Problemzerlegung

- Zerlege ein Problem P in einzelne Probleme $P_1,...,P_n$
- Löse jedes Problem P_i
- Füge die Lösungen zusammen zu P

Beispiele:
- Ungarischer Würfel
- Kurvendiskussion
- Integralrechnung

Problemzerlegung

Klausel als Problemzerlegung

$goal(X_1,...,X_n)$:- $subgoal_1(X_1,...,X_n)$,..., $subgoal_m(X_1,...,X_n)$.

„goal“ gilt (ist beweisbar)
falls alle „subgoals“ gelten (beweisbar sind).

um „goal“ zu beweisen,
beweise alle „subgoals“

Problemzerlegung

goal(X_1,...,X_n) :- subgoal$_1$(X_1,...,X_n) ,..., subgoal$_m$(X_1,...,X_n).

um „goal“ zu beweisen,
beweise alle „subgoals“

Problemzerlegung

```
erreichbar(Start,Ziel,Zeit)
:- s_bahn(Start,Zwischenziel,Abfahrt,Ankunft,_),
   erreichbar(Zwischenziel,Ziel,Zeit1),
   berechneZeit(Zeit1,Ankunft,Abfahrt,Zeit).
```


Problemzerlegung

```
berechneZeit(Zeit1,Ankunft,Abfahrt,Zeit).
```

Beabsichtigte Bedeutung:
Zeit = Zeit1 + (Ankunft - Abfahrt)

Problem:Spezielle Notation für Zeitangaben beachten

Problemzerlegung:

```
berechneZeit(Zeit1,Ankunft,Abfahrt,Zeit)
:- addiereZeit(Zeit1,Differenz,Zeit),
   subtrahiereZeit(Ankunft,Abfahrt,Differenz).
```

Mit geeignet definierten Prädikaten
`addiereZeit/3, subtrahiereZeit/3`

Unterschied zu prozeduralem Denken

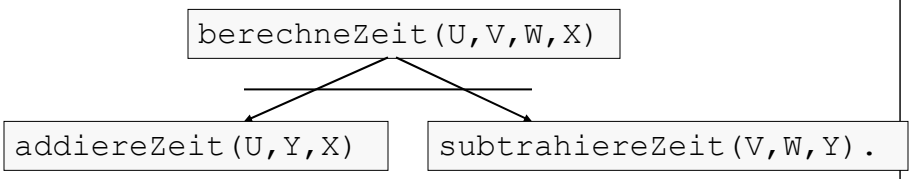
```
berechneZeit(Zeit1,Ankunft,Abfahrt,Zeit)
:- addiereZeit(Zeit1,Differenz,Zeit),
   subtrahiereZeit(Ankunft,Abfahrt,Differenz).
```

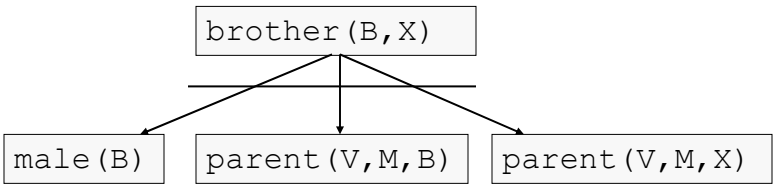
Logische Sicht:
Berechnung von „Differenz“ kann später erfolgen
(Bindung statt Wertzuweisung).

Die in Standard-Prolog eingebaute Arithmetik erfordert aus Effizienzgründen allerdings gebundene Eingangsparameter.

Problemzerlegung

Graphische Darstellung





Alternativen für Problemzerlegung

Zerlegung des Problems P in Probleme $P_1,...,P_n$

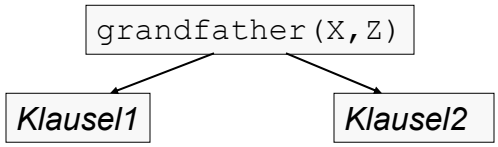
oder in Probleme $P'_1,...,P'_{n'}$

oder ...

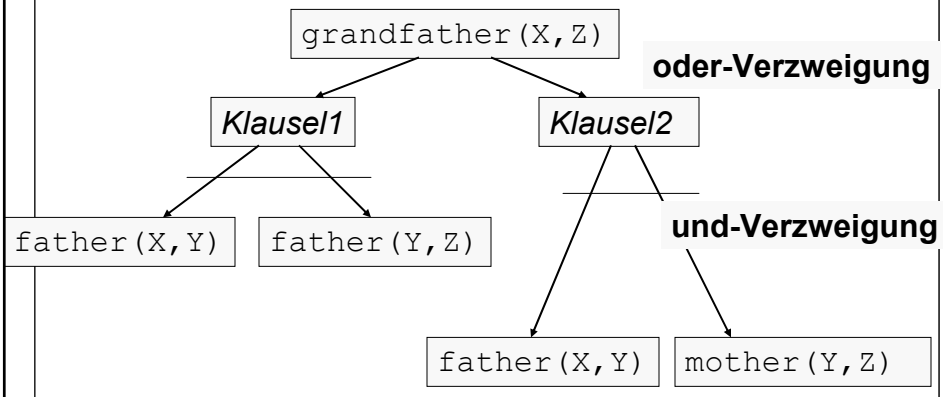
Klauseln einer Prolog-Prozedur bieten Alternativen

```
grandfather(X,Z):-father(X,Y),father(Y,Z).
grandfather(X,Z):-father(X,Y),mother(Y,Z).
```

