Einführung in die KI

Prof. Dr. sc. Hans-Dieter Burkhard
Vorlesung Winter-Semester 2004/05

Suchverfahren-Teil2:
- Problemzerlegung
- Spielbäume

1.4 Suche in Und-oder-Bäumen

Problemzerlegung
Und-oder-Baum
Lösungsbaum
Lösungsverfahren
Umformung in Zustandsraum-Suche

Beschränkung auf Bäume!

Problemzerlegung

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

Beispiele:
- Ungarischer Würfel
- Kurvendiskussion
- Integralrechnung
- Prolog: Klausel -> Subgoals
- Agenten: Verteiltes Problemlösen

Prolog: Klausel als Problemzerlegung

$\text{goal}(X_1,\ldots,X_n)$:- $\text{subgoal}_1(X_1,\ldots,X_n)$,..., $\text{subgoal}_m(X_1,\ldots,X_n)$.

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

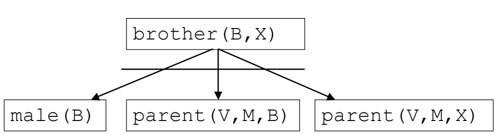
```
erreichbar(Start,Ziel,Zeit)
:- s_bahn(Start,Zwischenziel,Abfahrt,Ankunft,_),
   erreichbar(Zwischenziel,Ziel,Zeit1),
   addiereZeit(Zeit1,Ankunft,Abfahrt,Zeit).
```

Rekursion: wiederholte Problemzerlegung

Problemzerlegung

Graphische Darstellung

```
brother(B,X)
:- male(B), parent(V,M,B), parent(V,M,X)
```



Abhängigkeit von Teilproblemen

Unabhängige Teilprobleme
- Spielzüge eines Gegners, Seminarscheine, ...

Abhängige Teilprobleme
- Inhaltlich:
  - Bindungen von Variablen in Prolog-Klauseln,
  - Ressourcen, ...
- Zeitlich:
  - Fertigungsschritte,
  - Reihenfolge der Teil-Verhalten beim Doppelpass,..

Alternativen für Problemzerlegung

Zerlegung des Problems P in Probleme $P_1,\ldots,P_n$

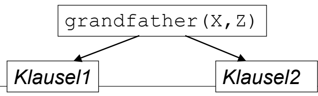
oder in Probleme $P'_1,\ldots,P'_{n'}$

oder ...

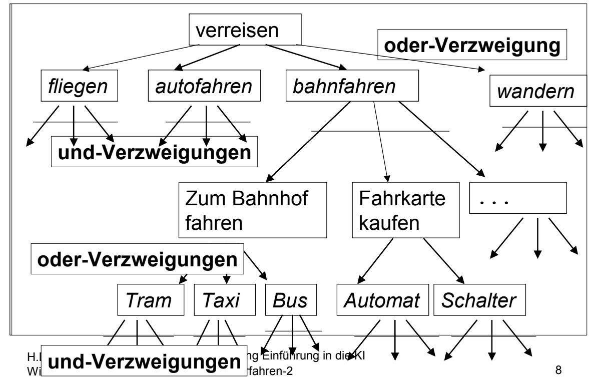
Klauseln einer Prolog-Prozedur bieten Alternativen

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

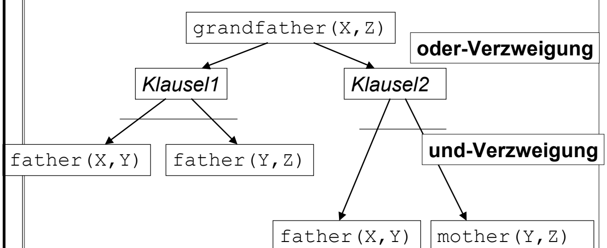
Graphische Darstellung



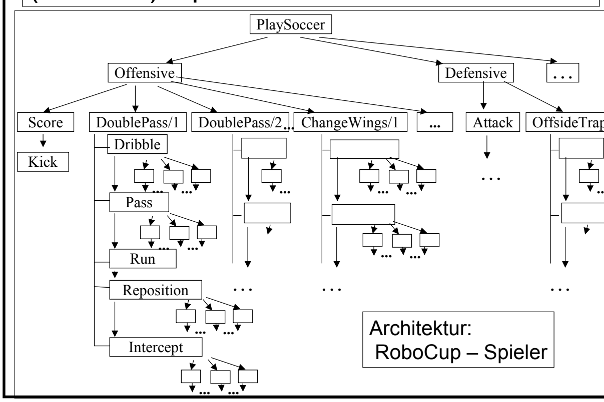
Kombinierte Verzweigungen



Kombinierte Verzweigungen

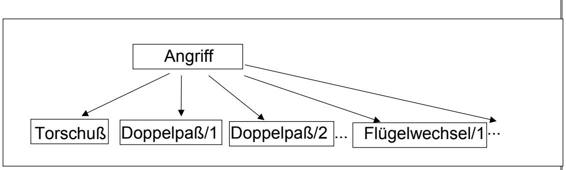


(Virtuelle) Optionen Hierarchie



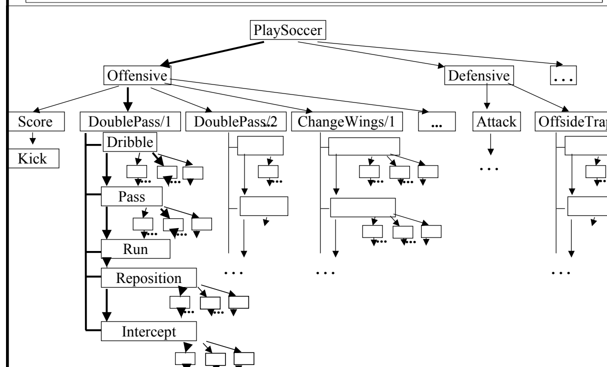
Oder-Verzweigungen

Auswahlmöglichkeiten in der Optionenhierarchie



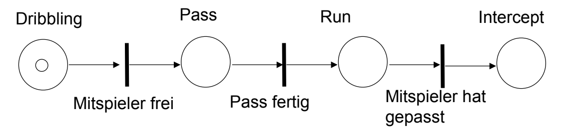
Deliberator trifft Auswahl auf allen Ebenen

Intention Subtree (Resultat des Deliberators)



Und-Verzweigungen

Ablauf komplexer Optionenen



Weiteres Problem: „Least commitment“:

Konkrete Werte erst spät festlegen

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

Fakt

- nichtterminale Knoten modellieren nicht zu lösende Probleme

Unerfüllbares Goal

(keine unifizierende Klausel)

