Abstrakte Datentypen

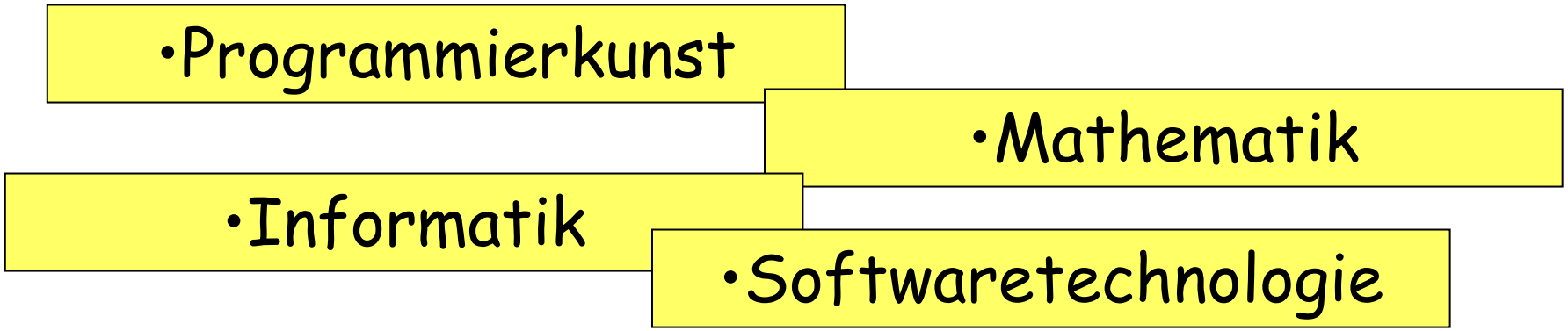
Implementation

Prototypen

Spezifikation

Formale Beschreibungen

Abstrakte Modelle



ABSTRAKTE DATENTYPEN

Beobachtung: Unterschiedliche Realisierungen

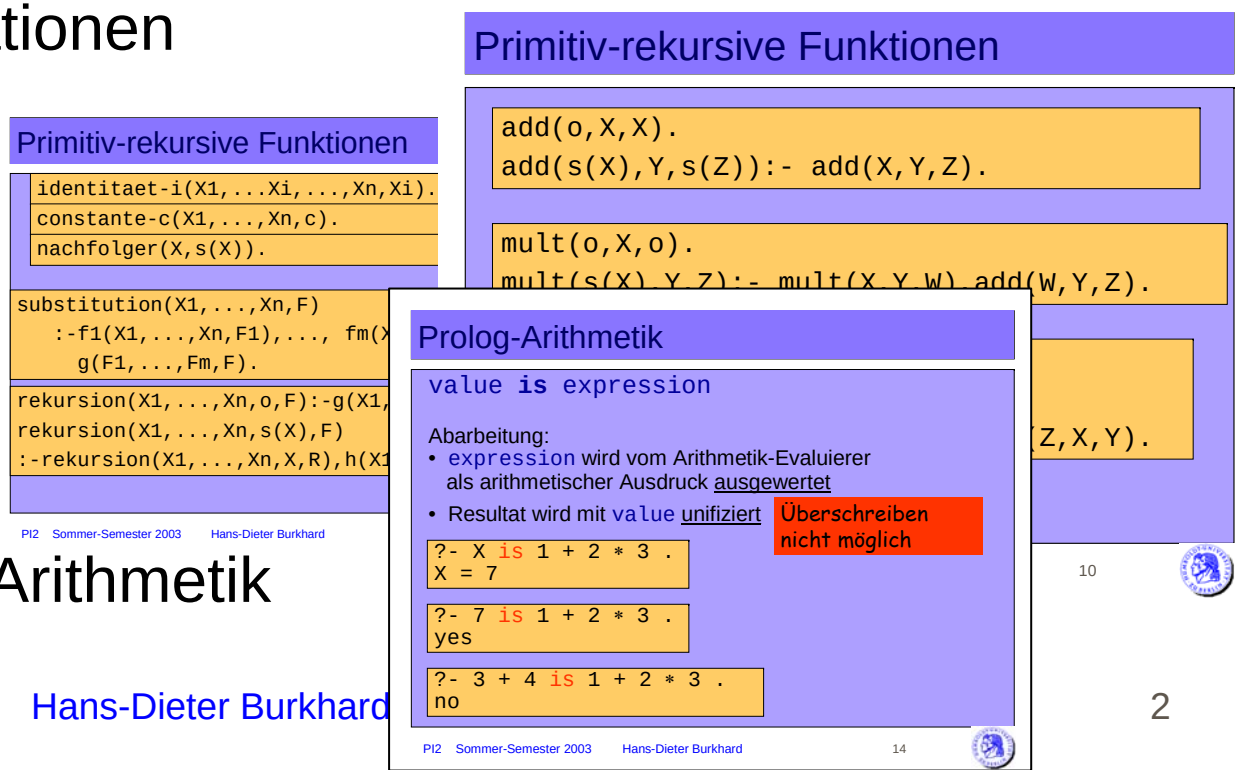
(Darstellung, Algorithmen, Fehlerbehandlung, ...)

für gleiche Aufgaben

Beispiel: Prolog-Arithmetik für natürliche Zahlen

– Mittels Deklarationen

– Mittels Prolog-Arithmetik



Mögliche Implementationen

```
?- nachfolger(a, b).
?- nachfolger(a, Nachfolger).
?- nachfolger(Vorgänger, b).
?- nachfolger(Vorgänger, Nachfolger).
```

```
nachfolger(X,Y):- integer(X), !, Y is X+1.
nachfolger(X,Y):- integer(Y), !, X is Y-1 .
nachfolger(0,1).
nachfolger(X,Y):- nachfolger(U,V), X is U+1, Y is V+1.
```

Fehlermöglichkeiten z.B.:

- Weder Zahl noch Variable
- Negative Zahl

Mögliche Implementationen

Argumente: Natürliche Zahl, Variable, sonstiges

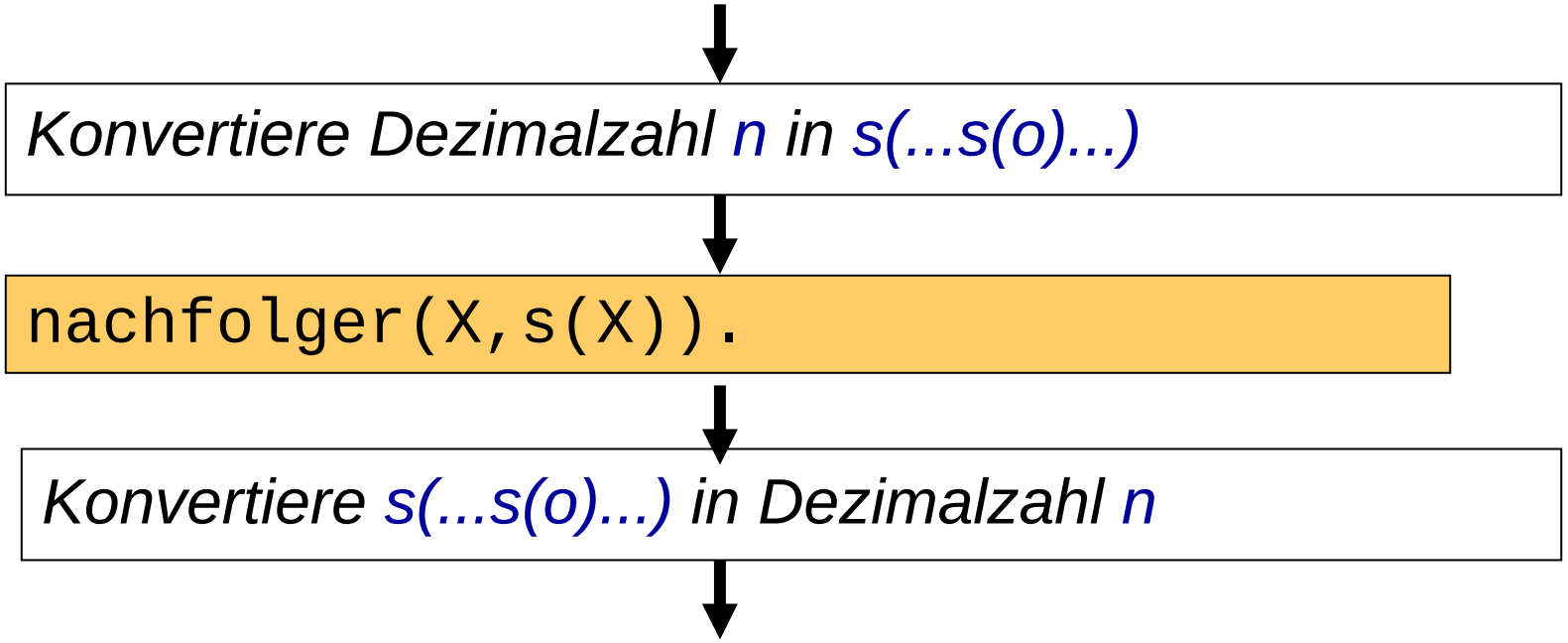
```
natural(X) :- integer(X), X >= 0 .

nachfolger(X,Y):- natural(X), natural(Y), !, Y is X+1 .
nachfolger(X,Y):- natural(X), var(Y), !, Y is X+1 .
nachfolger(X,Y):- var(X), natural(Y), !, X is Y-1 .
nachfolger(0,1) :- var(X), var(Y) .
nachfolger(X,Y):- var(X), var(Y), nachfolger(U,V), X is U+1, Y is V+1.
nachfolger(X,Y):- write('nicht im Bereich der natürlichen Zahlen') .
```

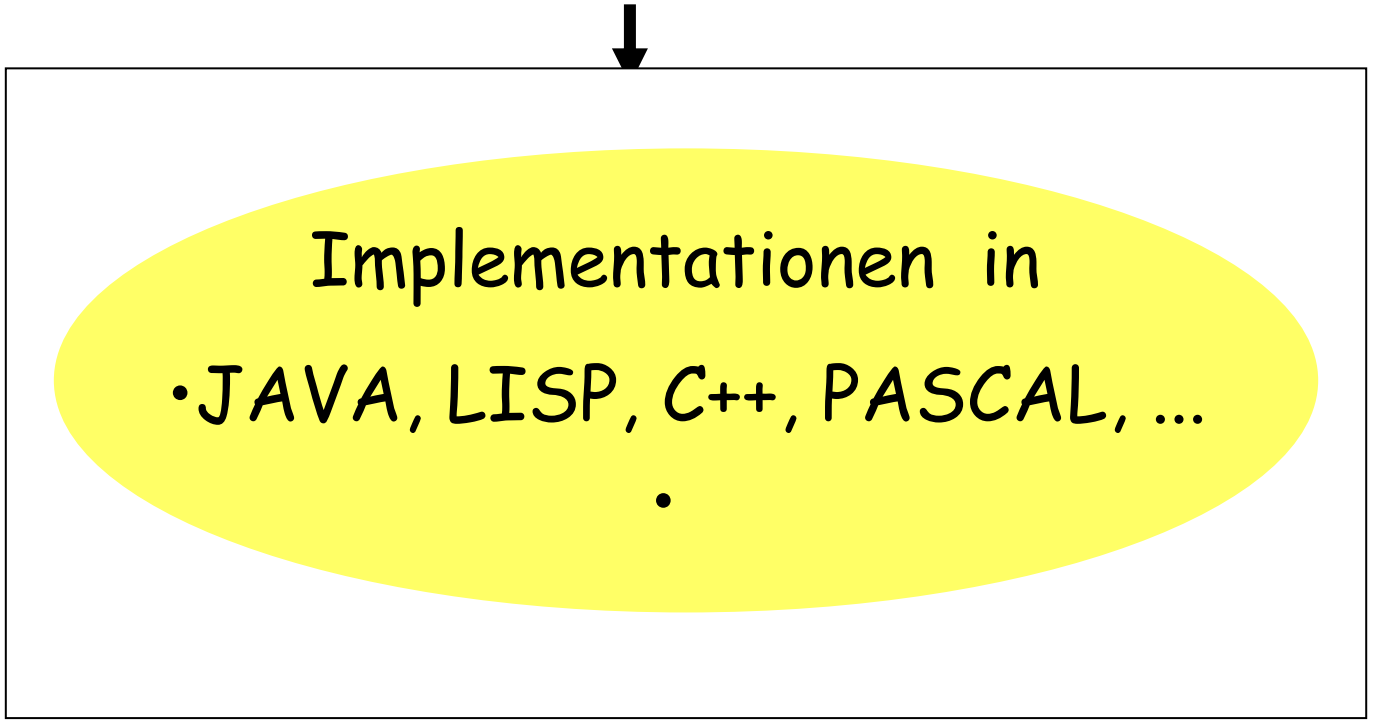
(es gibt noch weitere Ausnahmen ....)

- „Exceptions": Programmiersprache/-umgebung stellt Mittel für Ausnahmebehandlung bereit

Weitere Implementationen



Weitere Implementationen



Abstrakte Datentypen:

- Daten-Formate und Prozeduren
  in allgemeiner (mathematischer) Form
- Lösung von konkreten Details (Datenkapselung)

Abstrakte Datentypen

Mathematische Modelle:

- Strukturen (Definitionen)
- Algorithmen
- Komplexitätsaussagen

- Listen
  - Sortierverfahren
- Relationen
  - Prädikatenlogik

Datenkapselung:

- Interface
- Laufzeiteigenschaften

- Datenbank
- Prolog-Modul
- Java-Klasse

Implementation in ... von ... Version ...

Abstrakte Datentypen

Verallgemeinerung primitiver Datentypen

- boolean, integer, char, ...

Strukturen mit Operationen

- Mengen
- Relationen
- Liste (endliche Folge)
- Keller
- Warteschlangen
- Graph
- Baum
- ...

Mathematisches Modell
Algorithmen

Abstrakter Datentyp
→ **Funktionalität**

Datenstruktur
→ Programm

Liste

Definitionen: Länge n: Anzahl der Elemente

Liste als Funktion:

$L: \{1,...,n\} \rightarrow M$ M =*Menge der* (möglichen) *Elemente*

Liste als Aufzählung: $[L(1), L(2), ..., L(n)]$

Geordnete Listen, z.B.:

- Zahlen in aufsteigender Reihenfolge

- Wörter in lexikographische Anordnung

- Geordnete Liste bzgl. Ordnungsrelation $R \subseteq M \times M$

  Liste L ist *geordnet bzgl. R* ,

  falls gilt: $\forall i, j \in \{1,...,n\}: i < j \rightarrow R(L(i),L(j))$

Liste

Operationen:

Zugriff auf Elemente :

- Suchen, Einfügen, Löschen von Elementen

Bearbeitung von Listen :

- Verketten
- Reorganisation, Sortieren, ...

Sortierverfahren erzeugen eine Reihenfolge der Listenelemente entsprechend einer vorgegebenen Ordnungsrelation R. Unterschiedliche Sortierverfahren haben unterschiedliche Komplexität.

- *Quick-Sort*
- *Selection-Sort*
- *Merge-Sort*
- *Heap-Sort*
- *Bubble-Sort*
- *...*

Listen

Implementation als Datenstruktur:

Organisation bzgl. Speicherorganisation

Felder

Referenzen

- verkettete Liste: Referenz auf Nachfolger-Element
- Doppelt verkettete Liste

Rekursiv (Prolog)

Jeweils mit Methoden für Einfügen, Verketten, ...

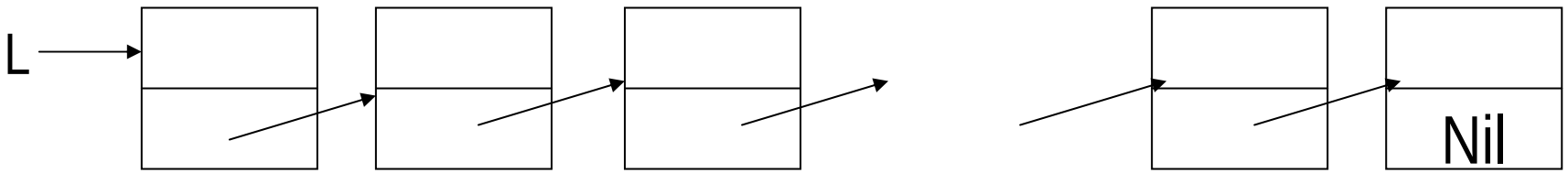
Geschickte Wahl der Datenstruktur

Listen-Operationen: member

Iterative Implementation für verkettete Liste

```
boolean lookup(List L, Element x)

E= L;
while (true)
{ if (E==NIL) {return false;}
  if (E.Value== x ) {return true;}
  E = E.Next;
}
```



Listen-Operationen: member

Rekursive Implementation für rekursive Liste

```
boolean lookup(List L, Element x)
 { if ( Head(L) ==  x ) { return true;}
   if ( Tail(L) == NIL)  { return false;}
  return lookup(Tail(L),x);
   }
```

```
member(X, [X | T] ) :- ! .
member(X, [H | T] ) :- member(X, T) .
```

Listen-Operationen: append

Iterative Implementation für verkettete Liste

```
Liste concatenate(Liste L1, Liste L2)
{ E = L1;
 while (E.Next != NIL) {E=E.Next;}
 E.Next = L2; return L1;
 }
```

Rekursive Implementation für rekursive Liste

```
append([ ], L, L ).
append([X | L1], L2, [X | L3] ) :- append(L1, L2, L3).
```

Streichen von Elementen

```
delete(X, [ ], [ ] ).
delete(X, [X | L ], Deleted ) :- delete(X, L , Deleted ).
delete(X, [Y | L ], [Y| Deleted] ) :- X \= Y, delete(X, L , Deleted ).
```

Streicht **alle** Vorkommen von X aus L.

Differenz-Listen

Andere Repräsentation von Listen:

Liste als Anfangsstück $[l_1,...,l_n]$
einer längeren Liste $[l_1,...,l_n, r_1,...,r_m]$,
vermindert um den Rest $[r_1,...,r_m]$.

Schreibweise z.B. $[l_1,...,l_n, r_1,...,r_m] - [r_1,...,r_m]$

Rest $[r_1,...,r_m]$ dabei beliebig,
z.B. auch $[l_1,...,l_n \mid [ ]] - [ ]$
allgemeinste Form: $[l_1,...,l_n \mid R] - R$

```
conv_DiffList_to_List(A-[ ],A).

conv_List_to_DiffList([ ],L-L).
conv_List_to_DiffList([X|L1], [X|L2]-L)  :- conv_List_to_DiffList(L1, L2-L).
```

Differenz-Listen

Liste als Anfangsstück $[l_1,...,l_n]$
einer längeren Liste $[l_1,...,l_n, r_1,...,r_m]$,
vermindert um den Rest $[r_1,...,r_m]$.

Verkettung von Differenz-Listen in einem Schritt:

```
append_dl(A-B, B-C, A-C).
```

A: $a_1,...,a_n$, $b_1,...,b_m$, $c_1,...,c_s$

A - B

A - C

B: $b_1,...,b_m$, $c_1,...,c_s$

B - C

C: $c_1,...,c_s$

```
?- append_dl( [a,b,c|R1] - R1, [1,2|R2] - R2 , L).
```

```
L = [a, b, c, 1, 2|_G173]-_G173
```

Differenz-Listen

Umkehrung einer Liste:

```
naive_reverse([ ],[ ]) .
naive_reverse( [H | T], L) :- naive_reverse(T,L1), append(L1,[H], L).
```

Quadratische Zeit

```
reverse_dl([ ], L - L) .
reverse_dl( [H | T], L - S ) :- reverse_dl(T,L – [H|S] ).

reverse( X,  Y) :- reverse_dl(X,Y – [ ] ).
```

Lineare Zeit

Listen-Subtraktor S als Akkumulator

```
reverse_acc([ ],R, R).
reverse_acc([ H| T], Acc, R ) :- reverse_acc( T, [H | Acc] , R) .
reverse(L,R) :- reverse_acc(L,[ ],R).
```

Quicksort

```
quicksort( [H | T], Sorted ) :-
    partition( T,H,Littles,Bigs),
    quicksort(Littles, LSorted),
    quicksort(Bigs,  BSorted ),
    append(LSorted,[H|BSorted],Sorted).
quicksort([ ], [ ]) .


partition( [A | M] ,H, [A | Ls] , Bs)      :- A =< H, partition(M,H,Ls,Bs).
partition( [A | M] ,H,   Ls,     [A | Bs]) :-  A > H, partition(M,H,Ls,Bs).
partition([ ], H, [ ], [ ]).
```

Quicksort mit Differenz-Liste

```
quicksort_dl( [H | T], Sorted - S ) :-
        partition( T,H,Littles,Bigs),
        quicksort_dl(Littles,   Sorted - [H|B] ),
        quicksort_dl(Bigs,   B - S).
quicksort_dl([ ], L - L) .

quicksort( X,  Y) :- quicksort_dl(X,Y – [ ] ).
```

Verwendet wie vorher:

```
partition( [A | M] ,H, [A | Ls] , Bs)      :- A =< H, partition(M,H,Ls,Bs).
partition( [A | M] ,H,   Ls,     [A | Bs]) :-  A > H,  partition(M,H,Ls,Bs).
partition([ ], H, [ ], [ ]).
```

Komplexität: **Laufzeit**

Zeit-Messungen

abhängig von Eingabewerten

"worst case" vs. *"average case"*

- *Benchmarks*:
  Vergleich von Algorithmen/Programmen
- *Profiling*:
  Analyse des Zeitverhaltens im Programm
- *Programm-Analyse:*
  Anzahl "elementarer" Anweisungen, Abschätzungen für Schleifen-Durchläufe bzw. Verzweigungen

90-10-Regel:

90% der Laufzeit in 10% des Codes verbraucht

Komplexitäts-Betrachtungen

Programm π auf Maschine *M* benötigt

m Operationen und *n* Byte Speicher

andere Maschine *M‘* simuliert *M*:

pro Operation von *M* benötigt *M‘* maximal:

a Operationen und *b* Byte Speicher

$\Rightarrow$ π auf *M′* benötigt maximal

a·m Operationen und *b·n* Byte Speicher

$\Rightarrow$Abstraktion von konkreter Maschine

(Unterschied: konstanter Faktor *a bzw. b*)

Bezug auf universelle Maschine, z.B. TURING-Maschine

Komplexitäts-Betrachtungen

Allgemeine Aussagen bezogen auf **Problem-Umfang**:
Anzahl der Variablen, Länge eines Ausdrucks, ...

***O*-Notation** (vgl. Landau-Symbole)

obere Schranke für Größen-Ordnung von Funktionen:

$T_\pi(n)$ sei Komplexität von π bei Problemgröße n.
Programm π hat Komplexität $O(f(n))$,
falls eine feste (!) Konstante c existiert,
so daß für *fast alle* Problemgrößen n gilt: $T_\pi(n) \leq c \cdot f(n)$

„fast alle“ = alle außer endlich viele
(d.h. alle ab einer festen Zahl n_0)

Zeit-Komplexität

Erfüllbarkeit eines AK-Ausdrucks mit *n* Variablen:
Zeit-Komplexität $O(2^n)$ Wertberechnungen

Branch-and-bound-Verfahren für Suche nach kürzestem Weg in einem Graphen mit *n* Knoten und mit *m* Kanten :
Zeit-Komplexität $O(n^2)$
andere Abschätzung mit $r = Max(m,n)$:
Zeit-Komplexität $O(r \cdot log(n))$

Suche in einer geordneten Liste der Länge *n*:
Zeit-Komplexität $O(log(n))$

Sortieren einer geordneten Liste der Länge *n*:
Zeit-Komplexität $O(n \cdot log(n))$
Quicksort: $O(n^2)$, im Mittel $O(n \cdot log(n))$

append(L1,L2,L) lineare Zeit bzgl. Länge von L1

naive_reverse(L,R) quadratrische Zeit bzgl. Länge von L

Exponentielle vs. polynomiale Komplexität

n	n^2	n^3	2^n
10	100	1000	1024 ($\sim 10^3$)
100	10000	1000000	$\sim 10^{30}$
1000	1000000	1000000000	$\sim 10^{300}$

bei Komplexität 2^n:

Steigerung der Rechenleistung um Faktor 1000

ermöglicht Steigerung der Problemgröße von n auf $n+10$

Speicher vs. Zeit

Suche in einer geordneten Liste der Länge *n*:

Zeit-Komplexität $O(log(n))$

mit spezieller Datenstruktur („Index“)

für Zugriff auf Liste, z.B. als binärere Baum

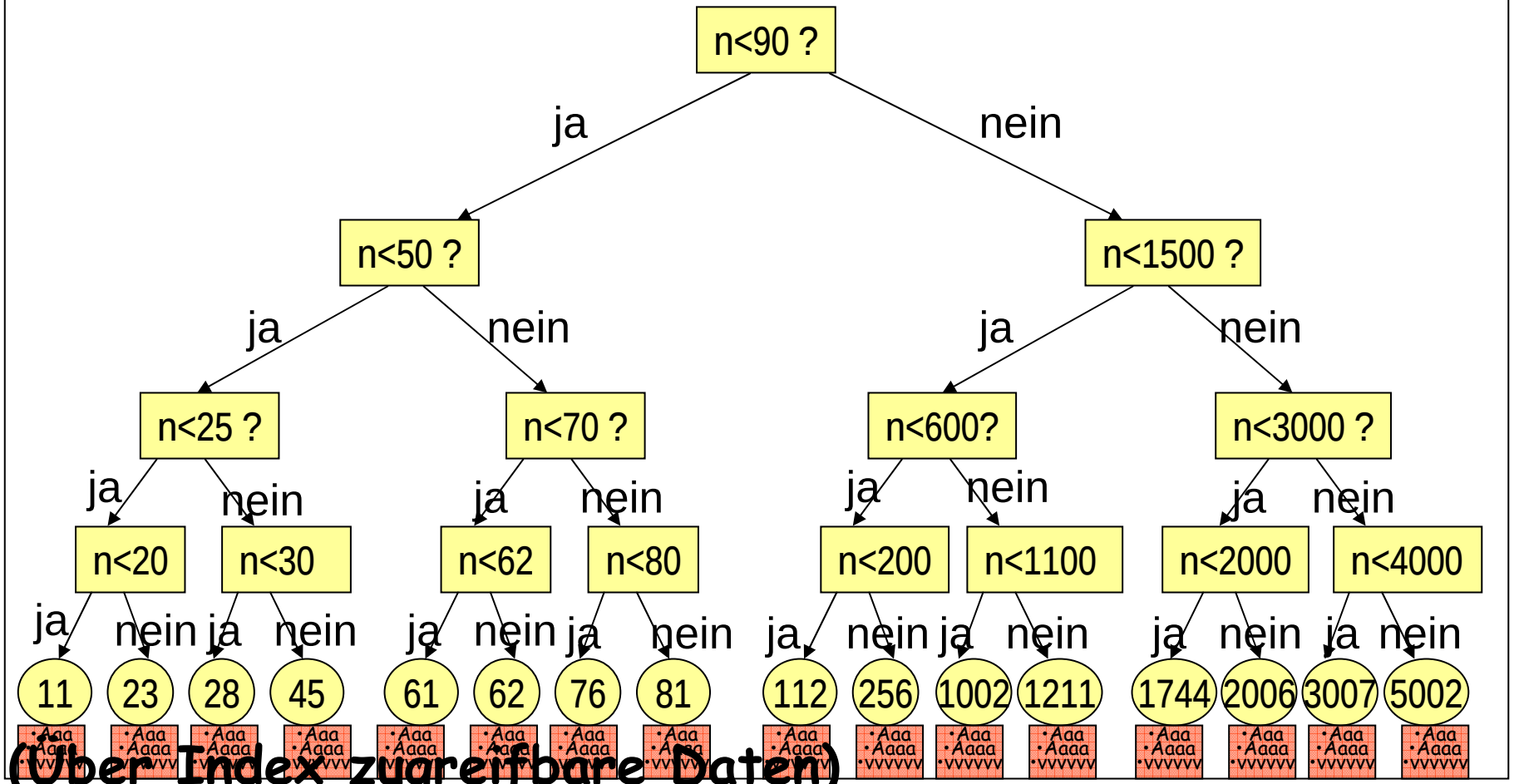
Allgemein:

Speicher-Zeit-“Trade-Off“

[11,23,28,45, 61,62,76,81, 112,256,1002,1211, 1744,2006,3007,5002]

Indexstruktur „Binärer Suchbaum“

[11,23,28,45, 61,62,76,81, 112,256,1002,1211, 1744,2006,3007,5002]

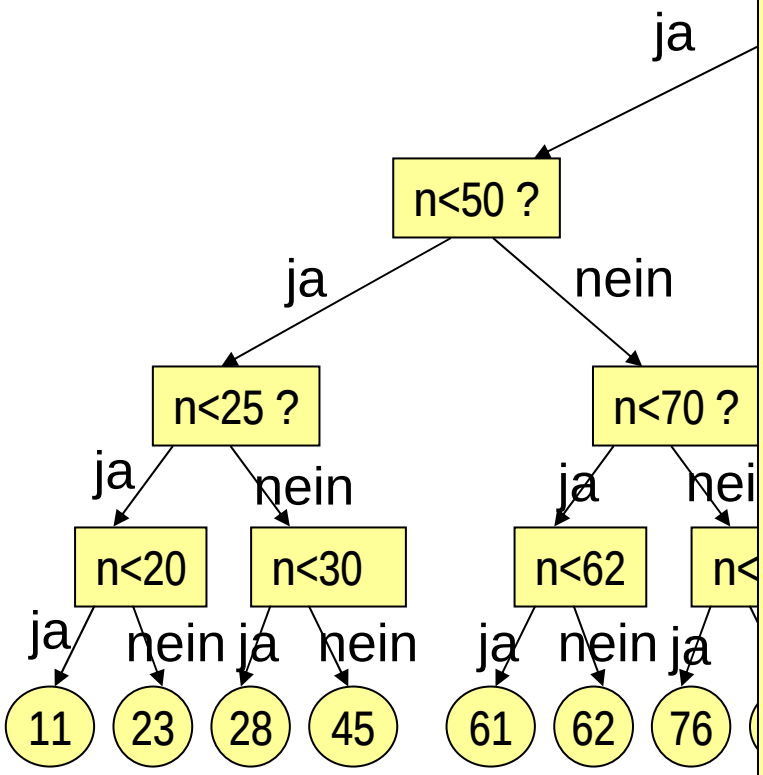


(Über Index zugreifbare Daten)

Indexstr

log(*n*) Tests bei günstig gewählter Aufteilung

[11,23,28,45,




```
if n<90 then
        if n <50 then
                if n<25 then
                        if n<20 then return 11
                        else return 23;
                else
                        if n<30 then return 28
                        else return 45;
        else
                if n<70 then
                        if n<62 then return 61
                        else return 62;
                else
                        if n<80 then return 76
                        else return 81;
else
        if n <1500 then
                if n<600 then
                        if n<200 then return 112
                        else return 256;
                else
                        if n<1100 then return 1002
                        else return 1211;
        else
                if n<3000 then
                        if n<2000 then return 1744
                        else return 2006;
                else
                        if n<4000 then return 3007
                        else return 5002;
```

Keller (Stapel, Stack, LIFO)

Liste K=[$K(1)$, $K(2)$, ..., $K(n)$] mit beschränktem Zugriff

Operationen:

pop: liefert oberstes Element $K(1)$
entfernt oberstes Element: $K' = [\ K(2), ..., K(n)\ ]$
(Fehler bei leerem Keller)

push(x): speichert Element x als oberstes Element:
$K' = [\ x, K(1), K(2), ..., K(n)\ ]$ (Fehler bei vollem Keller)

isEmpty: "true", falls Keller leer

isFull: "true", falls Keller voll

clear: Keller leeren

Keller

Anwendungen

Suchverfahren in Graphen (später mehr dazu):
Tiefe-Zuerst-Suche, Back-Tracking:
Speicherung der offenen Knoten (Liste *OPEN*)

Textverarbeitung: *„rückgängig“-Befehle*

Syntax-Analyse von Programmen

Auswertung von Ausdrücken

Laufzeitkeller

Auswertung von Ausdrücken

Darstellung in
umgekehrt polnischer Notation
(postfix-Darstellung)

x + y * (z-x) + 12 * x

= x y z x - * + 12 x * +

```
push(x)
push(y)
push(z)
push(x)
push(pop - pop)
push(pop * pop)
push(pop + pop)
push(12)
push(x)
push(pop * pop)
push(pop + pop)
```

Prozedur-Keller

Prozedur-Segmente enthalten:

Resultat

Parameter

Lokale Variable

Rücksprung-Adresse

```
int fak(int n)
 {  if (n==0) { return 1;}
    else { return n * fak(n-1); }
 }
```

Push bei Aufruf einer Prozedur

Pop bei Beendigung einer Prozedur

Keller: Einsatz für Handlungsplanung

Push:

Planschritte in umgekehrter Reihenfolge ablegen.

Pop:

Nächsten Planschritt bearbeiten:

- Einfache Aktion ausführen bzw.
- (Verfeinerten) Teilplan zur Umsetzung des Planschritts einkellern

Warteschlangen (Queue, FIFO)

Liste $Q = [\ Q(1), Q(2), ..., Q(n)\ ]$ mit beschränktem Zugriff

Operationen:

dequeue: liefert erstes Element $Q(1)$
entfernt erstes Element: $Q' = [\ Q(2), ..., Q(n)\ ]$
(Fehler bei leerer Schlange)

enqueue(x) :
speichert Element x als letztes Element:
$Q' = [\ Q(1), Q(2), ..., Q(n), x\ ]$
(Fehler bei voller Schlange)

isEmpty, isFull, clear ...

Warteschlangen (Queue, FIFO)

Implementation:

-- verkettete Liste
-- "Ring-Array" mit Positionsreferenz
`front`: aktuell erstes Element
`rear`: aktuell letztes Element

Anwendung:

- Suchverfahren in Graphen
  Breite-Zuerst-Suche:
  Speicherung der offenen Knoten (Liste *OPEN*)

Graphen

Definitionen

Ein *(gerichteter) Graph* ist ein Paar $G = [V, E]$ mit

einer Menge V von *Knoten* ("vertex", "node") und

einer Menge $E \subseteq V \times V$ von Kanten ("edge"):

"Die Kante $[v_1,v_2]$ führt von v_1 nach v_2."