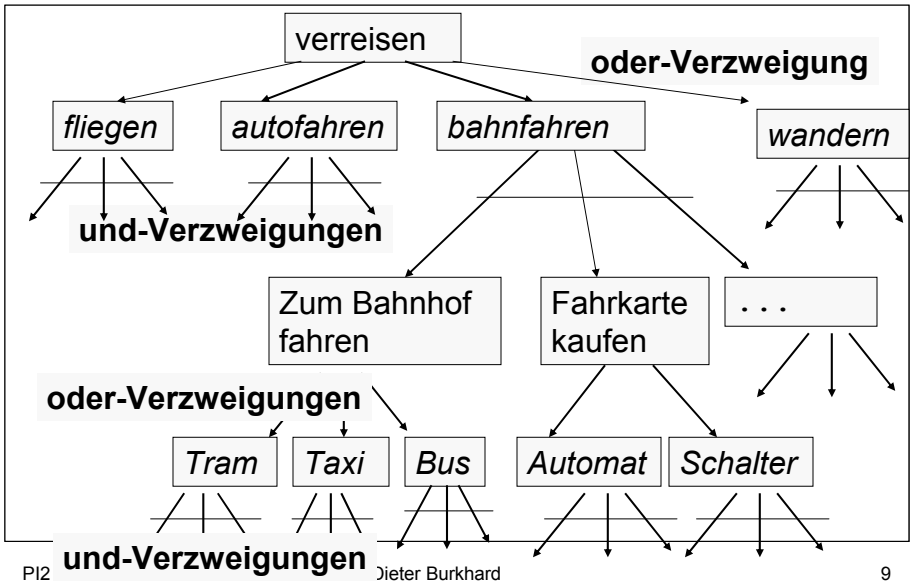
Graphische Darstellung



Alternativen für Problemzerlegung



Kombinierte Verzweigungen



Und-Oder-Baum

- Ein und-oder-Baum besteht (abwechselnd) aus
  - Knoten mit oder-Verzweigungen und
  - Knoten mit und-Verzweigungen
- Modell für Problemzerlegungen:
  - oder-Verzweigungen für alternative Möglichkeiten zur Problemzerlegung
  - und-Verzweigungen für Teilprobleme
- Modell für Prolog-Programm:
  - oder-Verzweigungen für alternative Klauseln einer Prozedur
  - und-Verzweigungen für subgoals einer Klausel

Und-Oder-Baum

Anfrage

Startknoten („**Wurzel**“) modelliert Ausgangsproblem

Knoten ohne Nachfolger („**Blätter**“) sind unterteilt in
- terminale Knoten („primitive Probleme“)
  modellieren unmittelbar lösbare Probleme
- nichtterminale Knoten
  modellieren nicht zu lösende Probleme

Fakt

Unerfüllbares Subgoal

(keine unifizierende Klausel)

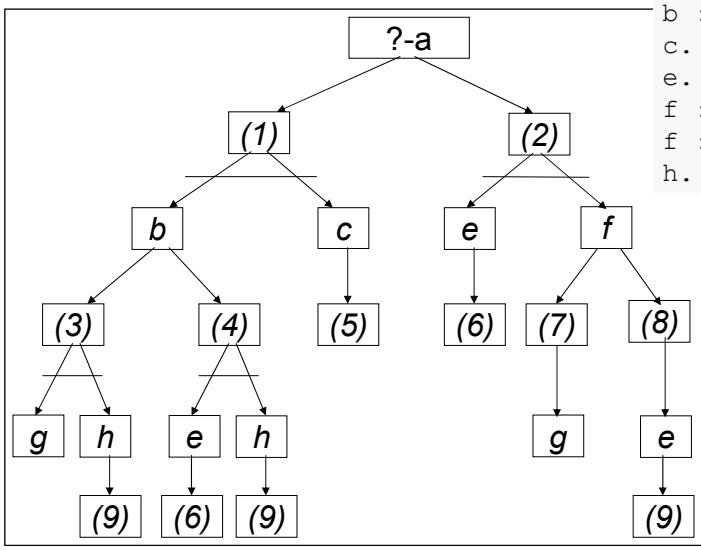
Innere Knoten sind unterteilt in
- Knoten mit und-Verzweigung
- Knoten mit oder-Verzeigung

Und-Oder-Baum

```
a :- b,c.        % (1)
a :- e,f.        % (2)
b :- g,h.        % (3)
b :- e,h.        % (4)
c.               % (5)
e.               % (6)
f :- g.          % (7)
f :- e           % (8)
h.               % (9)
```

Und-Oder-Baum

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



Lösbare/unlösbare Knoten

Lösbare Knoten:

1. terminale Knoten,
2. Knoten mit Und-Verzweigung:
   falls alle Nachfolger lösbar sind
3. Knoten mit Oder-Verzweigung:
   falls mindestens ein Nachfolger lösbar ist

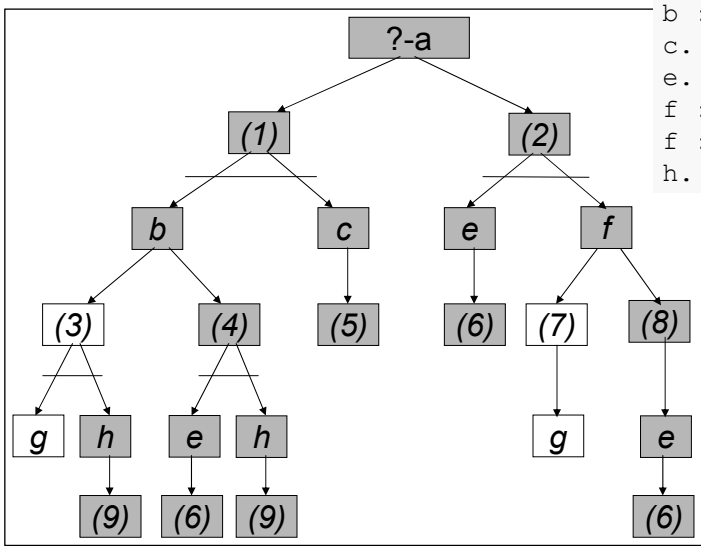
Unlösbare Knoten:

1. nichtterminale Knoten,
2. Knoten mit Und-Verzweigung: falls mindest. ein Nachfolger unlösbar,
3. Knoten mit Oder-Verzweigung: falls alle Nachfolger unlösbar sind

Für endliche Bäume ergibt sich bei Festlegung für die Blätter eine eindeutige Zerlegung in lösbare/unlösbare Knoten (Bedingungen sind jeweils komplementär)

Lösbare/unlösbare Knoten

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



Bottom-up-Konstruktionsalgorithmus

Anfang: $M_{LÖSBAR}$:= terminale Knoten
$M_{UNLÖSBAR}$:= nichtterminale Knoten

Zyklus:
Solange nicht alle Knoten untersucht wurden:
Wähle einen Knoten k, dessen Nachfolger alle untersucht wurden.
Falls bei k und-Verzweigung und alle Nachfolger von k in $M_{LÖSBAR}$ oder falls bei k oder-Verzweigung und ein Nachfolger von k in $M_{LÖSBAR}$:

$$M_{LÖSBAR} := M_{LÖSBAR} \cup \{k\} \ ,$$

andernfalls:

$$M_{UNLÖSBAR} := M_{UNLÖSBAR} \cup \{k\} \ .$$

Bottom-up-Konstruktionsalgorithmus

Ergebnis für endliche Und-oder-Bäume:
Zerlegung der Knoten in

- lösbare Knoten ($M_{LÖSBAR}$) und
- unlösbare Knoten ($M_{UNLÖSBAR}$)

Das Ausgangsproblem
kann genau dann durch Problemzerlegung gelöst werden,
wenn der Startknoten in $M_{LÖSBAR}$ ist.

Falls das Ausgangsproblem lösbar ist, kann ein <u>Lösungsbaum</u> konstruiert werden.

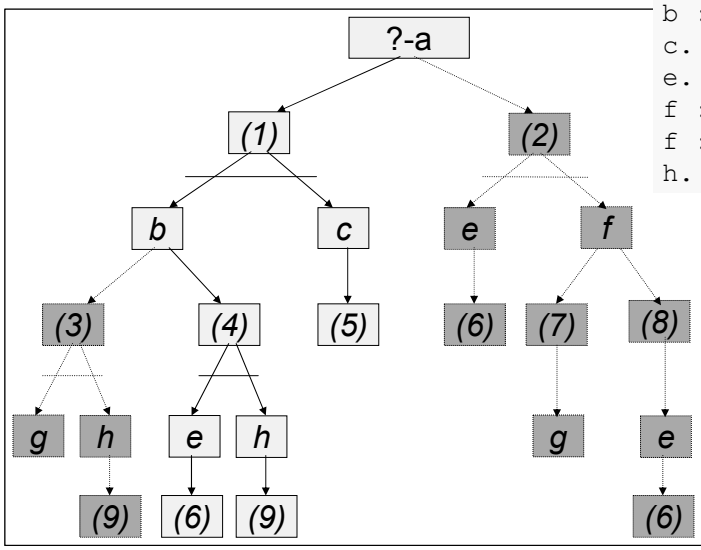
Lösungsbaum

Endlicher Teilbaum des Und-oder-Baums mit
folgenden Eigenschaften:
- Enthält nur lösbare Knoten,
- enthält Wurzelknoten,
- bei Und-Verzweigungen sind alle Nachfolger enthalten,
- bei Oder-Verzweigungen ist (genau) ein Nachfolger enthalten