Innere Knoten sind unterteilt in

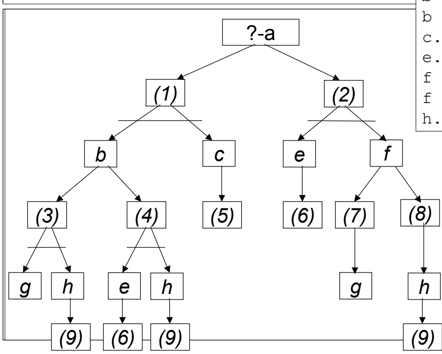
- Knoten mit und-Verzweigung — Subgoals einer Klausel
- Knoten mit oder-Verzeigung — Alternative Klauseln

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

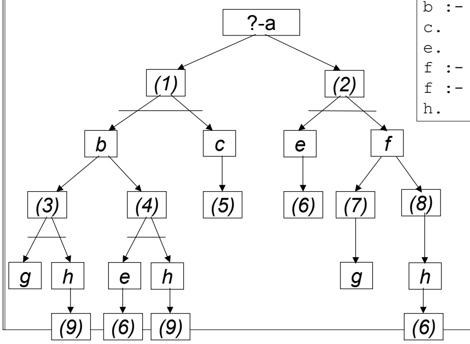
Unlösbare Knoten:

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 Knoten k, dessen Nachfolger alle untersucht wurden.
Falls k und-Verzweigung und alle Nachfolger von k in $M_{LÖSBAR}$
oder falls k oder-Verzw. und ein Nachfolger von k in $M_{LÖSBAR}$:

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

andernfalls:

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

Bottom-up-Konstruktionsalgorithmus

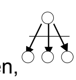
Ergebnis für endliche Und-oder-Bäume:
Zerlegung der Knoten in
lösbare Knoten ($M_{LÖSBAR}$) und
unlösbare Knoten ($M_{UNLÖSBAR}$)

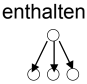
Das Ausgangsproblem
kann genau dann durch Problemzerlegung gelöst werden,
wenn der Startknoten in $M_{LÖSBAR}$ ist.

Falls Ausgangsproblem lösbar ist, kann ein Lösungsbaum 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

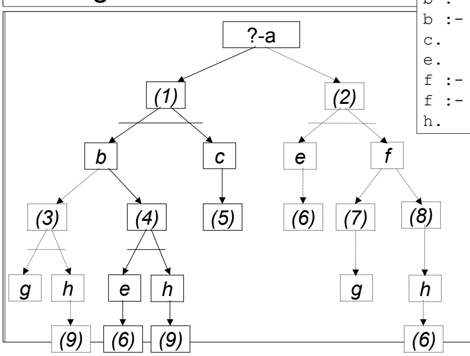




Modell für „Beweisbaum" in PROLOG

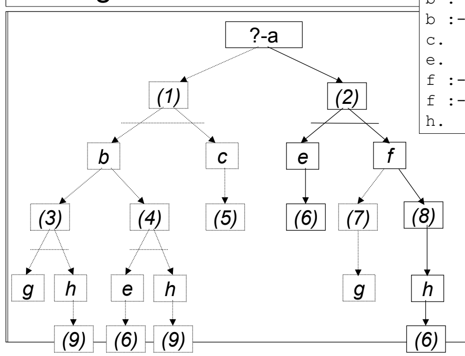
Lösungsbäume

```
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)
```

