Künstliche Intelligenz

Markus Klemm.net

SS 2015

Inhaltsverzeichnis

1 Prädikatenlogik

1.1 Problem

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

- Alle Vögel können fliegen.
- Wenn X eine Katze ist, dann ist X ein Haustier.
- Für jedes Land gibt es eine Hauptstadt.

In der Prädikatenlogik betrachten wir zunächst eine (unwichtige) Menge U (Universum), die alle zu betrachtenden Objekte (Vögel, Katzen, Länder) enthält. Davon betrachten wir Teilmengen, z.B. die Menge aller Vögel (also eine einstellige Relation) oder die Menge aller Verheirateten (also eine mehrstellige Relation).

1.2 Definition

Sei V eine Menge von Variablen, K eine Menge von Konstanten, F eine Menge von Funktionssymbolen.

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

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

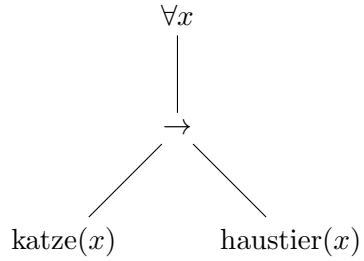
1.3 Definition

Sei eine Menge von Prädikatensymbolen gegeben. Die Formeln der Prädikatenlogik sind induktiv.

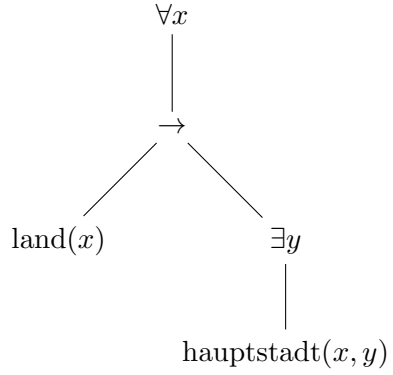
- Wenn $t_1, \ldots, t_n$ Terme und P ein Prädikatensymbol der Stelligkeit n ist, dann ist $P(t_1, \ldots, t_n)$ eine Formel.
- Wenn F, G Formeln sind, dann auch $F \wedge G, F \vee G, \neg F$.
- Wenn x eine Variable und F eine Formel sind, dann auch $\forall x F$, sowie $\exists x F$.

Beispiel

- katze(reni)
- $\forall x(\text{katze}(x) \rightarrow \text{haustier}(x))$ Syntaxbaum dazu



- $\forall x(\text{land}(x) \rightarrow \exists y \text{hauptstadt}(x, y))$



1.4 Semantik(informal)

Den Prädikatensymbolen müssen Relationen zugeordnet werden. Z.B: land = {deutschland, frankreich, spani
Wahrheitswert einer Formel:

- Die Formel $P(t_1, \ldots, t_n)$ ist wahr, 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, wenn es ein $x \in U$ gibt, so dass F für dieses x wahr ist.
- $\forall x F$ ist wahr, wenn für alle $x \in U \quad F$ für diese x wahr ist.

Beispiel

Symbole: vogel,fliegt

Relationen vogel $= \{\text{amsel}, \text{drossel}, \text{fink}, \text{star}\}$, fliegt $=$ vogel $\cup \{\text{maikäfer}, \text{A380}\}$

- vogel(amsel) ist wahr
- $\exists x$ vogel(x) ist wahr
- $\forall x$ vogel(x) ist falsch
- $\forall x$ (vogel$(x) \rightarrow$ fliegt(x)) ist wahr

Vorrang der Operatoren:

$$\neg$$
$$\forall \exists$$
$$\wedge, \vee$$
$$\rightarrow, \leftrightarrow$$

Beispiel land $= \{\text{deutschland}, \text{england}, \text{frankreich}, \text{spanien}\}$,
hauptstadt $= \{(\text{deutschland}, \text{berlin}), (\text{england}, \text{london}), (\text{frankreich}, \text{paris}), (\text{spanien}, \text{madrid})\}$

Rechenregeln

- $\neg \forall x \; F \equiv \exists x \neg F$
- $\neg \exists x \; F \equiv \forall x \neg F$
- Für $Q \in \{\forall, \exists\}$ und $\circ \in \{\wedge, \vee\}$ gilt $(QxF) \circ G \equiv Q(F \circ G)$, falls x in G nicht frei vorkommt.
- $\forall x F \wedge \forall x G \equiv \forall x (F \wedge G)$
- $\exists x F \vee \exists x G \equiv \exists x (F \vee G)$

Definition Eine Aussage F ist in bereinigter Pränexform wenn $F = Q_1 y_1 \ldots Q_n y_n G$, wobei $Q_i \in \{\forall, \exists\}$ und G keine Quantoren enthält und $y_1, \ldots, y_n$ paarweise verschieden sind.

Für jede Aussage gibt es eine äquivalente Formel in bereinigter Pränexform.

Beispiel

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

$$\begin{aligned}
&\neg(\forall x(\text{vogel}(x) \to \text{fliegt}(x))) && \equiv \\
&\neg(\forall x(\neg\text{vogel}(x) \vee \text{fliegt}(x))) && \equiv \\
&\exists x \neg(\neg\text{vogel}(x) \vee \text{fliegt}(x)) && \equiv \\
&\exists x(\text{vogel}(x) \wedge \neg\text{fliegt}(x))
\end{aligned}$$

Definition

- Ein Literal ist eine Atomformel oder eine negierte Atomformel.
- 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.

Beispiel

- Literale: $P(x,y), \neg P(x,y), Q(zx)$
- Hornklausel: $\neg P(x,y) \vee \neg Q(zx), \neg S(x) \vee T(y)$

Wichtiger Spezialfall Hornklausel mit genau einem positivem Literal.