Modell für „Beweisbaum“ in PROLOG

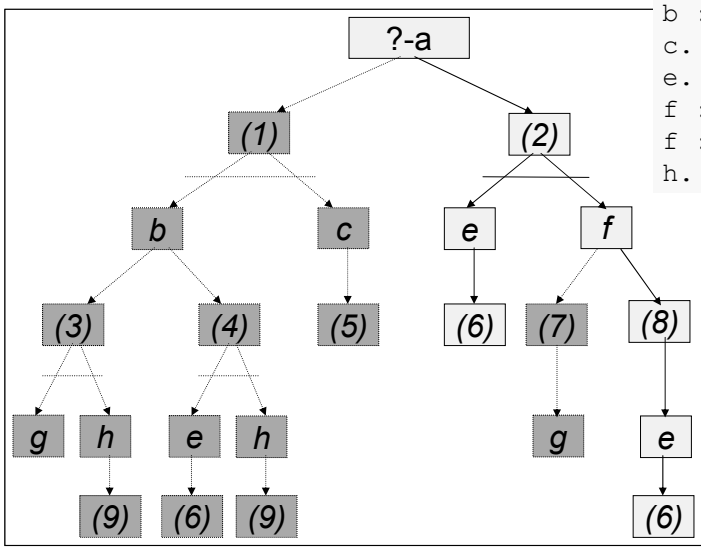
Lösungsbäume

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```

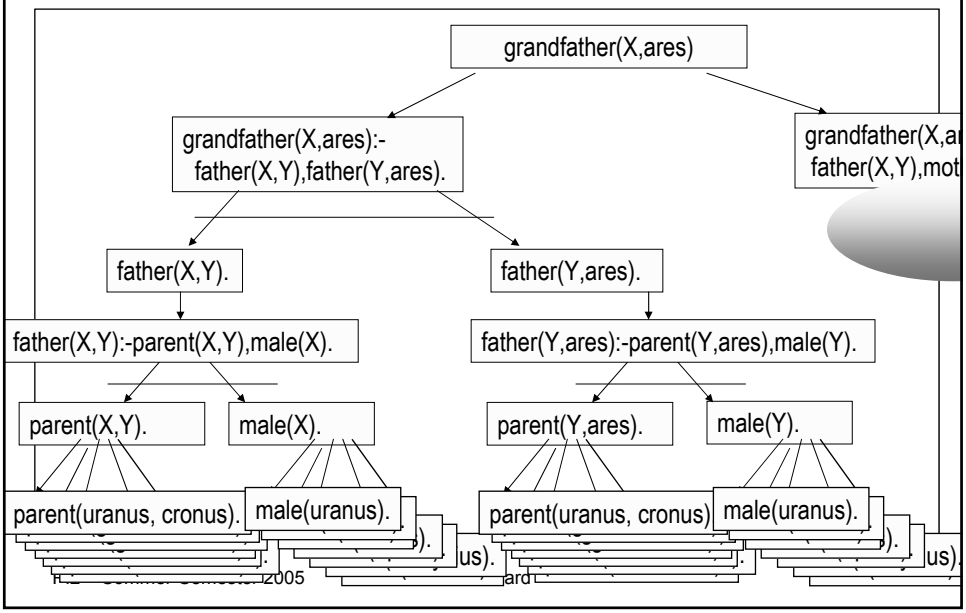


Lösungsbäume

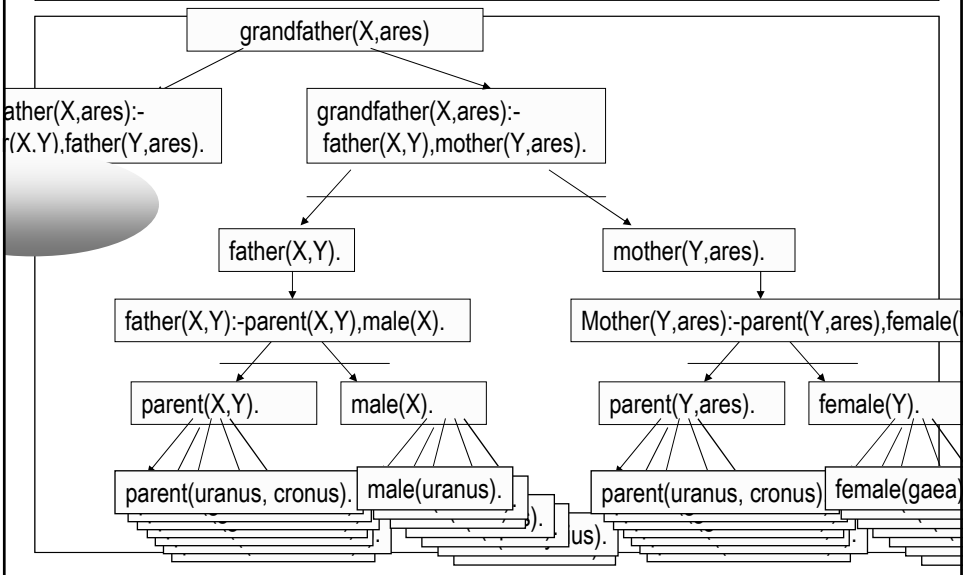
```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



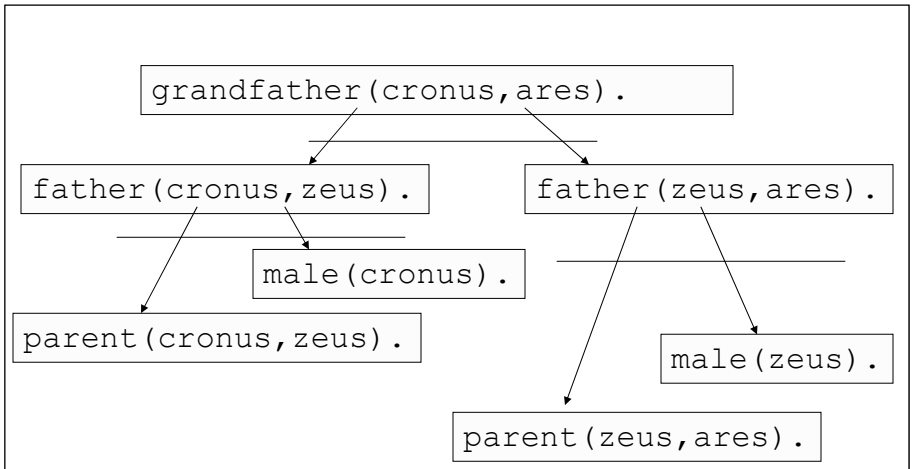
Und-oder-Baum modelliert Beweisversuche



Und-oder-Baum modelliert Beweisversuche



Lösungsbaum (Beweisb.) modelliert Beweis



Modell für nicht-deterministische Suche

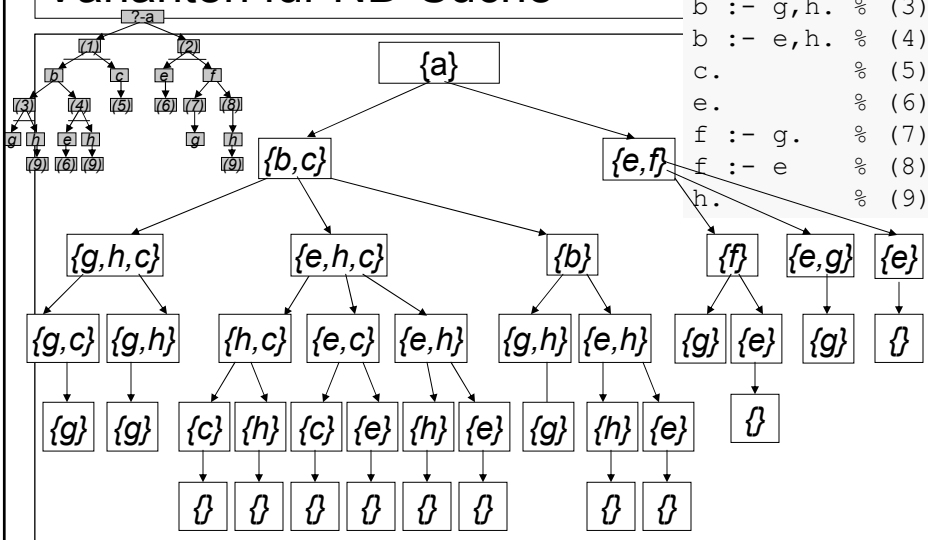
Vereinfachtes Schema (ohne Unifikation: „AK“)

1. subgoals=$\{g_1,...,g_n\}$
2. Falls subgoals = $\emptyset$: Erfolg.
3. Wähle g $\in$ subgoals .
4. Wähle Klausel k: g :- $g'_1,...,g'_m$ der Prozedur für g .
   Falls kein solches k existiert: Mißerfolg (des Versuchs).