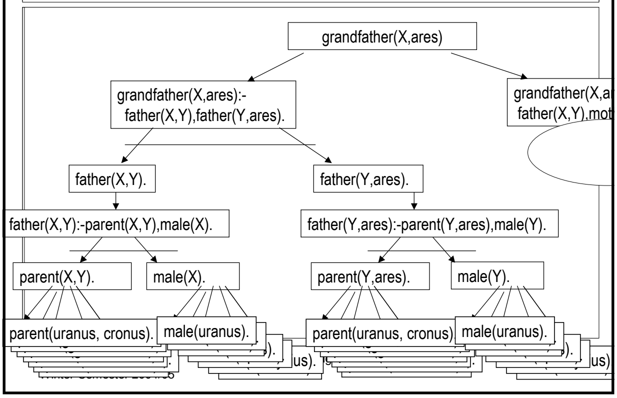


Lösungsbäume

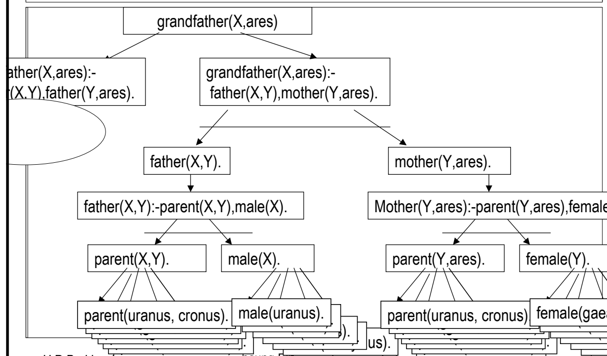
```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



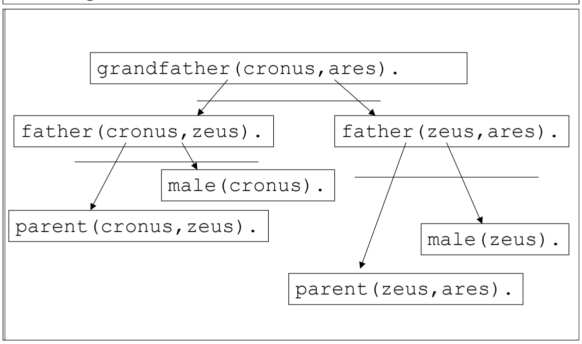
Und-oder-Baum modelliert Beweisversuche



Und-oder-Baum modelliert Beweisversuche



Lösungsbaum modelliert „Beweisbaum"



Modell für *nicht-deterministische* Suche

Beispiel: Vereinfachtes PROLOG-Schema

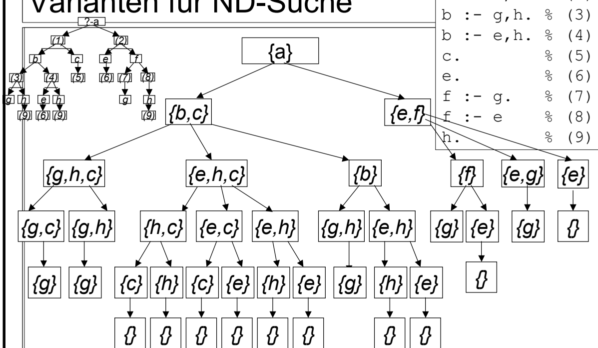
(ohne Unifikation: „Aussagen-Kalkül")

1. Initialisiere: 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 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

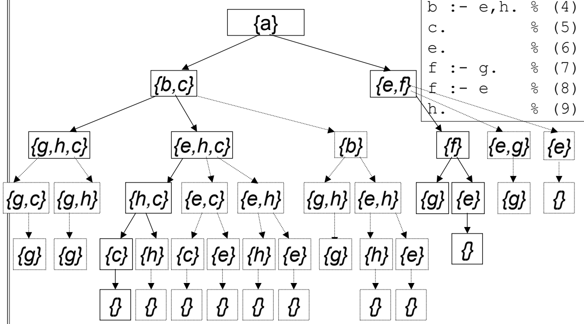
- Reihenfolge innerhalb einer Klausel (Und-Verzweigung)
  (alle subgoals müssen erfüllt werden)
  links vor rechts
- Reihenfolge innerhalb einer Prozedur (Oder-Verzweigung)
  (Alternativen für Beweis)
  oben vor unten

Zu zeigen wäre:

Wenn Beweis existiert, dann auch schon hierbei.

Einschränkung der Varianten

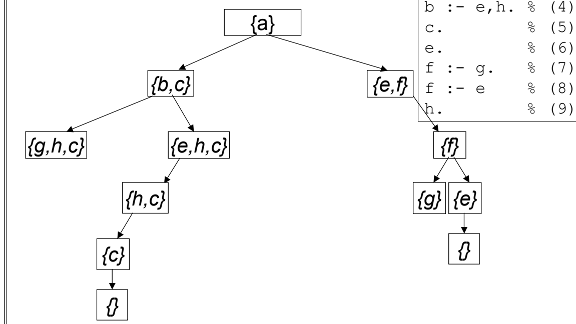
```
a :- b,c. % (1)
a :- e,f. % (2)
b :- g,h. % (3)
b :- e,h. % (4)
c.        % (5)
e.        % (6)
f :- g.   % (7)
f :- e    % (8)
h.        % (9)
```



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

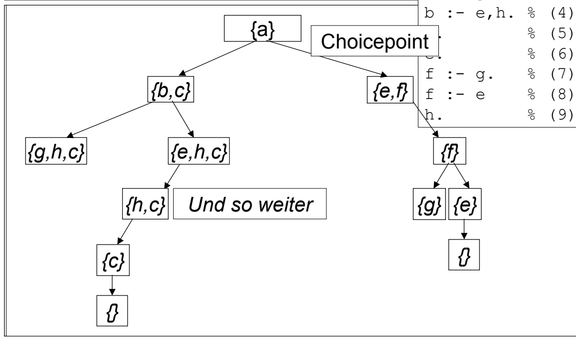
Wege werden nicht vollständig neu probiert, sondern nur stückweise.

Bei Alternativen: Einfügen eines „Choicepoint“

„Backtracking“:
Bei Fehlschlag am jüngsten „Choicepoint“ andere Alternative verfolgen

Backtracking

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

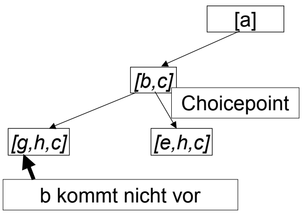
Linke Zweige zuerst

- Backtracking bei Fehlschlag:
  Rückkehr zu Alternative an Oder-Verzweigung

Nächster Zweig einer Oder-Verzweigung

Problem: Choicepoint in subgoal –Liste nicht repräsentierbar

Modelle für Prolog-Suche

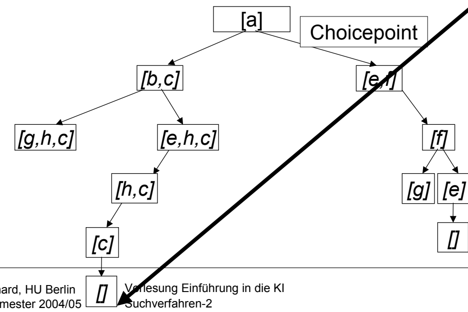


Problem: Choicepoint in subgoal –Liste nicht repräsentierbar

Quelle für Missverständnisse

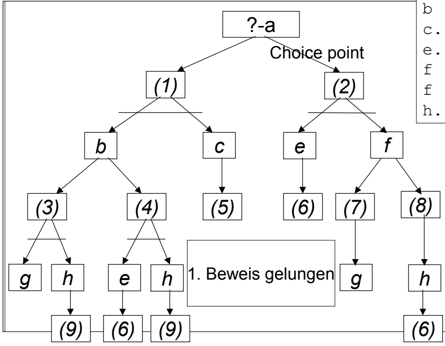
Bei erfolgreichem Beweis ist subgoal-Liste leer

Aber Baum noch nicht vollständig durchsucht

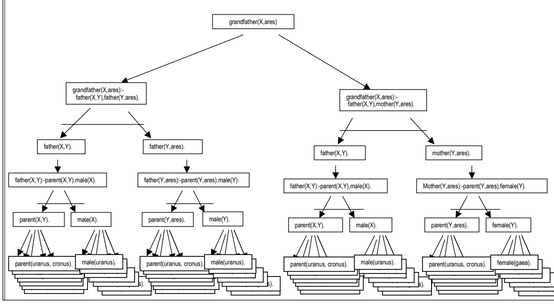


Abarbeitung im Und-oder-Baum