- Ein Literal: Fakt, z.B. $P(x,y)$.
- Mindestens zwei Literale: Dies lässt sich als Implikation darstellen.
  $\neg A_1 \vee \cdots \vee \neg A_{n-1} \vee A_n \equiv \neg(A_1 \wedge \cdots \wedge A_{n-1}) \vee A_n \equiv A_1 \wedge \cdots \wedge A_{n-1} \to A_n$.

Mit Hilfe der Hornklausel lassen sich Regeln formulieren z.B.

$$\begin{aligned}
&\forall x(\text{katze}(x) \to \text{haustier}(x)) && , \\
&\forall x(\text{hund}(x) \to \text{haustier}(x)) && , \\
&\text{katze(reni)}
\end{aligned}$$

Wenn diese Hornklausel mit $\wedge$ verknüpft werden, erhalten wir eine Hornformel. Diese Hornformel kann als Wissensbasis eines Expertensystems betrachtet werden.

Grundlegender Aufbau eines Expertensystems

2 Prolog-Programmierung

Prolog Programming in logic. In den 70ern und 80er Jahren als KI-Sprache entwickelt.

Prolog-Programme sind im Wesentlichen Hornformeln, bei denen alle Variablen allquantisiert sind und die in bereinigter Pränexform vorliegen.

Beispiel

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

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

Anfrage an Prolog:

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

2.1 Syntax

Prädikat Wort in Kleinbuchstaben (außerhalb einer Klammer).

Konstante Argument in Kleinbuchstaben.

Variablen Wort, das mit Großbuchstaben beginnt.

„$\leftarrow$:“ : „: $-$“ (wird impliziert von)

„$\wedge$“ : „,“

Jede Klausel wird mit „,“ abgeschlossen. Das Programm ist eine Menge von Klauseln, die implizit mit „$\wedge$“ verknüpft sind (Hornformel).

$\vee$-Verknüpfung: Die Formel $A \vee B \rightarrow C$ ist wegen $A \vee B \rightarrow C \equiv\neq (A \vee B) \vee C \equiv (\neq A \wedge \neg B) \vee C$ keine Hornklausel. Da jedoch: $A \vee B \rightarrow C \equiv (\neg A \vee C) \wedge (\neg B \vee C) \equiv (A \rightarrow C) \wedge (B \rightarrow C)$ eine Hornformel ist, lässt sich $A \vee B \rightarrow C$ in Prolog darstellen.

$\vee$-Operator: „;“

Beispiel haustier(X) : $-$katze(X); hund(x)

2.2 Unterschied zwischen Prolog und imperativen Programmiersprachen

Eine Variable kann sowohl Ergebnis als auch Ausgabe sein.

```
1 katze(reni).
2 katze(mimi).
3 ? - katze(reni).
4 true.
5 ? - X=reni, katze(X).
6 true.
7 ? - katze(X).
8 X=reni;
9 X=mimi.
```

2.3 Behandlung von Existenzquantoren

Da in Prolog alle Variablen allquantisiert sind, können existenzquantifizierte Variablen nicht unmittelbar dargestellt werden. Existenzquantoren können jedoch durch Skolemisierung. Dazu wird die Formel zuerst in bereinigte Pränexform gebracht.

Einfacher Spezialfall Die Formel in bereinigter Pränexform hat die Gestalt

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

Diese kann erfüllbarkeitsäquivalent dargestellt werden durch

$$P(a)$$

wobei a eine noch nicht verwendete Konstante ist (Skolemkonstante).

Weiterer Fall Wenn die Formel die Gestalt

$$\forall x \exists y P(x, y)$$

besitzt, dann lässt sich diese erfüllbarkeitsäquivalent darstellen durch

$$P(x, f(x))$$

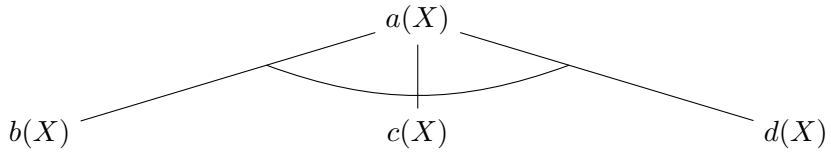
wobei f ein noch nicht verwendetes Funktionssymbol ist (Skolemfunktion).

2.4 Auswertestrategie von Prolog

Die Struktur eines Prolog-Programms lässt sich durch einen Und-Oder-Baum darstellen.

2.4.1 Und-Verknüpfung

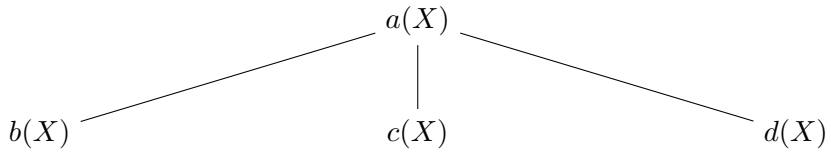
```
1 a(X) :- b(X), c(X), d(X).
```



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

2.4.2 Oder-Verknüpfung

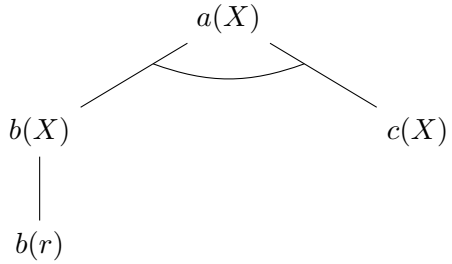
```
1  a(X) :- b(X); c(X); d(X).
```



Die Anfrage $a(X)$ ist wahr genau dann, wenn eines der Teilziele wahr ist $b(X), c(X), d(X)$ wahr ist.