Allgemeiner (Mehfachkanten):

$G = [V, E , f]$ mit f: $f: E \rightarrow V \times V$ (Inzidenz-Funktion)

2. Bei einem *ungerichteten Graphen* $G = [V, E]$ ist die Relation E symmetrisch:

$[v_1,v_2] \in E \leftrightarrow [v_2,v_1] \in E$

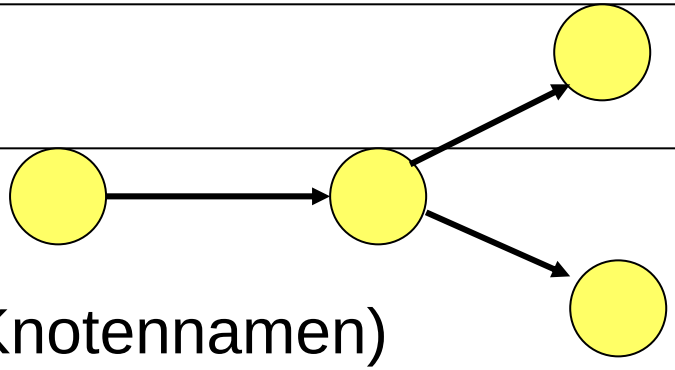
Graphen

3. *Knoten-beschrifteter Graph*: 4-Tupel $G = [V, E, A, \alpha]$ mit
   $[V, E]$ ist ein Graph,
   A ist eine Menge (von Beschriftungen).
   α ist ein Funktion $\alpha: V \rightarrow A$

4. *Kanten-beschrifteter Graph*: 4-Tupel $G = [V, E, B, \beta]$ mit
   $[V, E]$ ist ein Graph,
   B ist eine Menge (von Beschriftungen).
   β ist ein Funktion $\beta: E \rightarrow B$

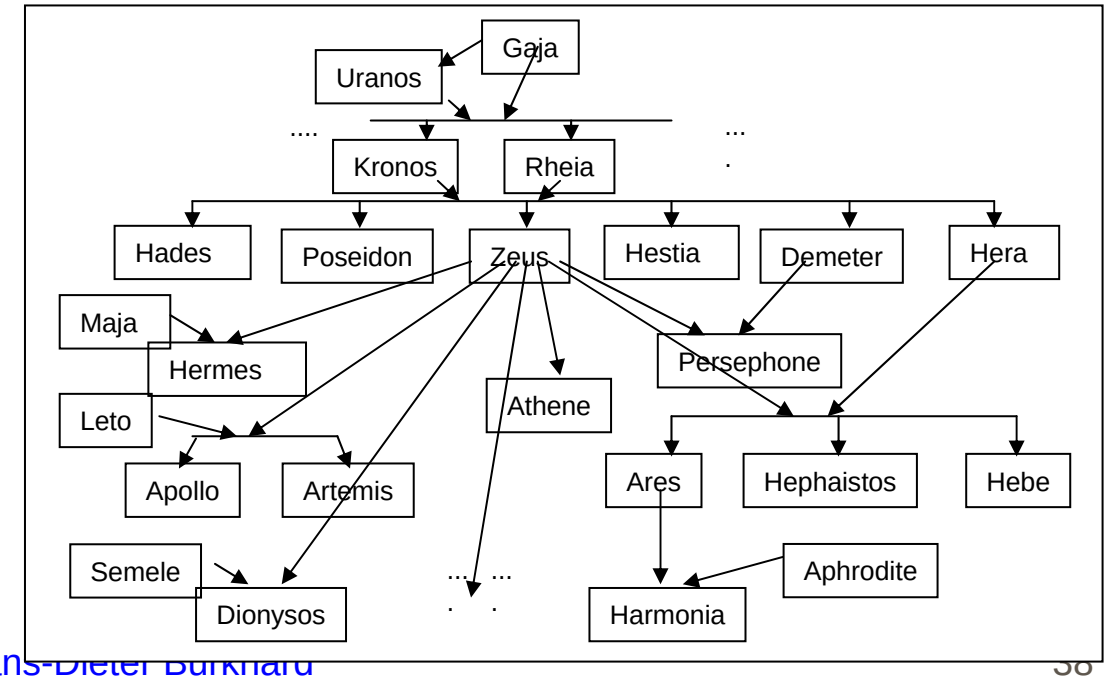
5. *Beschrifteter Graph:* 6-Tupel $G = [V, E, A, \alpha, B, \beta]$ mit
   $[V, E, A, \alpha]$ ist ein Knoten-beschrifteter Graph,
   $[V, E, B, \beta]$ ist ein Kanten-beschrifteter Graph.

Darstellungsformen



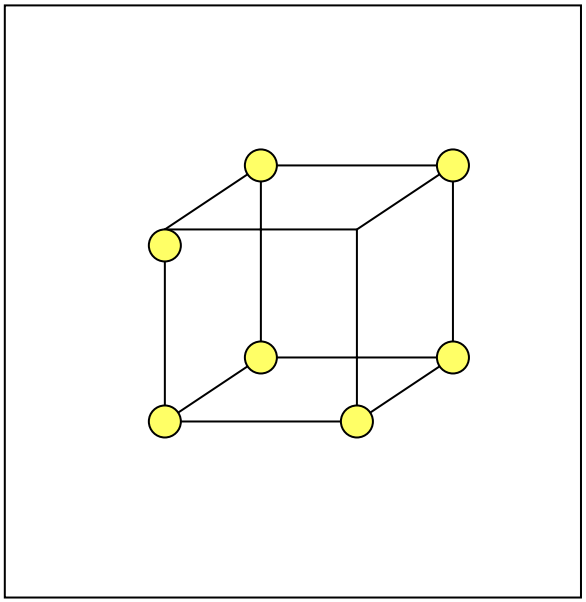
Graphisch:

- Knoten (z.B.) als Kreise (mit Knotennamen)
- Kanten als Bögen bzw. Pfeile (mit Kantennamen)
- Beschriftungen an Kreisen bzw. Bögen



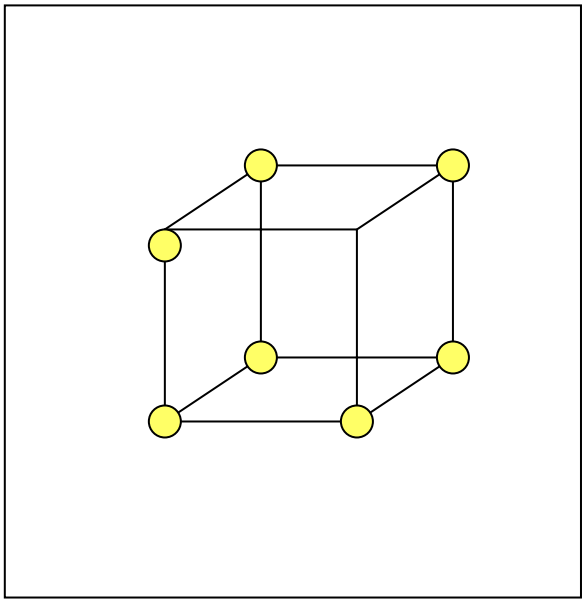
Darstellungsformen

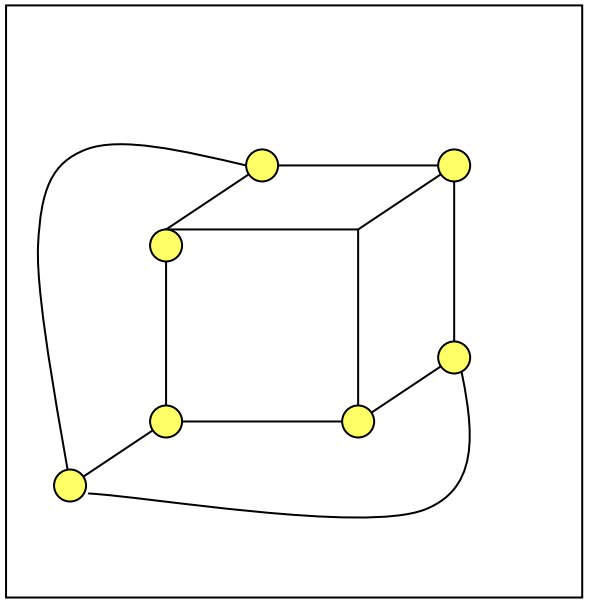
Planarer Graph: 2-D-Darstellung ohne Kreuzung von Kanten



Darstellungsformen

Planarer Graph: 2-D-Darstellung ohne Kreuzung von Kanten

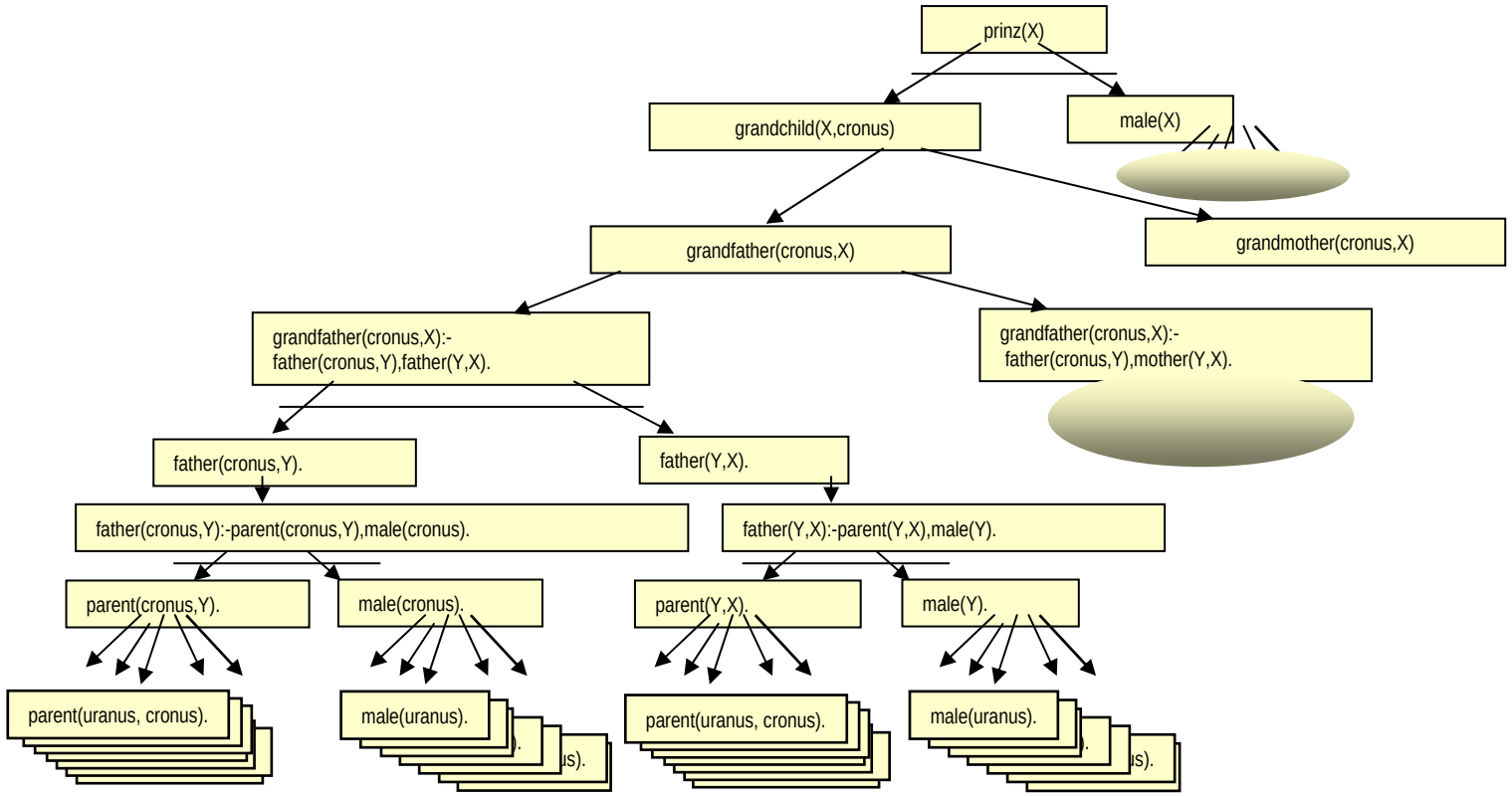




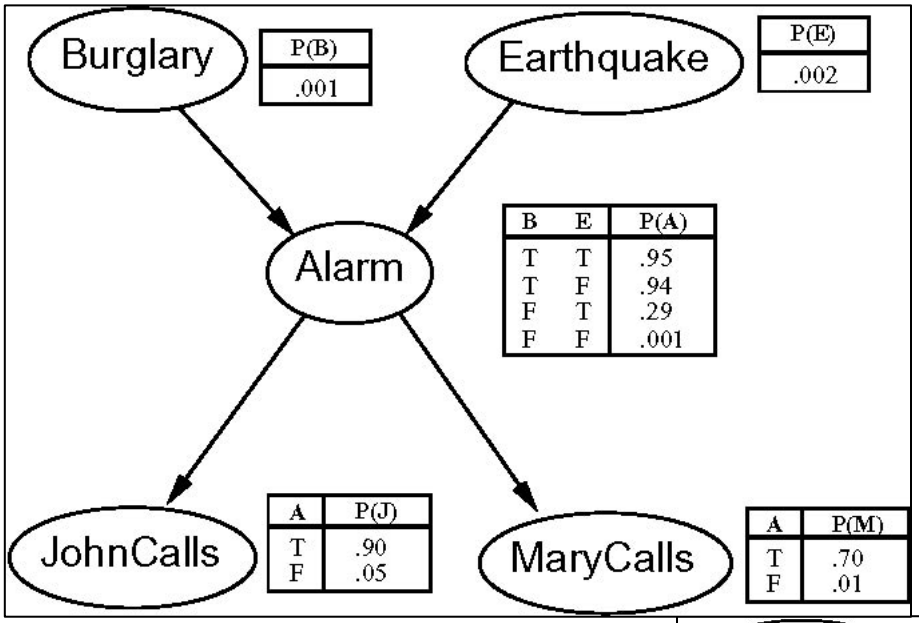
Beweisbaum

2 Sorten von Knoten: „bipartiter Graph“

an Und-Verzweigungen bzw. Oder-Verzweigungen

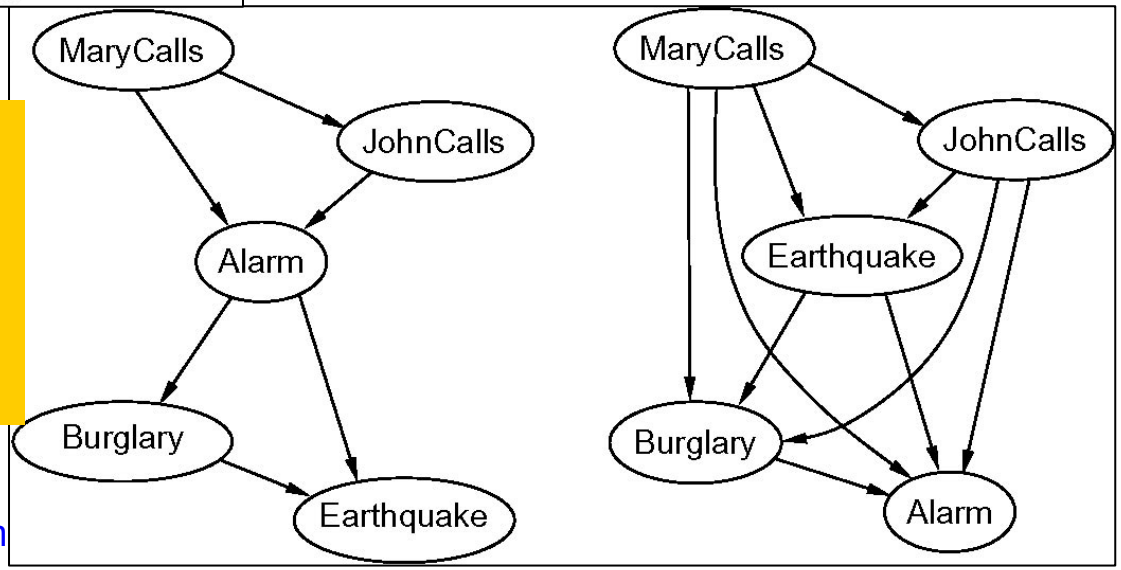


Kausale Abhängigkeiten: Belief-Netze

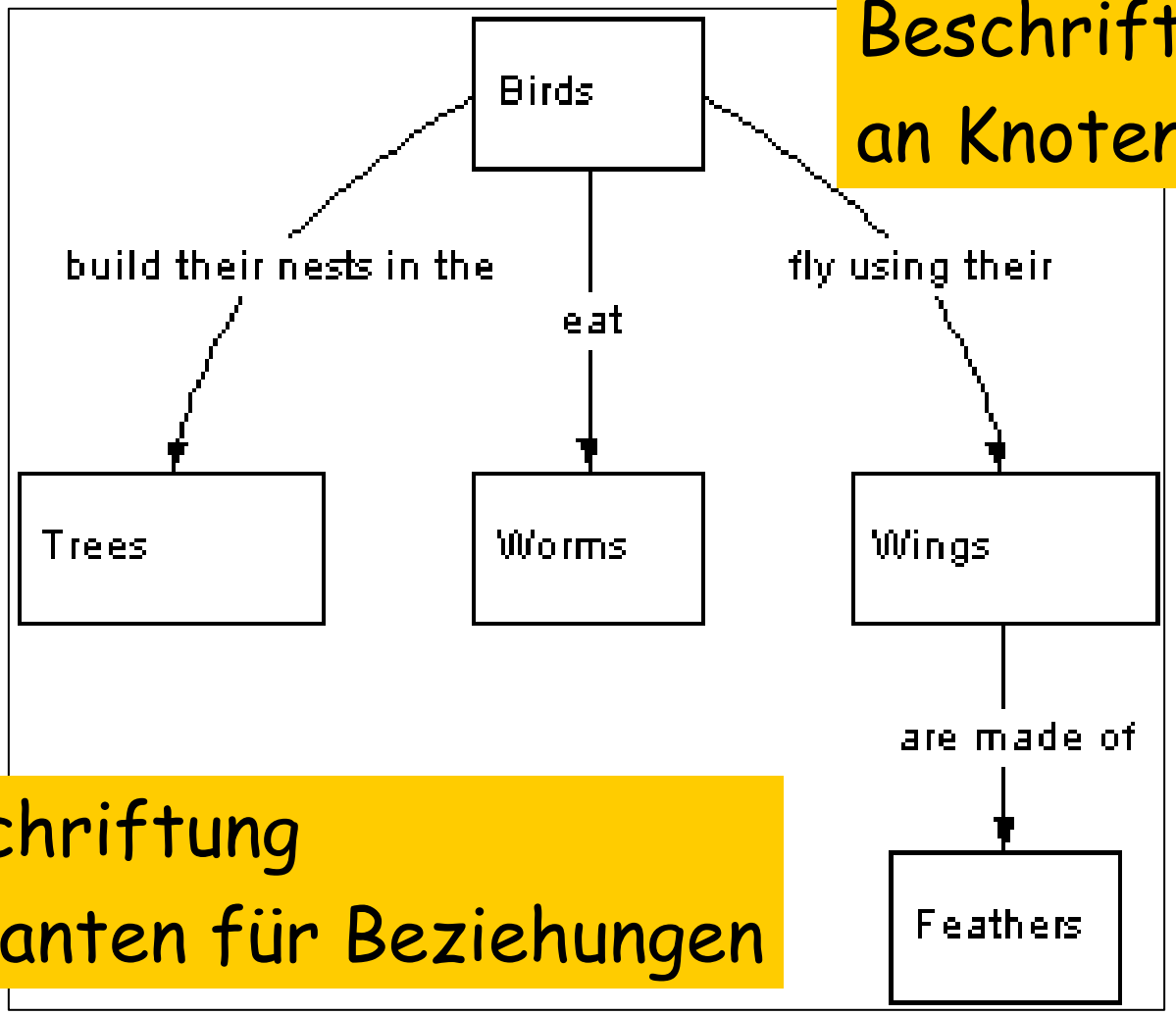


Beschriftung
an Knoten für bedingte
Wahrscheinlichkeiten

Unterschiedliche
Ansätze für
Abhängigkeiten



Sprachliche Zusammenhänge: Semantische Netze

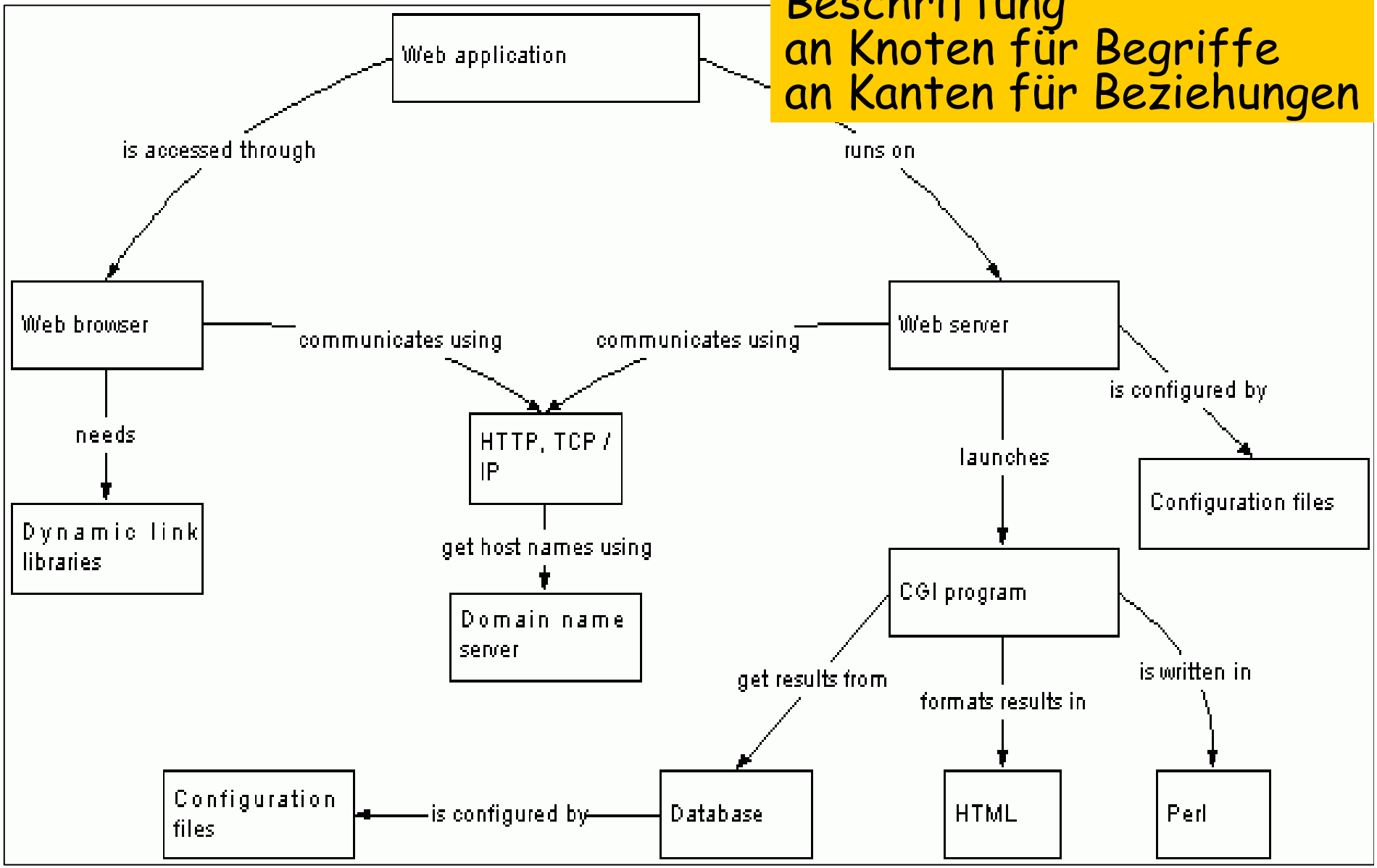


Beschriftung
an Knoten für Begriffe

Beschriftung
an Kanten für Beziehungen

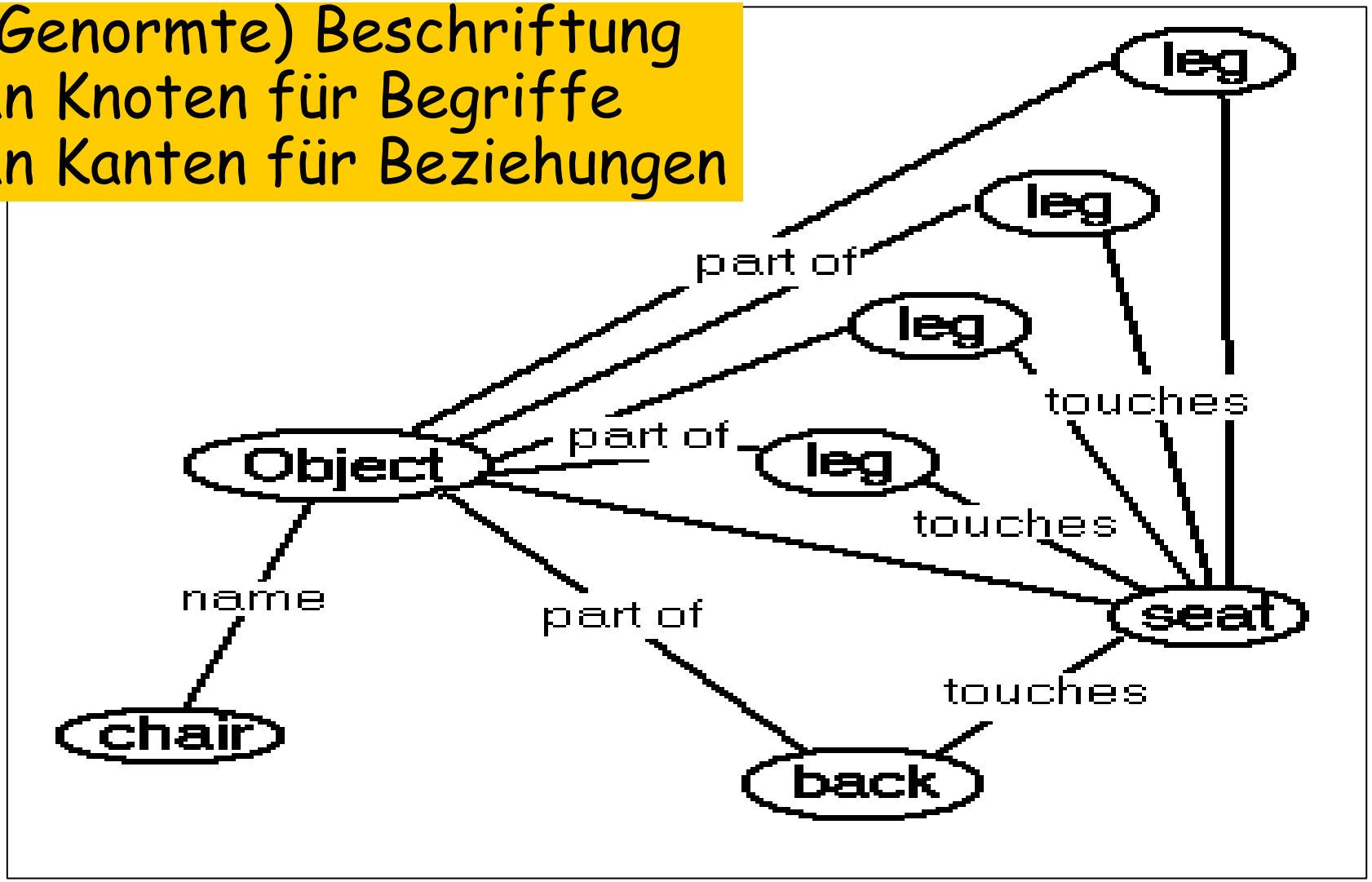
Inhaltliche Zusammenhänge

Beschriftung
an Knoten für Begriffe
an Kanten für Beziehungen



Inhaltliche Zusammenhänge

(Genormte) Beschriftung
an Knoten für Begriffe
an Kanten für Beziehungen



UML: Unified Modeling Language

- Graphische Notationen für Objektorientierte Modellierung
- Hilfsmittel für Erzeugung von Code
- Hilfsmittel für Dokumentation und Test

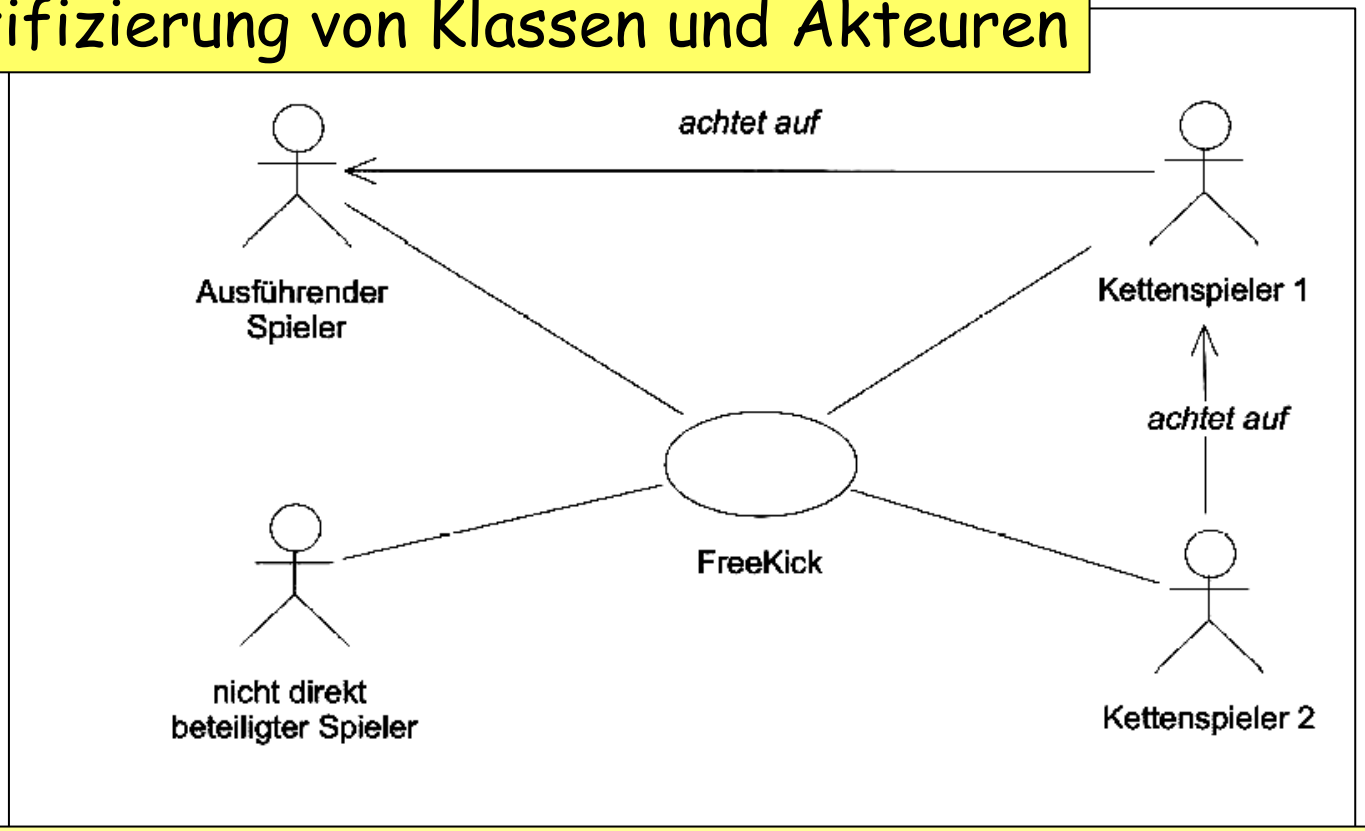
Graphische Darstellungen für ...

- Anwendungsfall-Beschreibung (use-case)
- Klassen/Objekte (Struktur)
  - –Vererbung
  - –Assoziationen: Beziehungen
  - –Aggregationen: Bestandteile („Container“)
- Sequenzdiagramm (Ablauf von Interaktionen)
- Zustandsdiagramm (Ablauf innerhalb eines Objekts)
- Aktivitätsdiagramm (Zusammenarbeit)

... und vieles mehr

Anwendungsfall-Beschreibung (use-case)

Funktionale Anforderungen
Erste Identifizierung von Klassen und Akteuren



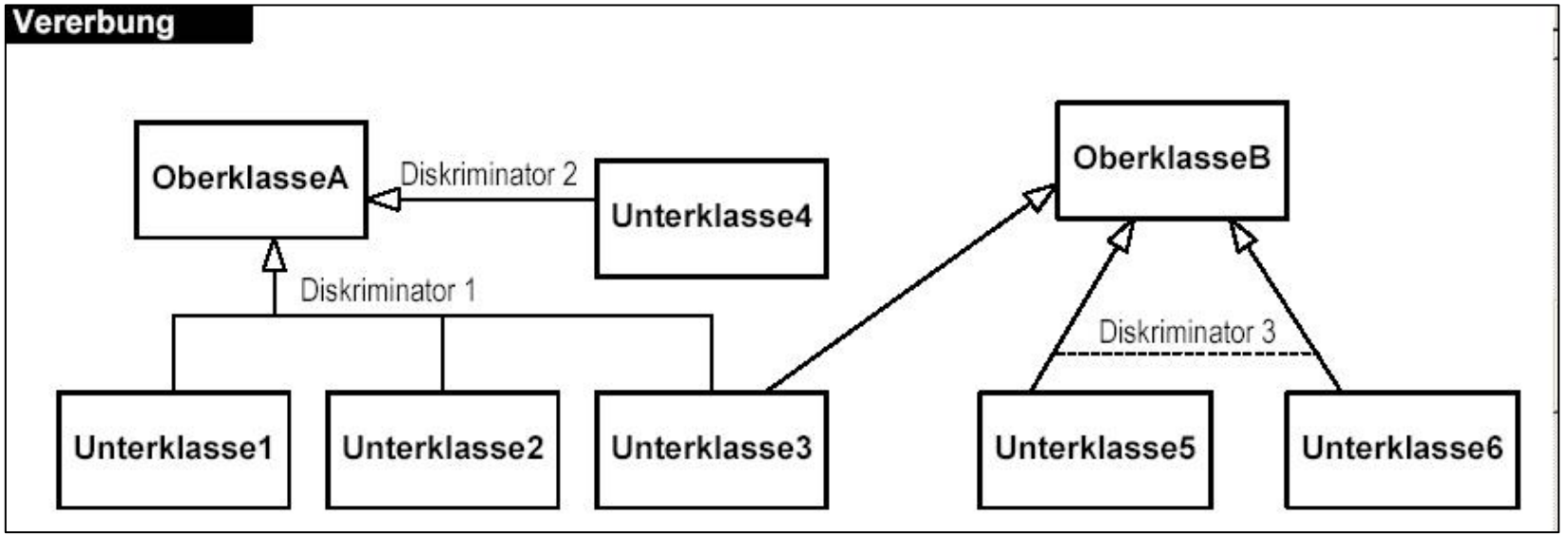
Beispiele:

Diplomarbeit Meinert/Sander: Der Intentionagent (HU, 2001)

Beschreibung von Strukturen: Klassen/Objekte

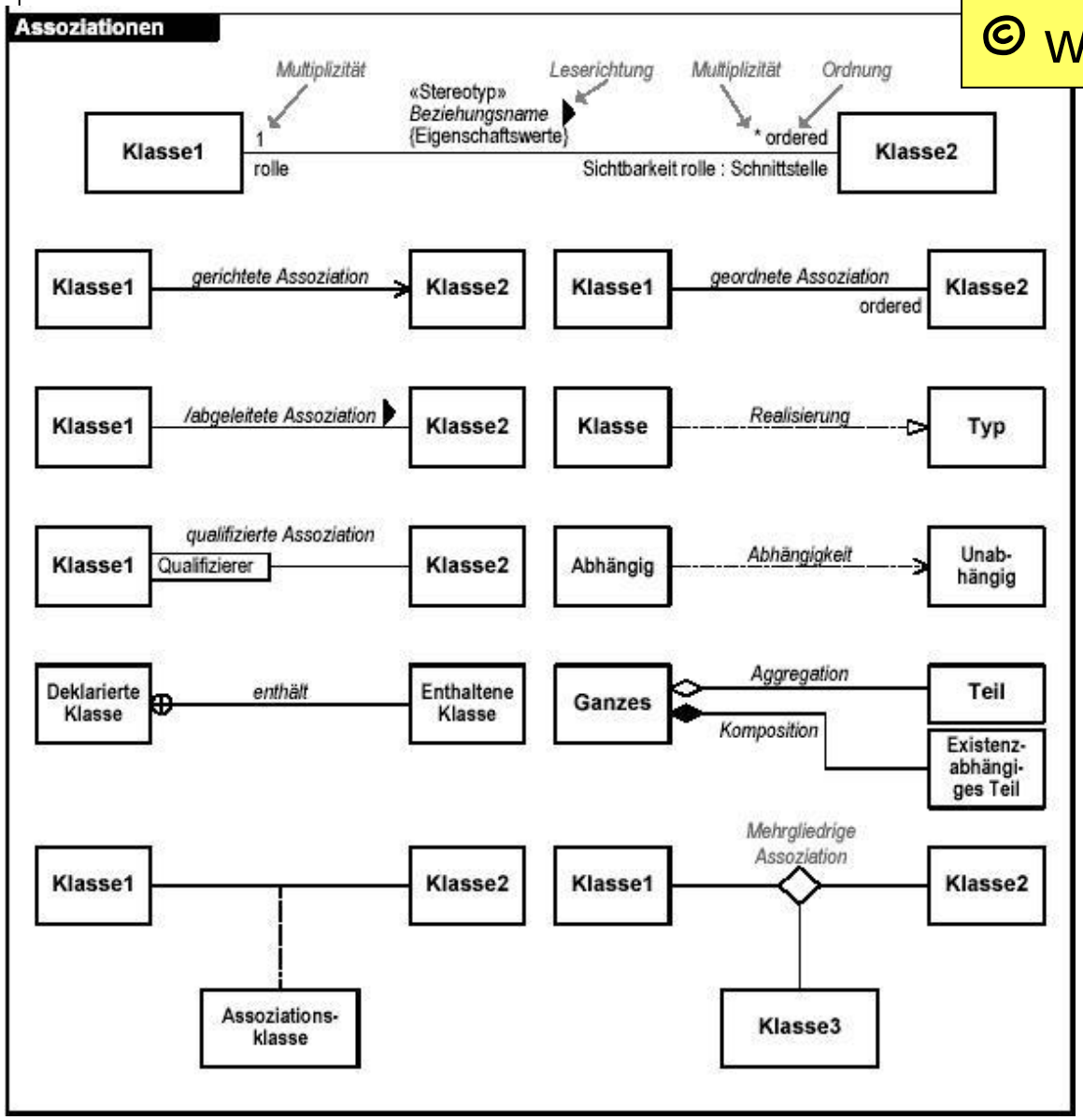
Diagramme zu den Beispielen (Stand 2003):