```
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



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;
```

Choicepoints

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 erfolgloser 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

Transformation in Zustandsraumsuche:
Subgoal-Listen sind Zustände eines Zustandsraums

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

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

Andere Repräsentation der Zustände:
Betrachtung als verschachtelte Listen
$\mathcal{L} = [L_1,...,L_m]$
$= [[g_{11}, ..., g_{1n}] , ..., [g_{i1}, ..., g_{i,ni}], ... , [g_{m1}, ..., g_{m,nm}]]$

Algorithmus für systematische Suche

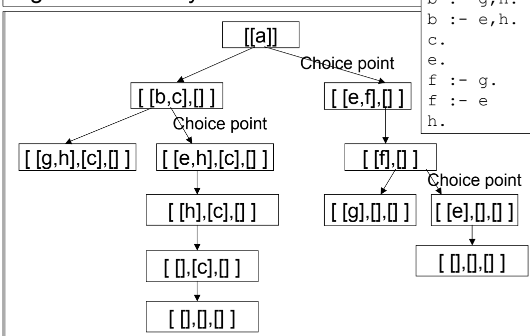
(0) (Start) $\mathcal{L}$: = [[Ausgangsproblem(e)]] .
(1) Falls $\mathcal{L}$ = [[ ], ...,[ ]] : EXIT(yes) .
(2) Sei L_i erste nicht-leere Subgoal-Liste aus $\mathcal{L} = [L_1,...,L_m]$.
Sei g_{i1} erstes Element aus $L_i = [g_{i1}, ..., g_{i,ni}]$:
- g_{i1} aus L_i entfernen: $L'_i := [g_{i2}, ..., g_{i,ni}]$.
- 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: *Choice Point* setzen.
Weiter bei **(2)** .

(4) Backtracking: Rücksetzen zum jüngsten *Choice Point* :
$\mathcal{L}$ zurücksetzen auf Stand vor *Choice Point* , weiter bei **(3)** .
Falls kein *Choice Point* existiert: $\mathcal{L}$ =[ ] , EXIT(no).

Algorithmus für systematische Suche

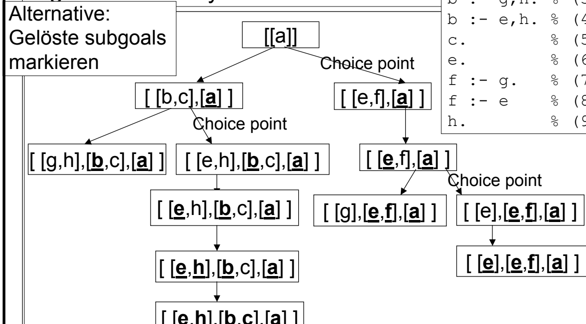
```
a :- b,c.  % (1)
a :- e,f.  % (2)
b :- g,h.  % (3)
b :- e,h.  % (4)
c.         % (5)
e.         % (6)
f :- g.    % (7)
f :- e     % (8)
h.         % (9)
```



Algorithmus für systematische Suche