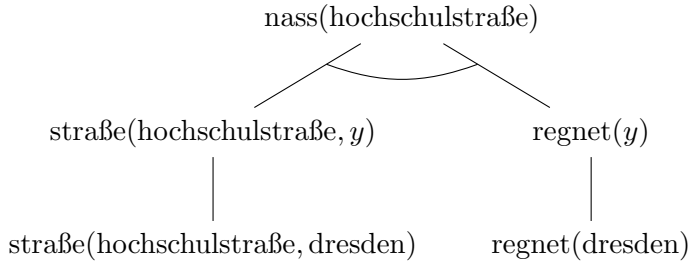
2.4.3 Suche nach einer Lösung

(d.h. X ist nicht instanziert):
X wird nach unten weitergereicht, bis es auf einer Blattebene an eine Konstante gebunden wird, mit der das zugehörige Teilziel war wird. Der so gefundene Lösungskandidat wird dann nach oben propagiert und für weitere Teilziele verwendet.



Wenn mit dem gefunden Kandidaten weitere, mit $\wedge$ verknüpfte Teilziele nicht erfüllbar werden, sucht Prolog nach weiteren Lösungen. Dazu wird eine Tiefensuche mit Backtracking ausgeführt.

Beispiel Aufgabe 4 Anfrage: nass(hochschulstraße)



Definition Zwei Atome P, Q heißen unifizierbar, wenn es eine Ersetzung der in P, Q verkommenden Variablen gibt, so dass $P \equiv Q$.

Beispiel regnet(y), regnet(dresden) sind unifizierbar durch y/dresden.
$P(a, X, Y), P(a, b, c)$ sind unifizierbar durch $X/b, Y/c$.
$P(X, X), P(a, b)$ sind nicht unifizierbar.

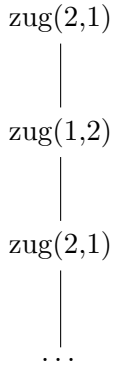
2.4.4 Durchsuchen des Baums

Für ein Ziel P wird das Prolog-Programm von oben nach unten durchsucht, bis eine linke Seite einer Klausel(Kopf) mit P unifiziert. Dadurch wird ein neues Ziel erzeugt. Wenn dieses neue Ziel false liefert, wird ein Backtracking ausgeführt, indem eine weitere linke Seite gesucht wird, die mit dem Ziel unifiziert. Innerhalb jeder Klausel wird die rechte Seite von links nach rechts durchsucht.

Problem bei der Tiefensuche Beispiel:

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

Suchbaum



Da der Suchbaum unendlich ist, liefert die Tiefensuche keine Lösung.

Lösung: Klauseln anders anordnen: Abbruchbedingung der Rekursion nach oben:

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

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

```
1 strasse(regen,nass).
2 strasse(X,trocken) :- X \== regen.
3 ?-strasse(sonne,Y).
4 y=trocken.
```

3 Computeralgebra

3.1 Arithmetik in Prolog

Arithmetische Ausdrücke werden mit dem Operator „is“ ausgewertet. Dabei muss die rechte Seite instanztiiert sein. Beispiel

```
1 ?- X is 3+4.
2 X=7.
3 ? 10 is 1+2.
4 false.
5 ? 2 is 1+X.
6 Error.
```

Beispiel Berechnung von $n!$

```
1 fac(0,1).
2 fac(N,M) :- N1 is N-1,
3             fac(N1,F),
4             M is N * F.
```

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

Prolog ist in der Lage, mit Hilfe des Unifikationsoperators arithmetische Ausdrücke zu zerlegen, wobei die Priorität der Operatoren beachtet wird.

```
1 ? - x + 3 * y = A+B.
2 A=x, B=3*y.
3 ? x+y+z = A+B.
4 A= x+y, B = z.
```

Anwendung Symbolisches Differenzieren

$$\frac{dx}{dx} = 1 \wedge \frac{c}{dx} = 0 \wedge \frac{dx \cdot c}{dx} = \frac{dx}{dx} \cdot c \wedge \frac{d(f \pm g)}{dx} = \frac{df}{dx} \pm \frac{dg}{dx}$$

```
1 diff(X,X,1).
2 diff(C,X,0) :- atomic(C), C \== X.
3 diff(-F,X,-DF) :- diff(F,X,DF).
4 diff(C*F,X,C*DF) :- diff(C,X,0), diff(F,X,DF).
5 diff(F+G,X,DF+DG) :- diff(F,X,DF), diff(G,X,DG).
6 diff(F-G,X,DF-DG) :- diff(F,X,DF), diff(G,X,DG).
```

Wie lassen sich Terme wie $1 * x + 0$ vereinfachen?
Ansatz: Prädikat $s/2$

```
1  s(0+A,A).
2  s(A+0,B) :- s(A,B).
3  s(1*A,A).
4  s(A*1,A).
5  s(A*0,0).
6  s(0*A,0).
7  s(A+B,C) :- number(A), number(B), C is A+B.
8  s(A–B,C) :- number(A), number(B), C is A–B.
9  s(A*B,C) :- number(A), number(B), C is A*B.
10 s(A/B,C) :- number(A), number(B), C is A/B.
11 s(A+B,C) :- s(A,SA), s(B,SB), s(SA+SB,C).
12 s(A,A).
```

Idee: Mit dem Prädikat $s/2$ werden die Terme rekursiv zerlegt, mit $s0/2$ werden elementare Vereinfachungen vorgenommen. Dadurch wird von oben nach unten ein Syntaxbaum aufgebaut und von unten nach oben vereinfacht.

3.2 Listen

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 H und den Elementen in T besteht.

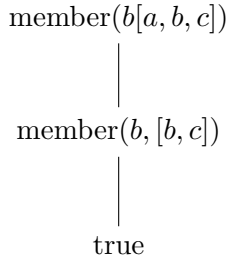
Alternativ kann die Liste auch in der Form $[x_1, \ldots, x_n]$ aufgeschrieben werden.