© http://www.oose.de/downloads/uml-notationsuebersicht.pdf



Klassendiagramm: Klassen und ihre Beziehungen

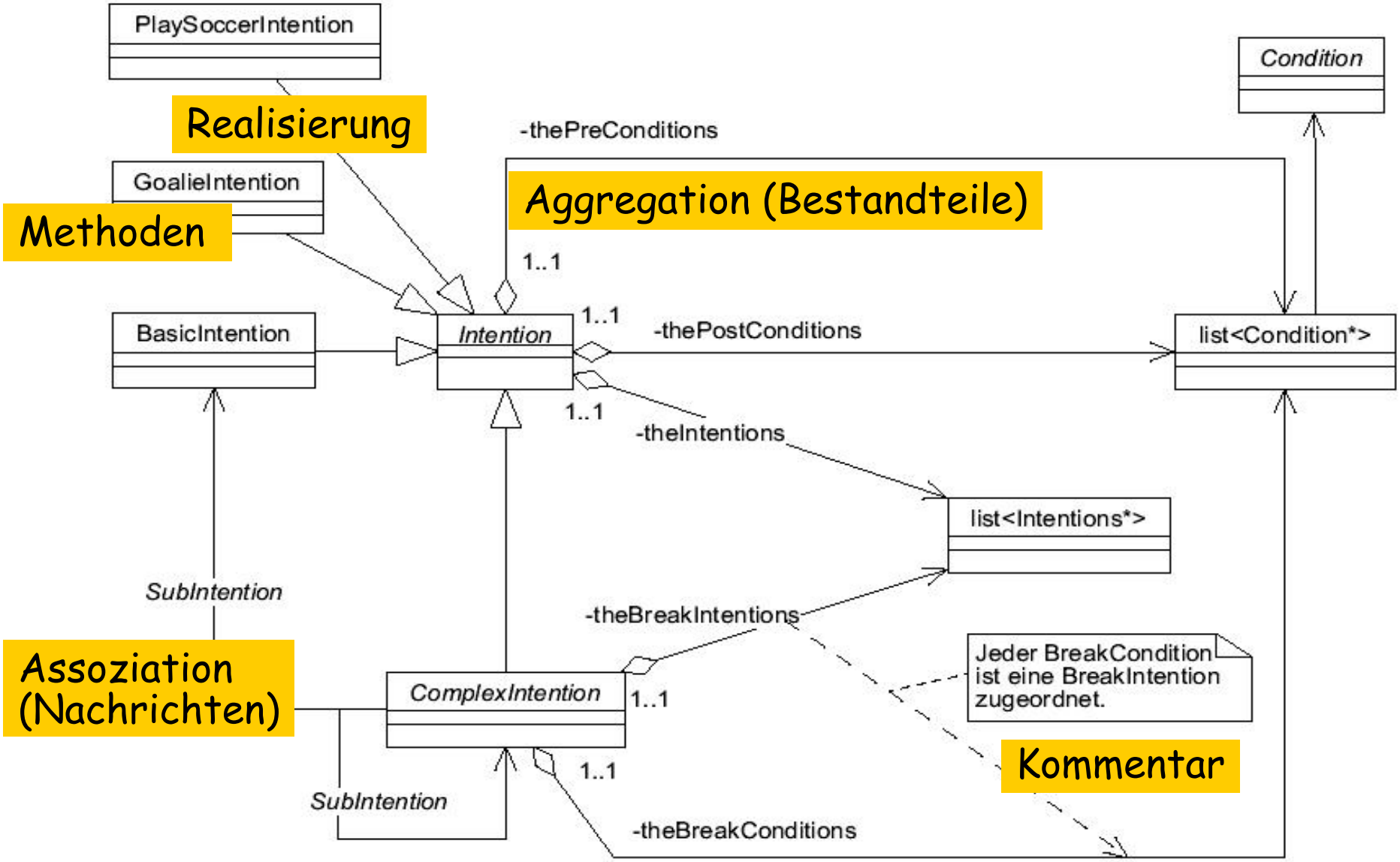
Beschreibung von Strukturen: Klassen/Objekte



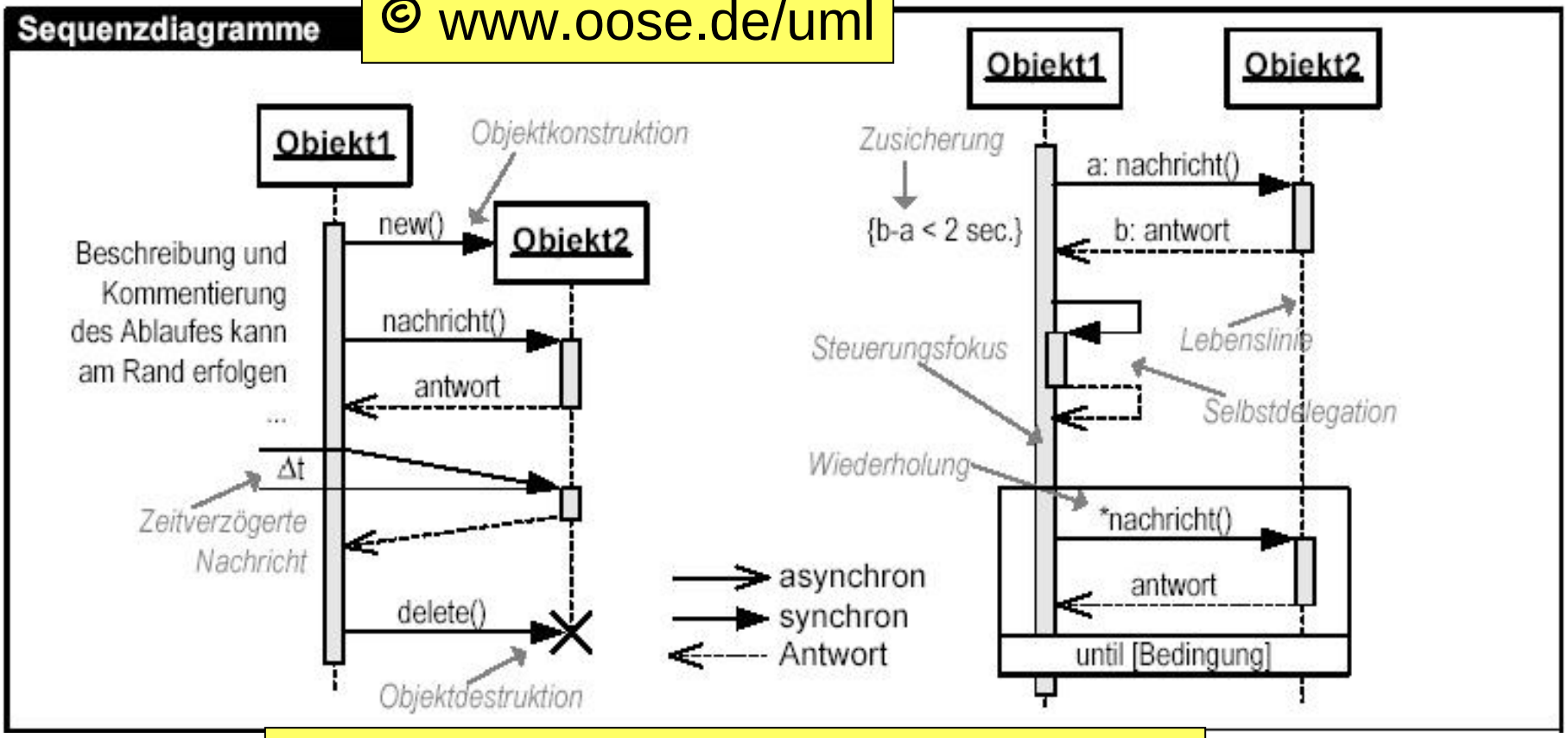
© www.oose.de/uml

Objektdiagramm:
Objekte und
ihre Beziehungen

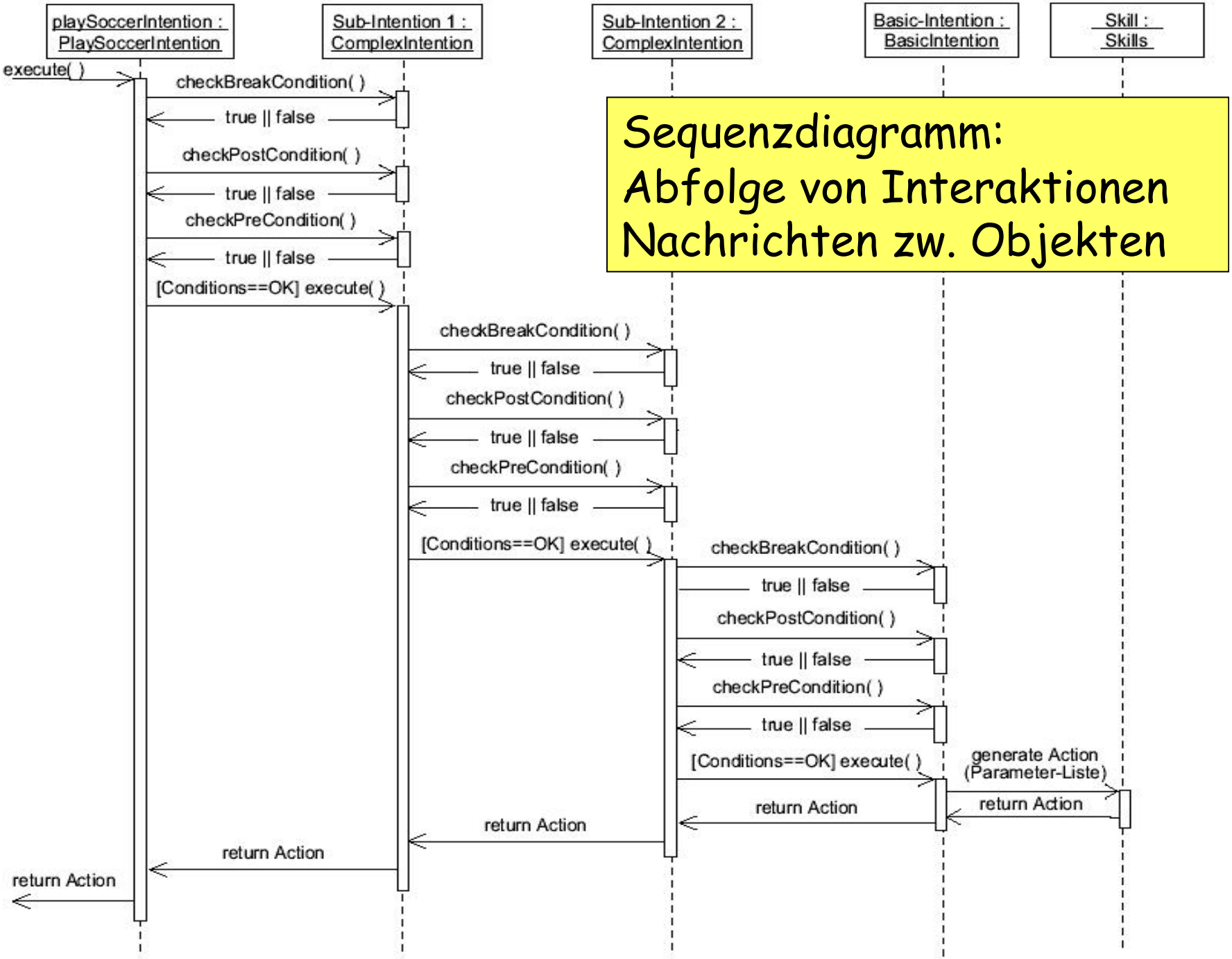
Beschreibung von Strukturen: Klassen/Objekte



Beschreibung von Abläufen: Sequenzdiagramme

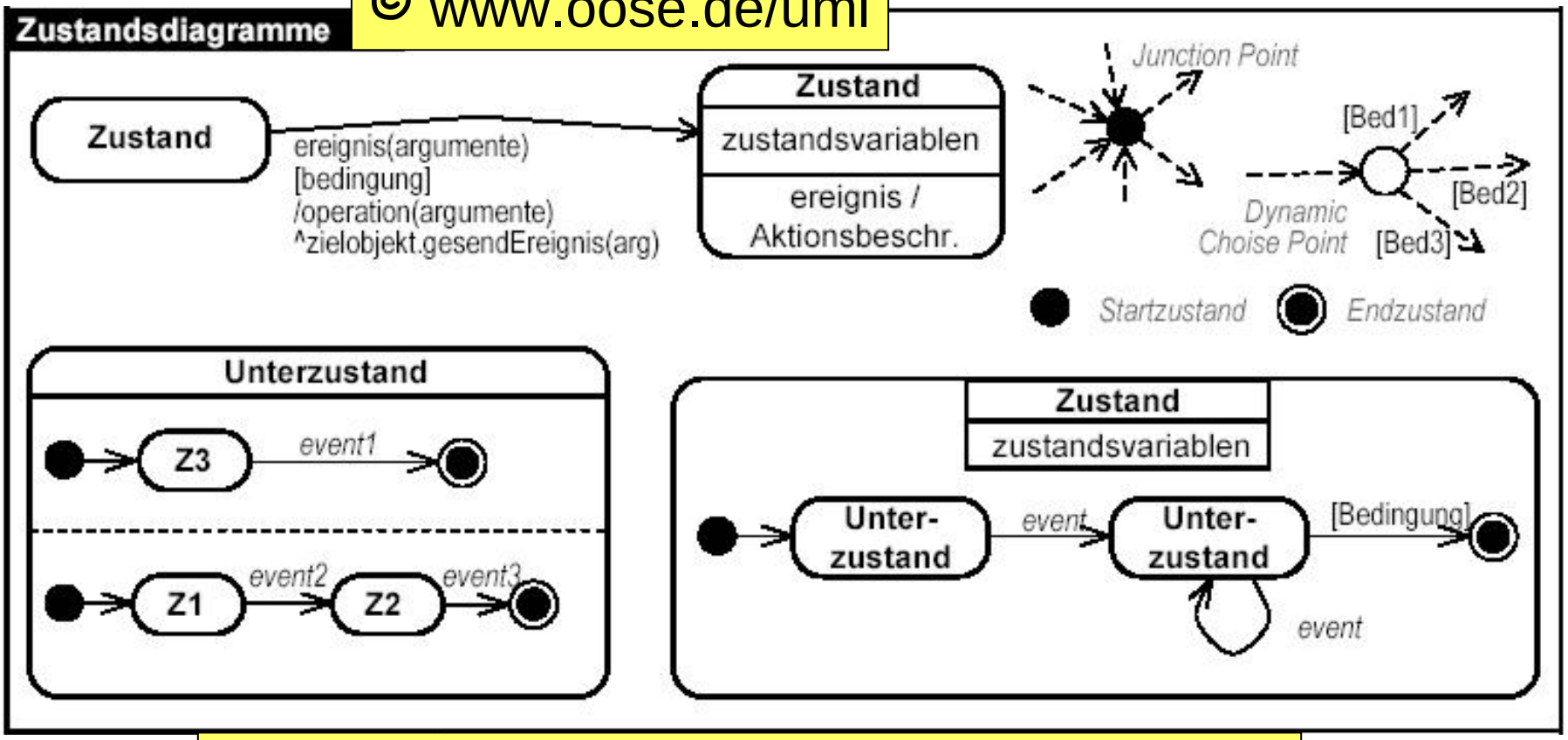


Sequenzdiagramm:
Abfolge von Interaktionen
Nachrichten zwischen Objekten



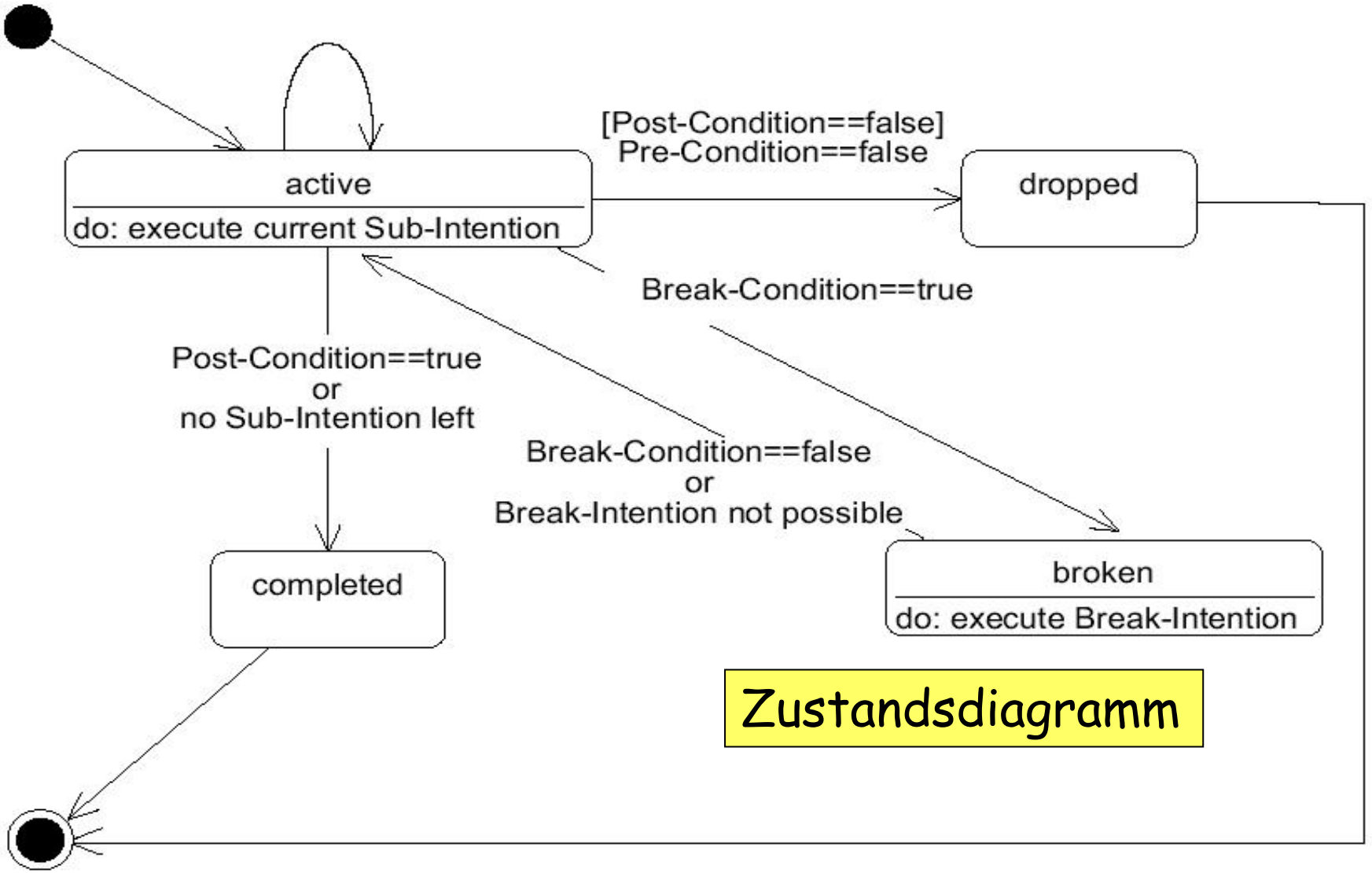
Sequenzdiagramm: Abfolge von Interaktionen Nachrichten zw. Objekten

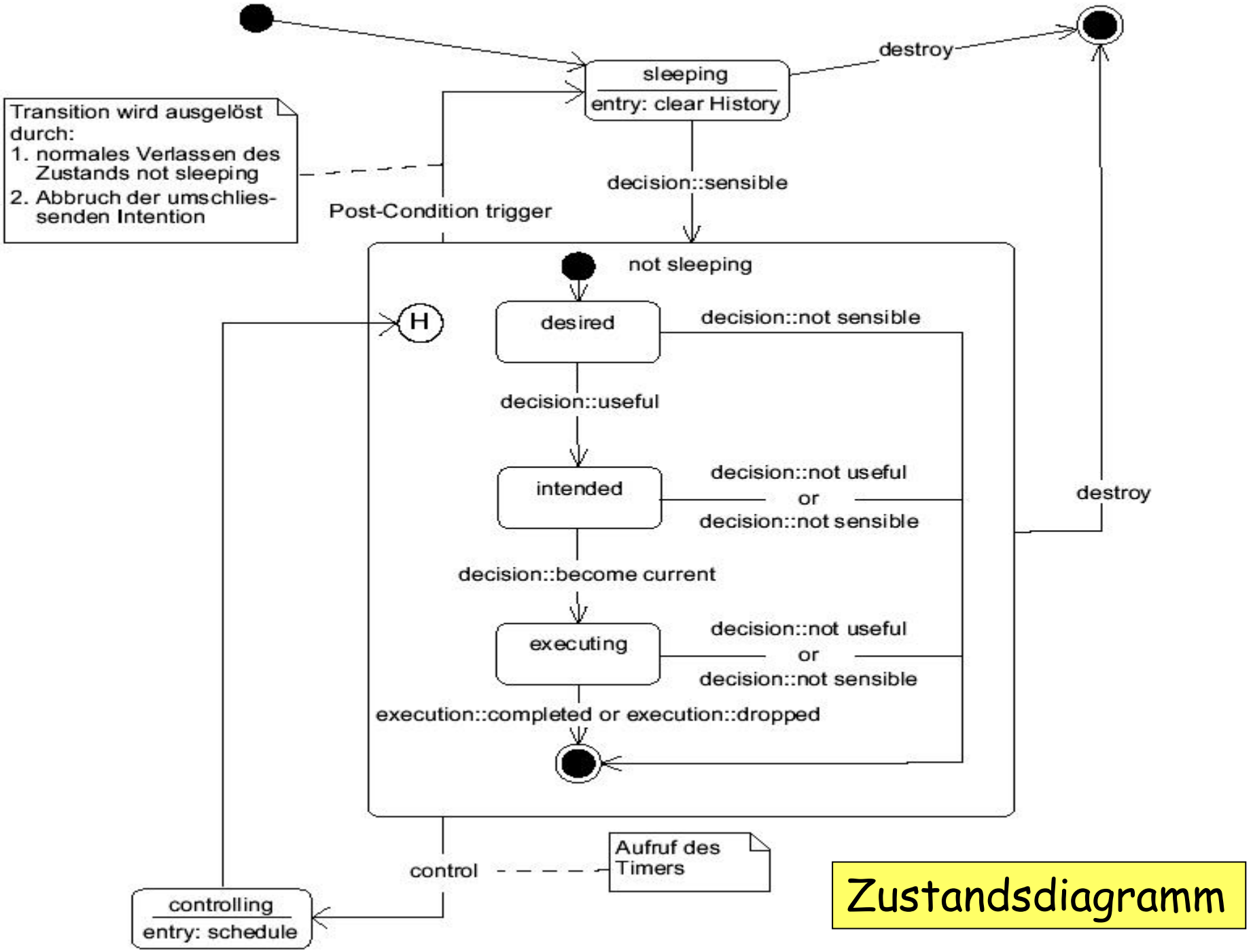
Beschreibung interner Abläufe: Zustandsdiagramm



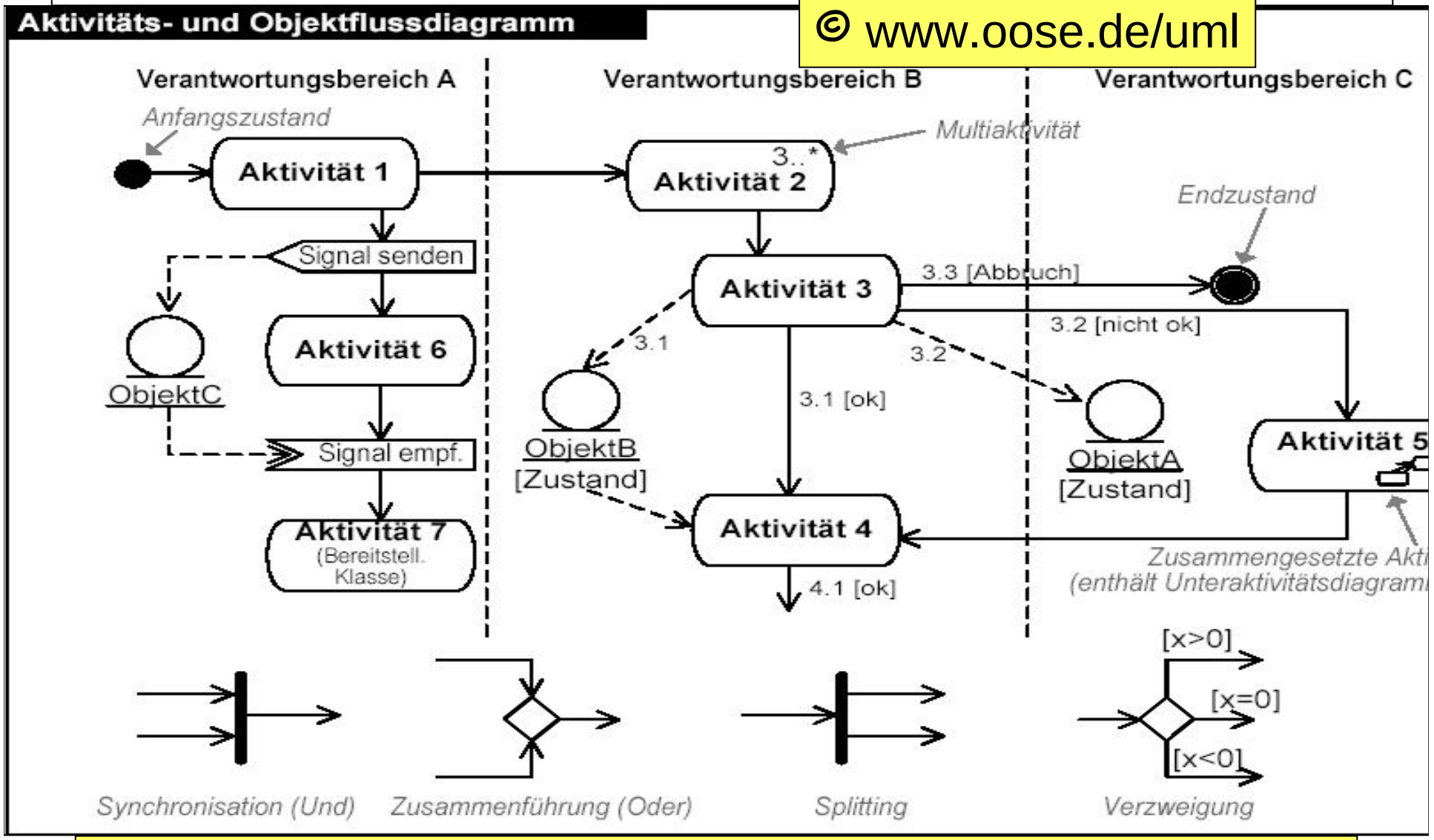
Zustandsdiagramm:
Abfolge von Ereignissen/Zuständen
während der Lebensdauer eines Objekts

Beschreibung interner Abläufe: Zustandsdiagramm



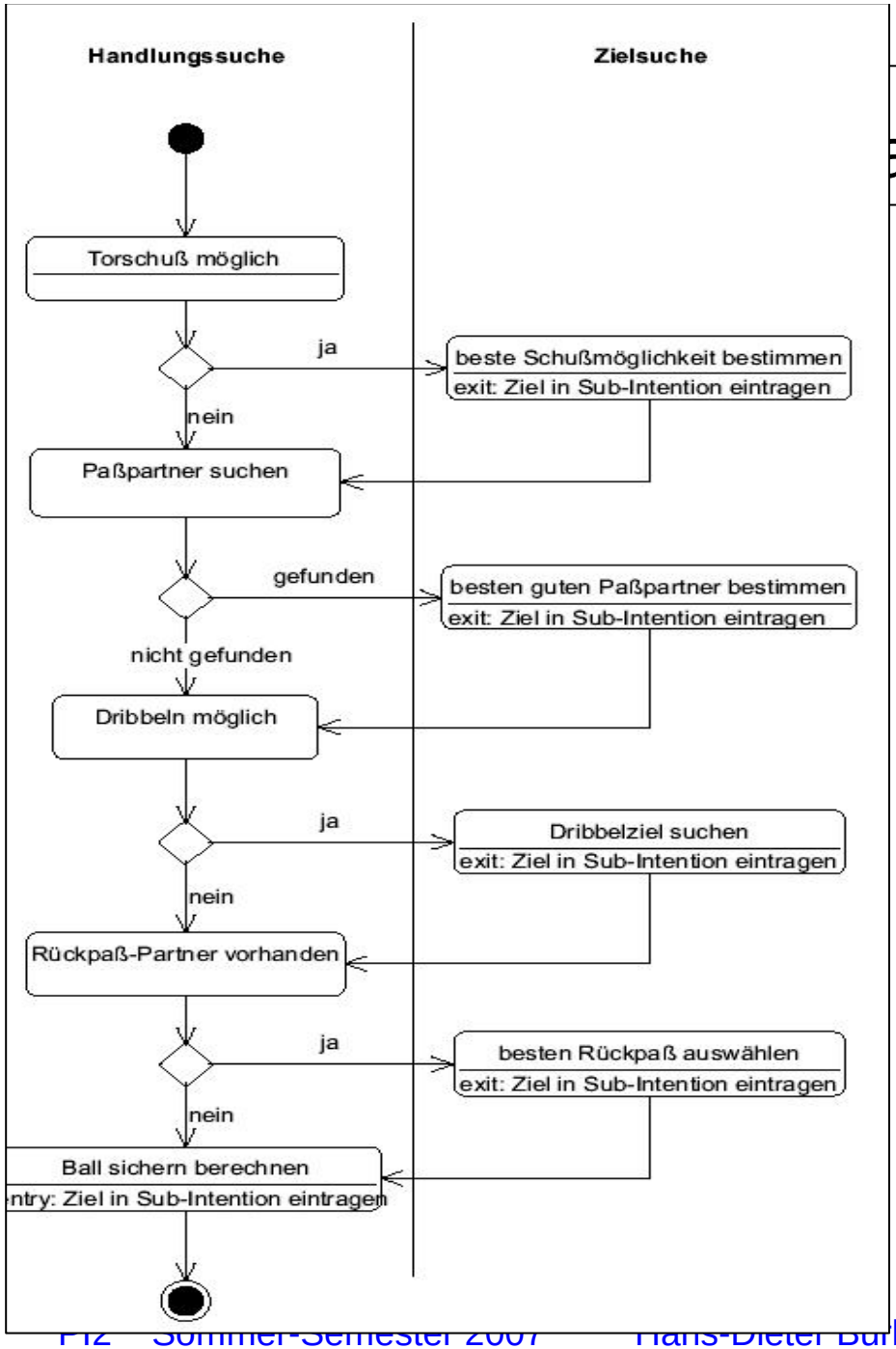

sleeping
entry: clear History
destroy
Transition wird ausgelöst durch:
1. normales Verlassen des Zustands not sleeping
2. Abbruch der umschliessenden Intention
Post-Condition trigger
decision::sensible
not sleeping
H
desired
decision::not sensible
decision::useful
intended
decision::not useful
or
decision::not sensible
destroy
decision::become current
executing
decision::not useful
or
decision::not sensible
execution::completed or execution::dropped
control
Aufruf des Timers
controlling
entry: schedule
Zustandsdiagramm