```
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)
```

Alternative:
Gelöste subgoals markieren



Effizientere Implementation

Bekannt sind:

- Reihenfolge von Klauseln in Prozeduren
- Reihenfolge von subgoals in Klauseln

Zur Laufzeit nicht gesamte Listen speichern,
sondern nur Referenz auf jeweils nächsten Eintrag

g_i statt $[g_i, ..., g_n]$,

Weitere Probleme für Prolog

Behandlung von Variablenbindungen (Unifikation).

Später mehr dazu

Eingriffe in den Beweisablauf (cut).

Effiziensteigerung (vorzeitige Speicherfreigabe), z.B.
- –last call Optimierung („lco“)
- –deterministische Klauseln („dco“)

Prolog-Compiler: Übersetzung in optimiertes Programm
(WAM = Warren abstract machine)

Algorithmen für Problemzerlegung

Analog zu Prolog-Variante:
- Transformation in Zustandsraum
- Anwendung entsprechender Verfahren

Betrachtung der (abgewickelten) Und-Oder-Bäume

Prinzipiell auch für Und-Oder-Graphen möglich,
aber schwer überschaubar

1.5 Suche in Spielbäumen

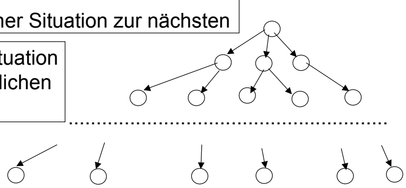
Spielbäume
2-Personen-Nullsummen-Spiele
Minimax-Strategie
Pruning-Verfahren
Heuristische Verfahren

Spiele mit n Spielern P1,...,Pn

Darstellung des Spiels als Spielbaum:
- – Knoten:
  mit Spielsituationen markiert
- – Kanten:
  Züge von einer Situation zur nächsten

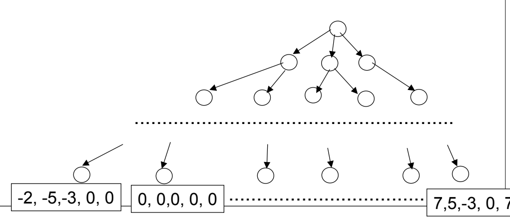
Graph schwierig
zu managen

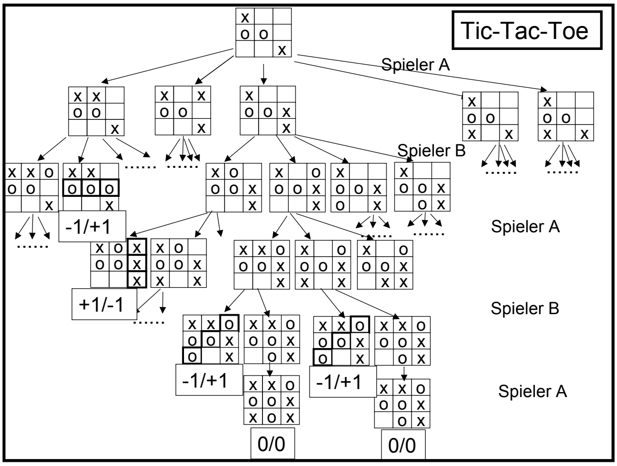
Ggf. gleiche Situation
an unterschiedlichen
Knoten



Spiele mit n Spielern P1,...,Pn

–Endknoten mit Bewertung für jeden Spieler markiert
- •Positive Zahl: Gewinn
- •Negative Zahl: Verlust

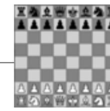




Schach

Durchschnittlich 30 Zugvarianten in jeder Situation

1. eigener Zug:		30 Nachfolgeknoten
1. gegnerischer Zug:	30^2 =	900 Nachfolgeknoten
2. eigener Zug:	30^3 =	27000 Nachfolgeknoten
2. gegnerischer Zug:	30^4 =	810000 Nachfolgeknoten
...		
5. gegnerischer Zug:	30^{10} ~	$6*10^{14}$ Nachfolgeknoten
...		
10. gegnerischer Zug:	30^{20} ~	$3,5*10^{29}$ Nachfolgeknoten
...		



Weitere Spieltypen

Spiel mit unvollständiger Information: Skat, Poker,...

Eigene Unsicherheit vermindern
Gegnerische Unsicherheit erhöhen

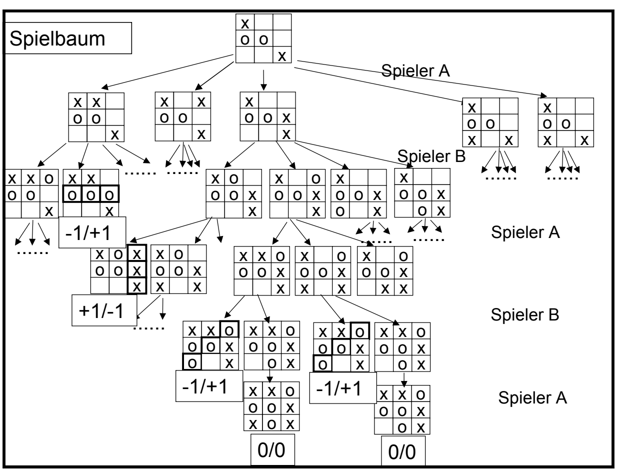
Emotionen modellieren
Emotionen beeinflussen

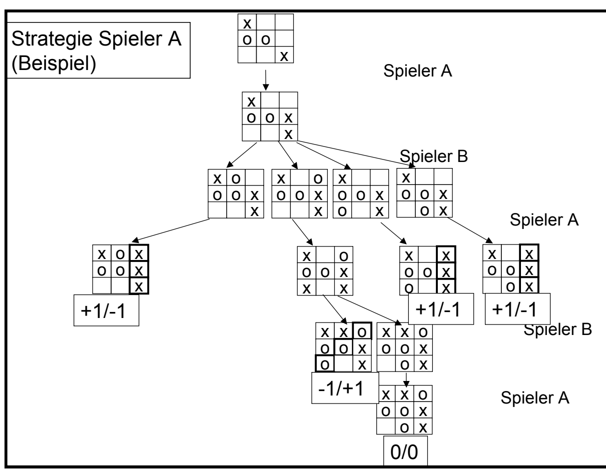
Spiel mit Zufallseinfluss: Monopoly, ...
(Würfel als „weiterer Spieler“)

Nullsummenspiel: Gewinne = –Verluste

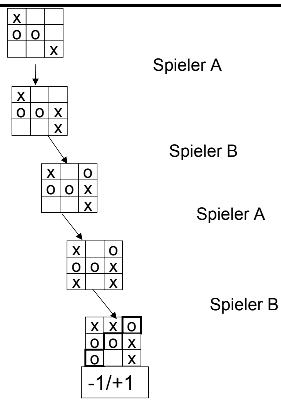
π_i : Spielstrategie (policy) des Spielers P_i

- Vorgabe eines Zuges für jede Situation
  π_i : Situationen $\rightarrow$ Spielzüge des Spielers P_i
- Entspricht einem Teilbaum des Spielbaums
  - 1 Nachfolger für den eigenen Zug gemäß Strategie
  - k Nachfolger für k mögliche Züge anderer Spieler
- Spielstrategien $\pi_1,...,\pi_n$ aller Spieler ergeben einen Weg im Baum zu einem Endzustand





Zugfolge:
Strategie Spieler A