Beispiel für Listen: $[a, b, c], [a|[b, c]], [a, b, c, 1, 2, [u, v]]$

Beispiel für ein Prädikat auf einer Liste: member/2 z.B.

```
1 member(X,[X|_]).
2 member(X,[_|T]) :- member(X,T).
```

Suchbaum für die Anfrage member$(b, [a, b, c,])$.:

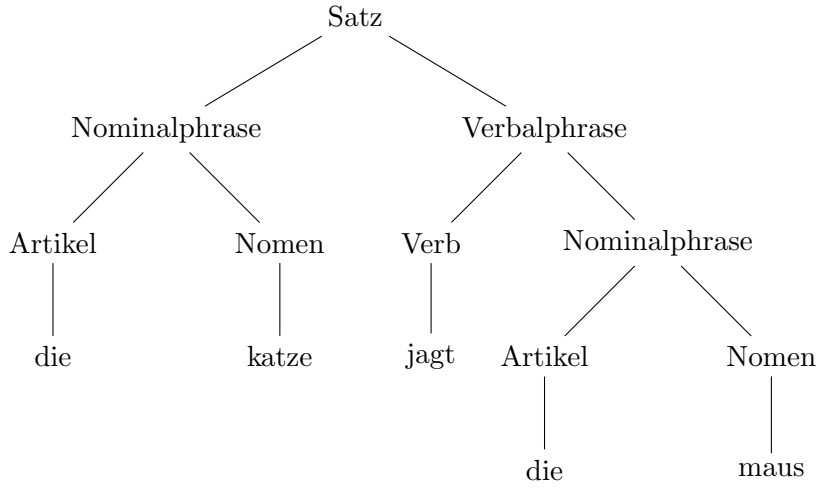


Weiteres Listenprädikat: append/3
append($L1, L2, L3$) ist wahr genau dann, wenn die Verkettung der Listen $L1, L2$ gleich $L3$ ist.

Beispiel append($[a, b, c], [d, e, f], L$) liefert $L = [a, b, c, d, e, f]$
append($[1, 2, 3], L, [1, 2, 3, 4, 5]$) liefert $L = [4, 5]$
append($L1, L2, [1, 2, 3]$) liefert $L1 = [], L2 = [1, 2, 3]; L1 = [1], L2 = [2, 3]$; usw.

4 Sprachverarbeitung

Beispielgrammatik:



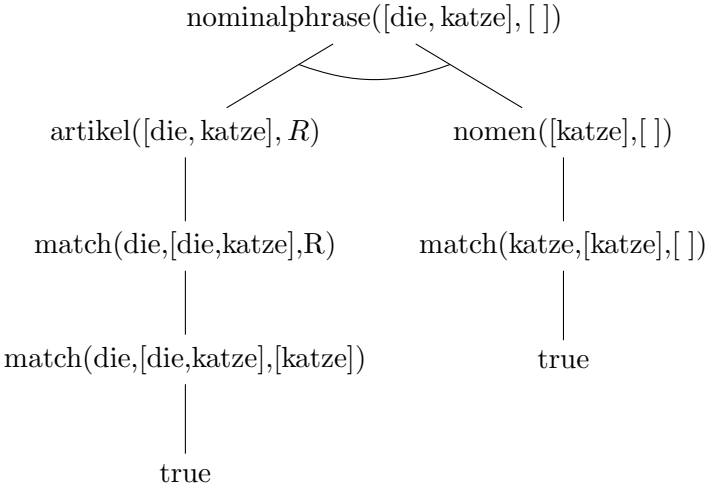
Zur Darstellung von Strings verwenden wir Listen, z.B. [die,katze,schläft].
Zur Darstellung der Grammatik verwenden wir einen Recursive-Descent-Parser.

```
1  satz(In,Rest) :-
2      nominalphrase(In,R),
3      verbalphrase(R,Rest).
4  nominalphrase(In,Rest) :- artikel(In,R), nomen(R,Rest).
5  verbalphrase(In,Rest) :- verb(In,R), nominalphrase(R,Rest).
6  artikel(In,Rest) :- match(die,In,Rest).
7  verb(In,Rest) :-
8      match(jagt,In,Rest).
9  nomen(In,Rest) :-
10     match(katze,In,Rest);
11     match(maus,In,Rest).
12 match(X,[X|Rest],Rest).
```

```
1  match(katze,[katze,schlaeft],R).
```

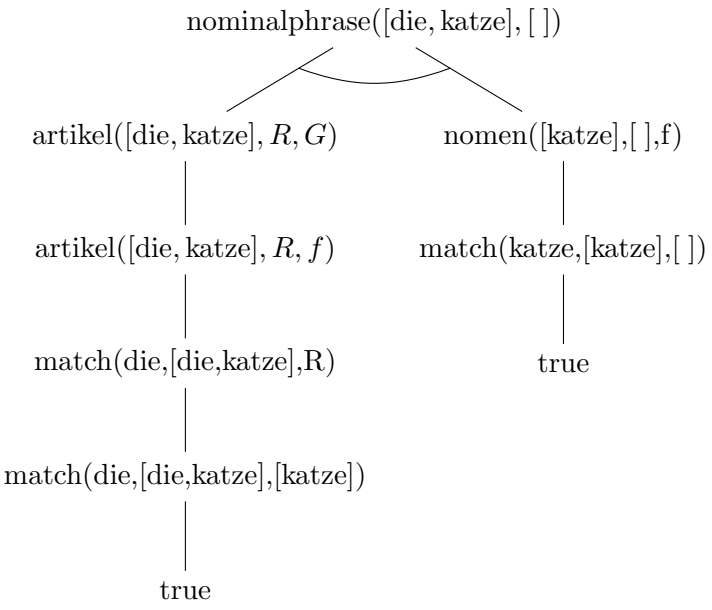
```
2 ? R= schlaeft.
3 match(katze,[maus],R).
4 ? false.
```