Beschreibung von Kooperation: Aktivitätsdiagramme

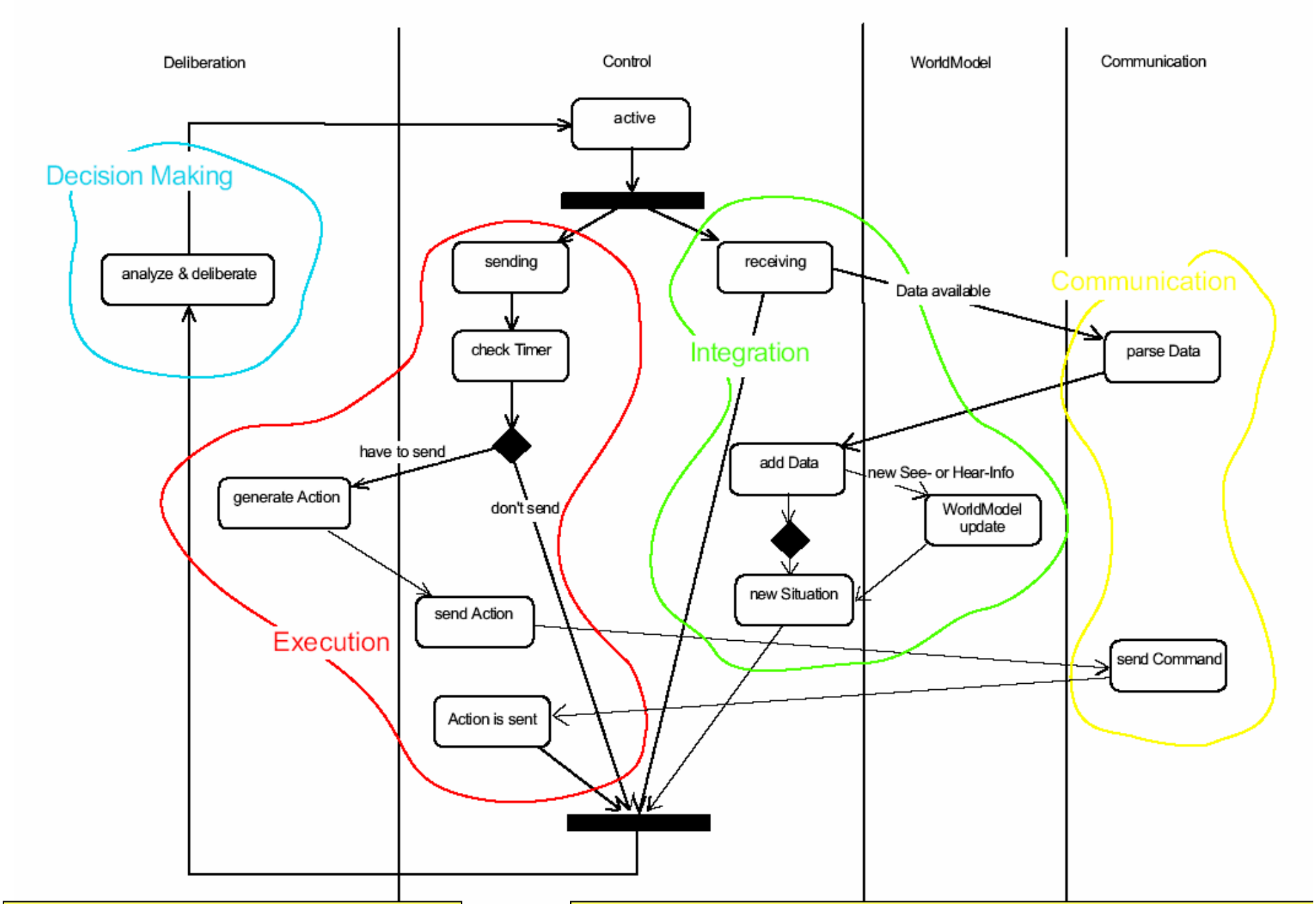


Zusammenarbeit von Objekten, Parallele Arbeit

ation: Aktivitätsdiagramme



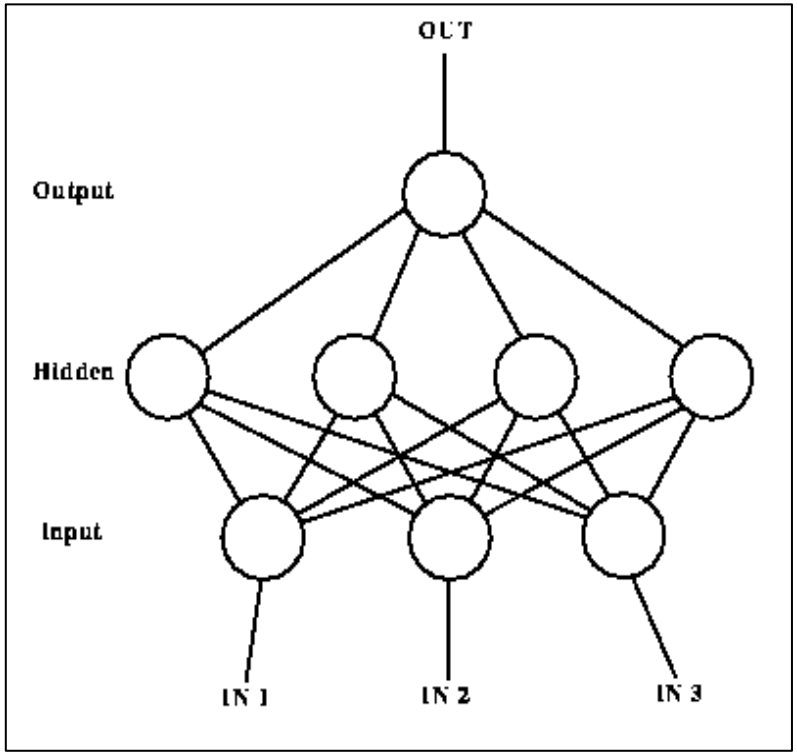
Aktivitätsdiagramm

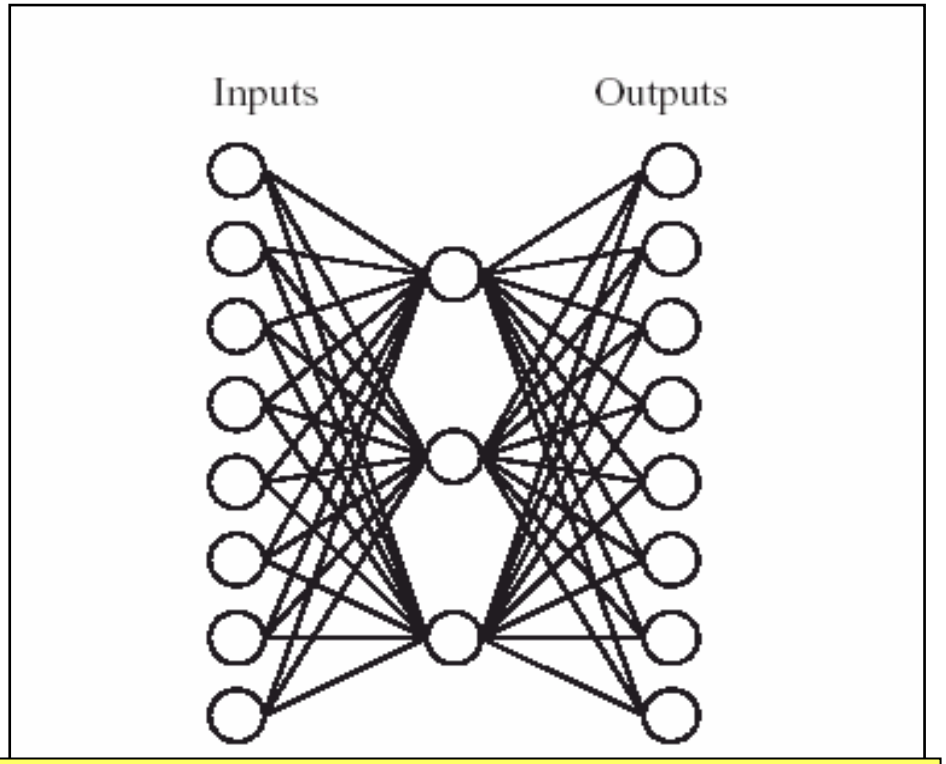


Aktivitätssdiagramm

Identifizierung von Komponenten

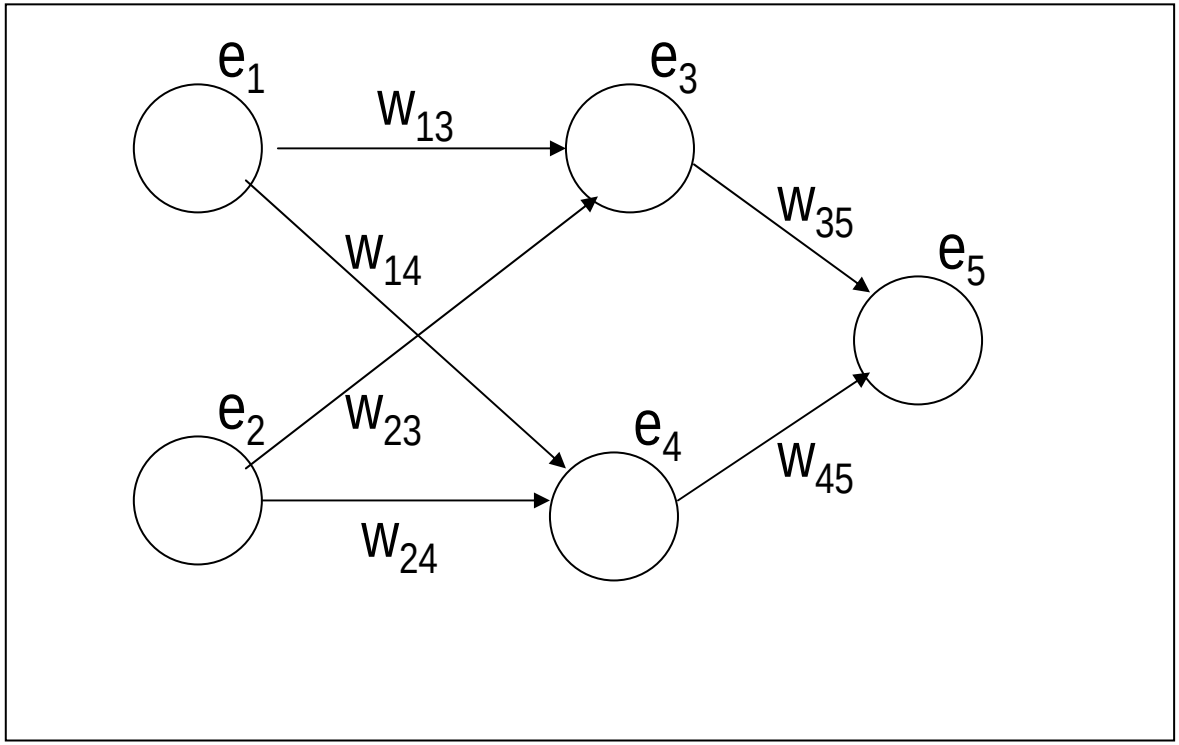
„Massive Parallelität“: Neuronale Netze





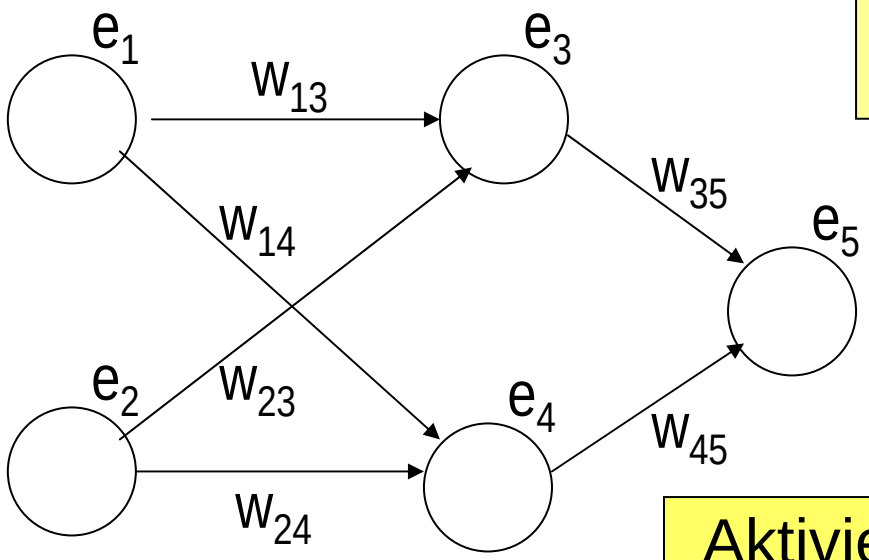
- Knoten: Neuronen
  Neuronen können erregt („aktiviert“) sein
- Kanten: Übertragung von Aktivierungen
  an Nachbar-Neuronen

„Massive Parallelität“: Neuronale Netze



- Kanten sind gewichtet
- Übertragene Aktivierung abhängig von Gewicht w_{ij}

„Massive Parallelität“: Neuronale Netze



$w_{ij} = 0$, falls keine Kante zwischen e_i und e_j

Aktivierung zum Zeitpunkt t :

$act_t : E \rightarrow R$

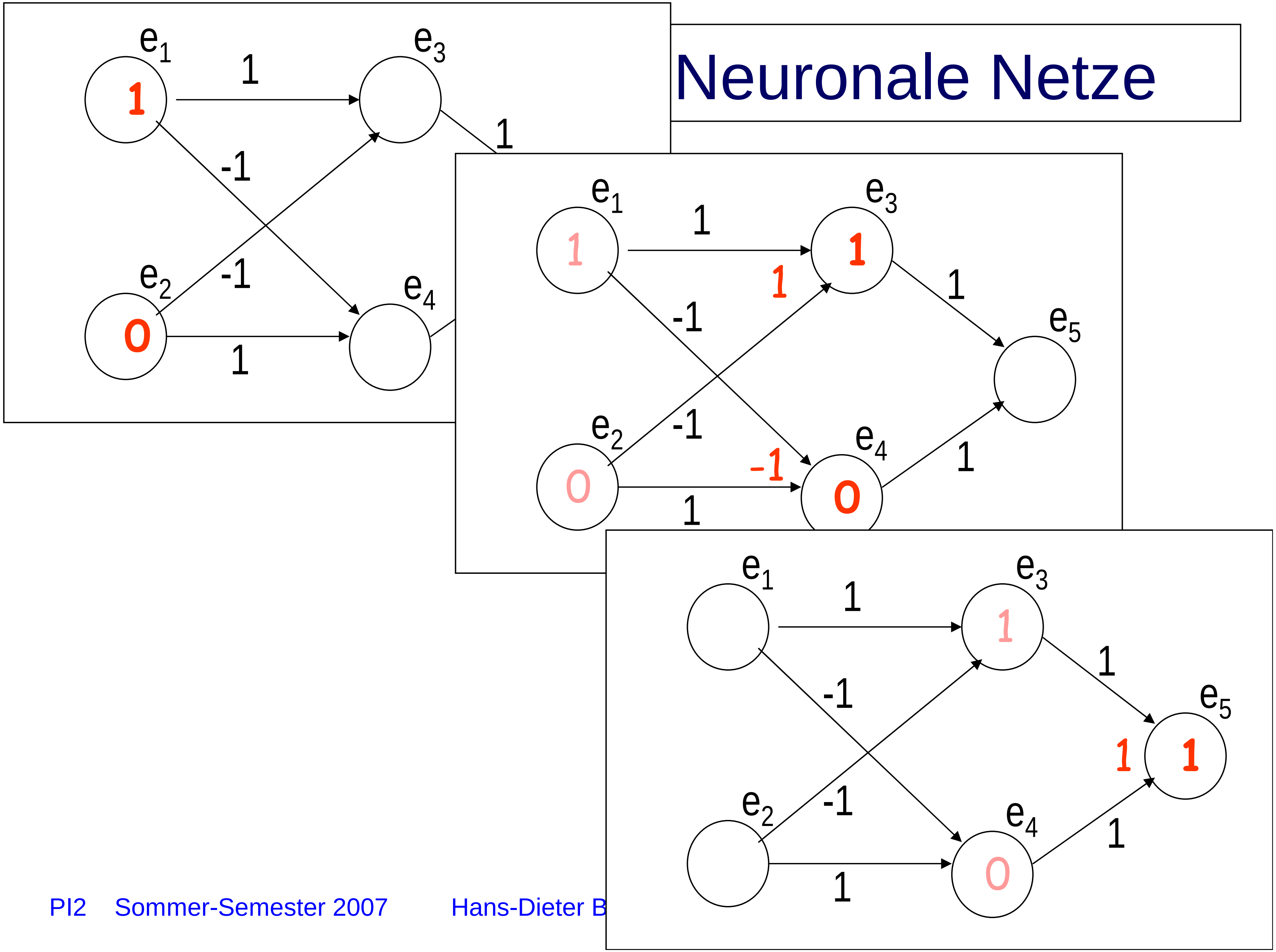
Einfache Propagierung von Aktivierungen z.B. gemäß:

$act_{t+1}(e_i) = 1$,

falls $w_{1i}*act_t(e_1)+w_{21}*act_t(e_2) + ... + w_{5i}*act_t(e_5) > 0$

$act_{t+1}(e_i) = 0$, sonst

Neuronale Netze



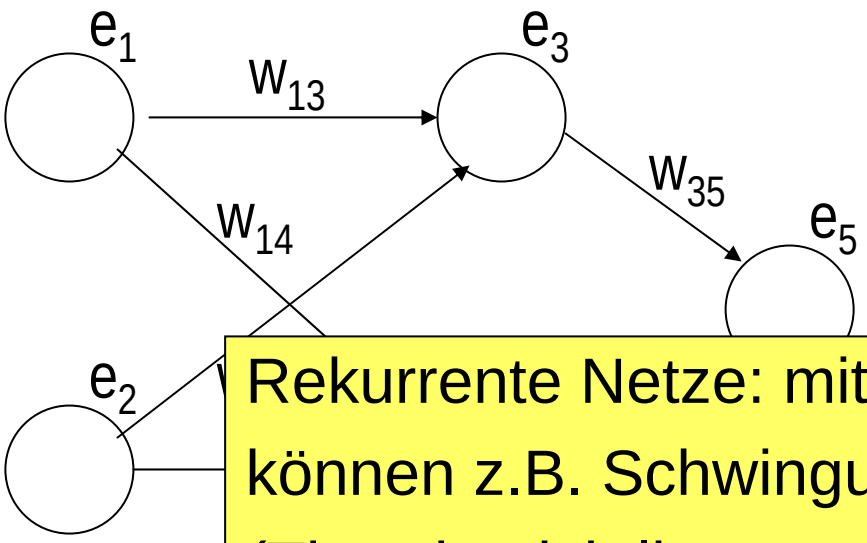
„Massive Parallelität“: Neuronale Netze

Komplexere Propagierung von Aktivierungen:

$act_{t+1}(e_i) = f(w_{1i} * act_t(e_1) + w_{21} * act_t(e_2) + ... + w_{5i} * act_t(e_5))$

z.B. mit $f(x) = \tanh(x) = (e^x - e^{-x}) / (e^x + e^{-x})$

oder $f(x) = sig(x) = 1 / (1 + e^{-x})$

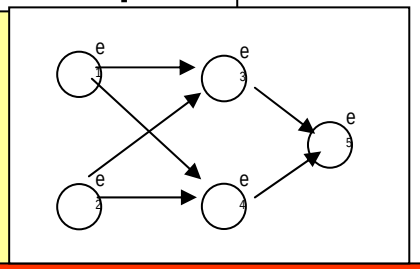


Rekurrente Netze: mit Rückkopplungen,
können z.B. Schwingungen erzeugen
(Theorie nichtlinearer dynamischer Systeme)

Statische/dynamische Modelle

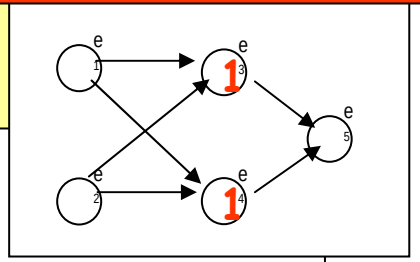
- Beschreibung statischer Zusammenhänge
  - Semantisches Netz
  - Klassendiagramm
  - Zustandsdiagramm (als Struktur)
  - Neuronales Netz (als Struktur)

Modell:
Graph



- Beschreibung dynamischer Abläufe
  - Aktueller Zustand im Zustandsdiagramm
  - Aktivierung im Neuronalen Netz

Unterscheiden!



Modell:
Wechselnde Beschriftungen des Graphen

Erfordert Angaben für Übergänge (Transitionen) zwischen Zuständen, Aktivierungen usw.

Darstellungsformen für Graphen

Adjazenz-Matrix:

- Ausgangs-Knoten v als Zeilen
- Eingangs-Knoten v′ als Spalten

Matrix-Elemente:

$m_{v,v'} = 1$, falls $[v,v'] \in E$

$m_{v,v'} = 0$, falls $[v,v'] \notin E$

bzw. bei Kanten-Beschriftungen auch:

$m_{v,v'} = \beta([v,v'])$

dabei spezieller Wert für $[v,v'] \notin E$, z.B. $\beta([v,v'])=0$

Darstellungsformen

Inzidenz-Matrix:

- Knoten v als Zeilen
- Kanten e als Spalten

Matrix-Elemente:

$m_{v,e} = +1$, falls $e = [v, v'] \in E$

$m_{v,e} = -1$, falls $e = [v', v] \in E$

$m_{v,e} = 0$, sonst

Weitere Definitionen zu Graphen

$G' = [V',E']$ ist Teilgraph von $G=[V,E]$,
falls $V' \subseteq V$ und $E' \subseteq E$ mit $E' \subseteq V' \times V'$

Einschränkung von $G=[V,E]$ auf $V' \subseteq V$:

$$G|V' =_{Def} [V', E \cap V' \times V']$$

Für $e = [v,v'] \in E$: $\quad source(e) =_{Def} v$, $\quad target(e) =_{Def} v'$

Eingangsvalenz (fan-in), Ausgangsvalenz (fan-out)

$$fan\text{-}in(v) =_{Def} card (\{ e \mid \exists v' \in V: e= [v',v] \in E \}) = card (\{ e \in E \mid target(e) = v \})$$

$$fan\text{-}out(v) =_{Def} card (\{ e \mid \exists v' \in V: e= [v,v'] \in E \}) = card (\{ e \in E \mid source(e) = v \})$$

Weg (Pfad) in einem Graphen

Gerichteter Weg

Als Folge von Knoten

$p = v_o, ..., v_n$ mit $\forall i (i \in \{ 1,...,n\} \rightarrow [v_{i-1},v_i] \in E)$

n ist die *Länge* des Weges $(n \geq 0)$

v_o = Anfang(p) v_n =Ende(p)

Als Folge von Kanten

$p = e_1, ..., e_n$ mit $\forall i(i \in \{1,...,n-1\} \rightarrow target(e_i)=source(e_{i+1}))$

Ungerichteter Weg:

$p = v_o, ..., v_n$ mit $\forall i(i \in \{1,...,n\} \rightarrow [v_{i-1},v_i] \in E \vee [v_i,v_{i-1}] \in E)$

Weg (Pfad) in einem Graphen

Einfacher Weg:

$p = v_0, ..., v_n$ mit $\forall i \, \forall j \in \{ 0,...,n \}: v_i = v_j \rightarrow i = j$

Masche:

Zwei unterschiedliche einfache Wege p und q der Länge >0
mit Anfang(p)=Anfang(q) , Ende(p)=Ende(q)

(„Einfache“ Masche: Teilwege bilden keine Masche)

Zyklus in gerichteten Graphen:

Einfacher Weg p der Länge >0
mit Ausnahmebedingung Anfang(p)=Ende(p)

d.h. $\forall i \, \forall j \in \{ 0,...,n \}: v_i = v_j \rightarrow i = j \vee \{i,j\} = \{0,n\}$)

DAG (directed acyclic Graph): gerichteter Graph ohne Zyklen

Erreichbarkeit, Zusammenhang

v′ ist *erreichbar* von v,
falls ein gerichteter Weg p existiert
mit Anfang(p)=v , Ende(p)=v′ .

G heißt *zusammenhängend*,
wenn zu je zwei Knoten $v,v' \in V$
ein ungerichteter (!) Weg p existiert
mit Anfang(p)=v , Ende(p)=v′

G heißt *stark zusammenhängend*,
wenn zu je zwei Knoten $v,v' \in V$
ein gerichteter (!) Weg p existiert
mit Anfang(p)=v , Ende(p)=v′

Die Relation
„Erreichbarkeit“
ist die
reflexive,
transitive Hülle
der Relation E .

wenn jeder Knoten $v' \in V$
von jedem Knoten $v \in V$
erreichbar ist.

Probleme für Graphen

Ist ein Knoten v von einem Knoten v_0 *erreichbar*?

Ist G *(stark) zusammenhängend*?

Ist G *azyklisch?*

Existiert ein *Hamilton-Kreis?*
(Hamilton-Kreis: Zyklus, der ganz V umfaßt).

Existiert ein *Euler-Kreis*?
(Euler-Kreis: Zyklus, der jede Kante genau einmal enthält).

k-Färbungsproblem:
Existiert eine Knotenbeschriftung $\alpha: V \rightarrow \{1,...,k\}$ mit
$\forall v\ \forall v'([v,v'] \in E \rightarrow \alpha(v) \neq \alpha(v'))$

Konstruktion der erreichbaren Knoten

Sei $G=[V,E]$ ein Graph, $v_0 \in V$.

$M(v_0) =_{Def} \{ v \mid v \text{ erreichbar von } v_0 \}$

$M_i(v_0)$ sind die mit Wegen der Länge kleiner gleich i erreichbaren v

Konstruktion:

- Schritt 0: $M_0(v_0) := \{ v_0 \}$
- Schritt i: $M_i(v_0) := M_{i-1}(v_0) \cup \{ v' \mid \exists\, v \in M_{i-1}(v_0) : [v,v'] \in E \}$
- Abbruch, falls $M_i(v_0) = M_{i-1}(v_0)$

Satz:

Der Algorithmus bricht nach höchstens card(V) Schritten ab. Beim Abbruch gilt $M_{i-1}(v_0) = M(v_0)$.

Konstruktion der erreichbaren Knoten

Folgerungen

Für endliche Graphen G ist entscheidbar,

- ob ein Knoten v‘ von einem Knoten v erreichbar ist.
- ob G stark zusammenhängend ist.
- ob G zusammenhängend ist.

Wenn v‘ von einem Knoten v erreichbar ist, so auch mit einem Weg der Länge $l < card(V)$.

Weitere Sätze und Komplexitätsbetrachtungen in Theoretischer Informatik

Baum

Eingangsvalenzen stets 1 oder 0 (Wurzel)

(Gerichteter) Baum: T = [V,E,r]

ist ein gerichteter Graph [V,E]

mit speziellem Knoten (Wurzel) $r \in V$,

von dem aus alle anderen Knoten

auf genau einem Weg erreichbar sind.

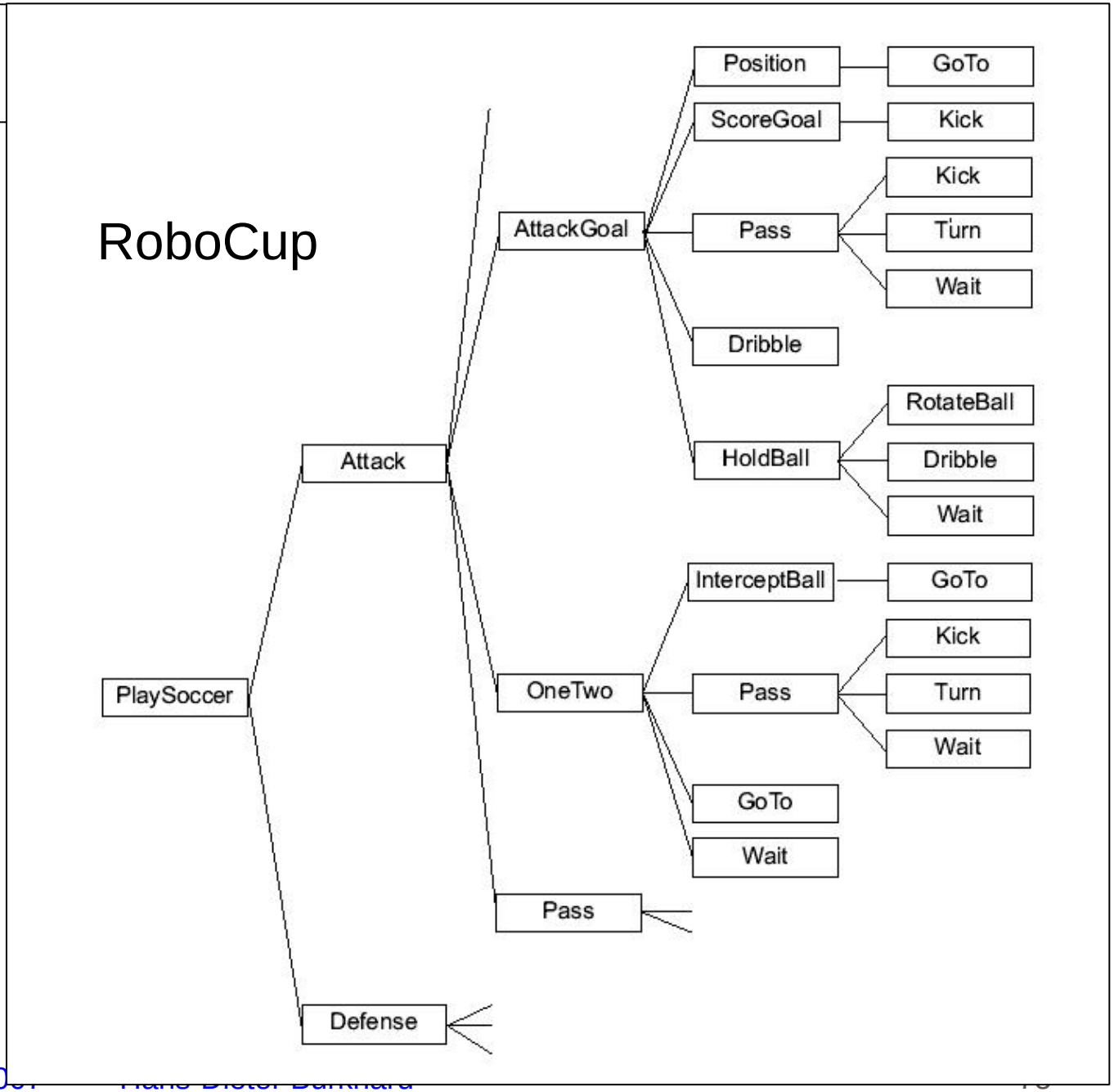
- gerichtet
- zusammenhängend
- ohne Zyklen, ohne Maschen
- Wurzelknoten r eindeutig bestimmt

„Ungerichteter Baum“:
jeder Knoten könnte Wurzelknoten sein

Bäume

Hierarchien:

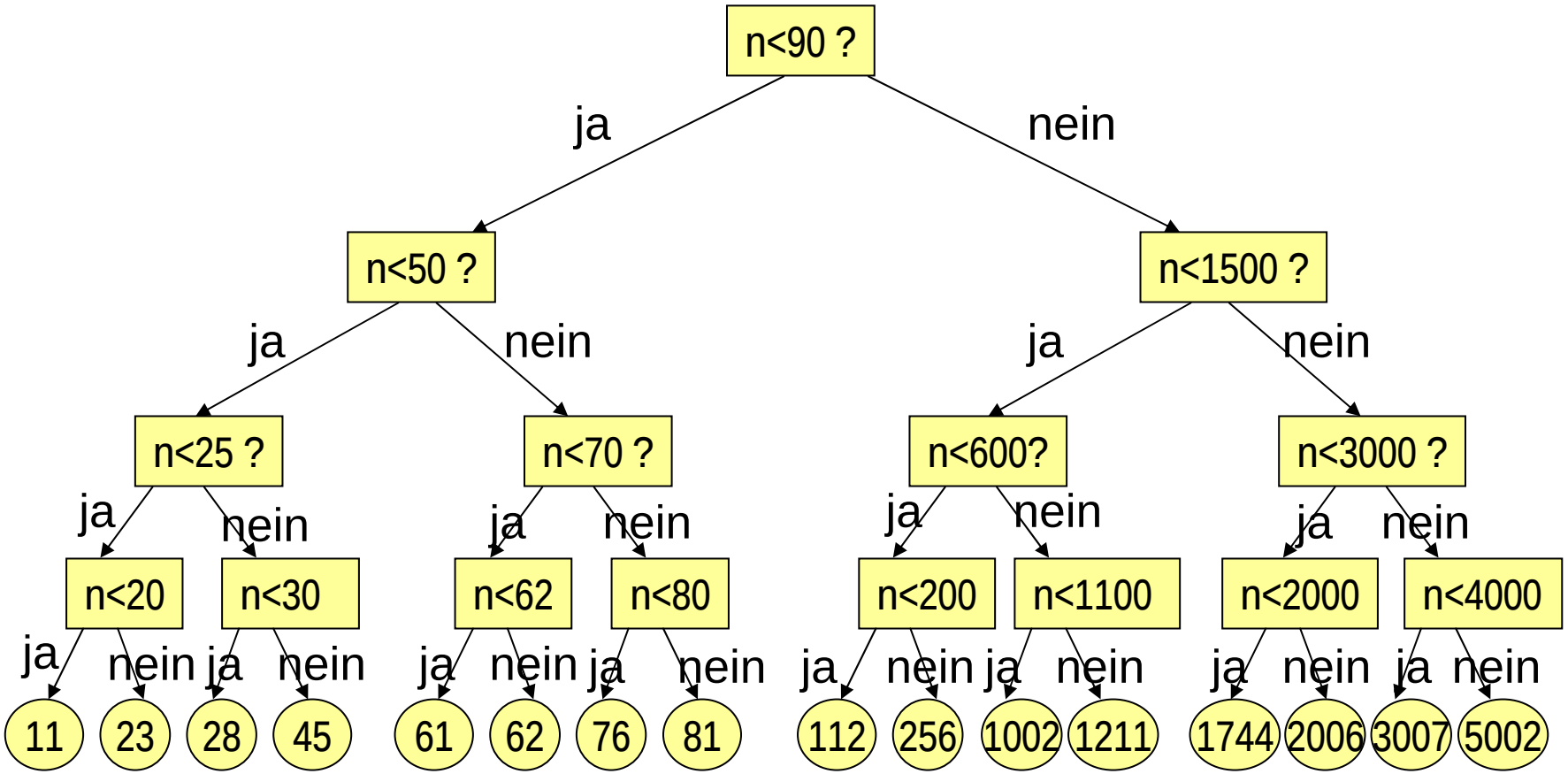
Absichten
und
Unterabsichten



Bäume

Suchbaum

Beschriftungen analog zu Graphen



Baum: Rekursive Definition

Anfang:

Ein Knoten v ist ein Baum: $T=[\ \{v\}, \varnothing, v\ ]$

Rekursionsschritt:

Wenn v ein Knoten ist

und wenn $T_i = [V_i, E_i, w_i\ ]$ $(i=1,...,n)$ Bäume sind,

deren Knotenmengen paarweise disjunkt sind,

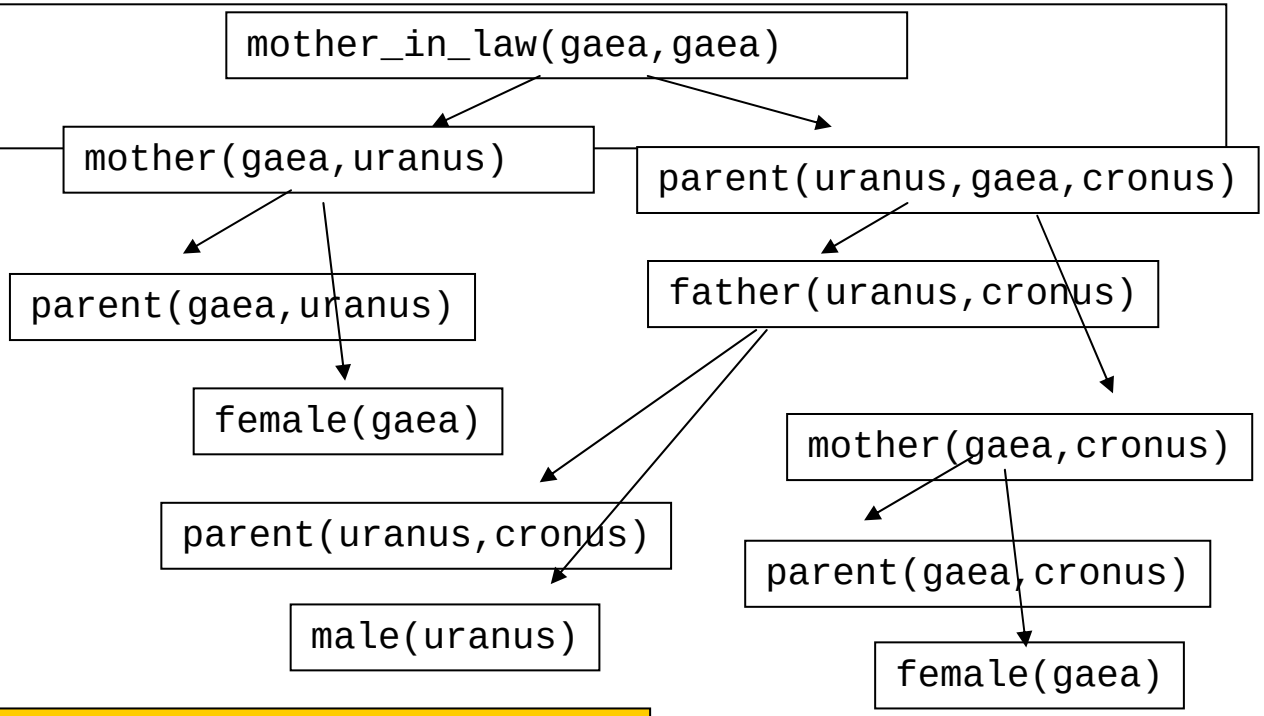
so ist

$T = [\ V_1 \cup ... \cup V_n \cup \{v\}\ ,\ E_1 \cup ... \cup E_n \cup \{\ [v, w_i\ ] \mid i = 1,...,n\ \}\ ,\ v\ ]$

ein Baum.