5. subgoals := $(\{g_1,...,g_n\} - \{g\}) \cup \{g'_1,...,g'_m\}$.
   Weiter bei 2.

Alle Varianten (in 3 und 4) probieren,
falls keine zum Erfolg führt: „Nicht beweisbar“

Varianten für ND-Suche

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



Alle Varianten (**jeden Weg von der Wurzel aus**) probieren, falls keine zum Erfolg führt: „Nicht beweisbar“

Prolog-Interpreter

Einschränkung der Varianten

- Reihenfolge innerhalb einer Klausel (Und-Verzweigung)

  (alle subgoals müssen erfüllt werden)

  links vor rechts
- Reihenfolge innerhalb einer Prozedur (Oder-Verzweigung)

  (Alternativen für Beweis)

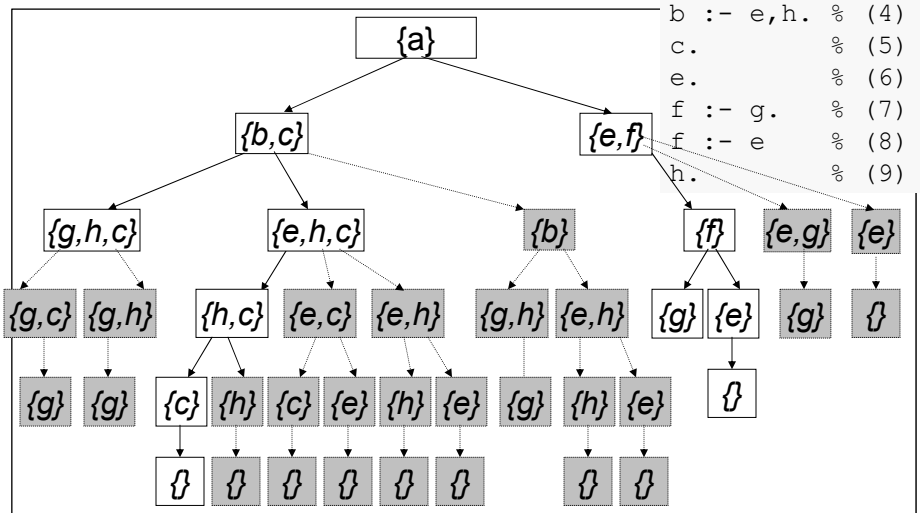
  oben vor unten

Zu zeigen wäre (Vollständigkeit):

Wenn Beweis existiert, dann auch schon hierbei.

Einschränkung der Varianten

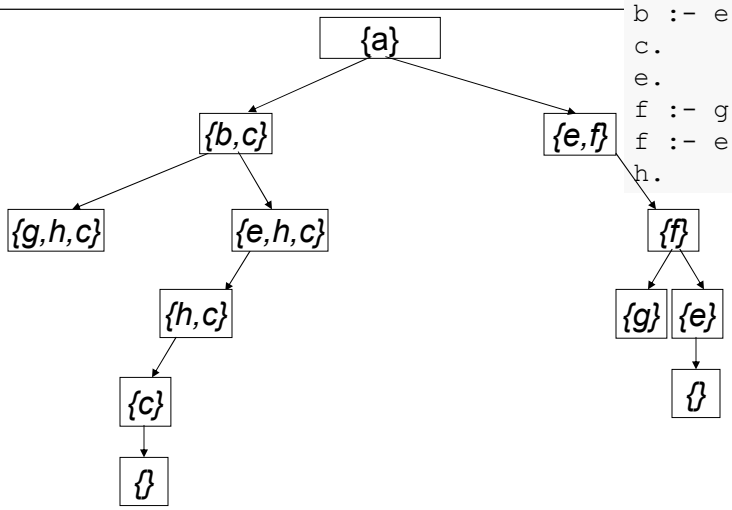
```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