Beispiel Suchbaum für die Anfrage nominalphrase([die, katze], [ ])



Genus-Kongruenz Wenn das Nomen „kater“ und der Artikel „der“ in obige Grammatik eingefügt werden, können auch grammatikalisch falsche Sätze, wie [die,kater,jagt,der,maus] abgeleitet werden.

```
1 nominalphrase(In,Rest) :- artikel(In,R,Genus), nomen(R,Rest,Genus).
2 artikel(In,Rest,m) :- match(der,In,Rest).
3 artikel(In,Rest,f) :- match(die,In,Rest).
4 nomen(In,Rest,m) :- match(kater,In,Rest).
5 nomen(In,Rest,f) :-
6       match(katze,In,Rest);
7       match(maus,In,Rest).
```



4.1 Wertigkeit von Verben

Zwei wichtige Klassen von Verben sind:

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

Das Verb bestimmt den Kasus des zugehörigen Objekts. Zum Beispiel benötigen jagen, fangen ein Akkusativobjekt (das Subjekt sieht wen oder was?)

4.2 Dialogsystem

Drei wichtige Aufgaben eines Dialogsystems

- Frage des Benutzers analysieren und den Sinn verstehen
- Eine Antwort suchen
- Antwort konstruieren und dem Benutzer antworten.

Einfacher Ansatz, um die Frage des Benutzers zu verstehen: Grammatik für mögliche Fragen des Benutzers konstruieren, die Parameter für die relevanten Bestandteile der Frage enthält. Anwendbar für einfach und gleichartig strukturierte Fragen, z.B. nach Zugverbindungen. Aus der analysierten Frage erzeugt das Dialogsystem eine interne Repräsentation und durchsucht eine Wissensbasis und erzeugt daraus die Antwort.

$$\text{place-question} \rightarrow (\text{in}|\varepsilon)(\text{what}|\text{which})\text{Place 1 is the Place 2}(\text{on}|\text{in}|\text{located in}|\varepsilon)$$

$$\text{place 1} \rightarrow \text{street}|\text{town}|\text{city}$$

$$\text{place 2} \rightarrow \text{hotel}|\text{restaurant}|\text{shop}$$

5 Problemlösen durch Suche

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

Beispiel Schach: Ca 30 Möglichkeiten pro Halbzug. Bei 50 Halbzügen enthält der Suchbaum $\sum_{d=0}^{50} 30^d = \frac{30^{51}-1}{30-1} \approx 7,4 \cdot 10^{73}$ Knoten.
Bei 10000 Computern, die 10^9 Knoten s^{-1} erzeugen und durchsuchen können, ergibt sich für die Rechenzeit $2,3 \cdot 10^{53}$ Jahre.

5.1 Uninformierte Suche

Bereits bekannt: Breiten- und Tiefensuche
findall(X,P,L): sucht alle X, für die P wahr ist, und erzeugt daraus die Liste L.
not(P): ist wahr genau dann, wenn P false liefert (Negation by failure).

5.1.1 Nachteil der Breitensuche

Exponentieller Speicherbedarf für Suchbäume.

5.1.2 Nachteil der Tiefensuche

- Wenn bereits besuchte Knoten nicht speichert werden: Zyklen möglich.

  Lösung: In der Tiefensuche werden nur die Knoten auf dem aktuellen Pfad gespeichert. Der Platzbedarf liegt dann in $\mathcal{O}(d)$, wobei d die Länge des aktuellen Pfades ist.

- Der gefundene Weg ist nicht notwendig der kürzeste Weg.

- Die Tiefensuche kann sich in unendlichen oder sehr langen Pfaden verlieren.

5.2 Iterative Tiefensuche (IDDFS, Iterative Deepering Depth First Search)

Wir verwenden eine Tiefenschranke, die sukzessive erhöht wird, bis das Ziel gefunden wird.
Da in einem Suchbaum mit Verzweigungsfaktor $b > 1$ fast alle Knoten Blätter sind, fällt die meiste Rechenzeit zum Durchsuchen der Blattebene an. Durch eine genaue Rechnung lässt sich zeigen, dass die Laufzeit der iterativen Tiefensuche nur einen kleinen Faktor höher ist, als die der Tiefensuche.

5.3 Planungsprobleme

Gegeben Regeln, wie Zustände verändert werden können, Startzustand und Zielzustand.

Gesucht Weg vom Start zum Ziel.

Beispiel Affe-Banane-Problem: Gegen sei ein Raum mit einer Tür, einem Fenster, einem Affen und einer Banane. Im Ausgangszustand ist der Affe an der Tür, auf der gegenüber liegenden Seite ist das Fenster mit dem Stuhl in einer Ecke am Fenster und die Banane hängt in der Mitte des Raums von der Decke.
Problem: Der Affe ist nicht hoch genug um die Banane direkt zu greifen.
Regeln. Affe kann herumlaufen, den Stuhl verschieben, auf den Stuhl steigen.
Ziel: Affe auf dem Stuhl, unter der Banane.

Repräsentation eines Zustands, bzw. Knoten des Graphen:

- Position des Affen
- Position des Stuhls
- Position der Banane
- Affe auf Stuhl

[Anmerkung: Knoten repräsentieren somit den Gesamtzustand des Raumes]
Positionen: Tür, Mitte, Fenster.
Die Kanten des Graphen sind durch Regeln gegeben.

5.4 Informierte bzw. heuristische Suche

Ziel Informationen über das Suchproblem nutzen, um schneller zum Ziel zu kommen. Dazu 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 :

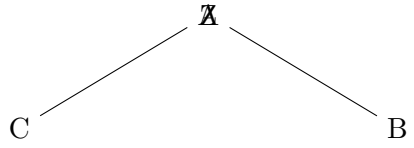
$$f(v) = 0$$

Die Knoten mit der niedrigsten Bewertung werden zuerst verfolgt.