Bäume

Beweisbaum



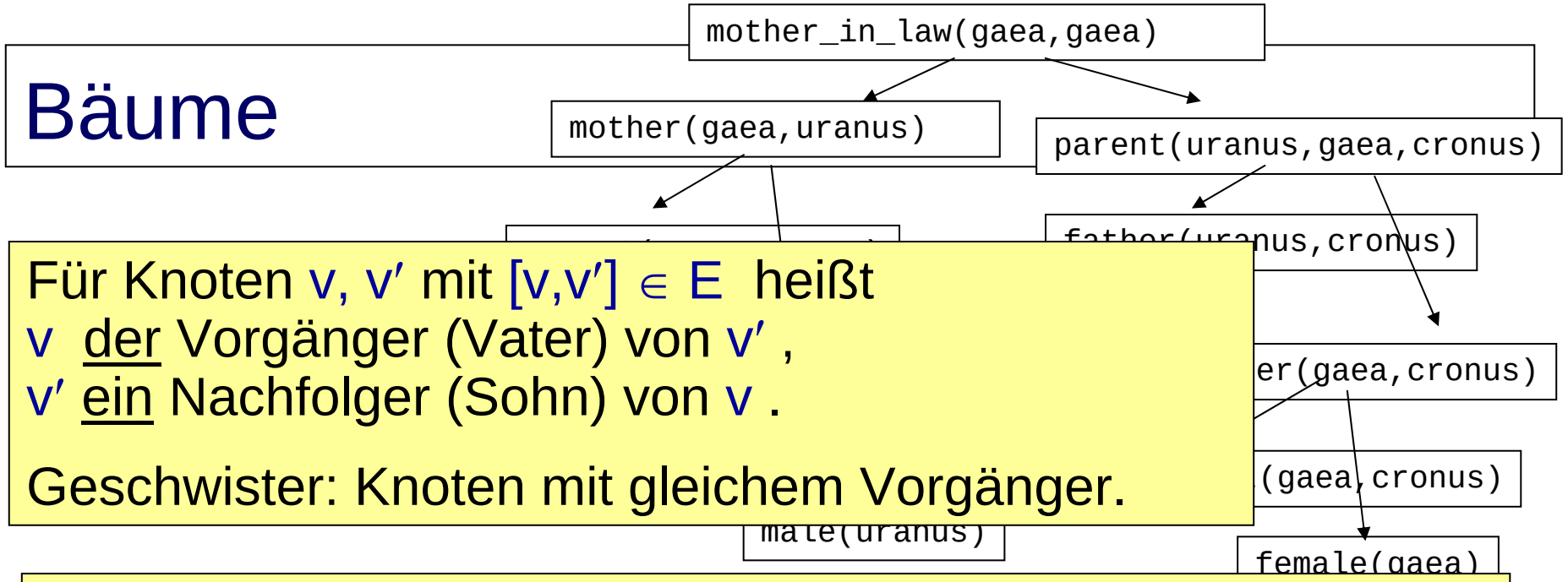
Knoten ohne Nachfolger: *Blätter*.

Knoten mit Nachfolger: *innere Knoten*.

Der Knoten ohne Vorgänger: *Wurzel.*

Tiefe: Entfernung von der Wurzel

Bäume



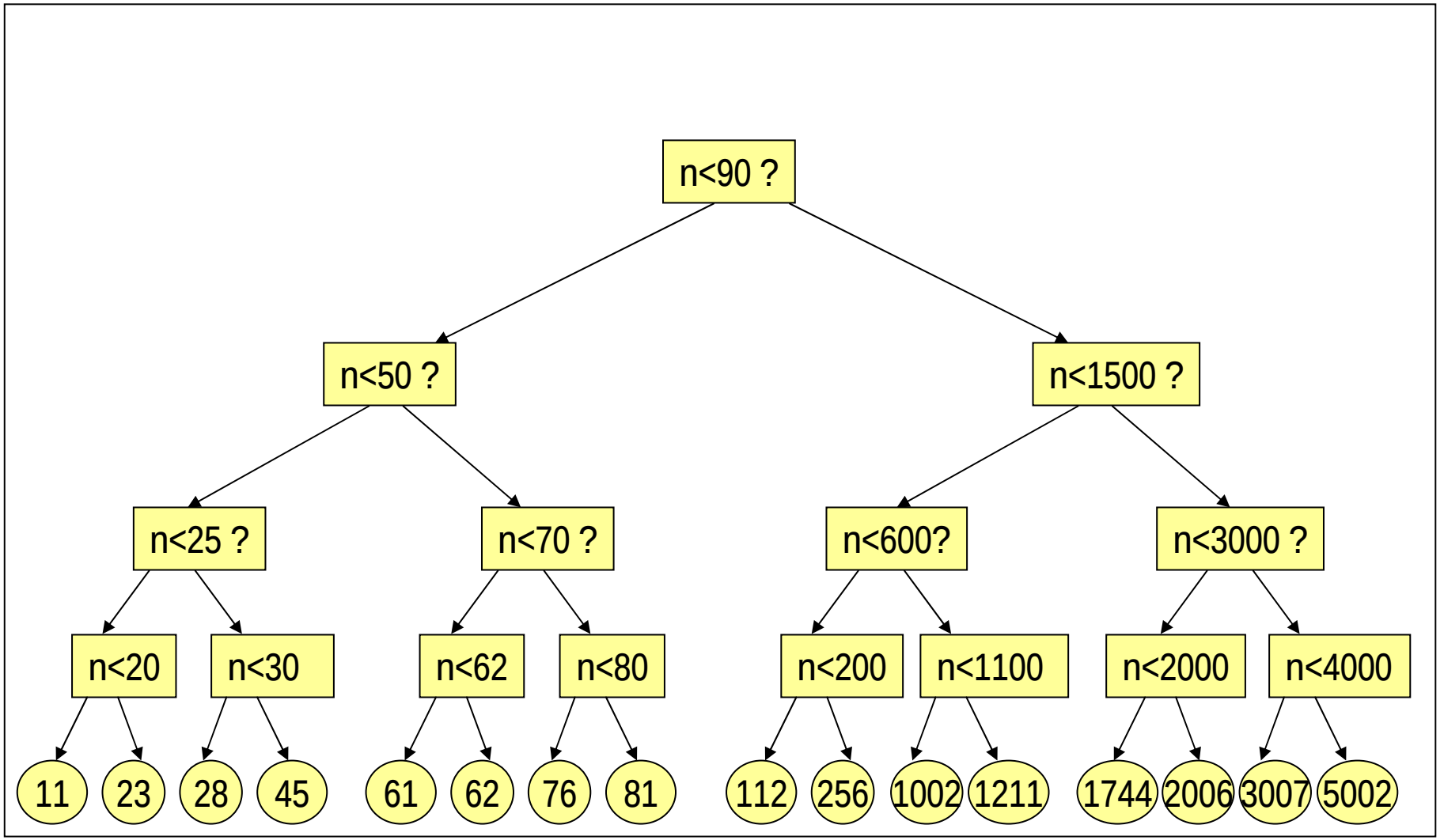
Für Knoten v, v′ mit [v,v′] ∈ E heißt
v der Vorgänger (Vater) von v′ ,
v′ ein Nachfolger (Sohn) von v .

Geschwister: Knoten mit gleichem Vorgänger.

Geordnete Bäume:
Geschwister geordnet (Darstellung: von links nach rechts)

binäre Bäume:
- innere Knoten haben genau 2 Nachfolger
- weiterhin bei Ordnung der Beschriftungen:
  geordnet bzgl. linker/rechter Nachfolger,
  Unterbäume konsistent mit Ordnung)

Binärer Suchbaum



Bäume

Bei Tiefe d und Verzweigungszahl b in den Knoten:

b^d Knoten in jeder Schicht

Baum der Tiefe d hat dann insgesamt

$1 + b + b^2 + ... + b^d$ Knoten

letzte Schicht hat mehr Knoten
als der ganze vorherige Baum (falls b >1)

Rekursion für Bäume

Wenn $T = [V, E, r]$ ein Baum ist,

so gilt für jeden Nachfolger v von r :

Der in v beginnende Teilgraph ist ein Baum:

$T|M(v) =_{Def} [M(v), E \cap M(v) \times M(v), v]$ Einschränkung

Rekursive Beweise/Definition in Bäumen

Strukturelle Induktion (vgl. freie Halbgruppen)

Wenn gilt (als Beweis oder Definition):

(A) H gilt für Wurzelknoten r .

(R) Wenn H für Knoten v gilt, so gilt H für alle Nachfolger v‘ von v .

Dann gilt: H gilt für alle Knoten.

Implementation von Bäumen

Liste von Paaren [*Knoten, Vorgänger*]

pre(Sohn,Vater)

Liste von Paaren [*Knoten, Liste der Nachfolger*]

succ(Vater,[Sohn-1,...,Sohn-n])

Rekursiv für binäre Bäume (andere analog) durch

Struktur: Knoten,rechter Teilbaum,linker Teilbaum

(bzw. mit Knotenbeschriftung)

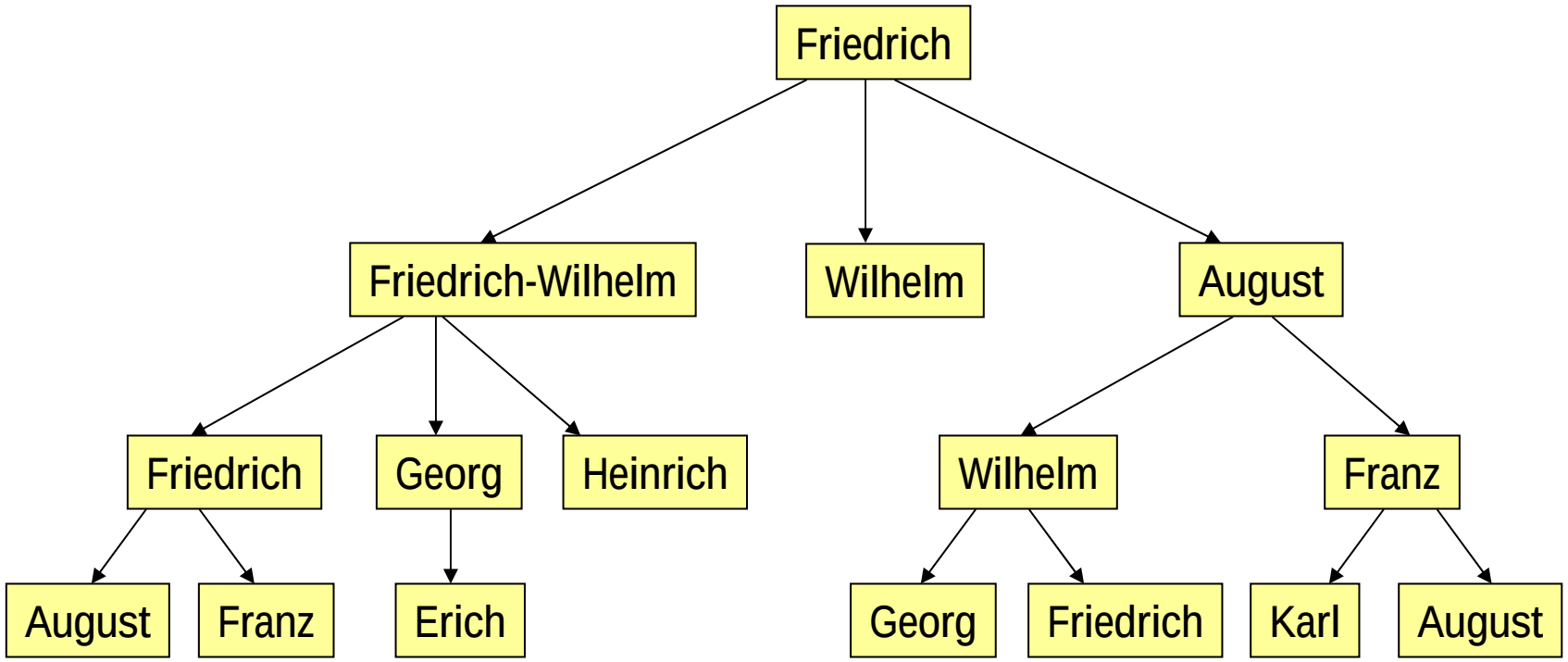
- tree(Knotenbeschriftung,LinkerBaum,RechterBaum)
- leerer Baum: nil

Identifikation von Knoten

Unterscheiden

Identifikator eines Knotens

Beschriftung eines Knotens



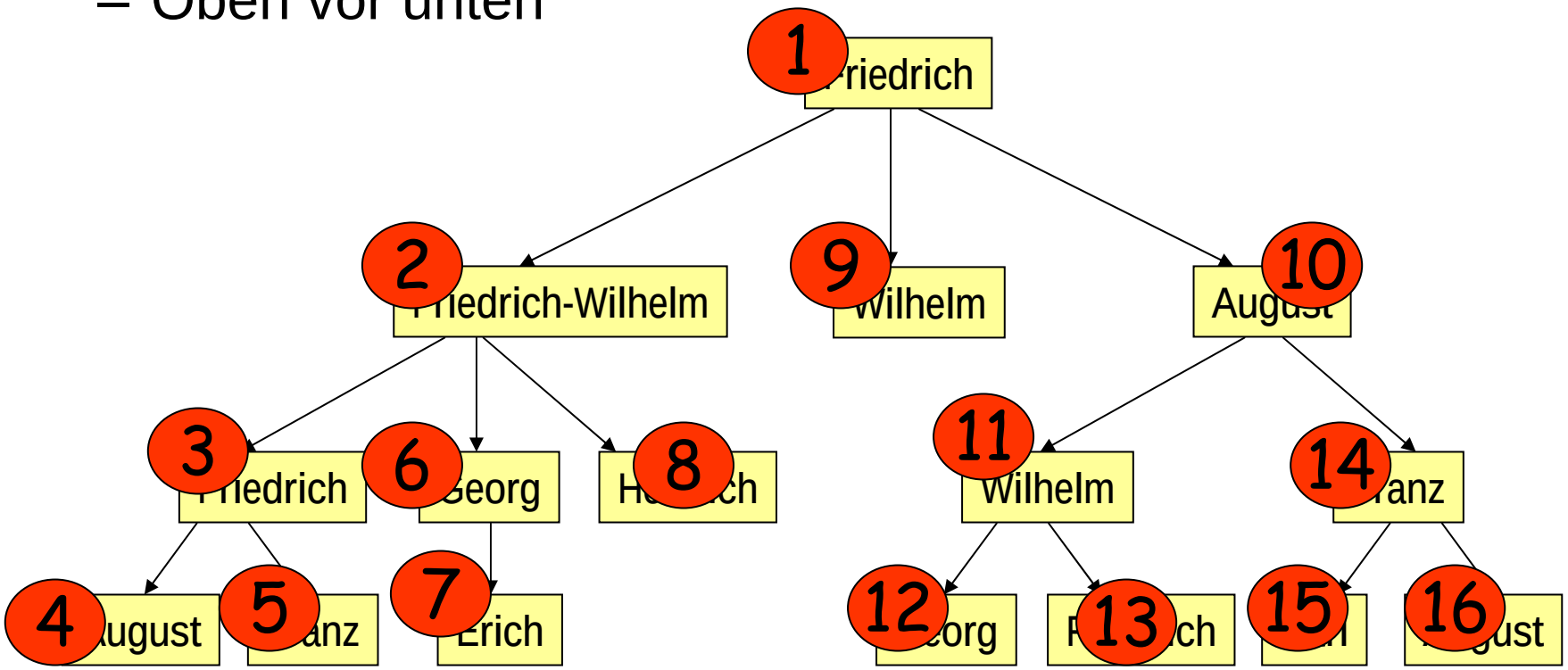
Identifikation von Knoten

Abzählen:

Links vor rechts

– Oben vor unten

„Tiefe zuerst"



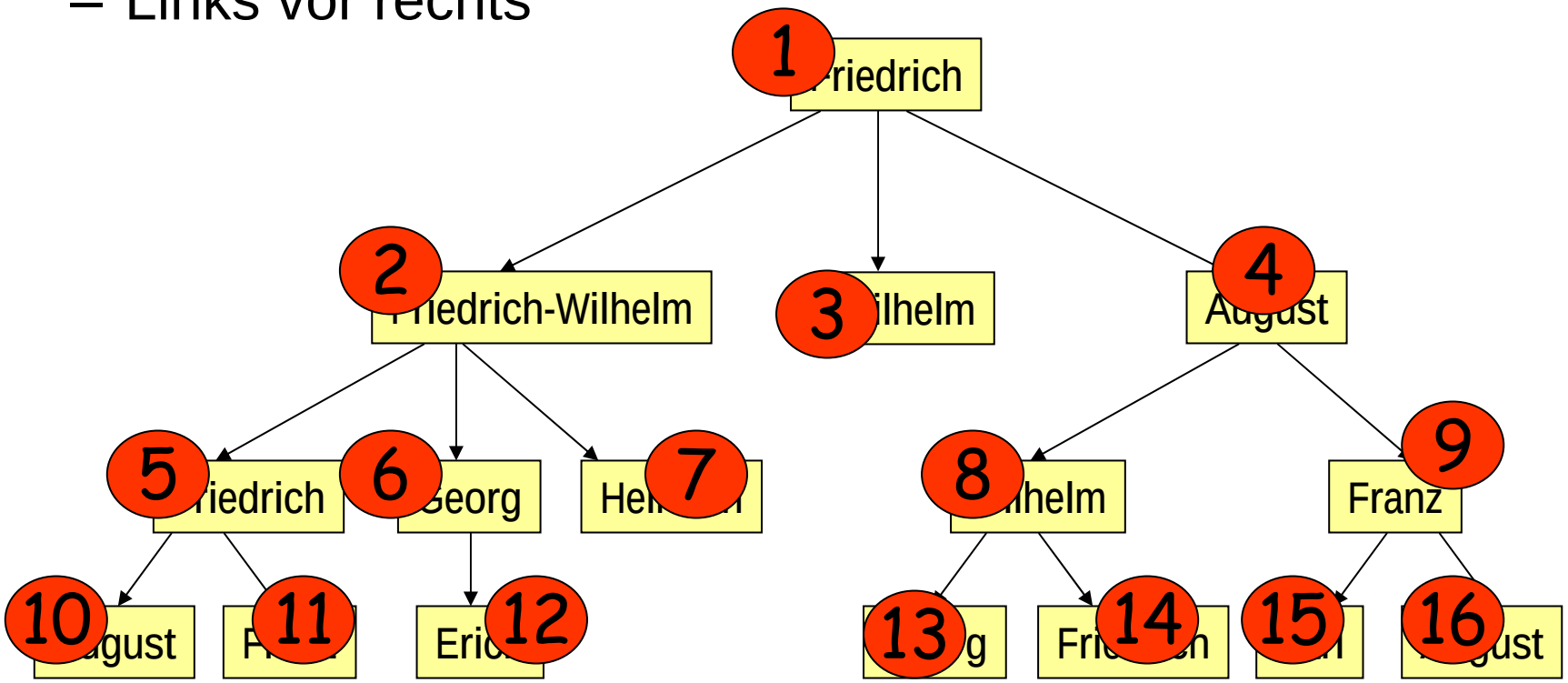
Identifikation von Knoten

Abzählen:

Oben vor unten

- Links vor rechts

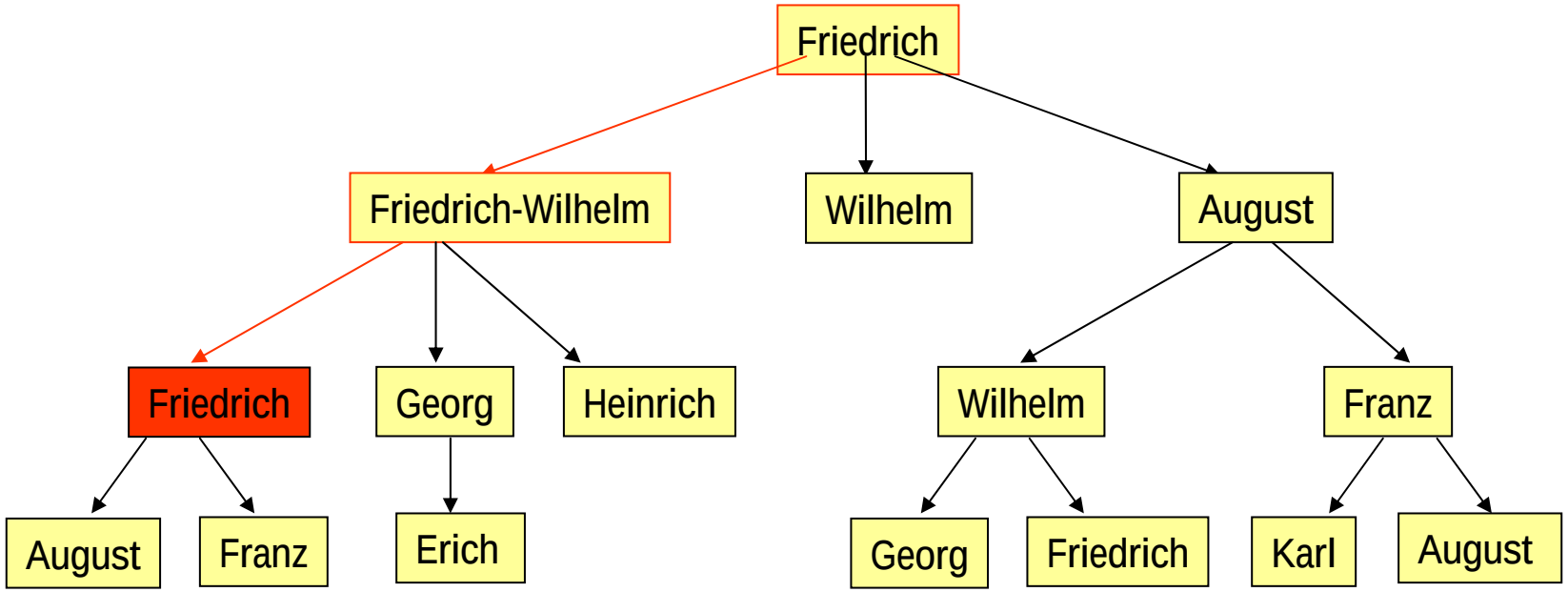
„Breite zuerst"



Identifikation von Knoten

Abzählung allein bestimmt nicht Position eines Knotens im Baum

Eindeutige Identifikation eines Knotens innerhalb eines Baumes: Weg von Wurzel zum Knoten als Identifikator



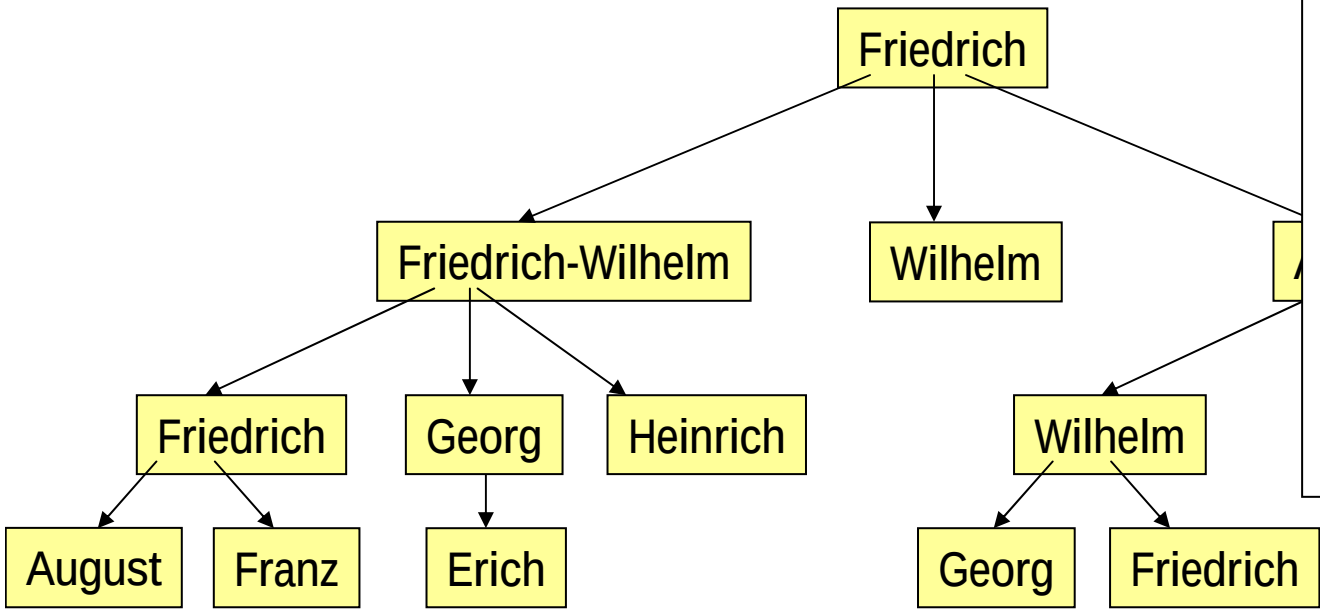
„Friedrich Sohn von Friedrich-Wilhelm dem Sohn von Friedrich“

Sequentialisierung

Rekursive Struktur:

(sohn)

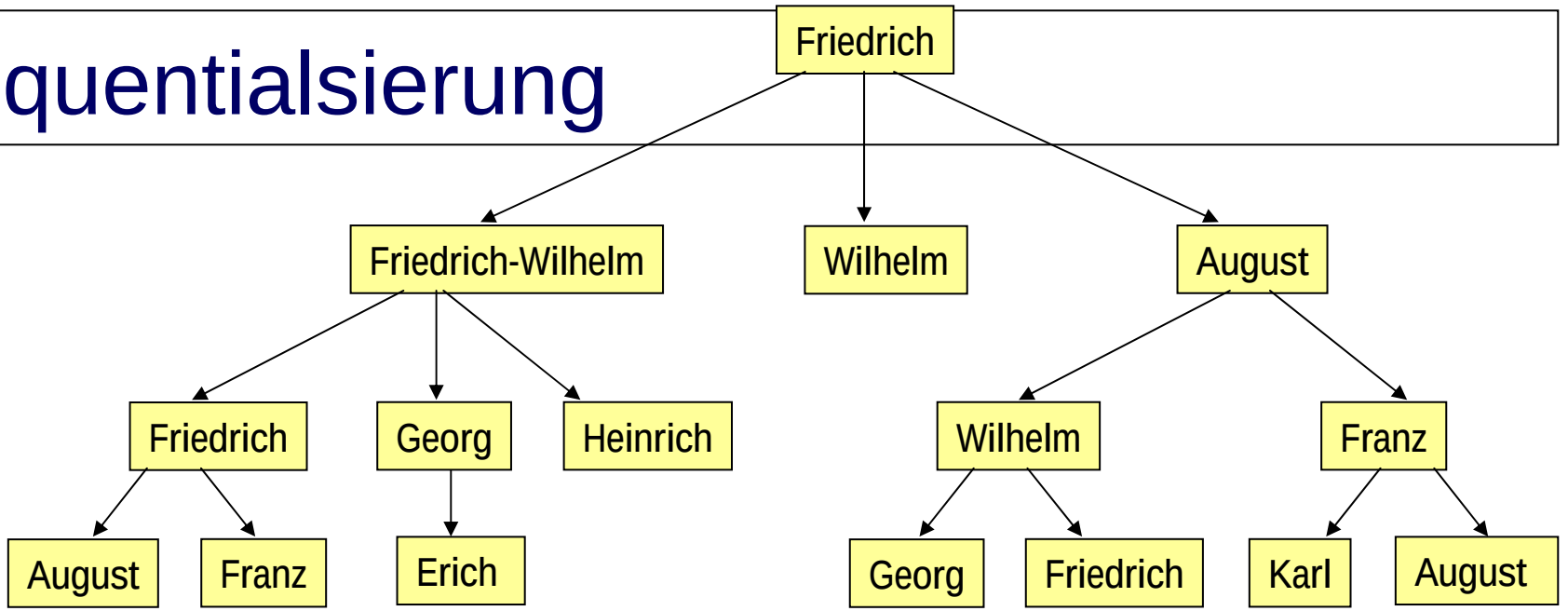
(vater(sohn)(sohn)...(sohn))




```
(Friedrich
      (FriedrichWilhelm
            (Friedrich
                  (August)
                  (Franz)
            )
            (Georg
                  (Erich)
            )
            (Heinrich)
      )
      (Wilhelm)
      (August
            (Wilhelm
                  (Georg)
                  (Friedrich)
            )
            (Franz
                  (Karl)
                  (August)
            )
      )
)
```

(Friedrich(FriedrichWilhelm(Friedrich(August)(Franz))(Georg(Erich))(Heinrich))(Wilhelm)(August(Wilhelm(Georg)(Friedrich))(Franz(Karl)(August))))

Sequentialsierung



(Friedrich(FriedrichWilhelm(Friedrich(August)(Franz))(Georg(Erich))(Heinrich))(Wilhelm)(August(Wilhelm(Georg)(Friedrich))(Franz(Karl)(August))))

Baumstruktur allein durch öffnende (Baumbeginn) und schließende (Baumende) Klammern definiert:

(((( )( ))(( ))( ))( )((( )( ))(( )( ))))

XML= Extensible Markup Language

Beschreibung hierarchischer (baumförmiger) Strukturen

- tags = Klammern
  - <name> für Beginn
  - </name> für Ende
- Weitere Konventionen
  (Inhalte, Attribute)
- Verarbeitungswerkzeuge

- Ursprung:
  electronic publishing
- Keine Formatanweisung
  (html)

http://www.w3schools.com/xml/default.asp

XML= Extensible Markup Language

```
<book>
      <title>My First XML</title>
      <prod id="33-657" media="paper"></prod>
     <chapter>Introduction to XML
          <para>What is HTML</para>
           <para>What is XML</para>
     </chapter>
     <chapter>XML Syntax
          <para>Elements must have a closing tag</para>
          <para>Elements must be properly nested</para>
     </chapter>
</book>
```

- Element: <name> ... *Inhalt* ... </name>
- Inhalt: andere Elemente, Text, auch gemischt oder leer

XML= Extensible Markup Language

Daten können als Attribute oder Elemente angegeben werden:

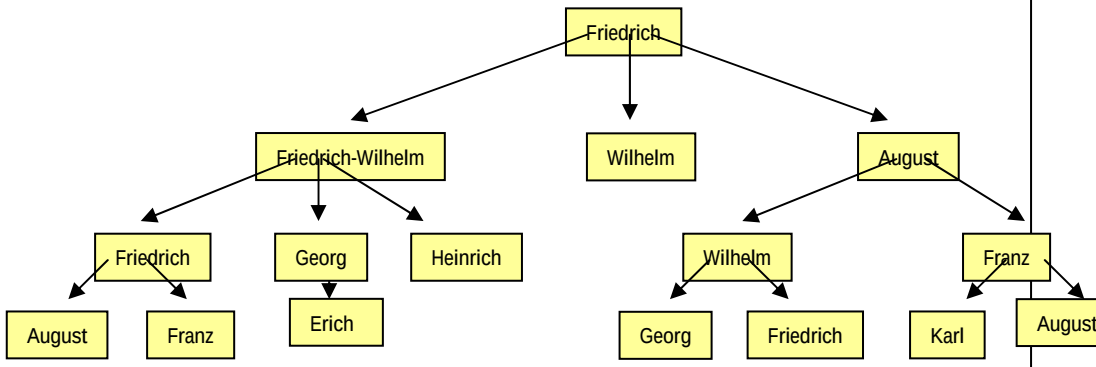
- Attribut

```
<person sex="female">
    <firstname>Anna</firstname>
    <lastname>Smith</lastname>
 </person>
```

- Element

```
<person>
    <sex>female</sex>
    <firstname>Anna</firstname>
     <lastname>Smith</lastname>
 </person>
```

XML= Extensible Mark




```
(Friedrich(FriedrichWilhelm(Friedrich(August)(
helm)(August(Wilhelm(Georg)(Friedrich))(Franz
```

```
<vater>Friedrich<vater>FriedrichWilhelm<vate
vater>Franz</vater></vater><vater>Georg<vate
nrich</vater></vater><vater>Wilhelm</vater><v
>Georg</vater><vater>Friedrich</vater></vater
vater>August</vater></vater></vater></vater>
```

```
<vater>Friedrich
    <vater>FriedrichWilhelm
        <vater>Friedrich
            <vater>August</vater>
            <vater>Franz</vater>
        </vater>
        <vater>Georg
            <vater>Erich</vater>
        </vater>
        <vater>Heinrich</vater>
    </vater>
    <vater>Wilhelm</vater>
    <vater>August
        <vater>Wilhelm
            <vater>Georg</vater>
            <vater>Friedrich</vater
            >
        </vater>
        <vater>Franz
            <vater>Karl</vater>
            <vater>August</vater>
        </vater>
    </vater>
</vater>
```

Suche (Retrieval) in einem Baum

Doppelt rekursiv für binären Baum

```
membertree(X,tree(X,_,_)).
membertree(X,tree(_,T1,_)):- membertree(X,T1) .
membertree(X,tree(_,_,T2)) :- membertree(X,T2) .
```

Geordneter binärer Baum

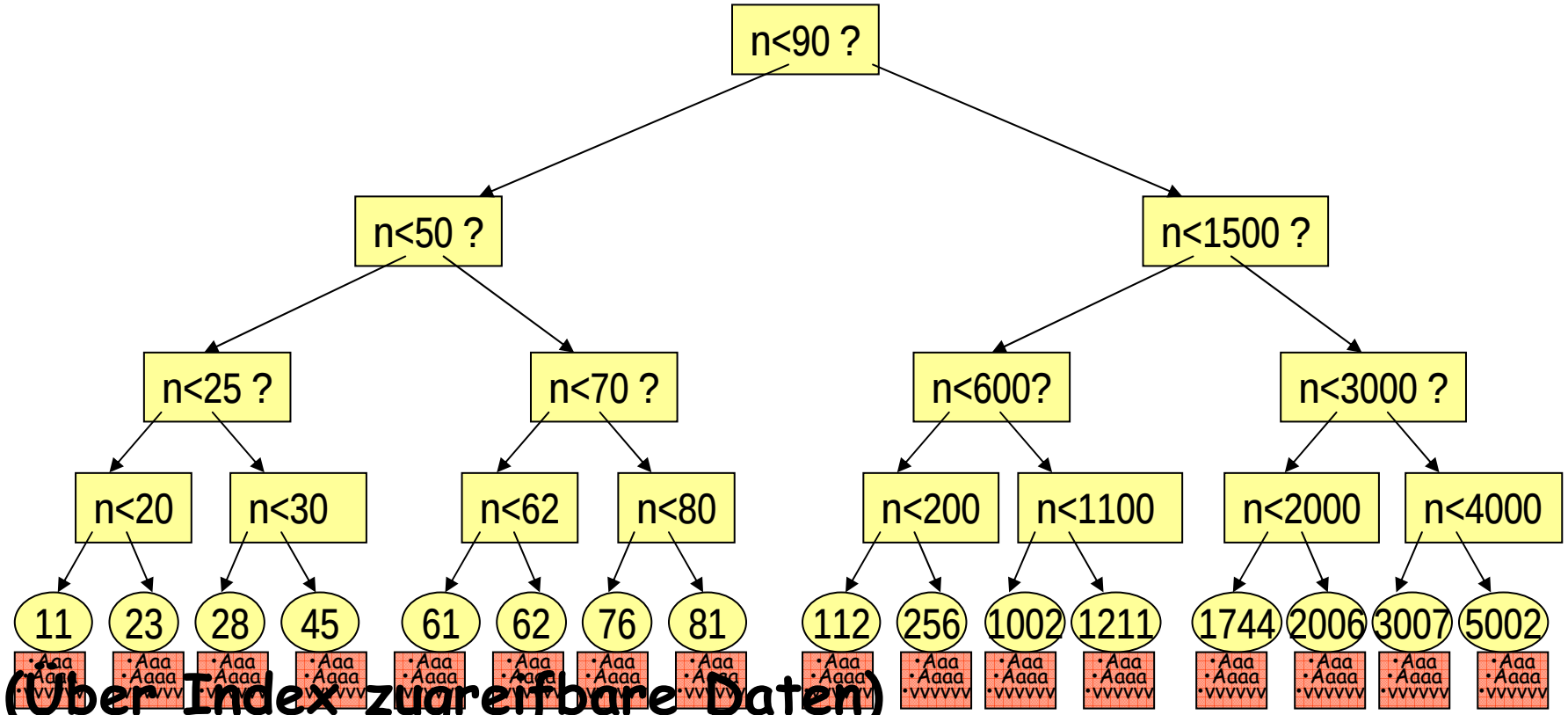
```
node retrieve(int x, node v)
 { if   (v == NIL) return NIL ;
   else if (x == v.value) return v ;
          else if (x < v.value)  return retrieve(x, v.left);
                 else                return retrieve(x, v.right);
 }
```

Suchbaum für Retrieval

Alternative zu Hash-Funktionen.