Strategie Spieler B
(Beispiel)



Gewinn/Verlust

$G_i(\pi_1,\ldots,\pi_n)$:

Gewinn/Verlust des Spielers P_i

wenn jeweils Spieler P_k die Spielstrategie π_k verwendet

Bewertung für alle Spieler als Vektor:
$[G_1(\pi_1,\ldots,\pi_n), G_2(\pi_1,\ldots,\pi_n), \ldots., G_n(\pi_1,\ldots,\pi_n)]$
Gesamtheit der Bewertungen als n-dimensionale Matrix

Ausgangspunkt
- für Festlegung der eigenen Strategie (Gewinn maximieren)
- für Verhandlungen über Koalitionen

(Problem: stabile Koalitionen)

Gefangenendilemma

Strategien: „leugnen“ oder „gestehen“

Resultatsmatrix

		Spieler 2	
		gestehen	leugnen
Spieler 1	gestehen	[5,5]	[0,10]
	leugnen	[10,0]	[3,3]

Modell für Koordination

- Koalitionsbildung
- Verhandlungen etc.

- einschließlich Kooperation (Koalitionen)
  z.B. Verhandlung über Strategie-Wahl $\pi_1,\ldots,\pi_n$ gemäß erwarteten Gewinnen $[G_1(\pi_1,\ldots,\pi_n), \ldots., G_n(\pi_1,\ldots,\pi_n)]$
- und Konflikt (Gegnerschaft)
- Ziele:
  - (Individuellen/Globalen) Gewinn optimieren

Probleme:
Welche Werte optimieren?

Modelle für Koordination

– Nash-Gleichgewicht:

Ein einzelner Spieler kann sich nicht verbessern, wenn er eine andere Strategie wählt (insbesondere ist individuelles Betrügen sinnlos).

– Pareto-Optimal:

Bei jeder anderen Wahl der Strategienmenge schneidet wenigstens ein Spieler schlechter ab (insbesondere kann kein Spieler besser abschneiden, ohne dass ein anderer schlechter abschneidet).

– Global-Optimal:

Summe über alle Gewinne optimal.

Probabilistische Modelle

Wahrscheinlichkeiten für
- Spielzustand (bei unvollst. Information)
- Zufallseinflüsse (Würfel)
- Strategien (Eigene/Gegnerische Züge)

Ergebnis als Erwartungswert

Hier nicht weiter verfolgen
⇨ Spieltheorie
⇨ Optimierung
⇨ BWL

Spielstrategien entwickeln

Im weiteren beschränken:

- 2-Personen-Nullsummenspiele
  - 2 konkurrierende Spieler A und B
  - Spieler ziehen abwechselnd
  - Gewinn(A) + Verlust(B) = 0

Angabe für Spieler A ausreichend

- Volle Information der Spieler
- Ohne Zufall
- Deterministische Strategien

Spielbaum für diese Spiele

Darstellung analog zu Und-Oder-Baum:

- A kann Zug wählen: „Oder-Verzweigung“
- A muss auf jeden Zug von B reagieren: „Und-Verzweigung“

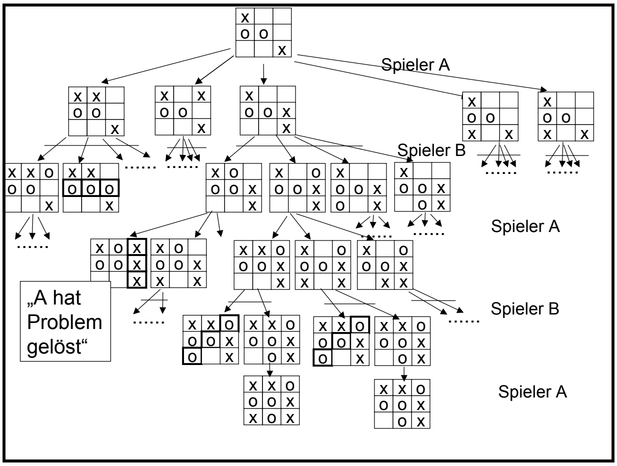
Bei Spielen mit Werten 1, -1 (Gewinn, Verlust)
volle Analogie zu Problemzerlegung:

- lösbare Knoten: A gewinnt
- unlösbare Knoten: A verliert

evtl. auch „unentschieden“ einbeziehen

Lösungsbaum liefert Gewinn-Strategie:

- A wählt jeweils Zug zu lösbarem Knoten,
- B muß dann ebenfalls zu lösbarem Knoten ziehen.



Optimalitätsannahme

Annahme: Beide Spieler spielen optimal

Mit Strategie π_A durch Spieler A erreichbarer Wert:

$G(\pi_A) := \text{Min}\{ G(\pi_A, \pi_B) \mid \pi_B \text{ Strategie für B} \}$

Im Spiel durch Spieler A erreichbarer Wert:

$G_A := \text{Max}\{ G(\pi_A) \mid \pi_A \text{ Strategie für A} \}$

