Da f zulässig ist, sind die Kosten bis zum Ziel mindestens FL
f([Start], FLimit),
idasLoop(Start, Goal, FLimit, ReturnPath).
```

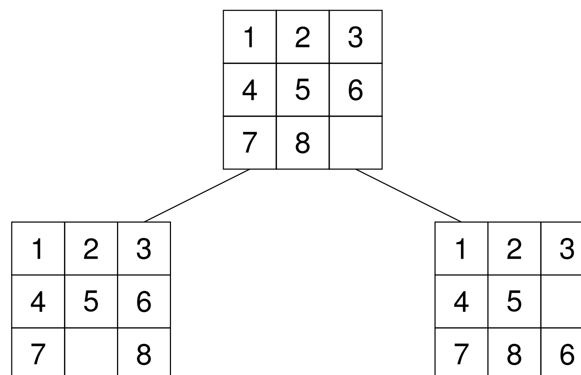
Anwendung 8-Punkte
Ziel: Plättchen in folgende Stellung zu bringen:

1	2	3
4	5	6
7	8	

Darstellung als Graph:

- Knoten: Brettstellungen
- Kanten: Übergänge, die durch das Verschieben eines Plättchen möglich sind.

Beispiel:



Geeignete heuristische Kostenschätzfunktionen h:

1. Hamming-Distanz bis zum Ziel:
   Die Hamming-Distanz zweier Plättchen ist die Anzahl der Plättchen, in denen sich die Stellungen unterscheiden.
   Beispiel:

8	1	3
4	5	7
6	2	

   $$h = 1 + 1 + 0 + 0 + 0 + 1 + 1 + 1 = 5$$

   Diese Heuristik ist zulässig, weil mindestens so viele Verschiebungen von Plättchen nötig sind, wie Plättchen falsch stehen.

2. Manhatten- oder Cityblock-Distanz:
   Anzahl der Verschiebungen, die nötig sind, um ein Plättchen auf direktem Weg zur Zielposition zu schieben, summiert über alle Plättchen.
   Beispiel:

8	1	3
4	5	7
6	2	

$$h = 3 + 1 + 0 + 0 + 0 + 3 + 3 + 2 = 12$$

Diese Heuristik ist zulässig, weil für jedes Plättchen mindestens diese Anzahl Verschiebungen (auf direktem Weg) nötig sind.

Anmerkung: Da jeder Knoten nicht eine Plättchen, sondern eine Stellung der Plättchen ist, entspricht diese berechnete Heuristik einem Knoten, also einer Stellung der Plättchen.

Implementierung Die Brettstellungen werden durch verschachtelte Listen dargestellt. Beispiel:

1	2	3
4	5	6
7	8	

wird dargestellt durch:

```
[ [1,2,3],
  [4,5,6],
  [7,8,9] ] % 9 steht für ein leeres Feld
```

Implementierung:

```
% Leerstelle in Zeile verschieben
sr( [9,A,B], [A,9,B]).
sr( [A,9,B], [A,B,9]).
shiftr(X,Y) :- sr(X,Y); sr(Y,X).  % findet mögliche Zeilenverschiebungen.

adjr( [X,  R2, R3], [Y,  R2, R3]) :- shiftr(X, Y).  % Verschiebe 1. Zeile, wenn sie Leerstelle enthält.
adjr( [R1, X,  R3], [R1, Y,  R3]) :- shiftr(X, Y).  % Verschiebe 2.
adjr( [R1, R2,  X], [R1, R2,  Y]) :- shiftr(X, Y).  % Verschiebe 3.

% Leerstelle in Spalte verschieben (Matrix transponieren, Spalten -> Zeilen. Damit Zeilenverschiebung durchführen)
adjc( [[A1,B1,C1], [D1,E1,F1], [G1,H1,I1]],
      [[A2,B2,C2], [D2,E2,F2], [G2,H2,I2]] ) :-
  adjr( [[A1,D1,G1], [B1,E1,H1], [C1,F1,I1]],
        [[A2,D2,G2], [B2,E2,H2], [C2,F2,I2]] ).

% Leerstelle in Zeile oder Spalte verschieben
adj(Board1, Board2) :-
  adjr(Board1, Board2);
  adjc(Board1, Board2).

% Zielstellung
goal( [[1,2,3],
```

```
23          [4,5,6],
24          [7,8,9]] ).