„Schlüssel“ als Suchbegriff (Knoten-Beschriftung).

Inhalt über Knoten zugreifbar („Anhang“, Verweis).



Zeit-Komplexität des Retrievals

Abschätzung: O(log(n))

Tatsächlicher Aufwand abhängig von

- Durchschnittliche Tiefe der Zweige
- Häufigkeit der Suche nach Schlüsseln

Optimierung:

Baum balancieren bzgl. Tiefe (AVL-Bäume)

AVL= Adelson-Velskij, Landis, 1962

Wege in einem Graphen

Sei $G=[V,E]$ ein Graph, $v_0 \in V$.

$L(v_0) =_{Def}$ Menge der in v_0 beginnenden Wege $p= v_0\, v_1\, v_2 \ldots v_n$

Für endliche gerichtete Graphen G gilt:

$L(v_0)$ ist eine reguläre Sprache über dem Alphabet V.

$L(v_0)$ ist endlich gdw. $G|M(v_0)$ azyklisch ist.

$G|M(v_0)$ = von v_0 aus erreichbarer Teilgraph

$G|M(v_0) =_{Def} [\, M(v_0),\, E \cap M(v_0) \times M(v_0)\,]$

$M(v_0) =_{Def}$ Menge der von v_0 erreichbaren Knoten

Erreichbarkeitsbaum

Sei $G=[V,E]$ ein Graph, $v_0 \in V$.

Erreichbare Zustände: $M(v_0) = \{ v \mid v \text{ erreichbar von } v_0 \}$

$L(v_0)$ = Menge der in v_0 beginnenden Wege $p= v_0 v_1 v_2 \ldots v_n$

„Abwickeln“ des Graphen in v_0 ergibt Erreichbarkeitsbaum:

Aufspalten von Maschen/Zyklen

$T(v_0) = [K, B, k_{v0} , V, \alpha, E, \beta]$ mit

$K = \{ k_p \mid p \in L(v_0) \}$

$\alpha(k_p)$ = der mit p erreichte Knoten

$B = \{ [k_p , k_{pv}] \mid p \in L(v_0) \wedge v \in V \wedge pv \in L(v_0) \}$

$\beta([k_p , k_{pv}])$ = letzte Kante auf Weg pv

$\alpha(k_{v0}) = v_0$

$\alpha(k_p) = v$ für $p= v_0 \ldots v_n , v_n = v$

$\beta([k_p , k_{pv}]) = [v_n,v]$ für $p= v_0 \ldots v_n$

Erreichbarkeitsbaum

Für jeden Weg in $G=[V,E]$ eigener Zweig in $T(v_0)$

Für Knoten k in $T(v_0)$:

- Name k_p :
  - gemäß Weg p in G
  - Kantenbeschriftungen auf Weg zu k_p in $T(v_0)$
- Beschriftung $\alpha(k_p)$: bei Weg p in G erreichter Knoten

$T(v_0)$ endlich gdw. $L(v_0)$ endlich

(für gerichtete Graphen: gdw. $G|M(v_0)$ azyklisch ist)

Binäre Relationen als Graph darstellen

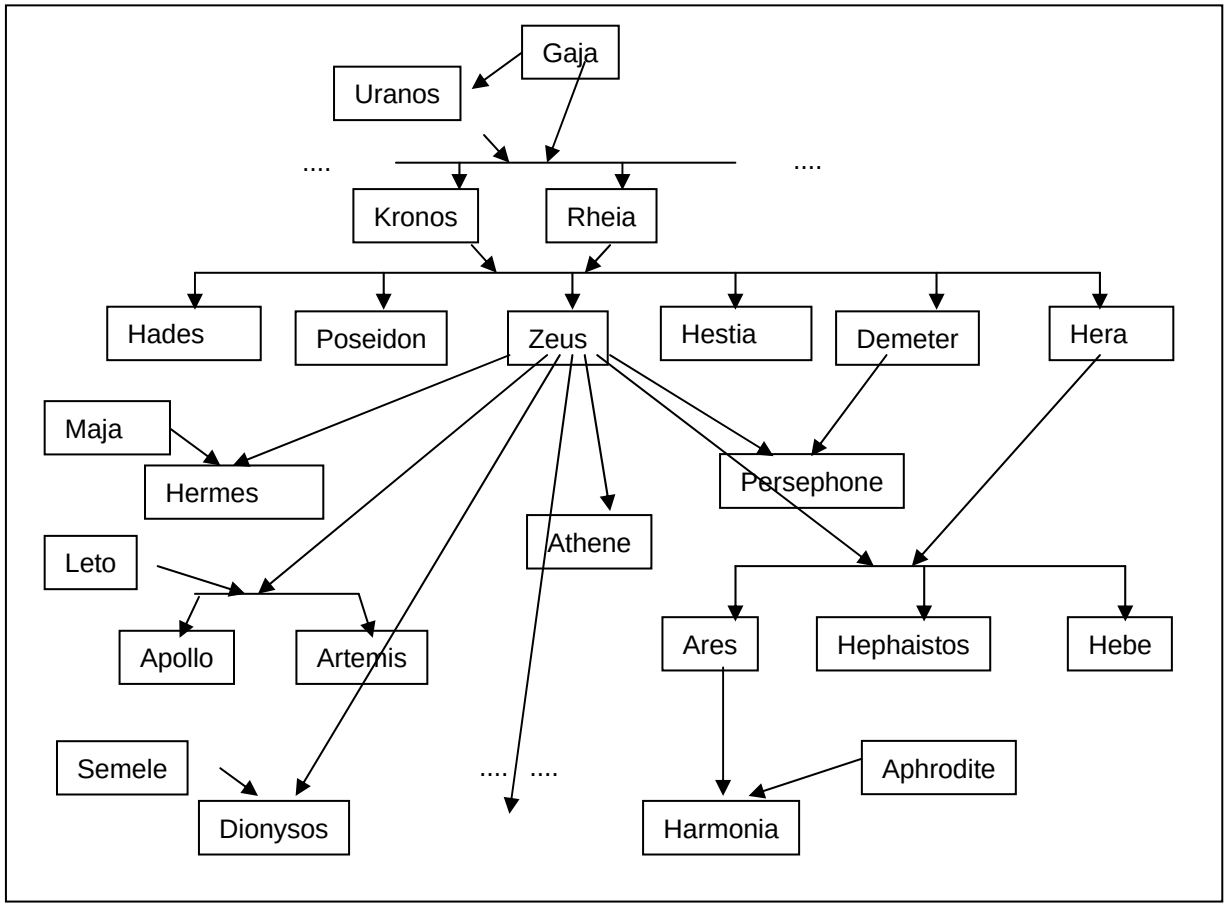
Binäre Relation R über Menge M: $R \subseteq M \times M$

- über natürlichen Zahlen:
  $<$, $\leq$, $>$, $\geq$, $=$, ..., $\equiv$ *mod*(*n*) , ...
- über Mengen:
  $\subset$, $\subseteq$,..., *gleichmächtig (d.h. card(M)=card(N))*, ...
- über Wörtern einer Sprache:
  suffix, *präfix*, ..., *gleiche Länge*, ...

Binäre Relationen als Graph darstellen

Binäre Relation R über Menge M: $R \subseteq M \times M$

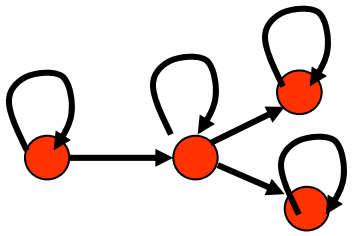
Graph G=[M,R]



Eigenschaften binärer Relationen

Reflexivität **(R)** $\forall a$: aRa

Schlingen $[v,v] \in E$ für alle $v \in V$

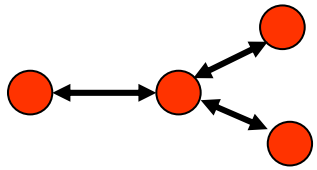


Irreflexivität **(Ir)** $\forall a$: $\neg aRa$

Eigenschaften binärer Relationen

Symmetrie **(S)** $\forall a,b: aRb \rightarrow bRa$

Kanten jeweils in beiden Richtungen bzw. ungerichtet

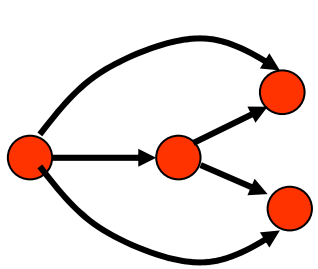


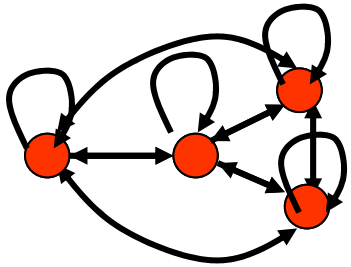
Asymmetrie **(aS)** $\forall a,b: aRb \rightarrow \neg bRa$

Antisymmetrie ("identitiv") **(anS)** $\forall a,b: aRb \wedge bRa \rightarrow a=b$

Eigenschaften binärer Relationen

Transitivität **(T)** $\forall a,b,c : aRb \wedge bRc \rightarrow aRc$





Direkte Verbindungen zwischen jeweils
allen Knoten auf gerichteten Wegen
(werden bei Darstellung gelegentlich weggelassen)

Eigenschaften binärer Relationen

Konnexität **(K)** $\forall a,b: aRb \vee bRa$

Kanten (in wenigstens einer Richtung) zwischen allen Knoten
(speziell: Schlingen an allen Knoten)

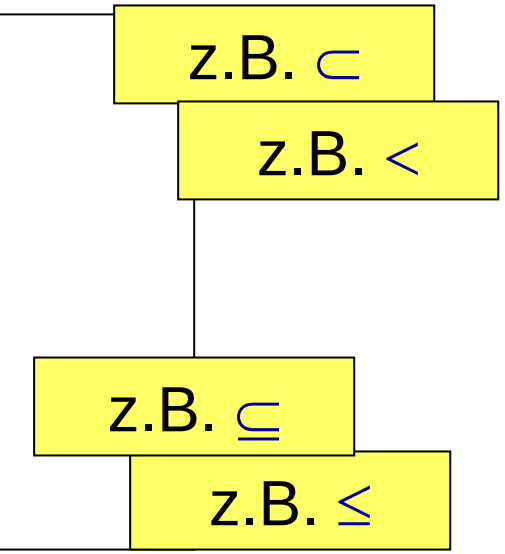
Linearität **(L)** $\forall a,b: aRb \vee bRa \vee a=b$

Kanten (in wenigstens einer Richtung) zwischen allen unterschiedlichen Knoten

Anordnungen

- irreflexive Halbordnung: **(Ir)**, **(T)**,(aS)
- irreflexive Ordnung: **(Ir)**, **(T)**, **(L)**, (aS)

- reflexive Quasiordnung: **(R)**, **(T)**
- reflexive Halbordnung: **(R)**, **(T)**, **(anS)**
- reflexive Ordnung: (R), **(T)**, **(anS)**, **(K)**



Anordnungen

Bezeichnungen nicht einheitlich

(Definition beachten):

- konnex vs. linear
- Halbordnung vs. Ordnung
- Halbordnung = "partielle Ordnung"
- Ordnung = "lineare" Ordnung, "totale" Ordnung, ...

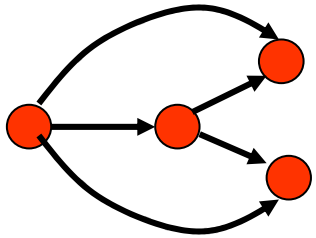
Anordnungen

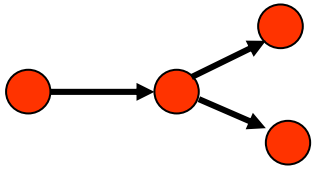
Hasse-Diagramm

Einschränkung des Graphen einer Anordnungsrelation:

Kanten nur von m nach n falls

$mRn \wedge m \neq n \wedge \neg \exists x \, (x \in M \wedge x \neq m \wedge x \neq n \wedge mRx \wedge xRn)$





Anordnungen

Kette: geordnete Teilmenge von M

Halbordnungen: Knoten liegen auf Linien („Ketten“)

„Partielle Ordnung“

Transitivität:
Alle Knoten auf einer Kette direkt verbunden

Ordnungen: alle Knoten liegen auf <u>einer</u> Linie („Kette“)

„Totale Ordnung“

Maximale (bzw. minimale) Elemente in $N \subseteq M$
für Halbordnung R über M :

$$m \in N \wedge \forall x \in N: \; mRx \rightarrow m=x$$

Anfangs-/Endpunkte der Ketten im Teil-Graphen für N

Äquivalenzrelationen

- Äquivalenzrelation **(R)**, **(S)**, **(T)**

Graph zerfällt in stark zusammenhängende Teilgraphen, in denen jeder Knoten mit jedem verbunden ist.

Eine Äquivalenzrelation R über einer Menge M definiert eine **Zerlegung** von M in disjunkte **Klassen**.

Für $a \in M$: $K(a) =_{Def} \{ b \mid aRb \}$, dabei gilt:

- $K(a) = K(b) \leftrightarrow aRb$
- $K(a) \cap K(b) = \emptyset \leftrightarrow \neg\, aRb$
- $M = \cup \{ K(a) \mid a \in M \}$

Klassen durch beliebige Repräsentanten eindeutig bestimmt.

Ähnlichkeitsrelation (Verträglichkeitsrelation

- Ähnlichkeitsrelation **(R)**, **(S)**

Graph überdeckt von stark zusammenhängenden Teilgraphen, in denen jeder Knoten mit jedem verbunden ist.

Eine Ähnlichkeitsrelation R über einer Menge M definiert eine **Überdeckung** $\mathcal{N}$ von M durch Mengen $A \subseteq M$ von untereinander ähnlichen Elementen:

$$\mathcal{N} =_{Def} \{ A \mid A \subseteq M \wedge A \text{ maximal} \wedge \forall a,b \in A \rightarrow aRb \}$$

Im Gegensatz zu Äquivalenzrelationen:

Die Mengen $B(a) =_{Def} \{ b \mid aRb \} \subseteq M$ sind nicht durch Repräsentanten bestimmt. Es kann insbesondere gelten:

aRb und gleichzeitig $B(a) \neq B(b)$

$\neg aRb$ und gleichzeitig $B(a) \cap B(b) \neq \emptyset$

Implementation von Graphen

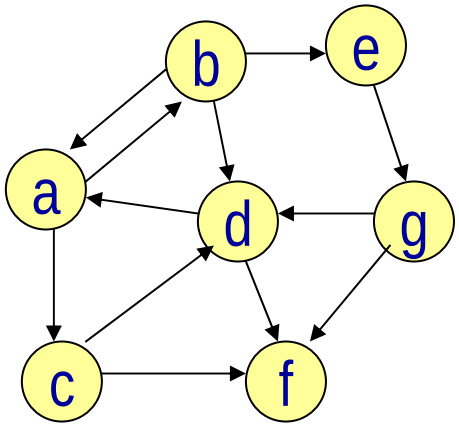
Und weitere.

ggf. zusätzlich Beschriftungen

Datenstruktur (Inzidenzmatrix, Adjazenzmatrix)

als (meist dünn besetzte) Matrix:

- Felder
- Listen

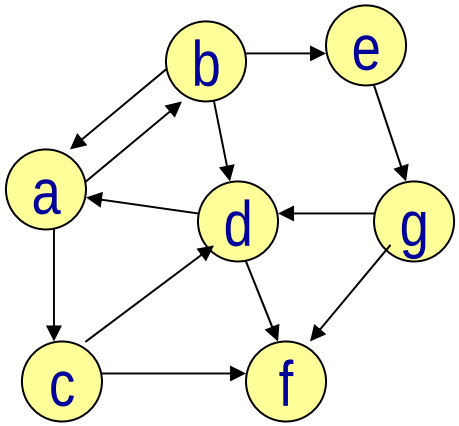


	a	b	c	d	e	f	g
a		1	1				
b	1			1	1		
c				1		1	
d	1					1	
e							1
f							
g				1		1	

Implementation von Graphen

- Datenbank von benachbarten Knoten

kante(knotenname1,knotename2)




```
kante(a,b).
kante(a,c).
kante(b,a).
kante(b,d).
kante(b,e).
kante(c,d).
kante(c,f).
kante(d,a).
kante(d,f).
kante(e,g).
kante(g,d).
kante(g,f).
```

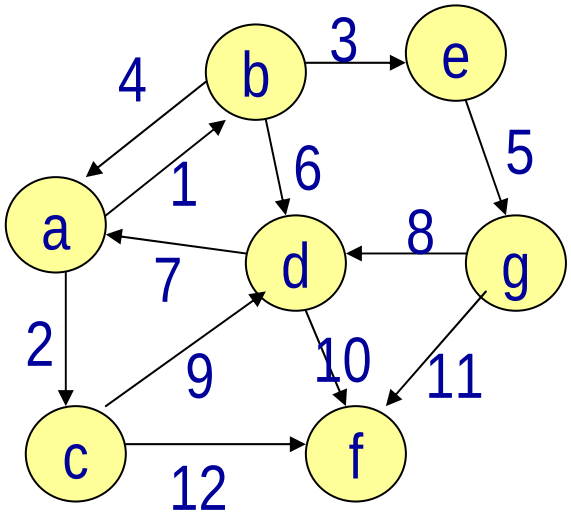
oder kante(kantenname, knotenname1,knotename2)

Implementation von Graphen

- Datenbank von Knoten-/Kantenbeziehungen

  eingangskante(knotenname,kantenname)

  ausgangskante(knotenname,kantename)

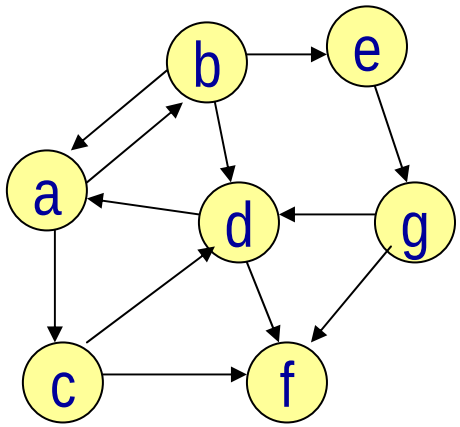



```
eingangskante(b,1).
eingangskante(c,2).
eingangskante(a,4).
eingangskante(d,6).
eingangskante(e,3).
eingangskante(d,9).
eingangskante(f,12).
eingangskante(a,7).
eingangskante(f,10).
eingangskante(g,5).
eingangskante(d,8).
eingangskante(f,11).
```

```
ausgangskante(a,1).
ausgangskante(a,2).
ausgangskante(b,4).
ausgangskante(b,6).
ausgangskante(b,3).
ausgangskante(c,9).
ausgangskante(c,12).
ausgangskante(d,7).
ausgangskante(d,10).
ausgangskante(e,5).
ausgangskante(g,8).
ausgangskante(g,11).
```

Graph als Liste von Adjazenzlisten

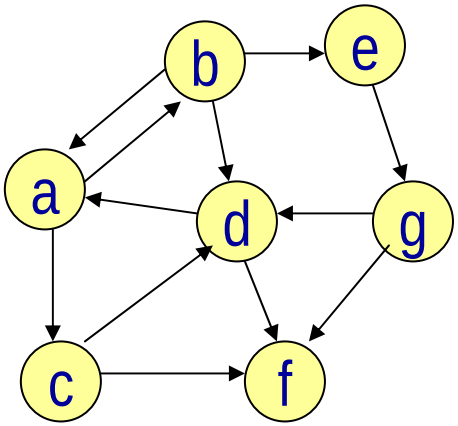
[a:[b,c], b:[a,d,e],c:[d,f],d:[a,f],e:[g],f:[],g:[d,f]]



- Berechnung von Kanten:

```
kante(X,Y,Graph)
        :- member(X:Nachbarn,Graph),member(Y,Nachbarn).
```

Datenbank der Adjazenzlisten




```
adjazenz(a,[b,c]).
adjazenz(b,[a,d,e]).
adjazenz(c,[d,f]).
adjazenz(d,[a,f]).
adjazenz(e,[g]).
adjazenz(f,[]).
adjazenz(g,[d,f]).
```

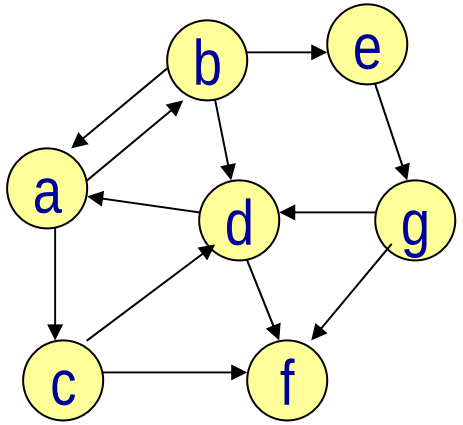
- Berechnung von Kanten:

```
kante(X,Y) :- adjazenz(X,Nachbarn),member(Y,Nachbarn).
```

Weg in einem Graphen

```
weg(Start,Start,_,[Start]).

weg(Start,Ziel,Graph,[Start|Weg])
  :- kante(Start,Nachbar,Graph),
     weg(Nachbar,Ziel,Graph,Weg).
```



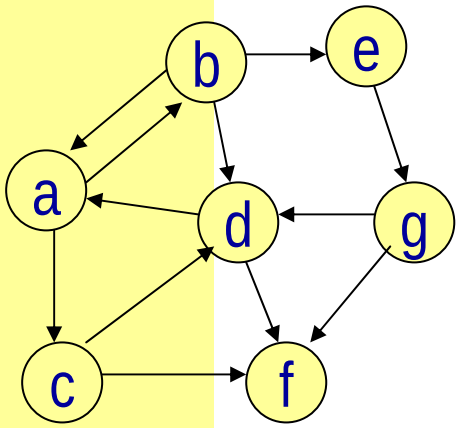

```
?- assert(graph([ a:[b,c], b:[a,d,e],c:[d,f],d:[a,f],e:[g],f:[],g:[d,f] ])).
?- graph(G),weg(a,b,G,W).
G = [a:[b, c], b:[a, d, e], c:[d, f], d:[a, f], e:[g], f:[], g:[d|...]]
W = [a, b] ;
...
W = [a, b, a, b] ;
...
W = [a, b, a, b, a, b] ;
...
W = [a, b, a, b, a, b, a, b]
```

Weg in einem Graphen

```
wegOhneZyklen(Start,Ziel,Graph,Weg)
        :-wegSuchen(Start,Ziel,Graph,Weg,[Start]).

wegSuchen(Start,Start,_,[Start],_).

wegSuchen(Start,Ziel,Graph,[Start|Weg],Bisherige)
  :- kante(Start,Nachbar,Graph),
     not(member(Nachbar,Bisherige)),
     wegSuchen(Nachbar,Ziel,Graph,Weg,[Nachbar|Bisherige]).
```




```
?- graph(G),wegOhneZyklen(a,f,G,W).
G = [a:[b, c], b:[a, d, e], c:[d, f], d:[a, f], e:[g], f:[], g:[d|...]]
W = [a, b, d, f] ;
...
W = [a, b, e, g, d, f] ;
...
W = [a, b, e, g, f] ;
...
W = [a, c, d, f] ;
...
W = [a, c, f] ;
```

Suche in Graphen

Beispiele:
- –Routenplanung
- –Fahrplanauskunft
- –Suche nach einem Beweis
- –Suche nach Gewinnstrategie
- –Planung

Modell für Problemlösen:
- Gegeben:
  - –Graph $G = [V,E]$
  - –„Anfangszustand“ $z_0 \in V$
  - –Menge von „Zielzuständen“ $Z_f \subseteq V$

- Probleme:
  - –Existiert ein Weg von z_0 zu einem $z_f \in Z_f$
  - –Konstruiere einen Weg von z_0 zu einem $z_f \in Z$
  - –Konstruiere optimalen Weg von z_0 zu einem $z_f \in Z_f$

    (bzgl. eines gegebenen Optimalitätskriteriums)

Planung: Modellierung als Graph

Mögliche Aktionen: $A = \{a_1,...,a_n\}$

Zustände (Knoten im Graphen):

V = durch Aktionen entstehende Situationen

Ausgangsituation: Anfangszustand z_0

Situationen, in denen Planungsziel erreicht ist: Zielzustände Z_f

Zustandsübergänge (Kanten im Graphen):

E = Übergänge zwischen Situationen durch Aktionen

$= \{ [v,v',a] \mid v,v' \in V\ a \in A \wedge v \text{ wird durch } a \text{ in } v' \text{ überführt} \}$

G ist ein Kanten-beschrifteter Graph mit Mehrfachkanten

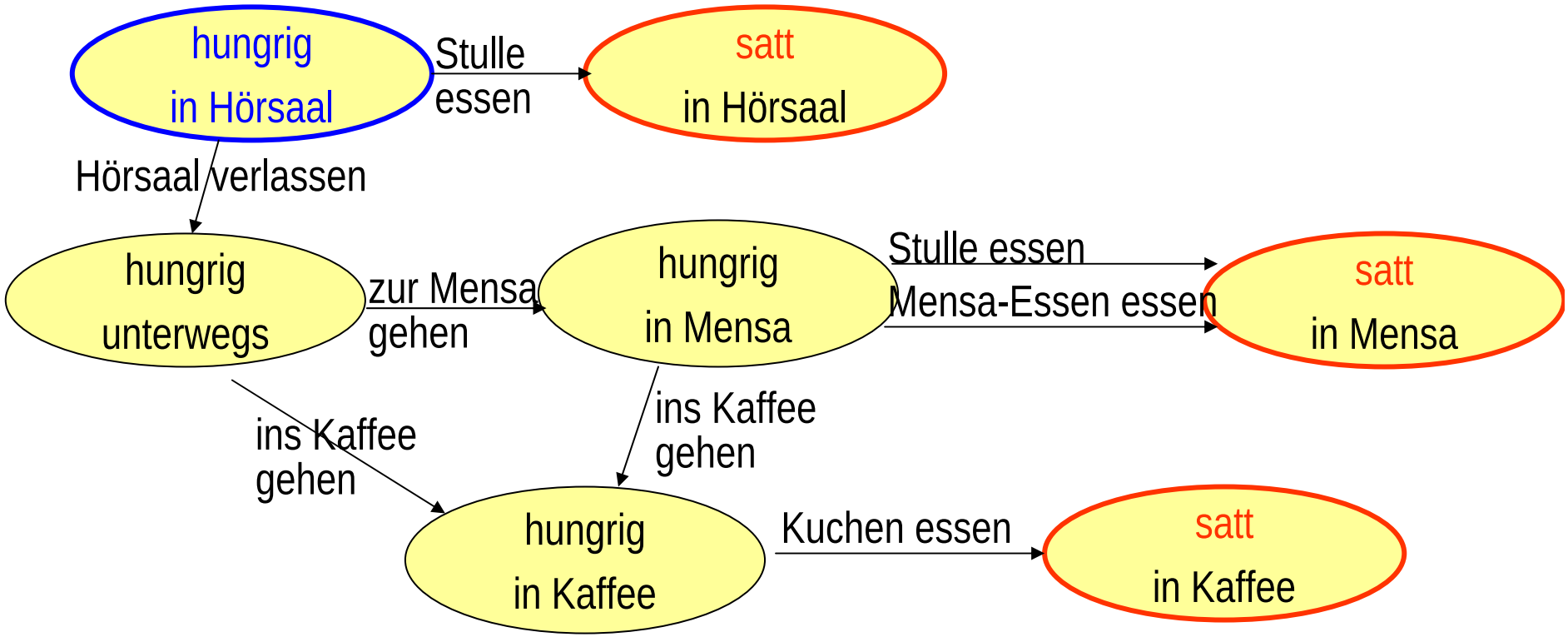
$G = [V,E,f,A,\beta]$

mit $f([v,v',a]) = [v,v']$, $\beta([v,v',a]) = a$

Planung: Modellierung als Graph

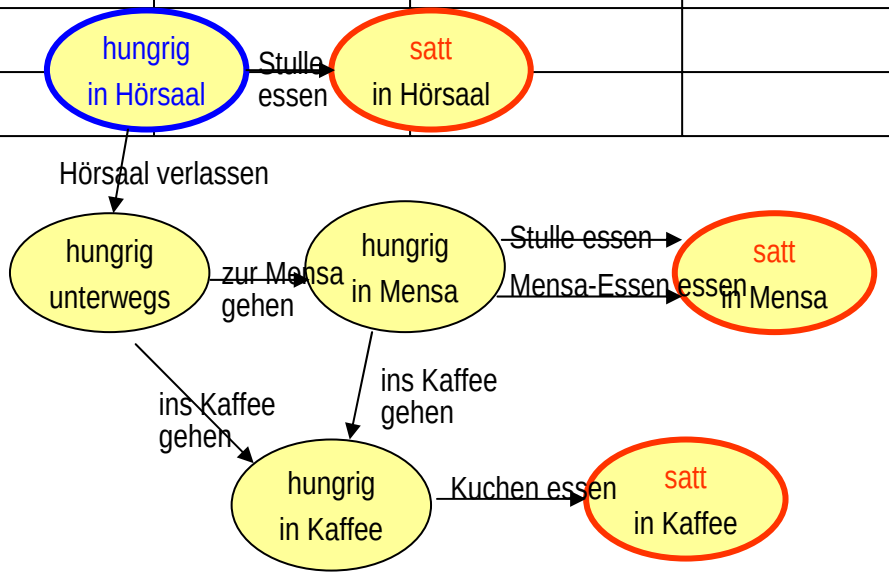
Ausgangszustand: *hungrig, im Hörsaal*

Bedingung an Zielzustände: *satt*



Übergangsmatrix

	Stulle essen	Hörsaal verlassen	zur Mensa gehen	Mensa-Essen essen	ins Kaffee gehen	Kuchen essen
hungrig, in Hörsaal	s., i.H.	h., u.				
hungrig, unterwegs			h., i.M.		h., i.K	
satt, in Hörsaal						
hungrig, in Mensa	s., i.M.			s., i.M.	h., i.K.	
hungrig, in Kaffee						s., i.K.
satt, in Mensa						
satt, in Kaffee						



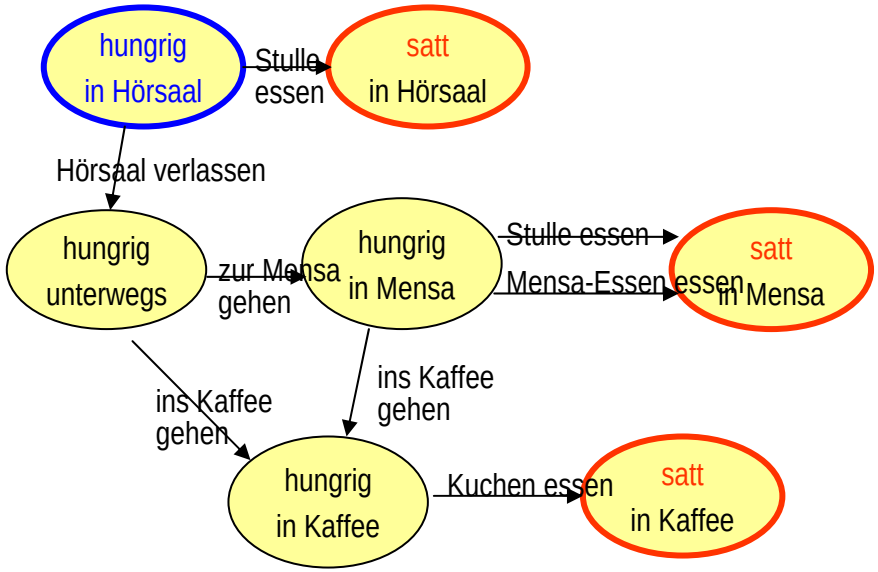
Übergangsmatrix

	Stulle essen	Hörsaal verlassen	zur Mensa gehen	Mensa-Essen essen	ins Kaffee gehen	Kuchen essen
hungrig, in Hörsaal	s., i.H.	h., u.				
hungrig, unterwegs			h., i.M.		h., i.K	
satt, in Hörsaal						
hungrig, in Mensa	s., i.M.			s., i.M.	h., i.K.	
hungrig, in Kaffee						s., i.K.
satt, in Mensa						
satt, in Kaffee						

Zeilen: Zustände z

Spalten: Aktionen a

Matrix-Element: von z durch a erreichter Zustand z´



- Transitionssystem
- Automat
- Akzeptor

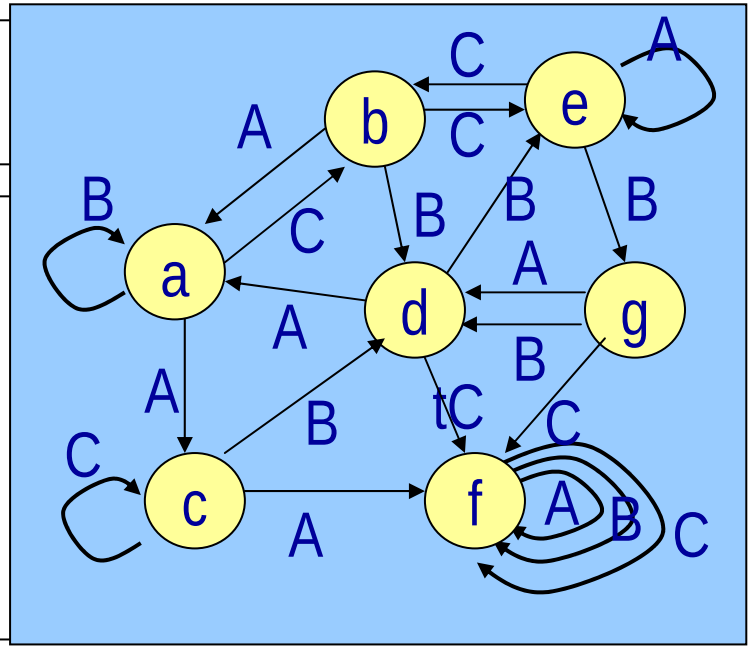
Transitionssystem

$T = [Z,X,\delta]$ mit

Z : Zustandsmenge

X : Eingangssignale

$\delta: Z \times X \rightarrow Z$ Überführungsfunktion



Graph mit Knoten für Zustände z aus Z
und Kanten für Übergänge von z nach $\delta(z,x)$

Erweiterung: $\delta: Z \times X^* \rightarrow Z$

$\delta(z,\lambda)=z$

$\delta(z,x_1...x_n\ x)= \delta(\delta(z,x_1...x_n), x)$

$\delta(z,x_1...x_n)$ ist der von z

mit der Folge $x_1...x_n$

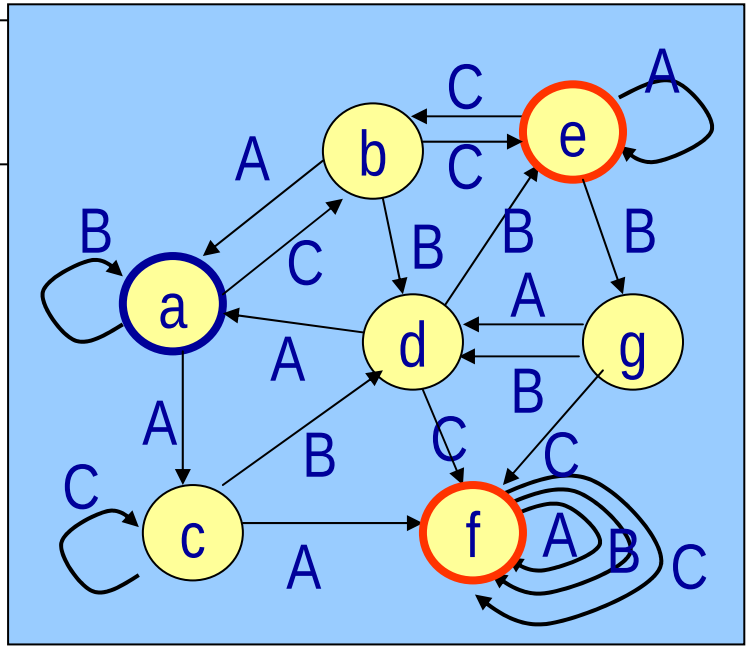
erreichte Zustand

Akzeptor

$T = [Z,X,\delta]$ mit

„Anfangszustand“ $z_0 \in Z$

Menge von „Zielzuständen“ $Z_f \subseteq Z$



Akzeptierte Sprache:

$L(T, z_0, Z_f) = \{ x_1...x_n \mid \delta(z_0,x_1...x_n) \in Z_f \} \subseteq X^*$

L ist <u>regulär</u>,

genau dann, wenn $T = [Z,X,\delta]$, $z_0 \in Z$, $Z_f \subseteq Z$ existieren

mit X,Z endlich und $L = L(T, z_0, Z_f)$.