Subgoals sind alle zu beweisen,
Reihenfolge links vor rechts

Einschränkung der Varianten

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



Subgoals sind alle zu beweisen,
Reihenfolge links vor rechts

Backtracking

Effizienzgewinn

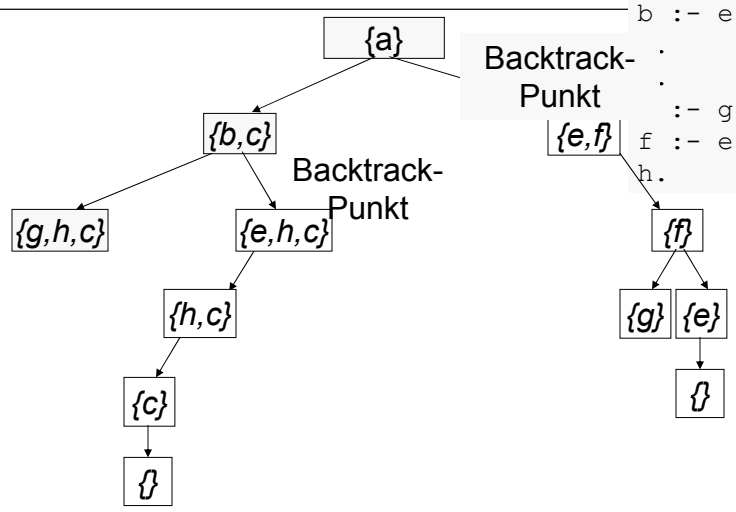
- Suchpfade werden nicht vollständig neu probiert, sondern nur stückweise.

- Bei Alternativen:
- Einfügen eines „Backtrack-Punktes“ („Choicepoint“)

- „Backtracking“:
  Bei Fehlschlag am jüngsten „Backtrack-Punkt“ andere Alternative verfolgen

Backtracking

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
.          % (5)
.          % (6)
  :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



Modelle für Prolog-Suche

Vereinfachtes Schema (ohne Unifikation: „AK“)

Liste

1. subgoals=[g_1,...,g_n] .
2. Falls subgoals = [ ] : Erfolg .
3. k sei nächste Klausel der Prozedur für g_1 :

   g_1 :- g'_1,...,g'_m .

   Falls kein solches k existiert: Backtracking.

   Falls kein Backtracking möglich: Misserfolg.
4. subgoals := [g'_1,...,g'_m ,g_2,...,g_n,].

   Weiter bei 2.

Prolog-Interpreter

- Und-Verzweigung: links vor rechts
- Teilziele der Reihe nach vollständig abarbeiten

  Verfolgen eines Zweiges in die Tiefe
- Oder-Verzweigung: oben vor unten

  Linke Zweige zuerst
- Backtracking bei Fehlschlag:

  Rückkehr zu Alternative an oder-Verzweigung

  Nächster Zweig einer Oder-Verzweigung

Modelle für Prolog-Suche

1. subgoals=[g_1,...,g_n] .
2. Falls subgoals = [ ] : Erfolg .
3. k sei nächste Klausel der Prozedur für g_1 :

   g_1 :- g'_1,...,g'_m .

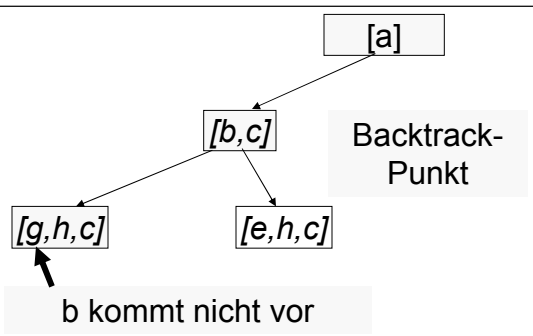
   Falls kein solches k existiert: Backtracking.

   Falls kein Backtracking möglich: Misserfolg.
4. subgoals := [g'_1,...,g'_m ,g_2,...,g_n,].

   Weiter bei 2.

Problem: Backtrack-Punkt in subgoal –Liste nicht repräsentiert

Modelle für Prolog-Suche

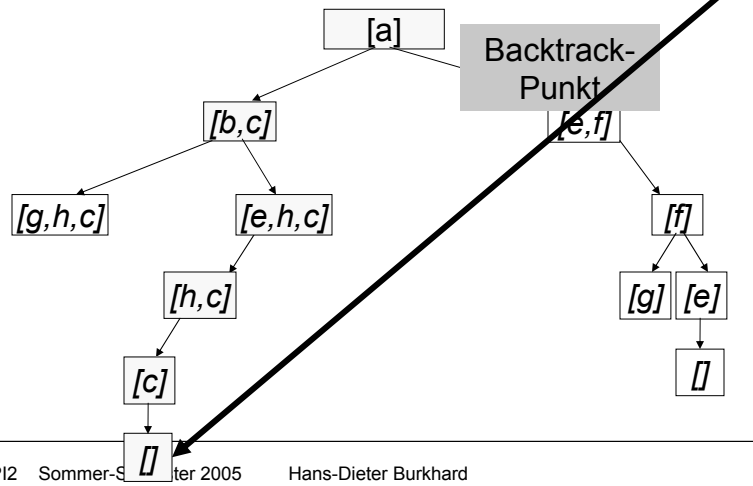


Problem: Backtrack-Punkt in subgoal –Liste nicht repräsentiert

Modelle für Prolog-Suche

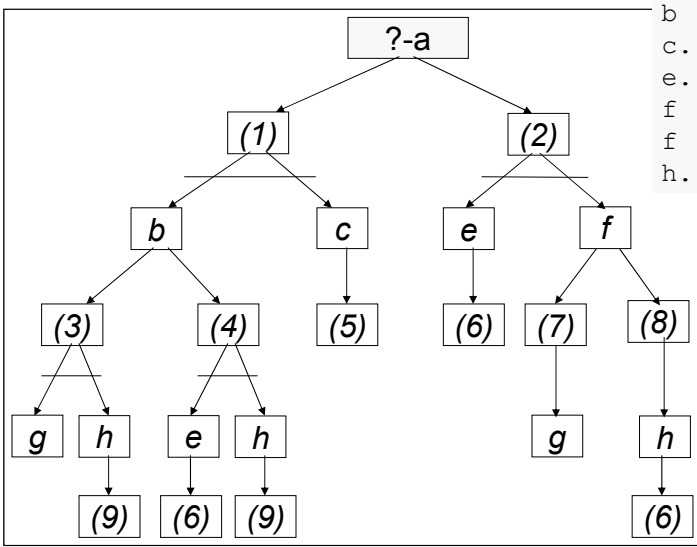
Quelle für Missverständnisse:

Bei erfolgreichem ersten Beweis ist subgoal-Liste leer



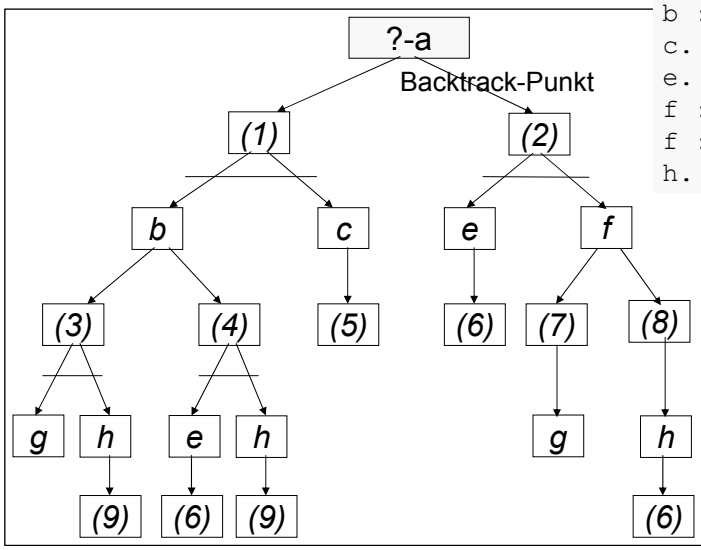
Abarbeitung im Und-oder-Baum

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



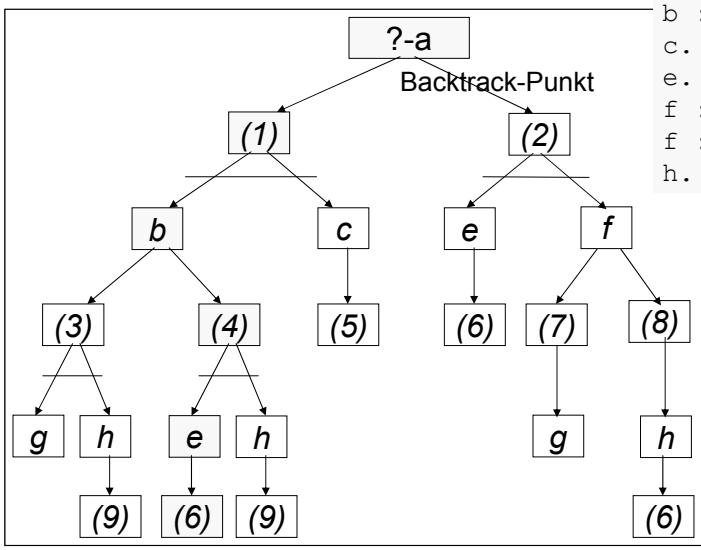
Abarbeitung im Und-oder-Baum

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



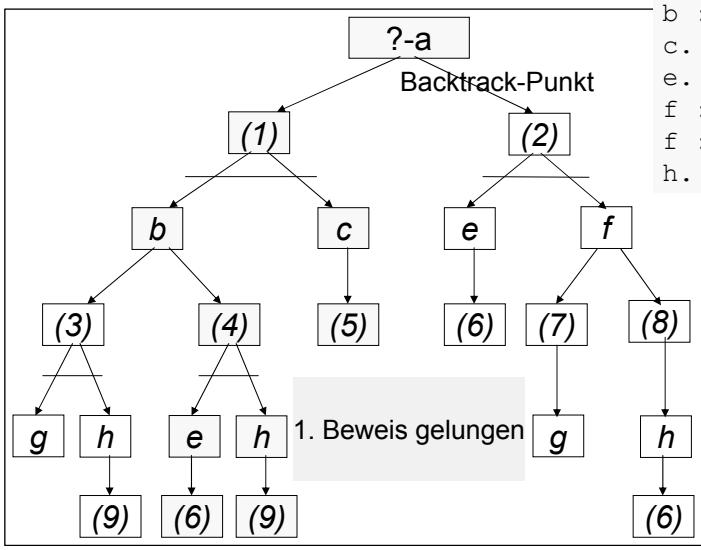
Abarbeitung im Und-oder-Baum

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



Abarbeitung im Und-oder-Baum

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



Algorithmus für systematische Suche

PROCEDURE solve(unsolved_goals: GOALLIST);

goals = $[g_1,\ldots,g_n]$ — Bezeichnet Liste von goals g_i

top(goals) = g_1 — Bezeichnet erstes Element

tail(goals) = $[g_2,\ldots,g_n]$ — Bezeichnet Rest-Liste

NIL = [ ] — Bezeichnet leere Liste

concatenate($[g_1,\ldots,g_n]$, $[g'_1,\ldots,g'_m]$) = [$g_1,\ldots,g_n$, $g'_1,\ldots,g'_m$]