25
26 % Pretty-Printer
27 printB(Board) :- maplist(writeln, Board), write('\n').
28 print(Boards) :- maplist(printB, Boards).
```

5.2.4 SPIELE MIT GEGNER

Vorlesung 19.06.2017

Es gibt zwei Spieler: Min, Max. Im Spielbaum stellen wir Min durch $\triangledown$, Max durch $\triangle$ dar. Im Spielbaum gibt es Endzustände und eine Nutzenfunktion, die den Nutzen eines Knotens für Max angibt.
Der einfachste Fall einer Nutzenfunktion ist eine Funktion mit drei Werten:

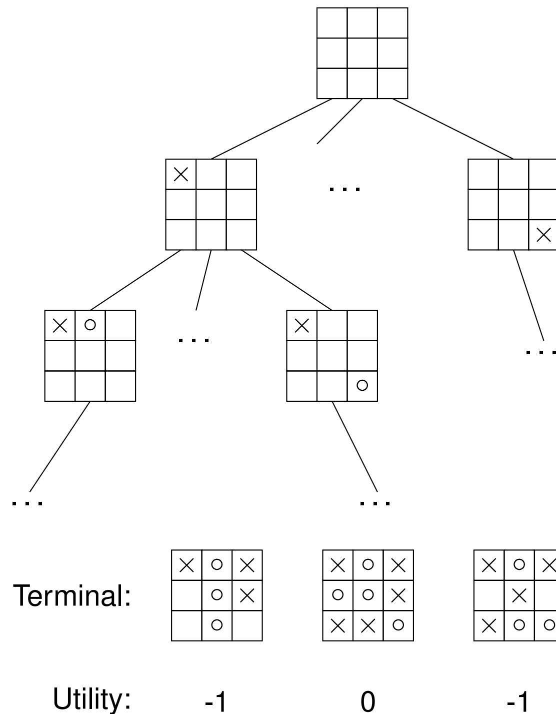
1: Max hat gewonnen

0: unentschieden

-1: Min hat gewonnen

Max versucht die Nutzenfunktion zu maximieren, Min versucht sie zu minimieren.
Für Endzustände lässt nich die Nutzenfunktion leicht angeben.

Beispiel Suchbaum für Tic-Tac-Toe, Max setzt $\times$, Min setzt $\circ$



5.2.4.1 MINIMAX

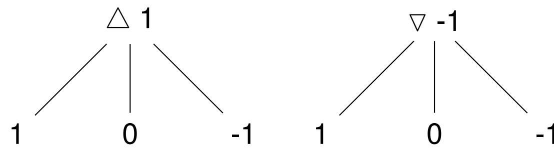
Da wir nicht wissen, wie der Gegner zieht, betrachten wir alle möglichen Züge des Gegners und nehmen an, dass dieser optimal spielt. Damit berechnen wir einen Nutzenwert für jeden Knoten (minmax-Wert). Dieser ergibt sich als:

- Maximum der Bewertungen der Nachfolgeknoten, wenn Max am Zug ist,

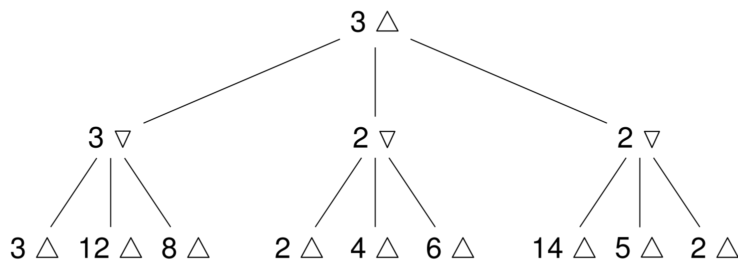
- Minimum der Bewertungen der Nachfolgeknoten, wenn Min am Zug ist.