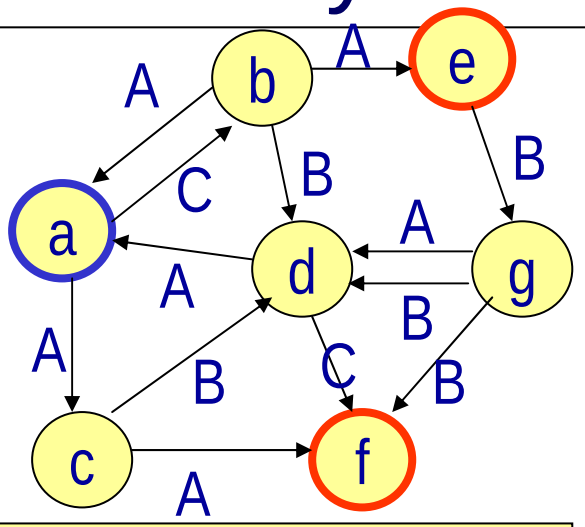
Nicht-deterministischesTransitionssystem

$T = [Z,X,f]$ mit

Z : Zustandsmenge

X : Eingangssignale

$f : Z \times X \rightarrow 2^Z$ Überführungsfunktion



Erweiterung: $f: 2^Z \times X^* \rightarrow 2^Z$

$f(M,\lambda)=M$

$f(M,x_1...x_n\ x)= f(f(M,x_1...x_n), x)$

$f(z,x_1...x_n)$ sind die von z mit der Folge $x_1...x_n$ erreichten Zustände

Akzeptierte Sprache:

$L(T, z_0, Z_f) = \{ x_1...x_n \mid f(z_0,x_1...x_n) \cap Z_f \neq \varnothing \}$

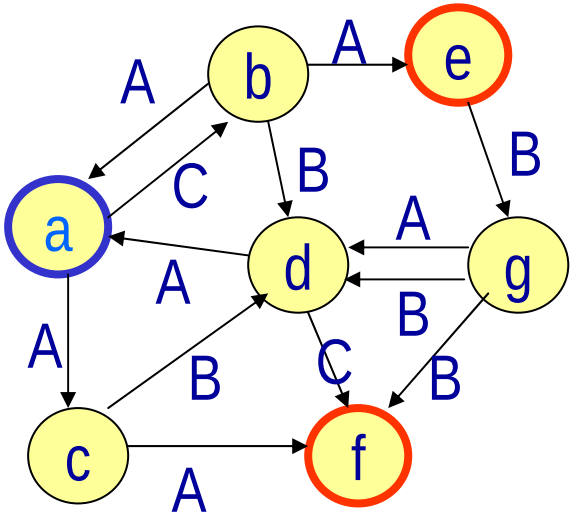
regulär, falls X,Z endlich

Transitionssystem: Akzeptor

```
accept(P):-initial(Z),accept(Z,P).

accept(Z,[X|P]) :- delta(Z,X,Z1),accept(Z1,P).

accept(Z,[ ])      :- final(Z).
```




```
initial(a).
final(e).
final(f).
```

```
delta(a,x_c,b).
delta(a,x_A,c).
delta(b,x_A,a).
delta(b,x_A,e).
delta(b,x_B,d).
...
delta(g,x_B,f).
```

„Nicht-deterministisch“

Komplexität (Anzahl der Zustände/Knoten)

8-er Puzzle: 9! Zustände

davon 9!/2 = 181.440 erreichbar

15-er Puzzle: 16! Zustände

davon 16!/2 erreichbar

ungarischer Würfel: $12 \cdot 4{,}3 \cdot 10^{19}$ Zustände

1/12 davon erreichbar: $4{,}3 \cdot 10^{19}$

Türme von Hanoi: 3^n Zustände für n Scheiben

lösbar in $(2^n) - 1$ Zügen

Dame: ca 10^{40} Spiele durchschnittlicher Länge

Schach: ca 10^{120} Spiele durchschnittlicher Länge

Go: 3^{361} Stellungen

Suchverfahren in Graphen

Graph: $G = [Z,E]$ mit

- Anfangszustand $z_0 \in Z$
- Zielzuständen $Z_f \subseteq V$

Probleme:

- Speicher reicht nicht für vollständigen Zustandsraum
- Aufwand für Erkennen von Wiederholungen

Lösungsmethode:

„Expansion des Zustandsraumes“:

Schrittweise Konstruktion und Untersuchung von Zuständen

„konstruieren – testen – vergessen“

Expansionsstrategien

- **Richtung**
  - Vorwärts, beginnend mit z_0
    (forward chaining, data driven, bottom up)
  - Rückwärts, beginnend mit Z_f
    (backward chaining, goal driven, top down)
  - Bidirektional

- **Ausdehnung**
  - Tiefe zuerst
  - Breite zuerst

- **Zusatzinformation**
  - blinde Suche
  - heuristische Suche

Aufgaben von Suchalgorithmen

- Probleme:
  - Existiert ein Weg von z_0 zu einem $z_f \in Z_f$
  - Konstruiere einen Weg von z_0 zu einem $z_f \in Z$
  - Konstruiere optimalen Weg von z_0 zu einem $z_f \in Z_f$

Güte von Suchalgorithmen

Bezogen auf Konstruktion von Wegen zum Ziel:

- **Korrektheit:**
  - Algorithmus liefert nur korrekte Wege.
- **Vollständigkeit:**
  - Algorithmus liefert (mindestens) alle korrekten Wege.
- **Optimalität:**
  - Algorithmus liefert optimale Wege.

Vollständigkeit kann auch schwächer gefasst werden (vgl. Existenzproblem): Algorithmus liefert einen korrekten Weg, falls eine Lösung existiert.

Komplexität von Suchalgorithmen

- bzgl. Komplexität des Verfahrens:
  - Zahl der Zustände insgesamt
  - Zahl der erreichbaren Zustände
  - Zahl der untersuchten Zustände
  - Suchtiefe
- bzgl. Gefundener Lösung

Graph: $G = [Z,E]$ mit

- Anfangszustand $z_0 \in Z$
- Zielzuständen $Z_f \subseteq V$

Zyklen, Maschen im Suchraum

Prolog:

```
erreichbar(X,Y) :- erreichbar(X,Z), nachbar(Z,Y).
erreichbar(X,X).

symmetrisch(X,Y):- symmetrisch(Y,X).
```

Test auf Wiederholungen:

Zeit-, Speicher- aufwändig

Beschränkung der Suchtiefe

Suche nach einem Weg

Expansion: Schrittweise Konstruktion des Zustandsraums

Datenstrukturen :

- Liste OPEN:

Ein Zustand (Knoten) heißt "offen" , falls er bereits konstruiert, aber noch nicht expandiert wurde (Nachfolger nicht berechnet)

- Liste CLOSED:

Ein Zustand (Knoten) heißt "abgeschlossen" , falls er bereits vollständig expandiert wurde (Nachfolger alle bekannt)

Zusätzliche Informationen:
z.B. Nachfolger/Vorgänger der Knoten
(für Rekonstruktion gefundener Wege)

Schema S (Suche nach irgendeinem Weg)

S0: (Start) Falls Anfangszustand z_0 ein Zielzustand: EXIT(„yes:“ z_0).
OPEN := [z_0] , CLOSED := [ ] .

S1: (negative Abbruchbedingung) Falls OPEN = [ ] : EXIT(„no“).

S2: (expandieren)
Sei z der erste Zustand aus OPEN.
OPEN := OPEN – {z} . CLOSED := CLOSED $\cup${z} .
Bilde die Menge Succ(z) der Nachfolger von z.
Falls Succ(z) = {} : Goto S1.

S3: (positive Abbruchbedingung)
Falls ein Zustand z_1 aus Succ(z) ein Zielknoten ist: EXIT(„yes:“ z_1).

S4: (Organisation von OPEN)
Reduziere die Menge Succ(z) zu einer Menge NEW(z)
durch Streichen von nicht weiter zu betrachtenden Zuständen.
Bilde neue Liste OPEN durch Einfügen der Elemente aus NEW(z) .
Goto S1.

Variable Komponenten in Schema S :

(Re-)Organisation von OPEN in S4

- V1. Bildung der Menge NEW(z) aus Succ(z) :
  (Auswahl der weiter zu betrachtenden Zustände)
  - alle Zustände aus Succ(z)
  - einige (aussichtsreiche)
  - nur die, die noch nicht in OPEN
  - nur die, die nicht in CLOSED
- V2. Sortierung von OPEN
  (bestimmt den nächsten zu expandierenden Zustand in S2)
  - NEW(z) sortieren
  - NEW(z) einfügen, z.B. an Anfang oder Ende,
  - OPEN (gesamte Liste) neu sortieren
- V3. Weitere Bedingungen
  - Beschränkung der Suchtiefe
  - Reduzierte Menge CLOSED

Blinde Suche mit Test auf Wiederholungen:

(1) Tiefe-Zuerst:

- V1: NEW(z) = Succ(z)-(OPEN ∪CLOSED)
- V2: NEW(z) an den Anfang von OPEN

Keller

(2) Breite-Zuerst:

- V1: NEW(z) = Succ(z)-(OPEN ∪CLOSED)
- V2: NEW(z) an das Ende von OPEN

Warteschlange

Für endliche Graphen:
korrekt und vollständig

Hoher Speicheraufwand
speziell für CLOSED

Vollständig im Sinne:
findet (eine) Lösung im Fall der Existenz

Blinde Suche ohne Test auf Wiederholungen:

Graph als „Abgewickelter Baum“

(1) Tiefe-Zuerst:

- V1: NEW(z) = Succ(z)
- V2: NEW(z) an den Anfang von OPEN

(2) Breite-Zuerst:

- V1: NEW(z) = Succ(z)
- V2: NEW(z) an das Ende von OPEN

Für endliche Graphen:

Tiefe-Zuerst: korrekt, aber nicht immer vollständig

Breite-Zuerst: korrekt und vollständig

Blinde Suche ohne Test auf Wiederholungen:

Graph als „Abgewickelter Baum“

Speicheraufwand für OPEN

- Tiefe-Zuerst: linear $d \cdot b$
- Breite-Zuerst: exponentiell b^d
  (bei b=fan-out, d=Tiefe)

Hoher Zeitaufwand für Wiederholungen

Backtracking

Implementierung von Tiefe-zuerst-Verfahren

Spezielle Organisation der Liste OPEN:

Referenz auf jeweils nächsten zu expandierenden Zustand in jeder Schicht

Nach Abarbeiten aller Zustände einer Schicht zurücksetzen (backtracking) auf davor liegende Schicht

Möglichkeit für Zyklenvermeidung mit reduzierter Menge CLOSED (nur für aktuellen Zweig):

– Beim Backtracking Rücksetzen von CLOSED auf früheren Stand

Iterative Tiefensuche

Stufenweise begrenzte Tiefensuche

- Stufe 1: begrenzte Tiefensuche bis zur Tiefe 1
- Stufe 2: begrenzte Tiefensuche bis zur Tiefe 2
- Stufe 3: begrenzte Tiefensuche bis zur Tiefe 3
- ...

„Depth-first-iterative deepening (DFID)“

Iterative Tiefensuche

DFID bis Tiefe d bei fan-out b erfordert insgesamt

$b^d + 2 \cdot b^{d-1} + 3 \cdot b^{d-2} + ... + (d-1) \cdot b$ Schritte

Vergleich mit Tiefe-Zuerst/ Breite-Zuerst bis Tiefe d :

$b^d + b^{d-1} + b^{d-2} + ... + b$ Schritte

DFID hat Speicherbedarf für OPEN wie Tiefe-zuerst

DFID findet Lösung wie Breite-zuerst

Heuristische Suche

Schätzfunktion $\sigma(z)$:

geschätzter Konstruktions-Aufwand

für Erreichen eines Zielzustandes von z aus

Heuristik: Zustände mit optimaler Schätzung bevorzugen

Heuristische Suche

(3) Bergsteigen/“hill climbing“ (ohne Test auf Wdh.):

- V1: NEW(z) = Succ(z)
- V2: NEW(z) nach Aufwand sortiert an Anfang von OPEN

(4) Bestensuche (ohne Test auf Wdh.):

- V1: NEW(z) = Succ(z)
- V2: OPEN $\cup$ NEW(z) nach Aufwand sortieren

Für endliche Graphen:

korrekt, aber nicht immer vollständig

Bei beiden Verfahren oft nicht alle Zustände aus NEW(z) weiter betrachten (Speicher sparen, „Pruning“)

Typische Probleme lokaler Optimierung

Vorgebirgsproblem:

steilster Anstieg führt auf
lokales Optimum ("Nebengipfel")

Plateau-Problem:

keine Unterschiede in der
Bewertung

Grat-Problem:

vorgegebene Richtungen
erlauben keinen Anstieg

Konsequenz: zwischenzeitliche Verschlechterungen zulassen

Suche nach "bestem Weg“

Bester/optimaler Weg:

Minimale Kosten bzgl. einer Kostenfunktion.

Unterschied zu Schätzfunktion σ

Suche nach "bestem Weg“

Kosten für Zustandsübergang (Kante)

$c: E \rightarrow \mathcal{R}^+$ (Kosten stets positiv!)

mit $c(e)$ = Kosten der Kante $e \in E$

bzw. $c(z,z')$ = Kosten der Kante $e=[z,z']$

Weg-Kosten als Summe von Kosten der Kanten.

Kosten eines Weges $s = e_1 \ldots e_n \in E^*$:

$$c(e_1 \ldots e_n) = \Sigma_{i=1,\ldots,n}\, c(e_i)$$

Kosten eines Weges $s = z_0\, z_1 \ldots z_n \in Z^*$

$$c(z_0\, z_1 \ldots z_n) = \Sigma_{i=1,\ldots,n}\, c(z_{i-1}, z_i)$$

Suche nach "bestem Weg“

Kosten für Erreichen des Zustandes z´ von z aus:

– Falls z´ von z erreichbar:

$g(z,z') := \text{Min}\{ c(s) / s \text{ Weg von z nach z´ }\}$,

– Andernfalls: $g(z,z') := \infty$

Vorläufigkeit der Kostenberechnung während Expansion:

$G' = [Z', E']$ sei (bekannter) Teilgraph von G

$g'(z, z', G') := \text{Min } \{ c(s) / s \text{ Weg in G´ von z nach z´ }\}$

$g'(z,z',G') \geq g(z,z')$

Suche nach "bestem Weg“

Verfahren “Generate and Test”:
Alle Wege im Graphen untersuchen.

$L(z_0)$

= Menge der in z_0 beginnenden Wege $p= v_0 \ldots v_n$

$L(z_0,Z_f)$

= Menge der in z_0 beginnenden Wege $p= v_0 \ldots v_n$ mit $v_n \in Z_f$

Kürzesten Weg in $L(z_0,Z_f)$ bestimmen.

Suche nach bestem Weg

Schema S (Suche nach irgendeinem Weg)

findet eventuell zuerst teure Wege

S0: (Start) Falls

OPEN := [z

S1: (negative Abbruchbedingung) Falls OPEN = [ ] : EXIT(„no“).

S2: (expandieren)

Sei z der erste Zustand aus OPEN.

OPEN := OPEN – {z} . CLOSED := CLOSED ∪{z} .

Bilde die Menge Succ(z) der Nachfolger von z.

Falls Succ(z) = {} : Goto S1.

S3: (positive Abbruchbedingung)

Falls ein Zustand z_1 aus Succ(z) ein Zielknoten ist: EXIT(„yes:“ z_1).

S4: (Organisation von OPEN)

Reduziere die Menge Succ(z) zu einer Menge NEW(z)

durch Streichen von nicht weiter zu betrachtenden Zuständen.

Bilde neue Liste OPEN durch Einfügen der Elemente aus NEW(z) .

Goto S1.

Suche nach "bestem Weg“

S0: (Start) Falls Anfangszustand z_0 ein Zielzustand: EXIT(„yes:“ z_0).
OPEN := [z_0] , CLOSED := [ ] .
S1: (negative Abbruchbedingung) Falls OPEN = [ ] : EXIT(„no“).
S2: (expandieren)
Sei z der erste Zustand aus OPEN.
OPEN := OPEN - {z} . CLOSED := CLOSED ∪{z} .
Bilde die Menge Succ(z) der Nachfolger von z.
z) = {} : Goto S1.
Reduziere die Menge Succ(z) zu einer Menge NEW(z)
durch Streichen von nicht weiter zu betrachtenden Zuständen.
lde neue Liste OPEN durch Einfügen der Elemente aus NEW(z) .

Lösungsidee jetzt:

Abbrechen, wenn alle offenen Wege teurer sind als aktuell gefundene Lösung

dafür:

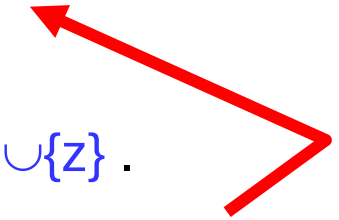
- Positive Abbruchbedingung von Schema S verändern
- Umstellung der Schritte in Schema S

Schema S‘ für Suche nach "bestem Weg“

S‘0: (Start) Falls Anfangszustand z_0 ein Zielzustand: EXIT(„yes:“ z_0).
OPEN := $[z_0]$, CLOSED := [ ] .

S‘1: (negative Abbruchbedingung) Falls OPEN = [ ] : EXIT(„no“).

S‘2: **(positive Abbruchbedingung)**
Sei z der erste Zustand aus OPEN.
Falls z ein Zielknoten ist: EXIT(„yes:“ z).



S‘3: (expandieren)
OPEN := OPEN - {z} . CLOSED := CLOSED $\cup${z} .
Bilde die Menge Succ(z) der Nachfolger von z.
Falls Succ(z) = {} : Goto S‘1.

S‘4: (Organisation von OPEN)
- $g´(z_0 ,z´,G´)$ für alle $z´ \in$Succ(z) berechnen (im aktuellen G´) .
- Neue Liste OPEN durch Einfügen der Elemente aus Succ(z):
  Sortieren von OPEN $\cup$ Succ(z) nach aufsteigendem $g´(z_0 ,z´,G´)$

Goto S‘1.

Schema S‘ für Suche nach "bestem Weg“

Satz :

Vor.: Es existiert $\delta > 0$ mit $c(z,z') > \delta$ für alle Kanten in G

Beh.: Falls Lösung existiert, so findet S‘ einen optimalen Weg

- „Verzweigen und Begrenzen“ (Branch and bound)
- „Dijkstra's Algorithmus“ (1959)

Verbesserungen möglich:

Streichen aus OPEN (bzw. Succ(z)):

- Zustände aus CLOSED
- mehrmaliges Auftreten von Zuständen

- Prinzip der Dynamischen Optimierung/Programmierung

Heuristische Suche nach „bestem Weg“

Problem:

Gleichzeitig **Kostenfunktion** g´(z, z´,G´) *(bisheriger Weg)*

und **Schätzfunktion** $\sigma(z)$ *(zukünftiger Weg)* berücksichtigen

Vorläufigkeit der Kostenberechnung während Expansion:

G´= [Z´, E´] sei (bekannter) Teilgraph von G

g´(z, z´,G´) := Min { c(s) / s Weg in G´ von z nach z´ }

Schätzfunktion $\sigma(z)$: geschätzter Konstruktions-Aufwand für Erreichen eines Zielzustandes von z aus

Heuristik: Zustände mit optimaler Schätzung bevorzugen

Schema S‘‘ für heurist. Suche nach "bestem Weg“

S‘‘0: (Sta … z_0).

OPE…

S‘‘1: (neg…

S‘‘2: **(positive Abbruchbedingung)**

Sei z d…

Falls z ei…

S‘‘3: (expandieren)

OPEN := OPEN - {z} . CLOSED := CLOSED ∪{z} .

Bilde die Menge Succ(z) der Nachfolger von z.

Falls Succ(z) = {} : Goto S‘1.

S‘‘4: (Organisation von OPEN)

– $g´(z_0 ,z´,G´) + \sigma(z)$ für alle $z´ \in$ Succ(z) berechnen (im aktuellen G´)

– Neue Liste OPEN durch Einfügen der Elemente aus Succ(z):

– Sortieren von OPEN ∪ Succ(z) nach aufsteigendem $g´(z_0 ,z´,G´) + \sigma(z)$

Goto S‘‘1.

Modifikation von Schema S‘ in Schritt 4:

$g´(z_0 ,z´,G´) + \sigma(z)$ anstelle von $g´(z_0 ,z´,G´)$

Verfälschung des Ergebnisses durch $\sigma(z)$ möglich.

Heuristische Suche nach „bestem Weg“

Schätzfunktion σ heisst *optimistisch* oder *Unterschätzung*,

falls $\sigma(z) \leq g(z,Z_f)$ für alle $z \in Z$.

Satz :

Vor.: Es existiert $\delta > 0$ mit $c(z,z') > \delta$ für alle Kanten in G

σ sei eine optimistische Schätzfunktion

Beh.: Falls Lösung existiert, so findet S‘ einen optimalen Weg

Schema S‘ für heurist. Suche nach "bestem Weg“

Streichen von CLOSED-Zuständen aus OPEN kann bei S‘ zu Problemen führen.

Benötigen schärfere Bedingungen an σ:

„konsistente Schätzfunktion“

Algorithmus A*

Heuristik vs. Kosten

	ohne Kosten c	mit Kosten c
ohne Heuristik σ	*Blinde Suche, irgendein Weg, S*	*Bester Weg, S‘*
mit Heuristik σ	*Heurist. Suche, irgendein Weg, S*	*S“ (A*)*

Gierige Suche (greedy search)

Allgemein: Eine aktuell beste Bewertung bevorzugen

In Suchverfahren:
Expansion bei aktuell besten Werten für g‘ und/oder σ

Varianten:

- OPEN vollständig sortieren (vgl. Bestensuche)
- NEW(z) sortiert an Anfang von OPEN (vgl. Bergsteigen)
- Nur den besten Zustand weiter verfolgen

    L**ocal greedy:** Variante von Bergsteigen

„Lokale Optimierung“

Probleme bzgl. Vollständigkeit/Korrektheit

Theoretische Grundlagen von PROLOG

Probleme:

Automatische Beweisverfahren

Was leistet der PROLOG-Interpreter ?

„Program = Logic + Control“

Prozedurale/imperative Sprachen:

- Abläufe formulieren
- Computer führt aus

von-Neumann-Maschine

Idee von deklarativen/logischen/funktionalen Programmiersprachen:

- Zusammenhänge formulieren
- Computer erschließt weitere Zusammenhänge

Built-in

Prädikate

Programm:

Nutzerdefinierte Prädikate

Interpreter:

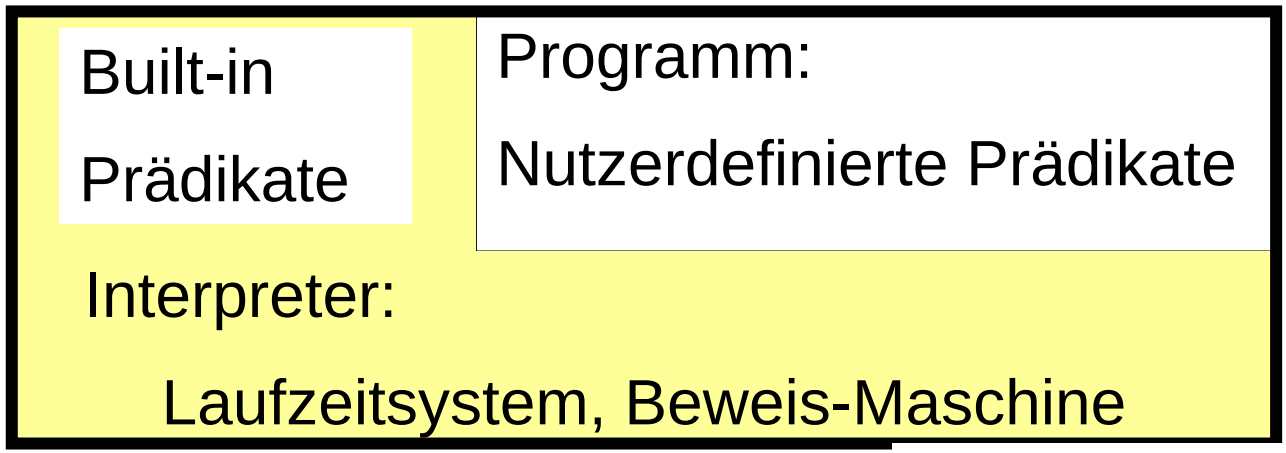
Laufzeitsystem, Beweis-Maschine

„Program = Logic + Control“

Idee von deklarativen/logischen/funktionalen Programmiersprachen:

- Zusammenhänge formulieren
- Computer erschließt weitere Zusammenhänge

Deklarative Semantik



Ablauf der Beweismaschine:

Prozedurale Semantik

Ziele der Software-Technologie:

... Der zweite Aspekt erfordert eine Sprache,
die im Idealfall ,,nahe am Problem" ist,
so daß die Konzepte der Problemlösung direkt und schlüssig formuliert werden können. ...
Die Verbindung zwischen der Sprache,
in der wir programmieren/denken,
und den Problemen und Lösungen,
die wir uns vorstellen können, ist sehr eng. ...

Bjarne Stroustrup:
Die C++ Programmiersprache, S. 10
(,,Philosophische Bemerkung")

Probleme mit Software:

Softwaresysteme gehören zu den komplexesten Gebilden,
die je von Menschenhand geschaffen wurden.
Strukturen und Abläufe in großen Systemen sind
im einzelnen oft nicht mehr überschaubar.
Man kann sie weder im vorhinein, beim Entwurf,
noch im nachhinein, beim Testen, beim Betrieb
und in der Wartung vollständig verstehen.

Denert: Software-Engeneering, S. 4

Probleme mit Software:

Das entscheidende Charakteristikum der industriell einsetzbaren Software ist, dass es für den einzelnen Entwickler sehr schwierig, wenn nicht gar unmöglich ist, alle Feinheiten des Designs zu verstehen.
Einfach ausgedrückt, überschreitet die Komplexität solcher Systeme die Kapazität der menschlichen Intelligenz.

Booch: Objektorientierte Analyse und Design

Prolog basiert auf Prädikatenlogik

Vorbild für „Formalismus“:

– exakt, präzise, (theoretisch) beherrscht

Aufbau:

–Zeichen

–Ausdrücke (rekursive Definition)

–Sätze („Theorie“) **Th**

•syntaktisch bestimmt:

Th = Abl(**Ax**)

Nach speziellen formalen Regeln erzeugbare Formeln

•semantisch bestimmt:

Th = allgemeingültige Sätze einer Struktur

Beweise

„Irreflexive, transitive Relationen sind asymmetrisch“

Inhaltlicher Beweis:

– Argumentation

Formaler (syntaktischer) Beweis:

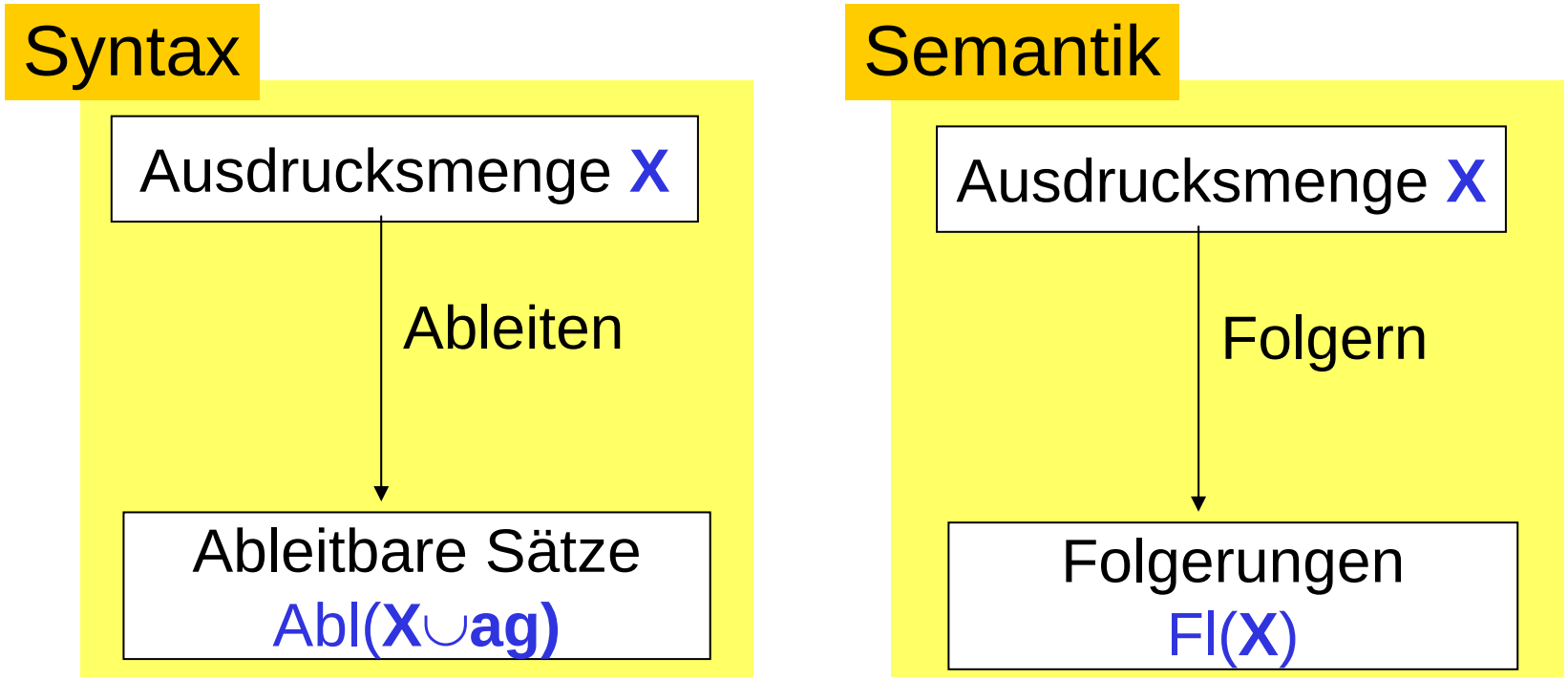
– Umformung von Ausdrücken („Kalkül“)

Theorembeweiser:

– (Syntaktisches) Verfahren zur Entscheidung, ob ein Ausdruck zu einer Satzmenge (Theorie) gehört:

$H \in Th$?

Axiomatische Behandlung der Logik



Korrektheit von Abl : $Abl(\mathbf{X} \cup \mathbf{ag}) \subseteq Fl(\mathbf{X})$

Vollständigkeit von Abl : $Abl(\mathbf{X} \cup \mathbf{ag}) \supseteq Fl(\mathbf{X})$

Äquivalenz von Abl : $Abl(\mathbf{X} \cup \mathbf{ag}) = Fl(\mathbf{X})$

Formales Ableiten (Resolutionsregel)

Für Klauseln („Disjunktion von Literalen“)

Voraussetzung:

K1 und K2 unifizierbar mittels Unifikator σ

K =Res(K1,K2,σ) entsteht durch

- „Vereinigung“ von σ(K1) und σ(K2)
- Streichen komplementärer Literale

Formales Ableiten (Resolution)

KA1: $\neg R(x,x)$

KA2: $\neg R(u,y) \vee \neg R(y,z) \vee R(u,z))$

K1: $R(c,d)$

K2: $R(d,c)$

K3 =Res(KA1,KA2,σ): $\neg R(w,y) \vee \neg R(y,w)$

mit $\sigma(u)= \sigma(z)= \sigma(x)= w$

K4 = Res(K1,K3,σ): $\neg R(d,c)$

mit $\sigma(w)=c \quad \sigma(y)=d$

K5 = Res(K2,K4,σ):

Enscheidbarkeit in der Logik

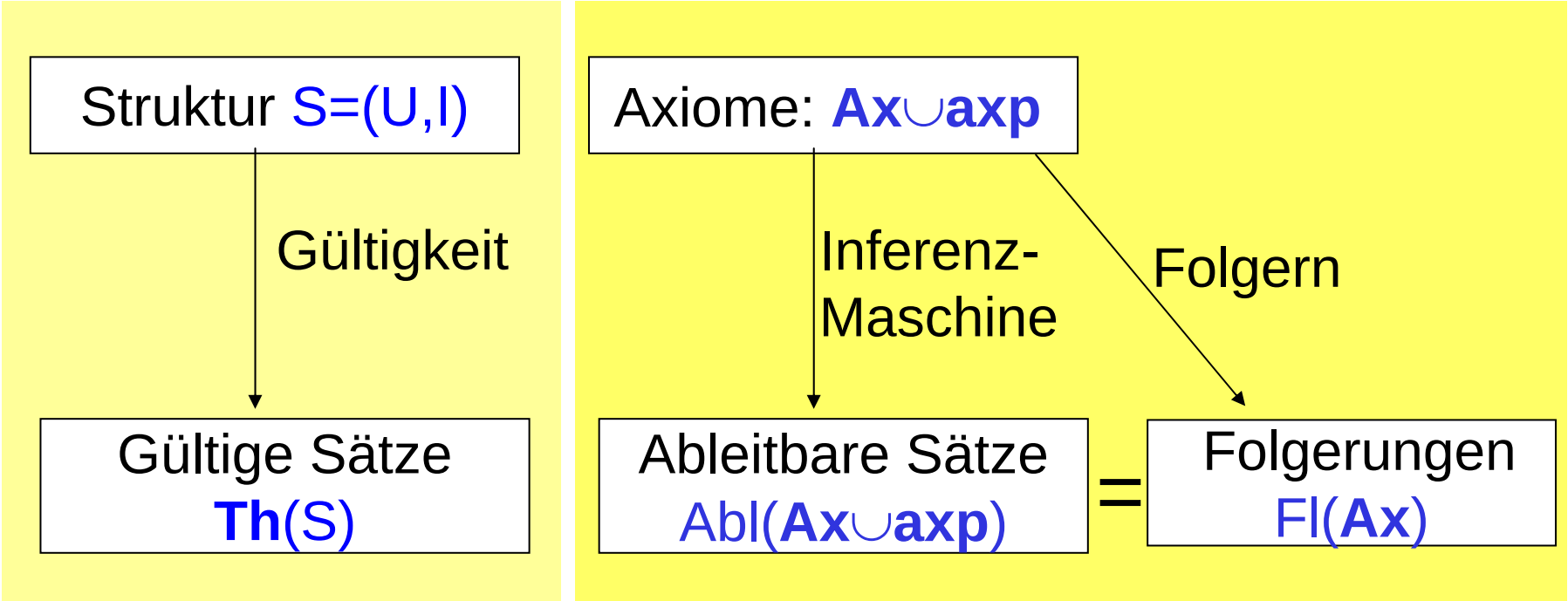
(rein logisch) allgemeingültige Sätze:
ag = Abl(**axp**) = Fl(∅)

H ∈ **ag** ?