Bezeichnet Verkettung von Listen

Algorithmus für systematische Suche

PROCEDURE solve(unsolved_goals: GOALLIST);

unsolved_goals — Liste ungelöster subgoals

klauseln(g) — Klauseln der Prozedur für g

```
ackermann(o,N,s(N)).
ackermann(s(M),o,V):- ackermann(M,s(o),V).
ackermann(s(M),s(N),V):- ackermann(s(M),N,V1), ackermann(M,V1,V).
```

subgoals(k) — Subgoals der Klausel k

```
ackermann(s(M),s(N),V):- ackermann(s(M),N,V1), ackermann(M,V1,V).
```


Algorithmus für systematische Suche

```
PROCEDURE solve(unsolved_goals: GOALLIST);
                    VAR k:KLAUSEL, g: GOAL;
BEGIN
  IF unsolved_goals = NIL THEN HALT(yes)
  ELSE g:= top(unsolved_goals);
    FORALL k ∈ klauseln(g) DO
                    (* Klauseln für g nacheinander rekursiv probieren*)
       solve(concatenate(subgoals(k),tail(unsolved_goals)))
                    (* subgoals der Klausel k weiter verfolgen *)
    END (*FORALL*)
  END (*IF*)
END solve;
```

Backtrack-Punkte

Reihenfolge: oben vor unten

Reihenfolge: links vor rechts

Aufruf mit solve([goal]); HALT(no).

Algorithmus für systematische Suche

Rekursive Prozedur-Aufrufe mit Subgoal-Listen.

Bei leerer Subgoalliste:

- Resultat „yes“
- Abbruch der Prozedur-Kette.

Nach vollständiger Abarbeitung einer Aufruf-Kette Rückkehr zur jüngsten Möglichkeit gemäß FORALL ... (Backtracking).

Wenn alle (FORALL-)Varianten erfolglos versucht:

- Resultat „no“
- Prozedur-Ketten vollständig abgearbeitet.

Algorithmus für systematische Suche

Algorithmus verwendet eigentlich zwei Listen
(gemäß LIFO-Prinzip: Keller/stacks)

- Liste unsolved_goals (offene Teilziele)

- Liste der offenen Prozedur-Aufrufe (Prozedurkeller) mit Alternativen für Backtracking'

Betrachtung als verschachtelte Listen

$\mathcal{L} = [\ L_1,\ldots,L_m\ ]$

$= [\ [g_{11}, \ldots, g_{1n}]\ , \ldots, [g_{i1}, \ldots, g_{i,ni}], \ldots\ , [g_{m1}, \ldots, g_{m,nm}]\ ]$

Algorithmus für systematische Suche

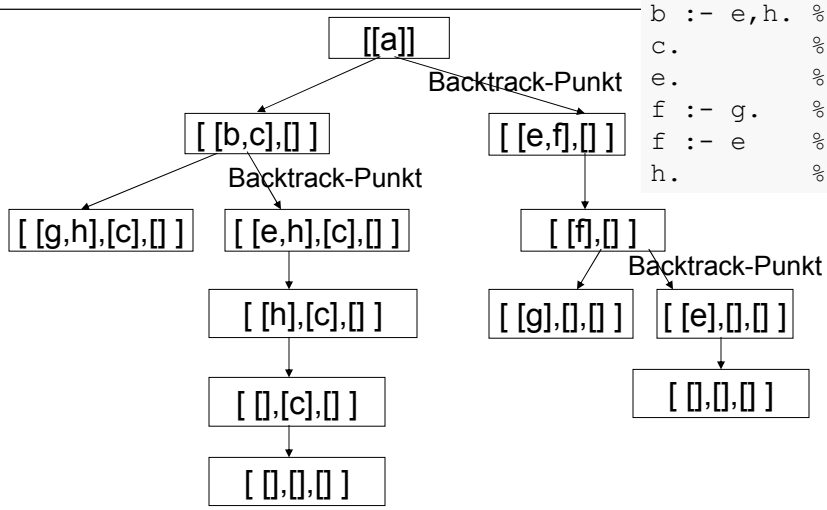
(0) (Start) $\mathcal{L}$: = [[Ausgangsproblem(e)]] .
(1) Falls $\mathcal{L}$ = [[ ], ...,[ ]] : EXIT(yes) .
(2) Sei L_i erste nicht-leere Subgoal-Liste aus $\mathcal{L}$ = [L_1,...,L_m] .
Sei g_{i1} erstes Element aus L_i = [g_{i1}, ..., $g_{i,ni}$] :

- g_{i1} aus L_i entfernen: L'_i := [g_{i2}, ..., $g_{i,ni}$] .
- Falls keine Klauseln für g_{i1}existieren: weiter bei **(4)** .

(3) Sei k die nächste abzuarbeitende Klausel für g_{i1} .
Falls k Fakt: weiter bei **(1)** .
Falls k Regel: g_{i1} :- g_1, ..., g_n :
$\mathcal{L}$:= [[g_1, ..., g_n], L_1,.., L'_i,..., L_m] .
Falls weitere Klauseln für g_{i1} existieren:
Backtrack-Punkt setzen. Weiter bei **(2) .**
(4) Backtracking: Rücksetzen zum jüngsten *Backtrack-Punkt* :
$\mathcal{L}$ zurücksetzen auf Stand vor *Backtrack-Punkt*, weiter bei **(3)** .
Falls kein *Backtrack-Punkt* existiert: $\mathcal{L}$ =[ ] , EXIT(no).

Algorithmus für systematische Suche

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



Algorithmus für systematische Suche

Alternative:
Gelöste subgoals markieren

```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```