Damit erhalten wir:

$$\text{minimax}(n) = \begin{cases} \text{utility}(n) & \text{, wenn } n \text{ ein Endzustand ist} \\ \max\{\text{minimax}(s) \mid s \text{ ist Nachfolger von } n\} & \text{, wenn } n \text{ ein Max-Knoten ist} \\ \min\{\text{minimax}(s) \mid s \text{ ist Nachfolger von } n\} & \text{, wenn } n \text{ ein Min-Knoten ist} \end{cases}$$

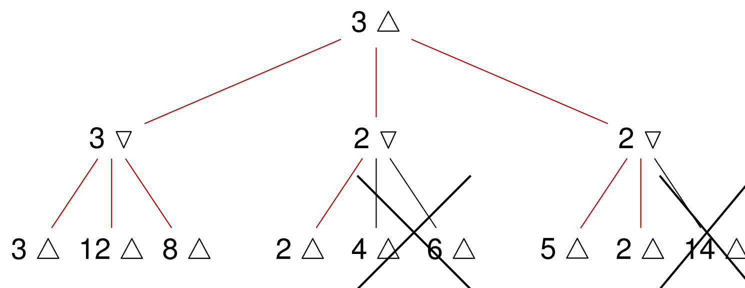


Beispiel



Laufzeit der Minimax-Berechnung: $O(b^d)$ für Spielbaum mit (konstanten) Verzweigungsfaktor b und Tiefe d.
Um die Laufzeit zu verkürzen rechnen wir Minimax-Werte nur für solche Knoten, die das Ergebnis verändern können.

Beispiel



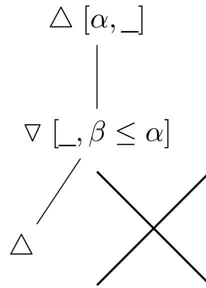
Die Tiefensuche kann im zweiten Zweig abgebrochen werden, da bereits ein (potentielles) Minimum für den minmax-Wert gefunden wurde. Dies ist schon geringer, als das Minimum vom ersten Zweig. Damit braucht der nicht rot markierte Pfad zu Knoten 4 und 6 nicht mehr betrachtet werden. Gleiches gilt für den Pfad zum Koten 14 im dritten Zweig.

5.2.4.2 ALPHA-BETA-ALGORITHMUS

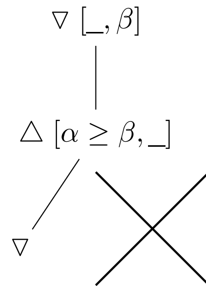
Der α-β-Algorithmus schneidet Zweige des Suchbaums ab, wenn die Werte der darin enthaltenen Knoten den Minimax-Wert des aktuellen Knotens nicht verändern.
Dann wird während der Suche für jeden Knoten ein Intervall $[\alpha, \beta]$ mitgeführt, das angibt, in welchem Bereich sich der Minimax-Wert befindet.

Regeln für das Beschneiden des Suchbaums (Pruning):

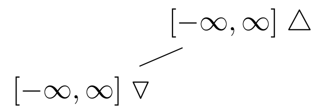
- Ist der β-Wert eines Min-Knotens $\leq$ α-Wert des Vater-Max-Kontens, so wird der Min-Knoten nicht weiter untersucht.



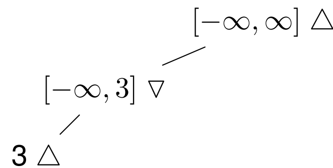
- Ist der α-Wert eines Max-Knotens $\geq$ β-Wert des Vater-Min-Knotens, so wird der Max-Knoten nicht weiter untersucht.