5.4.1 Gierige Suche

Die gierige Suche verwendet in jedem Schritt den Knoten, der dem Ziel am nächsten liegt bzw., der die günstigste Entfernungsschätzung bis zum Ziel hat.

Beispiel Suche nach Wegen zu einem Ort. Heuristische Bewertungsfunktion: $f(s) =$ Luftlinienentfernung von s zum Ziel.



Die kürzeste Strecke von A nach Z ist A-B-Z (100km). Wenn $f(C) < f(B)$, findet die gierige Suche jedoch die Strecke A-C-Z, die 11 km länger ist.

Folgerung Die gierige Suche ist nicht optimal.

5.5 A*-Suche

Die gierige Suche berücksichtigt nicht die Kosten, die bis zum Knoten s bereits entstanden sind. Wir führen daher eine Funktion g ein, die diese Kosten angibt, und eine Funktion h, die die Kosten bis zum Ziel schätzt.
Damit definieren wir die heuristische Bewertungsfunktion f durch

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

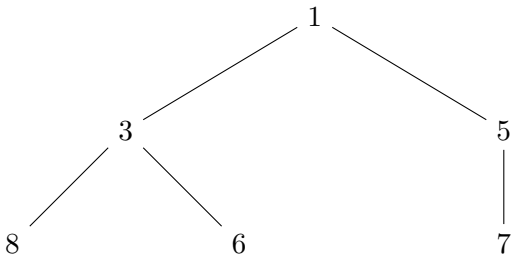
Obige gierige Suche ist der Spezialfall $g(s) = 0, h(s) =$ Luftlinienentfernung bis zum Ziel.

Definition Eine heuristische Kostenschätzfunktion h heißt zulässig, wenn h die Kosten bis zum Ziel nie überschätzt.
D.h. es gilt stets $h(s) \leq$ tatsächliche Entfernung von s bis zum Ziel.

Die A*-Suche ist folgender Algorithmus:
Für den aktuellen Knoten v werden die noch unbesuchten Nachbarn bestimmt und anhand ihrer heuristischen Bewertungsfunktion f, die wie oben konstruiert sein muss, in eine geeignete Datenstruktur eingefügt. In jedem Schritt wird die Suche mit einem Knoten mit minimaler Bewertung weitergeführt.

Naive Implementierung Wie bei der Breitensuche werden die noch zu besuchenden Knoten in einer Liste gespeichert. Diese wird anhand der f-Werte der Knoten sortiert, so dass am Kopf der Liste Knoten mit minimaler Bewertung steht.

Effiziente Implementierung Ein *Min-Heap* ist ein Binärbaum mit der Eigenschaft: Jeder Knoten besitzt einen Wert, der $\leq$ der Werte seiner Nachfolger ist.



Beispiel
Die Heap-Operationen

- Entfernen der Wurzel
- Hinzufügen eines neuen Knotens

benötigen die Laufzeit $\mathcal{O}(\log(n))$

Die A*-Suche ist optimal, vorausgesetzt, die Kostenschätzfunktion h ist zulässig.
Vorteil der A*-Suche: Bei guter Heuristik wird das Ziel oft deutlich schneller gefunden

als mit einer uninformierten Suche.
Nachteil: Wie bei der Breitensuche müssen im Worst-Case alle Knoten im Speicher gehalten werden.

5.6 IDA*-Suche

Ziel Vorteile der A^*-Suche mit denen der iterativen Tiefensuche kombinieren. Dazu führen wir eine Schranke für die heuristische Bewertungsfunktion f ein und erhöhen diese schrittweise.

Anwendung 8-Puzzle: Darstellung als Graph: Knoten: Brettstellungen. Kanten: geben an, wie durch Verschieben eines Plättchens eine neue Brettstellung erreicht werden kann.

Darstellung einer Brettstellung in Prolog:
Jede Zeile ist eine Liste aus drei Elementen, das Brett eine Liste aus drei Zeilen.

Geeignete Heuristiken für die Suche im 8-Puzzle:

1. Hamming-Distanz: Anzahl der Plättchen die falsch stehen. Diese Heuristik ist zulässig, weil mindestens so viele Verschiebungen notwendig sind, wie Plättchen falsch stehen.

2. Manhattan- oder Cityblock-Distanz: Anzahl Verschiebungen die nötig sind, um ein Plättchen auf direktem Weg zum Ziel zu schieben, summiert über alle Plättchen. Diese Heuristik ist zulässig, da für jedes Plättchen mindestens diese Anzahl an Verschiebungen nötig ist.

5.7 Spiele mit Gegner

Wir bewerten Stellungen durch eine Nutzenfunktion. Es gibt zwei Spieler:

- Max: Max versucht, die Nutzenfunktion zu minimieren.
- Min: Min versucht, die Nutzenfunktion zu minimieren.

Einfachster Fall einer Nutzenfunktion:

- 1: Max hat gewonnen
- 0: Unentschieden
- -1: Min hat gewonnen

Die Nutzenfunktion lässt sich im Allgemeinem nur für Endzustände des Spiels berechnen. Da wir nicht wissen, wie der Gegner zieht, betrachten wir alle möglichen Züge es Gegners und nehmen an, dass dieser optimal spielt. Damit berechnen wir einen Nutzwert für jeden Knoten (minimax-value). Dieser ergibt sich als

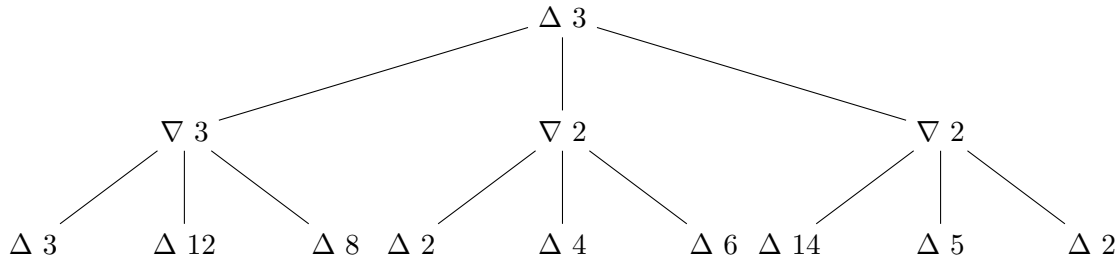
- Maximum der Bewertungen der Nachfolgeknoten, wenn Max am Zug ist.
- Minimum der Bewertungen der Nachfolgeknoten, wenn Min am Zug ist.