Spieler A ist *Maximiere*r (seines Gewinns)
Spieler B ist *Minimierer* (des Gewinns für A)

Optimale Strategie für Spieler A: π_A^* mit $G(\pi_A^*) = G_A$

Spieler A besitzt Gewinnstrategie,
falls G_A maximal möglichen Wert annimmt

Problemstellungen

- Besitzt Spieler A eine Gewinnstrategie?
- Konstruiere ggf. Gewinnstrategie für A
- Konstruiere optimale Strategie π_A^*

Gewinnstrategien existieren für

Wolf und Schafe (Schafe gewinnen)

Nim-Spiel (abhängig von Startsituation gewinnt A oder B)

Baumspiel (1.Spieler gewinnt immer)

Satz:
Jedes Spiel mit endlicher Baumstruktur
und nur Gewinn/Verlust
besitzt entweder eine Gewinnstrategie für A
oder eine Gewinnstrategie für B

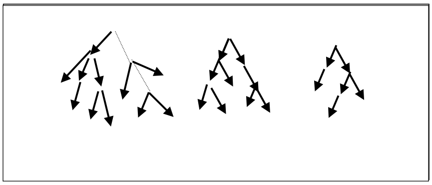
Baumspiel

Situationen: Menge von Bäumen

Start: Ein einziger Baum

Zug: Streiche einen Knoten und alle seine Vorgänger in einem Baum (übrig bleiben Teilbäume)

Gewinn: Wer letzten Baum streicht.



Wert von Spielsituationen

Welchen Zug soll Spieler als nächstes wählen?

Wert $G_A(s)$ einer Spielsituation s für Spieler A

$G_A(s) \rightarrow$ Gewinne

$G_A(s)$:= maximaler Wert, den Spieler A von dort aus mit seiner optimalen Strategie π_A* erreichen kann

Aus G_A kann umgekehrt π_A* konstruiert werden:

In Situation s wähle Zug

zu einer Folgesituation s´

mit optimaler Bewertung $G_A(s)$

Ermitteln der Werte von Spielsituationen

Credit-Assignment-Problem:

Wert einer Situation (bzw. eines Spielzugs) ist erst am Spielende bekannt

Immerhin: Iterative Abhängigkeit der Werte

Wenn A in s zieht:

$G_A(s)$ = Max {$G_A(s´)$ | s´ Nachfolgesituation von s }

Wenn B in s zieht:

$G_A(s)$ = Min {$G_A(s´)$ | s´ Nachfolgesituation von s}

Bei bekanntem Spielbaum: Bottom-Up-Konstruktion der Werte im Spielbaum

Minimax-Verfahren

Lernen der Werte von Spielsituationen

Immerhin: Iterative Abhängigkeit der Werte

Wenn A in s zieht:

$G_A(s)$ = Max {$G_A(s´)$ | s´ Nachfolgesituation von s }

Wenn B in s zieht:

$G_A(s)$ = Min {$G_A(s´)$ | s´ Nachfolgesituation von s}

Bei unbekanntem Spielbaum:

„Reinforcement-Lernen"

- Exploration
  (Erkunden von Möglichkeiten, d.h. Spielzüge ausprobieren)
- Sukzessives Verbessern der Bewertungen

Minimax-Verfahren

Voraussetzungen:

- Endlicher Spielbaum
- Endknoten mit Resultaten für Spieler A markiert

(1)Falls Startknoten markiert:

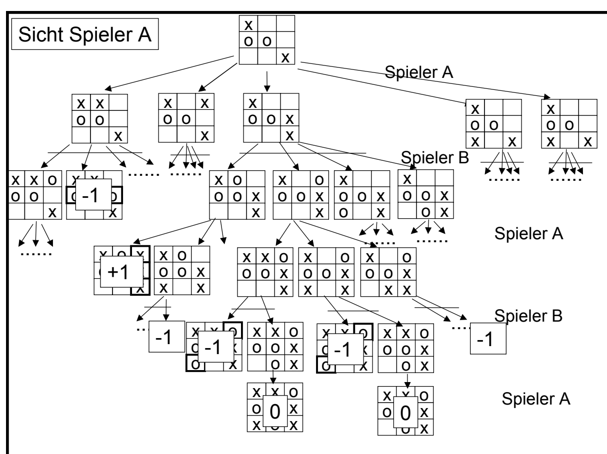
Exit: Wert der optimalen Strategie π_A* = Wert(Startknoten)

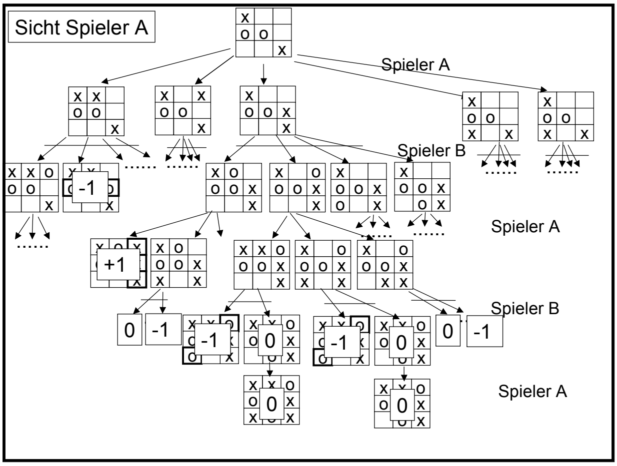
(2)Wähle unmarkierten Knoten k, dessen Nachfolger markiert sind

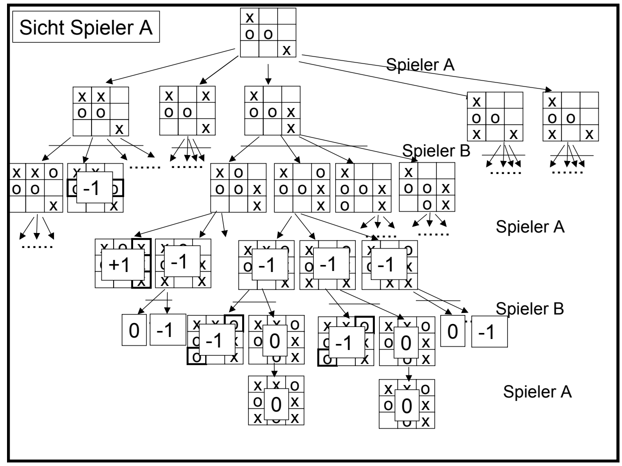
Wenn A in k zieht: Wert(k):=Max{Wert(k´) | k´ Nachfolger von k }

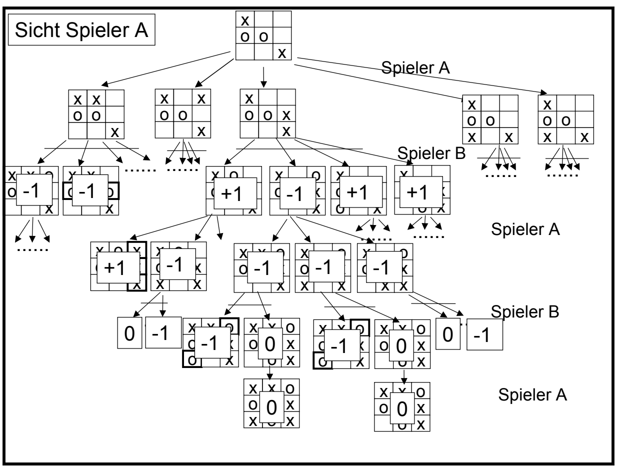
Wenn B in k zieht: Wert(k):=Min{Wert(k´) | k´ Nachfolger von k }

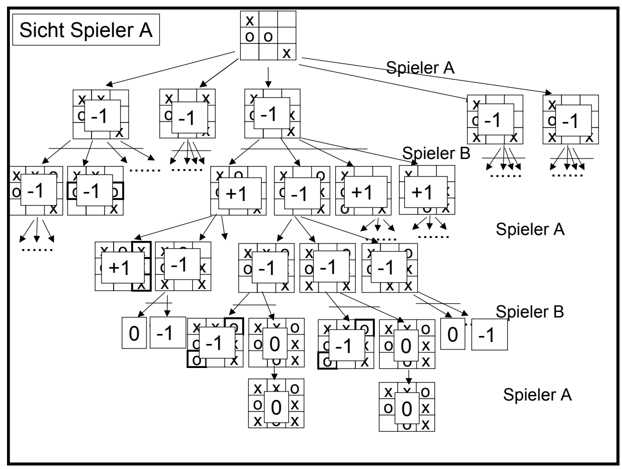
Weiter bei (1).

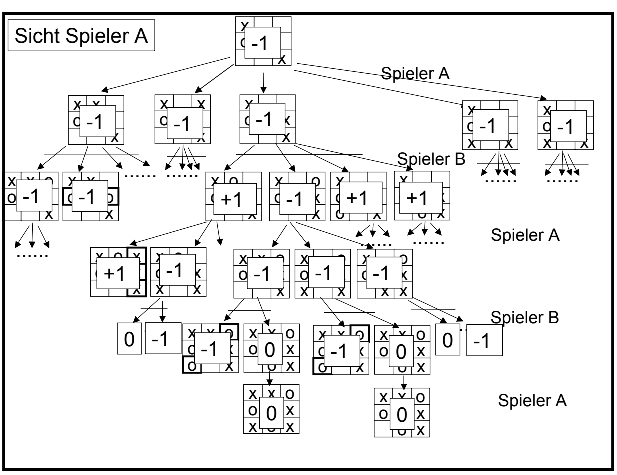


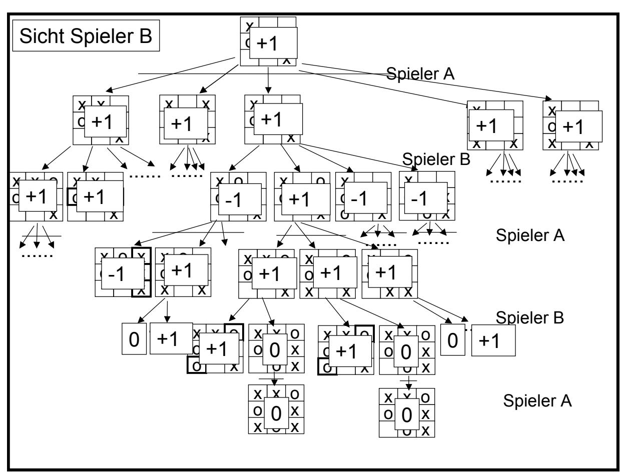

Sicht Spieler A