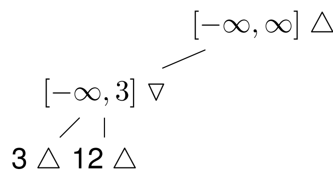
Beispiel Suchbaum von oben:
1. Schritt:



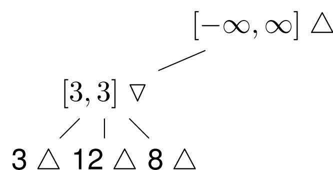
2. Schritt:



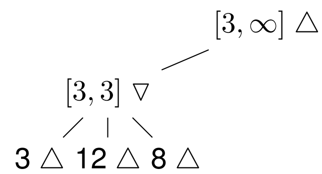
3. Schritt:



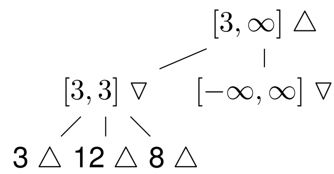
4. Schritt:



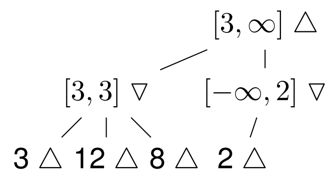
5. Schritt:



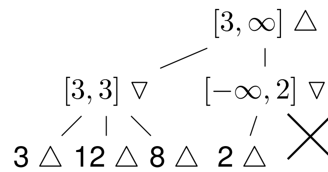
6. Schritt:



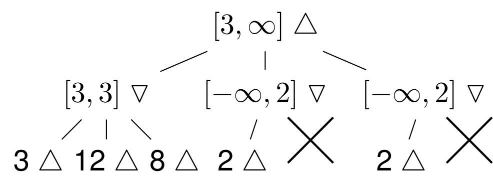
7. Schritt:



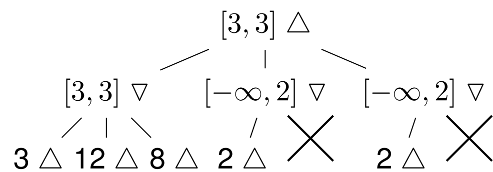
8. Schritt:



Analog:



Und damit letztendlich:



6 BAYES'SCHE NETZE

Vorlesung 26.06.2017

6.1 FORMELN UND BEGRIFFE

$$\text{Notation: } \mathbb{P}(A, B) = \mathbb{P}(A \cap B)$$

$$\mathbb{P}(B|A) = \frac{\mathbb{P}(A \cap B)}{\mathbb{P}(A)} \quad \text{bzw. } \mathbb{P}(B|A) \cdot \mathbb{P}(A) = \mathbb{P}(A, B)$$

Dies gilt nur, wenn B nur von A abhängig ist und nicht von weiteren. Sonst: Marginalisierung / disjunkte Zerlegung:

$$\mathbb{P}(A) = \mathbb{P}(A \cap (B \cup \overline{B})) = \mathbb{P}((A \cap B) \cup (A \cap \overline{B})) = \mathbb{P}(A \cap B) + \mathbb{P}(A \cap \overline{B}) = \mathbb{P}(A, B) + \mathbb{P}(A, \overline{B})$$

$$A,\ B \text{ unabhängig} \Leftrightarrow \mathbb{P}(A \cap B) = \mathbb{P}(A) \cdot \mathbb{P}(B) \text{ und auch: } \mathbb{P}(A \cap B|X) = \mathbb{P}(A|X) \cdot \mathbb{P}(B|X)$$

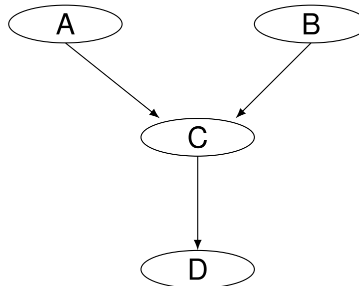
$$\mathbb{P}(A) = \mathbb{P}(A|B) \cdot \mathbb{P}(B) + \mathbb{P}(A|\overline{B}) \cdot \mathbb{P}(\overline{B})$$

$$\mathbb{P}(A|B) = 1 - \mathbb{P}(\overline{A}|B)$$

$$\mathbb{P}\left(\sum A_k\right) = \sum \mathbb{P}(A_k)$$

$$\mathbb{P}\left(\bigcup A_k\right) \leq \sum \mathbb{P}(A_k)$$

Ereignisse $A,\ B$ heißen BEDINGT UNABHÄNGIG gegeben C, wenn gilt $\mathbb{P}(A, B|C) = \mathbb{P}(A|C) \cdot \mathbb{P}(B|C)$. Dies ist gleichwertig damit, dass $A,\ B$ unabhängig bezüglich des W-Maßes $\mathbb{P}(\bullet|C)$ sind. Hinweis zur Berechnung von Wahrscheinlichkeiten:



$$\mathbb{P}(D) = \mathbb{P}(D, A, B, C) + \ldots + \mathbb{P}(D, \overline{A}, \overline{B}, \overline{C})$$

$$\mathbb{P}(D, A, B, C) = \mathbb{P}(D|C) \cdot \mathbb{P}(C|A \cap B) \cdot \mathbb{P}(A) \cdot \mathbb{P}(B)$$

6.2 GRUNDLAGEN

Beispiel Bob hat in seinem Haus eine Alarmanlage installiert. Seine Nachbarn John und Marry rufen ihn im Büro an, wenn sie Alarm hören. Bob modelliert deren Zuverlässigkeit durch

$$\mathbb{P}(J \mid Al) = 0,9 \qquad \mathbb{P}(M \mid Al) = 0,7$$

$$\mathbb{P}(J \mid \overline{Al}) = 0,05 \qquad \mathbb{P}(M \mid \overline{Al}) = 0,01$$

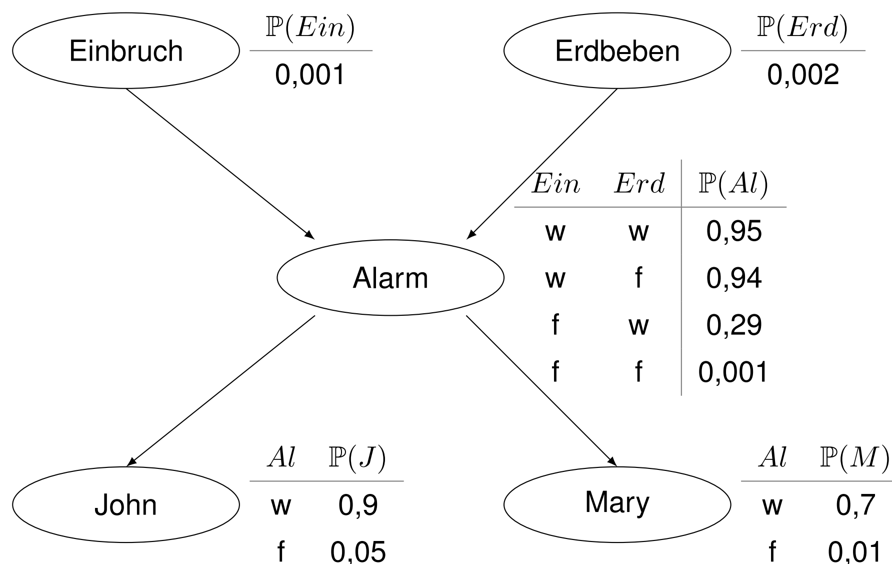
Sowohl ein Einbruch als auch ein Erdbeben können einen Alarm auslösen:

$$\mathbb{P}(Al \mid Ein,\ Erd) = 0,95 \qquad \mathbb{P}(Al \mid Ein,\ \overline{Erd}) = 0,94$$

$$\mathbb{P}(Al \mid \overline{Ein},\ Erd) = 0,29 \qquad \mathbb{P}(Al \mid \overline{Ein},\ \overline{Erd}) = 0,001$$

Ferner sei $\mathbb{P}(Ein) = 0,001$, $\mathbb{P}(Erd) = 0,002$.
Die Ereignisse Erd, Ein seien unabhängig.
Graphisch Darstellung als Bayes'sches Netz:

- Knoten: Ereignisse
- Kanten: Bedingte W'keiten



Ein	Erd	$\mathbb{P}(Al)$
w	w	0,95
w	f	0,94
f	w	0,29
f	f	0,001

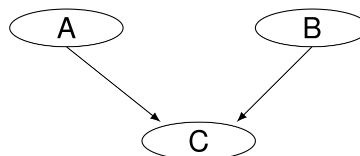
Al	$\mathbb{P}(J)$
w	0,9
f	0,05

Al	$\mathbb{P}(M)$
w	0,7
f	0,01

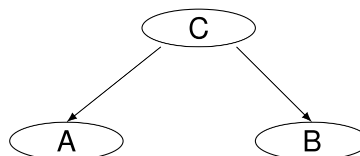
Wen John und Mary unabhängig auf einen Alarm reagieren, dann sind die Ereignisse J, M unabhängig gegeben Al.

6.3 SEMANTIK VON BAYES-NETZEN

Kanten beschreiben (bedingte) Unabhängigkeiten von Ereignissen:



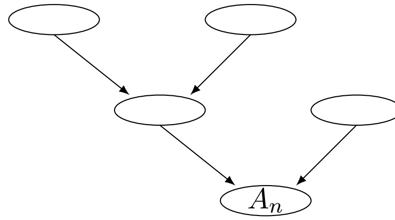
$\rightarrow A$, B sind unabhängig.



$\rightarrow A$, B sind unabhängig gegeben C.

Ein Bayes-Netz muss ein DAG (directed asymmetric graph) sein (enthält also keinen Loop).