Daraus erhalten wir eine rekursive Formel, um minimax zu berechnen:

$$\text{minimax}(n) = \begin{cases} \text{utility}(n) & \text{wenn } n \text{ ein Endzustand ist} \\ \max\{\text{minimax}(s) | s \in \text{succ}(n)\} & \text{wenn } n \text{ ein Max-Knoten ist} \\ \min\{\text{minimax}(s) | s \in \text{succ}(n)\} & \text{wenn } n \text{ ein Min-Knoten ist} \end{cases}$$

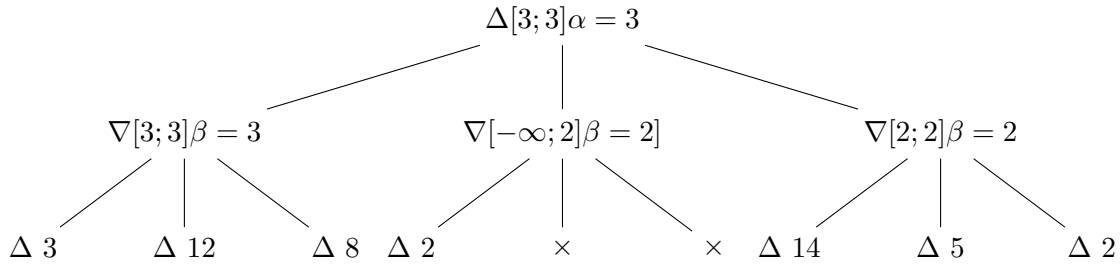
Dabei sei succ (n) die Menge der Nachfolger des Knotens n im Suchbaum.

Beispiel Δ : Max ∇ : Min



Die Minimax-Werte lassen sich mit einer Tiefensuche berechnen. Laufzeit dazu: $\mathcal{O}(b^d)$, wenn der Suchbaum die Tiefe d und den Verzweigungsfaktor b besitzt.
Nachteil dieser Art der Berechnung: Es müssen alle Knoten erzeugt und deren minimax-Werte berechnet werden. Wir können Rechenzeit einsparen wenn Knoten ausgelassen werden, deren minimax-Werte keine Auswirkung auf den aktuellen Knoten haben.

Beispiel Δ : Max ∇ : Min



Im Laufe der Berechnung werden folgende Werte aktualisiert:

- α-Wert: Untere Grenze des Minimax-Wertes eines Max-Knotens. Der α-Wert bleibt gleich oder wird größer.
- β-Wert: Obere Grenze des Minimax-Wertes eines Min-Knotens. Der β-Wert bleibt gleich oder wird kleiner.

In zwei Fällen kann der Suchbaum beschnitten werden:

- Ist der β-Wert eines Min-Knotens $\leq \alpha$-Wert des darüber liegenden Max-Knotens, dann muss der Min-Knoten nicht weiter untersucht werden.

Die Effizienz dieser α-β-Kürzung ist abhängig von der Reihenfolge, in der die Knoten besucht werden.

Vorteilhaft: Zuerst gute Knoten besuchen, z.B.: Schach: Erst Würfe prüfen, dann Bedrohungen, dann Vorwärtszüge, zuletzt Rückwertszüge.
Im besten Fall ist die Laufzeit $\mathcal{O}(b^{\frac{d}{2}})$.

5.8 Bewertungsfunktionen

Für Spiele wie Schach ist auch die Laufzeit der α-β-Suche zu groß. Die Suche wird daher abgebrochen bevor ein Endknoten erreicht wird. Innere Knoten werden dazu mit einer Funktion bewertet, z.B.

$$\#\text{ Bauern} + 3 \cdot \#\text{ Läufer} + 5 \cdot \#\text{ Türme} + 9 \cdot \#\text{ Damen}$$

6 Schließen mit Unsicherheit

6.1 Bayes-Netze

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

$$P(J|Al) = 0,9 \qquad P(M|Al) = 0,7$$
$$P(J|\bar{Al}) = 0,05 \qquad P(M|\bar{Al}) = 0,01$$

Aber auch ein Erdbeben kann den Alarm auslösen:

$$P(Al|Ein, Erd) = 0,95 \qquad P(Al|Ein, \overline{Erd}) = 0,94$$
$$P(Al|\overline{Ein}, Erd) = 0,29 \qquad P(Al|\overline{Ein}, \overline{Erd}) = 0,001$$

Ferner sei $P(Ein) = 0,001; P(Erd) = 0,002$. Ein, Erd seien unabhängig.