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A

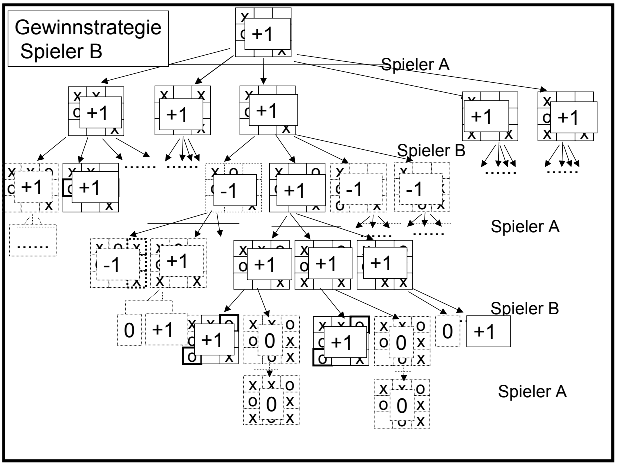

Sicht Spieler A
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A

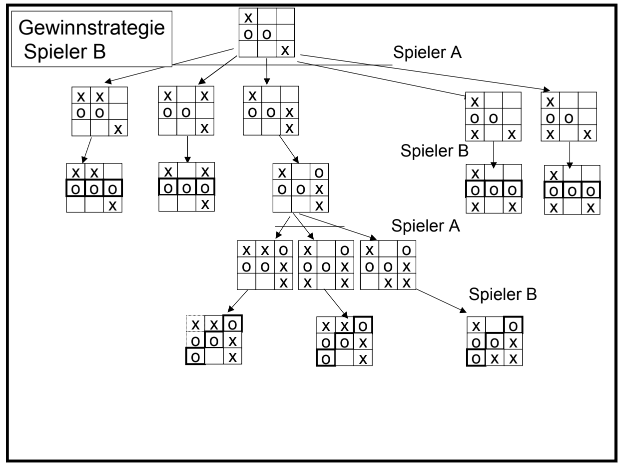

Sicht Spieler A
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A


Sicht Spieler A
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A


Sicht Spieler A
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A

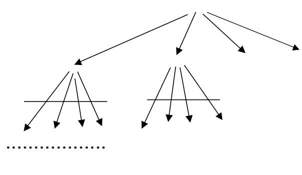

Sicht Spieler B
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A



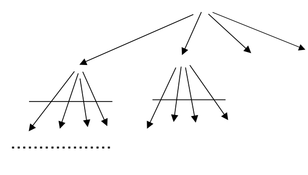


Züge ausschließen: Pruning-Strategien

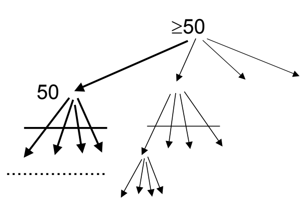
Idee: Wenn es bereits bessere Varianten gibt, müssen schlechtere nicht weiter verfolgt werden



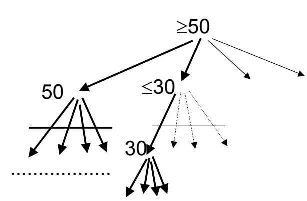
α-pruning (Züge von A ausschließen)



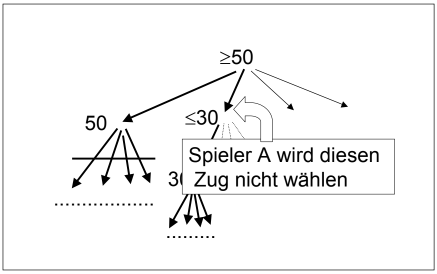
α-pruning (Züge von A ausschließen)



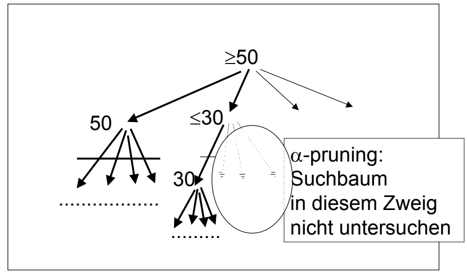
α-pruning (Züge von A ausschließen)



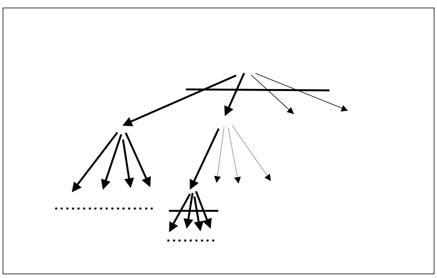
α-pruning (Züge von A ausschließen)



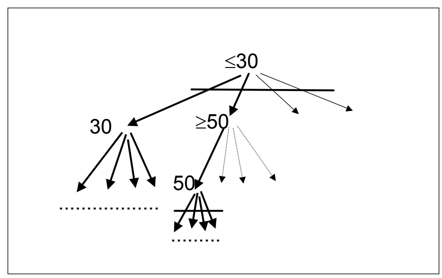
α-pruning (Züge von A ausschließen)



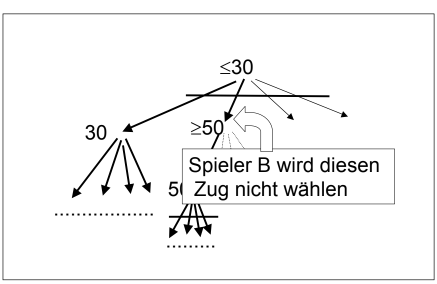
β-pruning (Züge von B ausschließen)



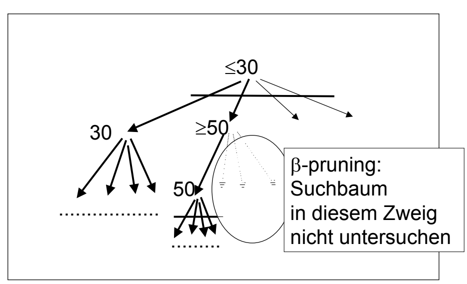
β-pruning (Züge von B ausschließen)



β-pruning (Züge von B ausschließen)



β-pruning (Züge von B ausschließen)



Effizienz von Pruning-Strategien

abhängig von der Reihenfolge:

- im ungünstigsten Fall keine Einsparung:
  Es bleibt bei b^d Endknoten für Verzweigungsfaktor b, Tiefe d
- im günstigsten Fall („beste Züge jeweils links"):
  Aufwand ungefähr $2 * b^{(d/2)}$

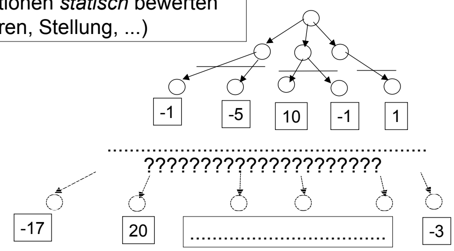
$2 * b^{(d/2)} - 1$	für gerade d,
$b^{((d+1)/2)} + b^{((d-1)/2)} - 1$	für ungerade d.

d.h. Doppelte Suchtiefe mit gleichem Aufwand möglich

Weitere Verfahren
zur Auswahl günstiger Expansionsreihenfolge

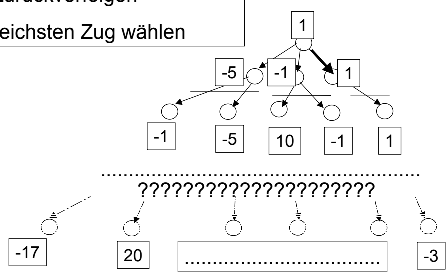
Heuristische Suche

1. Spielbaum teilweise entwickeln von aktueller Situation ausgehend
2. dort Situationen *statisch* bewerten (z.B. Figuren, Stellung, ...)



Heuristische Suche

3. gefundene Werte *dynamisch* gemäß Minimax zurückverfolgen
4. aussichtsreichsten Zug wählen



Woher kommen Bewertungen?

Statische Bewertung von Situationen:

- Analogie zu Schätzfunktionen
- Vorhersage des erreichbaren Gewinns

Bauer:	1
Läufer, Springer:	3
Turm:	5
Dame:	9
Bauernstellung:	0,5
Königsstellung:	0,5

Unterschiedliche Bewertungsfaktoren
Verdichtung zu einer Zahl, z.B. gewichtete Summe

Lernen von Gewichten anhand von Beispielen

Bis zu welcher Tiefe entwickeln?

Horizont-Effekt:

- bei Abbruch in „unruhiger Situation" falsche Bewertung
- Ausweg:
  - keine feste Tiefenbeschränkung,
  - in unruhigen Situationen weiterentwickeln (umgekehrt: in eindeutig schlechten Situationen frühzeitig abbrechen)
- *alpha-beta-pruning*