Dann können die Knoten topologisch sortiert werden. Dann gilt:

$$\mathbb{P}(A_n | A_1, \ldots, A_{n-1} = \mathbb{P}(A_n | \text{ Eltern}(A_n)$$



Daraus folgt:

$$\begin{aligned}\mathbb{P}(A_1,\dots,A_n) &= \mathbb{P}(A_n|A_1,\dots,A_{n-1})\cdot\mathbb{P}(A_1,\dots,A_{n-1})\\ &= \mathbb{P}(A_n|\ \mathrm{Eltern}(A_n))\cdot\mathbb{P}(A_1,\dots,A_{n-1})\\ &= \dots\\ &= \prod_{k=1}^{n}\mathbb{P}(A_k|\ \mathrm{Eltern}(A_k))\end{aligned}$$

Dabei ist $\mathbb{P}(A_k|\ \mathrm{Eltern}(A_k)) := \mathbb{P}(A_k)$, wenn A_k keine Eltern besitzt.

Beispiel

$$\begin{aligned}\mathbb{P}(J,Al,Ein,Erd) &= \mathbb{P}(J|Al)\cdot\mathbb{P}(Al|Ein,Erd)\cdot\mathbb{P}(Ein)\cdot\mathbb{P}(Erd)\\ &= 0,00000171\end{aligned}$$

$$\begin{aligned}\mathbb{P}(J,Ein) = &\ \mathbb{P}(J,Al,Ein,Erd)+\mathbb{P}(J,\overline{Al},Ein,Erd)\\ &+\mathbb{P}(J,Al,\overline{Erd},Ein)+\mathbb{P}(J,\overline{Al},Ein,\overline{Erd})\\ = &\ \dots = 0,000849\end{aligned}$$

Entsprechend ergibt sich $\mathbb{P}(J) = 0,052$.
$\mathbb{P}(Ein|J) = \frac{\mathbb{P}(Ein,J)}{\mathbb{P}(J)} = 0,016$. Diese Zahl ist so gering, da die WK für den Einbruch sehr gering ist.

6.4 KOMPLEXITÄT DER INFERENZ IN BAYES’SCHEN NETZEN

Vorlesung 03.07.2017

Die exakte Inferenz (Berechnung von W’keiten) ist NP-vollständig. D.h., es ist kein effizienter (polynomieller) Algorithmus zur Berechnung der W’keiten bekannt.

Bei einem Bayes’schen Netz mit den Knoten $A_1,\dots,A_n$ muss $\mathbb{P}(A_n)$ im Allgemeinen durch eine disjunkte Zerlegung folgender Form sein:

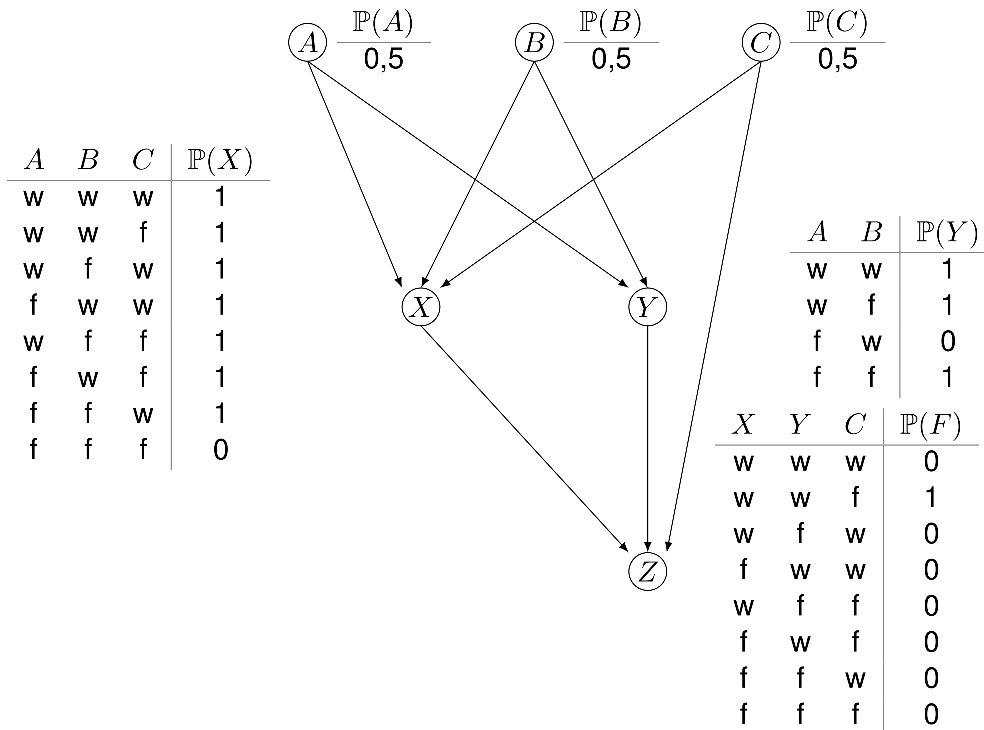
$$\mathbb{P}(A_n) = \mathbb{P}(A_n,A_1,\dots,A_{n-1})+\dots+\mathbb{P}(A_n,\overline{A}_1,\dots,\overline{A}_{n-1})$$

Diese Summe besteht aus 2^{n-1} Summanden. Der Aufwand zur Berechnung von $\mathbb{P}(A_n)$ wächst daher exponentiell in n.