6.1.1 Graphische Darstellung als Bayes-Netz

Wenn John und Mary unabhängig auf einen Alarm reagieren gilt

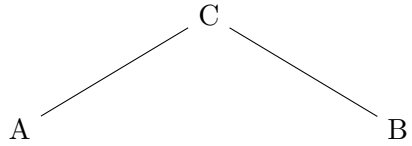
$$P(J, M|Al) = P(J|Al) \cdot P(M|Al)$$

Wenn wir annehmen, dass John nicht auf einen Einbruch, sondern nur auf einen Alarm reagiert, gilt ferner

$$P(J, Ein|Al) = P(J|Al) \cdot P(Ein|Al)$$

6.1.2 Semantik von Bayes-Netzen

Kanten beschreiben (bedingte) Unabhängigkeit von Ereignissen



Ein Bayes-Netz muss ein DAG (directed acyclic graph) sein. Dann können die Knoten topologisch sortiert werden. Dann gilt

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

Es folgt

$$\begin{aligned}
&P(A_1,\ldots,A_n) = \\
&P(A_n|A_1,\ldots,A_{n-1}) \cdot P(A_1,\ldots,A_{n-1}) &&= \\
&P(A_n|A_1,\ldots,A_{n-1}) \cdot P(A_{n-1}|A_1,\ldots,A_{n-2}) \quad \cdot P(A_1,\ldots,A_{n-2}) &&= \\
&\vdots \\
&\prod_{i=1}^{n} P(A_i|A_1,\ldots,A_{i-1}) = \\
&\prod_{i=1}^{n} P(A_i|\text{Eltern}(A_i)) \\
&(\text{wobei} P(A_1|A_1,\ldots,A_0) := P(A_1))
\end{aligned}$$

ÜA: Berechne P(J,Al,Ein,Erd)

UÄ: Berechne P(J,Ein): Für diskunkte Ereignisse $A_1,\ldots,A_n$ gilt

$$P(\sum_{i=1}^{n} A_i) = \sum_{i=1}^{n} P(A_i)$$

Diskunkte Zerlegung:

$$P(A) = P(A \cap B + A \cap \bar{B}) = P(A,B) + P(A,\bar{B})$$

$P(J,Ein) = P(J,\bar{Al},Ein,\bar{Erd}) + P(J,\bar{Al},Ein,Erd) + P(J,Al,Ein,\bar{Erd}) + P(J,Al,Ein,Erd) = 0,000849$

Ebenso: $P(J) = 0,052$

Damit ergibt sich $P(Ein|J) = \frac{P(Ein,J)}{P(J)} = 0,016$

Ebenso $P(Ein|J,M) = \frac{P(Ein,J,M)}{P(J,M)} = 0,284$

6.2 NP-hart

Eine Sprache L heißt NP-vollständig, wenn:

- L $\in$ NP
- L ist NP-hart

NP-hart bedeutet, dass jedes Problem in NP effizient übersetzt werden kann in ein Entscheidungsproblem für L.
Die exakte Inferenz in Bayes'schen Netzen ist NP-hart.

Eines der klassischen NP-vollständigen Probleme ist 3KNF-SAT = $\{F|F$ ist eine Formel in konjunktiver Normalform(KNF) mit 3 literalen pro Klausel und F ist erfüllbar $\}$.

Beispiel Eine Formel in 3KNF-SAT:

$$(A \vee B \vee \neg C) \wedge (\neg A \vee B \vee \neg D)$$

Ein Algorithmus für die exakte Inferenz in BAYES'schen Netzen (Berechnung von Wahrscheinlichkeiten) lässt sich nutzen, um 3KNF-SAT zu entscheiden.

Beispiel $F = (A \vee B \vee C) \wedge (C \vee D \vee \neg A) \wedge (B \vee C \vee \neg D)$
Aus dieser Formel lässt sich folgendes BAYES'sches Netz erzeugen:

Der Knoten 2 besitzt folgende bedingte Wahrscheinlichkeiten:

C	D	A	P(2)
f	f	f	1
f	f	w	0
f	w	f	1
f	w	w	1
w	f	f	1
w	f	w	1
w	w	f	1
w	w	w	1

Der Knoten $\wedge$ bestitzt die bedingten Wahrscheinlichkeiten:

1	2	3	P($\wedge$)
f	f	f	0
f	f	w	0
	$\vdots$		0
w	w	w	1

Dann gilt: F ist erfüllbar $\Leftrightarrow P(\wedge) > 0$.

Da die exakte Inferenz in BAYES'schen Netzen NP-hart ist, gibt keinen effizienten (d.h. polynomiellen) Algorithmus zur Berechnung der Wahrscheinlichkeiten. Ausweg: Approximative Inferenz.

Gegeben eine Verteilung, wie kann man Stichproben daraus ziehen?

Beispiel Würfel: 6 Ausfälle, Verteilung ist $(\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6})$
Wir ziehen eine Zufallszahl gleichverteilt in $[0, 1]$. Das Intervall $[0, 1]$ wird nun unterteilt in 6 Intervalle, die dem Würfel der Zahlen 1-6 entsprechen.

6.3 Algorithmus direktes Sampling

Vorraussetzung Netz ist topologisch sortiert.

```
1 for i = 1 to n
2   Erzeuge einen Wert fuer
3   P(X_i | Eltern(X_i))
4   gemaess der bedingten Verteilung von X_i,
5   gegeben die bereits erzeugten Werte fuer Eltern (X_i).
```

Auf diese Weise wird eine Stichprobe $(x_1, \ldots, x_n)$ erzeugt.

Korrektheit des Verfahrens $P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(X_i | \text{Eltern}(X_i))$
Dies ist die Wahrscheinlichkeit der Verteilung gemäß Formel aus letzter Vorlesung.

6.4 Berechnen von bedingten Wahrscheinlichkeiten

6.4.1 Ablehnungssampling

```
1 for i = 1 to k
2   Erzeuge eine Stichprobe x mit dem
3   Algorithmus direktes Sampling
4   Verwerfe x, wenn x nicht konsistent ist
5   mit der Bedingung e, fuer die P(X | e) berechnet
6   werden soll
7 Berechne, wie oft das Ereignis X in den nicht verworfenen Samples vorkommt.
```

Der Algorithmus Ablehnungs-Sampling ist ungeeignet, wenn $P(e)$ klein ist. Für $e = e_1, \ldots, e_n$ gilt insbesondere, dass $P(e)$ exponentiell kleiner wird.