$$SAT = \{F \mid F \text{ ist eine erfüllbare Formel der Aussagenlogik}\}$$

SAT ist NP-vollständig. Man kann zeigen: Aus einem effizienten Algorithmus für Bays’sche Netze ließe sich ein effizienter Entscheidungsalgorithmus für SAT konstruieren.

Beispiel Sei $F = (A \vee B \vee C) \wedge (A \vee \neg B) \wedge \neg C$
(Vorgehen: Eingangsknoten auf $\frac{1}{2}$ setzen, prüfen, ob Ergebnis erfüllbar ist)
Daraus konstruieren wir folgendes Bays'sches Netz:



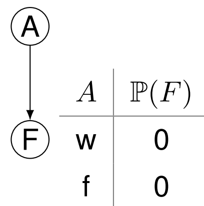
A	B	C	$\mathbb{P}(X)$
w	w	w	1
w	w	f	1
w	f	w	1
f	w	w	1
w	f	f	1
f	w	f	1
f	f	w	1
f	f	f	0

A	B	$\mathbb{P}(Y)$
w	w	1
w	f	1
f	w	0
f	f	1

X	Y	C	$\mathbb{P}(F)$
w	w	w	0
w	w	f	1
w	f	w	0
f	w	w	0
w	f	f	0
f	w	f	0
f	f	w	0
f	f	f	0

Es gilt: $\mathbb{P}(F) > 0 \Leftrightarrow F$ ist erfüllbar.

Beispiel Sei $F = A \wedge \neg A$



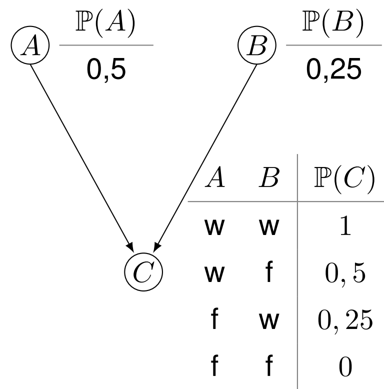
A	$\mathbb{P}(F)$
w	0
f	0

$\Rightarrow F$ ist nicht erfüllbar

6.4.1 AUSWEG DER NP-VOLLSTÄNDIGKEIT DER EXAKTEN INFERENZ

Für speziell strukturierte Netze gibt es einen effizienten Algorithmus. Ferner lässt sich die W'keit durch SAMPLING approximieren. Einfachster Ansatz dann: Beginnend mit dem Startknoten des Netzes werden Ereignisse entsprechend ihrer W'keiten gewürfelt.
Dieses Sampling wird n-mal wiederholt und es wird ein Mittelwert als Schätzer der W'keit berechnet.

Beispiel



Sampling Durchlauf:

1. $A = w$, $B = f$, $C = w$
2. $A = w$, $B = f$, $C = f$
3. $A = f$, $B = f$, $C = f$
4. $A = f$, $B = w$, $C = f$
5. $A = w$, $B = f$, $C = w$
6. $A = f$, $B = f$, $C = f$

Als Schätzer für $\mathbb{P}(C)$ ergibt sich $\hat{\mathbb{P}}(C) = \frac{2}{6} = \frac{1}{3}$.
Vorteil/Nachteil der Methode:

+ Einfach zu implementieren
– Ungeeignet für Netze, in denen Knoten mit sehr geringer Wahrscheinlichkeit vorkommen.