- Entscheidbar im AK
- Unentscheidbar im PK1

  (aber aufzählbar, da axiomatisierbar)

Formalisierung einer Domäne

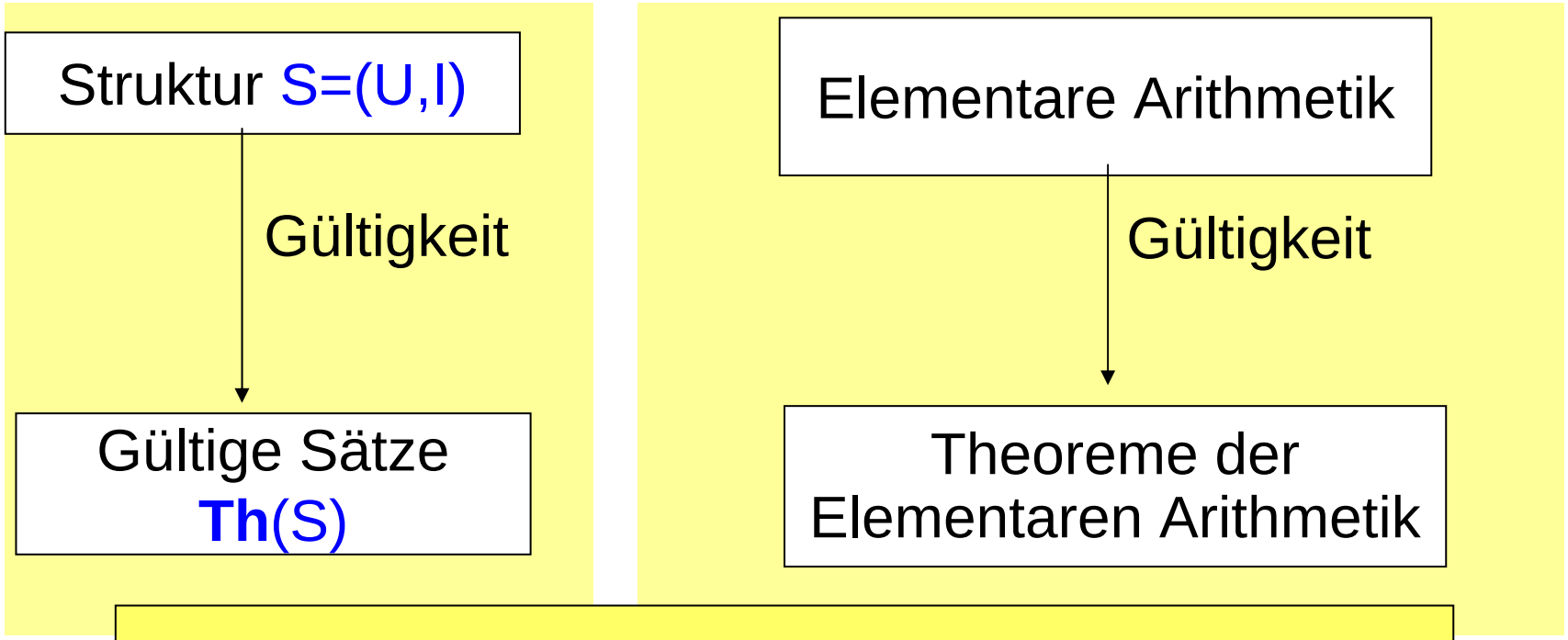


Korrektheit der Formalisierung : $Abl(\mathbf{Ax} \cup \mathbf{axp}) \subseteq \mathbf{Th}(S)$

Vollständigkeit der Formalisierung: $Abl(\mathbf{Ax} \cup \mathbf{axp}) \supseteq \mathbf{Th}(S)$

Äquivalenz der Formalisierung : $Abl(\mathbf{Ax} \cup \mathbf{axp}) = \mathbf{Th}(S)$

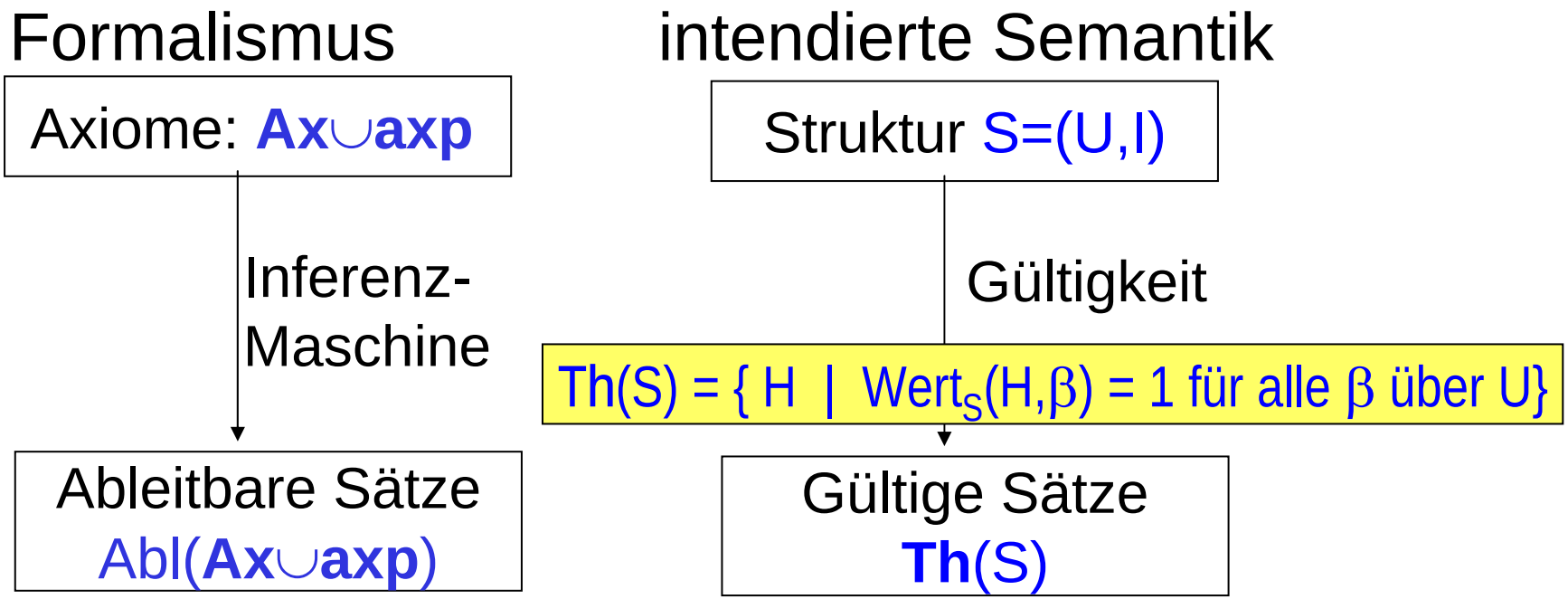
Beispiel: Formalisierung der Arithmetik



$Th(S) = \{ H \mid Wert_S(H,\beta) = 1 \text{ für alle } \beta \text{ über } U \}$

β : Belegung der Variablen
mit Individuen aus dem Universum U

Formalisierung einer Domäne

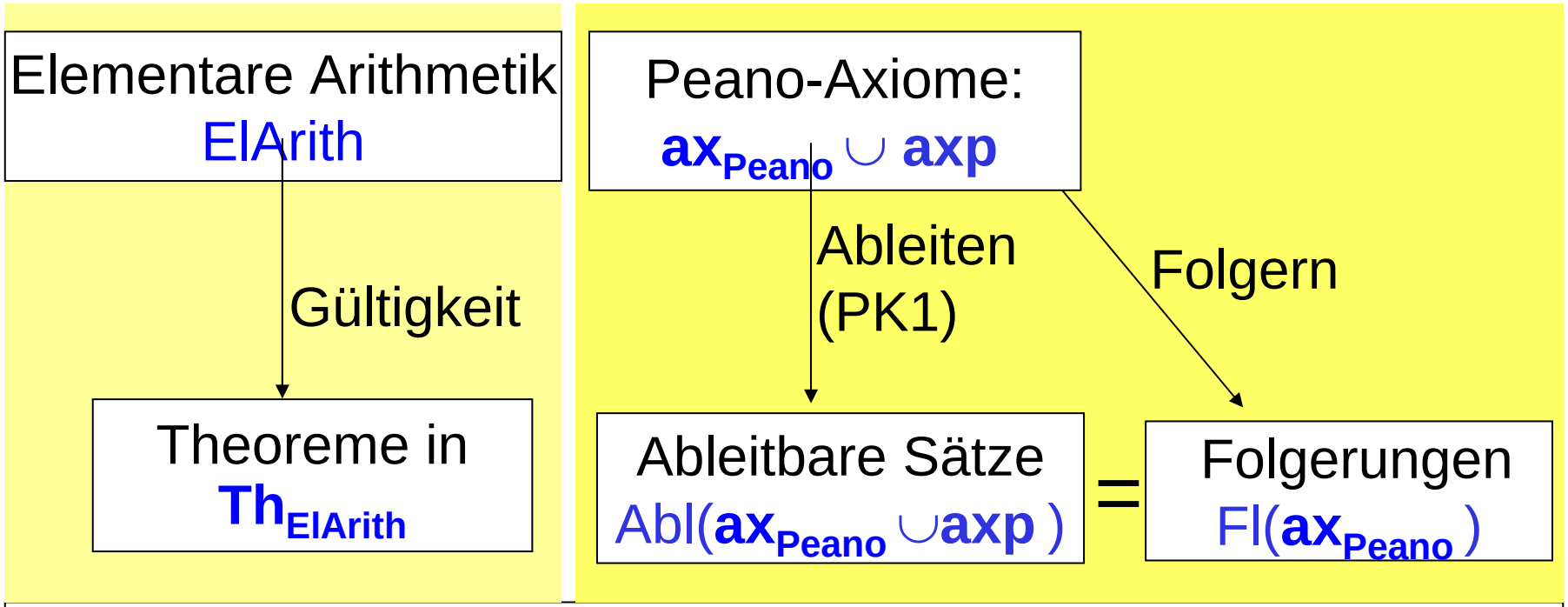


Korrektheit der Formalisierung : $Abl(\mathbf{Ax} \cup \mathbf{axp}) \subseteq \mathbf{Th}(S)$

Vollständigkeit der Formalisierung: $Abl(\mathbf{Ax} \cup \mathbf{axp}) \supseteq \mathbf{Th}(S)$

Äquivalenz der Formalisierung : $Abl(\mathbf{Ax} \cup \mathbf{axp}) = \mathbf{Th}(S)$

Beispiel: Formalisierung der Arithmetik



Korrektheit der Formalisierung :

$$Abl(\mathbf{Ax_{Peano} \cup axp}) \subseteq \mathbf{Th_{ElArith}}$$

Unvollständigkeit der Formalisierung:

$$Abl(\mathbf{Ax_{Peano} \cup axp}) \neq \mathbf{Th_{ElArith}}$$

Adäquatheit der Formalisierung

Adäquatheit:

Die wesentlichen Aspekte werden in der Struktur S

bzw. den Axiomen **Ax** korrekt erfaßt.

–Parallelen-Axiom der Geometrie

–Stetigkeitsdefinition in der Analysis

–Modellierung eines Staubsaugers

Syntax des PK1

Terme: Individuen (Konstante, Variable, Funktionen)

Prädikate (atomare Formeln):

Relationen $R(x_1,...,x_n)$ (wahr/falsch)

Ausdrücke:

logische Beziehungen zwischen Prädikaten mittels

- Aussagenlogischen Operatoren $\neg \wedge \vee \leftrightarrow \rightarrow$
- Quantifikation von Variablen $\forall \exists$

  z.B. $\forall x \exists y\, R(x, x_1,...,x_n) \wedge \neg R(y, x_1,...,x_n)$

Positives Literal: nicht negierte atomare Formel

Negatives Literal: negierte atomare Formel

Syntax des PK1

Ableiten: Ausdrücke umformen mit Ableitungsregeln

z.B. Abtrennungsregel (modus ponens)

$$\frac{H_1 \ , \ H_1 \rightarrow H_2}{H_2}$$

H *ableitbar aus* **X ,**

falls Ableitungsfolge für H aus **X** existiert

X |- H oder: H $\in$ **X**|- oder: H $\in$ Abl(**X**)

Allgemeingültigkeit, Erfüllbarkeit

Eine Struktur S = [U, I]

- beschreibt eine Menge U von Objekten:
  Individuenbereich, Universum
- legt mit der Interpretation I die Bedeutung von Funktionszeichen und Prädikatenzeichen über U fest.

Eine Belegung β ordnet jeder Variablen ein Objekt aus U zu.

Es gilt $Wert_{[U,I]}(H, \beta) = W$, falls der Ausdruck H in der Struktur S = [U, I] bei der Belegung β wahr ist.

Allgemeingültigkeit, Erfüllbarkeit

Belegung β erfüllt den Ausdruck H in der Struktur S = [U, I] ,

falls $Wert_{[U,I]}$ (H, β) = W.

(Rein logische) Erfüllbarkeit eines Ausdrucks H:

H heißt erfüllbar, falls β, U,I existieren mit $Wert_{[U,I]}$ (H, β) = W.

ef : Menge aller erfüllbaren Ausdrücke

(Rein logische) Allgemeingültigkeit eines Ausdrucks H:

H heißt allgemeingültig, falls $Wert_{[U,I]}$ (H, β) = W für alle β, U,I .

ag : Menge aller allgemeingültigen Ausdrücke

Freie Variable werden bei Allgemeingültigkeit wie generalisierte Variable behandelt.

Allgemeingültigkeit, Erfüllbarkeit

H ist allgemeingültig gdw. ¬H nicht erfüllbar ist.

H ist erfüllbar gdw. ¬H nicht allgemeingültig ist.

ag axiomatisierbar:

Es gibt aufzählbares Axiomensystem **axp** mit **ag** = Abl(**axp**)

Satz von Church:

Erfüllbarkeit/Allgemeingültigkeit sind unentscheidbar.

Menge der allgemeingültigen Ausdrücke: axiomatisierbar.

Menge der nicht erfüllbaren Ausdrücke: axiomatisierbar.

Menge der nicht allgemeingültigen Ausdrücke: nicht axiomat.

Menge der erfüllbaren Ausdrücke: nicht axiomatisierbar.

Axiomatisierbar = aufzählbar = partiell entscheidbar
M entscheidbar gdw. M und U-M aufzählbar

Folgern im PK1

Eine Struktur $S = [U, I]$ und eine Belegung β

sind ein *Modell* für eine Menge **X** von Ausdrücken,

wenn für alle $H \in \mathbf{X}$ gilt:

β erfüllt H in der Struktur $S = [U, I]$, d.h. $Wert_{[U,I]}(H, \beta) = W$.

Es sei **X** eine Menge von Ausdrücken, H ein Ausdruck.

H *folgt aus* **X** , falls gilt:

Jedes Modell von **X** ist ein Modell von H .

$\mathbf{X} \models H$ oder: $H \in \mathbf{X}\models$ oder: $H \in Fl(\mathbf{X})$

Folgern im PK1

Fl ist syntaktisch beschreibbar mittels Abl

Fl = Abl „modulo **ag**“

Fl und Abl sind monoton:

$$\mathbf{X} \subseteq \mathbf{Y} \Rightarrow \mathrm{Fl}(\mathbf{X}) \subseteq \mathrm{Fl}(\mathbf{Y})$$

$$\mathbf{X} \subseteq \mathbf{Y} \Rightarrow \mathrm{Abl}(\mathbf{X}) \subseteq \mathrm{Abl}(\mathbf{Y})$$

$\mathbf{ag} = \mathrm{Fl}(\emptyset) = \mathrm{Abl}(\mathbf{axp})$

$$(H \rightarrow G) \in \mathrm{Fl}(\mathbf{X}) \Leftrightarrow G \in \mathrm{Fl}(\mathbf{X} \cup \{H\})$$

$$H \in \mathrm{Fl}(\mathbf{X}) \Leftrightarrow (\wedge\mathbf{X} \rightarrow H) \in \mathbf{ag}$$

Formale Theorien im PK1

Als *Theorie* wird eine bezüglich Folgern (Ableiten) abgeschlossene Menge **Th** von Ausdrücken bezeichnet:

$$\mathbf{Th} \;=\; Fl(\mathbf{Th}) \;=\; Abl(\mathbf{Th} \cup \mathbf{ag})$$

Theorie mit *semantisch bestimmter Satzmenge*:

Gegeben ist eine Struktur $S = [U, I]$ mit

$$\mathbf{Th} \;=\; \{ F \mid F \text{ allgemeingültig in } S\}$$

$$= \{ F \mid Wert_{[U,I]}(F, \beta) = W \text{ für alle } \beta \text{ über } U \}$$

Theorie mit *syntaktisch bestimmter Satzmenge*:

Gegeben ist ein Ausdruckmenge **Ax** („Axiome“) mit

$$\mathbf{Th} \;=\; Abl(\mathbf{Ax} \cup \mathbf{axp}) \qquad (\; = \; Fl(\mathbf{Ax}) \;)$$

Formalisierung mittels PK1

Ziel: Axiome zur Beschreibung von Sachverhalten finden

Analoges Ziel: **Computerverarbeitung ermöglichen**

1. semantisch definierte Theorie **Th** bestimmen
   (Universum, Relationen/Funktionen, Interpretation)
2. axiomatische Beschreibung von **Th** :
   Axiome **Ax** mit **Th** = Abl(**Ax** $\cup$ a**xp**)

   (z.B. PROLOG-Programm)

Anwendung PK1

Formalisierung: Axiome **X**

Problem durch Ausdruck H beschreiben

Entscheiden, ob

$$H \in Fl\ (\mathbf{X})$$

d.h. ob $H \in Abl(\mathbf{X} \cup \mathbf{axp}\ )$

Programme dafür: Theorembeweiser

Beweise

– Positiver Kalkül: Allgemeingültigkeit entscheiden

$H \in \mathbf{Th}$?

– Negativer Kalkül: Unerfüllbarkeit untersuchen

$\{\neg H\} \cup \mathbf{Th}$ widersprüchlich ?

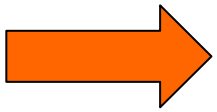
– Deduktiver Kalkül:
  Erweitern der Axiome um H zu finden (foward chaining)

– Testkalkül:
  Reduktion von H auf Axiome (backward chaining)

Beispiel: Resolution (PROLOG) ist *Negativer Testkalkül*

Beweise

$H \in Fl(\mathbf{X})$

gilt gdw. $H \in Abl(\mathbf{X} \cup \mathbf{axp})$  Positiver Kalkül

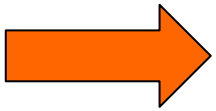
$H \in Fl(\mathbf{X})$

gilt gdw. $(\wedge\mathbf{X} \rightarrow H) \in \mathbf{ag}$

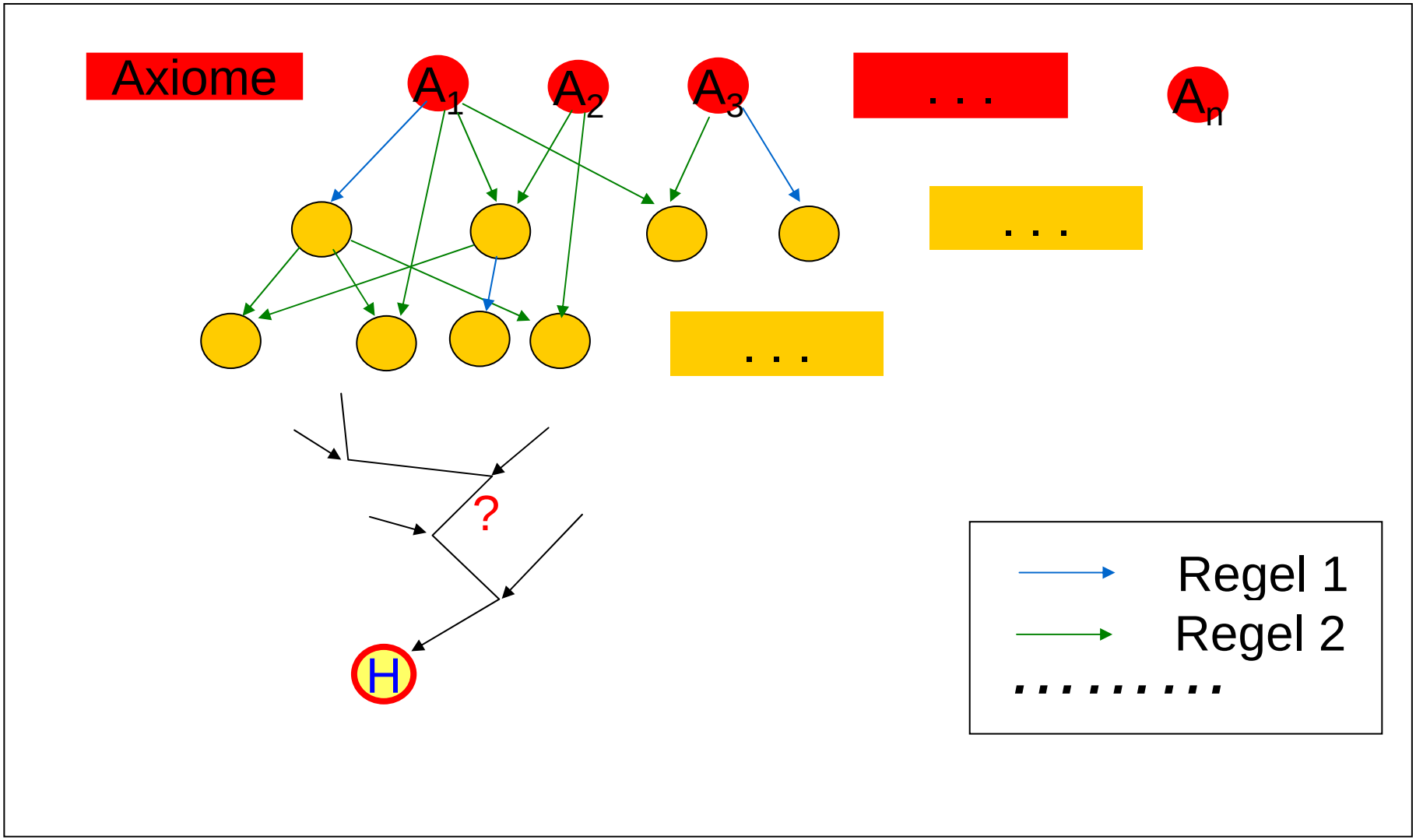
gilt gdw. $\neg(\wedge\mathbf{X} \rightarrow H) \notin \mathbf{ef}$

gilt gdw. Skolemform von $(\wedge\mathbf{X} \wedge \neg H)$ nicht erfüllbar

gilt gdw. Klauselform von $(\wedge\mathbf{X} \wedge \neg H)$ nicht erfüllbar

 Negativer Kalkül

Positiver deduktiver Kalkül



Suchraum bei deduktivem Kalkül

Klassischer AK : 15 Axiome für **ag**, 2 Regeln

(ist entscheidbar, allerdings NP)

Klassischer PK1:

- abzählbar viele Axiome für **ag** + weitere für **Th** ,
- 7 Regeln

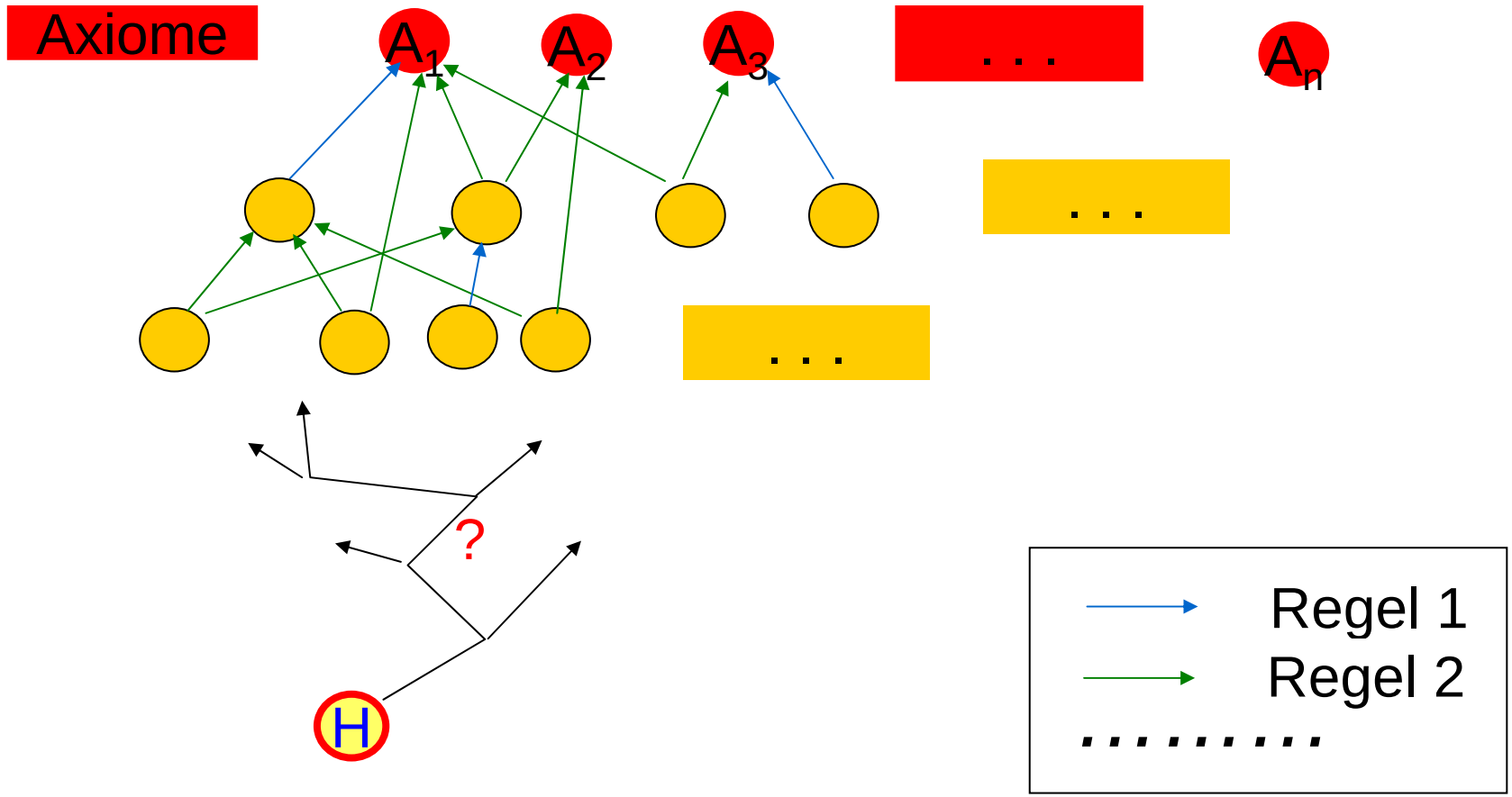
Vollständigkeit als Nachteil:

- Alle allgemeingültigen Ausdrücke im Suchraum **Th** :

  ag = Fl(∅) ⊆ Fl(**Th**) = **Th**
- Mit H viele weitere Ausdrücke im Suchraum **Th** :

  z.B. $H \vee G$ für beliebiges G

Positiver Test-Kalkül



Normalformen

ein Ausdruck H heißt *bereinigt*, falls gilt:

- keine Variable x kommt in H sowohl frei als auch gebunden vor,
- hinter den Quantoren in H vorkommende Variable sind verschieden.

ein Ausdruck H heißt *pränex*, falls gilt:

- H hat die Form $Q_1\, x_1 \ldots Q_n\, x_n\, H'$,

  wobei $Q_1, \ldots, Q_n$ Quantoren sind und H' keine Quantoren enthält.

ein Ausdruck H heißt *pränexe Normalform*, falls gilt:

- H hat pränexe Form $Q_1\, x_1 \ldots Q_n\, x_n\, H'$,
- H' ist eine konjunktive Normalform (KNF):

$$H' = \bigwedge \{ \{ \bigvee L_{ij} \mid i=1...n \} \mid j=1...m_n \}$$

(die L_{ij} sind Literale)

Literal: atomare Formel oder negierte atomare Formel

Normalformen

Sätze

1. Zu jedem Ausdruck H_1 existiert ein Ausdruck H_2 in bereinigter Form.
2. Zu jedem Ausdruck H_2 in bereinigter Form existiert ein semantisch äquivalenter Ausdruck H_3 in bereinigter pränexer Form
3. Zu jedem Ausdruck H_3 in bereinigter pränexer Form existiert semantisch äquivalenter Ausdruck H_4 in bereinigter pränexer Normalform.

Zu jedem Ausdruck H existiert ein semantisch äquivalenter Ausdruck G in bereinigter pränexer Normalform.

Skolem-Normalform

Eine Skolem-Normalform ist eine pränexe Normalform, deren Variable sämtlich generalisiert sind.

Umformung einer bereinigten pränexen Normalform G

in eine Skolem-Normalform F:

1. ∃-Quantoren einführen für alle freien Variablen in G.
   (erfüllbarkeits-äquivalente Umformung !!)
2. Elimimation aller ∃-Quantoren durch Skolem-Funktionen.

Die entstehende Formel F ist erfüllbarkeits-äquivalent zu G.

Klauselform

Eine Klausel ist eine Menge von Literalen $K = \{L_1, ..., L_n\}$.

Vorteil der Klauselnotation: Mengenschreibweise beseitigt rein formale Unterschiede äquivalenter Ausdrücke bzgl. Kommutativität/Idempotenz.

Umformung einer Skolem-Normalform F in Klauselform:

1. Verteilung der Allquantoren auf die Alternativen.

   (Semantisch äquivalente Umformung)

2. Gebundenene Umbenennungen derart, dass sich insgesamt

   alle Quantoren auf unterschiedliche Variablen beziehen.

   (Semantisch äquivalente Umformung)

3. Darstellung der Alternativen als Klauseln (Menge von Literalen).

   Darstellung von F als Menge von von Klauseln **Kl** :

$$\mathbf{Kl} = \{ \{ L_{ij} \mid j = 1, ...m_i \} \mid i = 1, ..., n \}$$

Klauselform

A1: $\forall x \neg R(x,x)$

A2: $\forall x\ \forall y\ \forall z\ (\ R(x,y) \wedge R(y,z) \rightarrow R(x,z))$

$\forall x\ \forall y\ \forall z\ (\neg(R(x,y) \wedge R(y,z)) \vee R(x,z))$

$\forall x\ \forall y\ \forall z\ (\neg R(x,y) \vee \neg\ R(y,z) \vee R(x,z))$

Klauseln für A1:

KA1: $\neg R(x,x)$

Klauseln für A2:

KA2: $\neg R(u,y) \vee \neg\ R(y,z) \vee R(x,u))$

Klauselform

Negation bei Resolutionsbeweis

$$H: \quad \forall x\, \forall y\, (\, R(x,y) \rightarrow \neg R(y,x)\,)$$

$$\neg H: \quad \neg \forall x\, \forall y\, (\, R(x,y) \rightarrow \neg R(y,x)\,)$$

$$\exists x\, \exists y\, \neg(\neg\, R(x,y) \vee \neg\, R(y,x)\,)$$

$$\exists x\, \exists y\, (\, R(x,y) \wedge R(y,x)\,)$$

$$R(c,d) \wedge R(d,c)$$

Klauseln für $\neg H$:

K1: $R(c,d)$

K2: $R(d,c)$

Substitution

Substitution σ :

Abbildung der Individuenvariablen in die Menge der Terme.

Variablenumbenennung: Ersetzung von Variablen durch Variable.

$\sigma(L)$: Ergebnis der Substitution σ für ein Literal

(rekursiv definiert).

Hintereinanderausführung von Substitutionen:

$$\sigma_1 * \sigma_2 (x) = \sigma_2(\sigma_1(x))$$

$\sigma(L)$ ist jeweils „Spezialfall“ von L .

L ist die allgemeinste Form bzgl. aller $\sigma(L)$ für beliebige σ.

Substitution, Unifikation

Eine Substitution σ heißt *Unifikator*

für eine Menge $\{L_1, ..., L_n\}$ von Literalen,

falls gilt: $\sigma(L_1) = \sigma(L_2) = ... = \sigma(L_n)$

(syntaktische Gleichheit).

Ein Unifikator σ heißt *allgemeinster Unifikator*

(m.g.u. = most general unifier), falls für jeden Unifikator σ_0 eine Substitution σ_{00} existiert mit

$\sigma_0 = \sigma * \sigma_{00}$ (σ_0 ist spezieller als σ).

Der allgemeinste Unifikator

ist eindeutig bis auf Umbenennungen.

Substitution, Unifikation

Der allgemeinste Unifikator σ einer Menge $\{ L_1,...,L_n \}$ von Literalen beschreibt die Menge der gemeinsamen Spezialisierungen (Beispiele).

Werden die Variablen der Literale zuvor separiert, so ergibt sich ein allgemeinerer Unifikator.

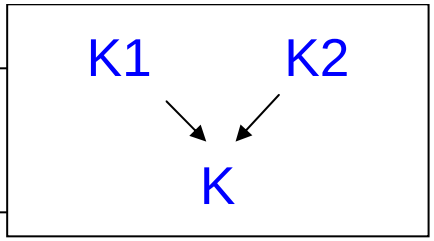
Satz

Es ist entscheidbar, ob ein Unifikator existiert.

Zu jeder unifizierbaren Menge von Literalen existiert ein allgemeinster Unifikator.

Ein allgemeinster Unifikator kann ggf. konstruiert werden.

Resolutionsregel



K_1 und K_2 Klauseln ohne gemeinsame Variablen

(sonst: Separation durch Variablenumbenennungen)

Literale $L'_1,...,L'_{n'} \in K_1$ und $M'_1,...,M'_{m'} \in K_2$, $n',m' > 0$,

die mittels eines Unifikators σ unifizierbar sind.

Durch Resolution von K_1 und K_2 entsteht die *Resolvente*

$$K = \sigma ((K_1 - \{L'_1,...,L'_{n'}\}) \cup (K_2 - \{M'_1,...,M'_{m'}\}))$$

$$\frac{\{L_1,..,L_n,L'_1,..,L'_{n'}\}, \{M_1,.., M_m,M'_1,..,M'_{m'}\}, \ \sigma(L'_1)=...=\sigma(L'_n) = \sigma(\neg M'_1)=...=\sigma(\neg M'_m)}{\sigma (\{ L_1,..,L_n , M_1,..,M_m \})}$$

Resolutionsregel

Zerlegung in zwei Regeln.

Einfache Resolutionsregel:

$$\frac{\{L_1,..,L_n,L'\},\ \{M_1,..,M_m,M'\},\ \sigma(L')=\sigma(\neg M')}{\sigma(\{L_1,..,L_n,M_1,..,M_m\})}$$

Faktorisierungsregel:

$$\frac{\{L_1,..,L_n,L_1',L'_2\},\ \sigma(L_1')=\sigma(L_2')}{\sigma(\{L_1,..,L_n,L_1'\})}$$

Spezialfälle

Schnittregel

$$\frac{A \vee B,\ \neg B' \vee C\ ,\ \sigma(B)=\sigma(B')}{\sigma\ (A \vee C)}$$

Modus ponens

$$\frac{B,\ B \rightarrow C}{C}$$

Indirekter Beweis

$$\frac{B,\ \neg\ B}{}$$

Unerfüllbarkeitsnachweis

Res(**K**) = Menge aller mit Resolution in endlich vielen Schritten aus Klauselmenge **K** ableitbarer Klauseln

Klauseln(H) = Klauseldarstellung eines Ausdrucks H (nicht eindeutig)

Satz: $H \notin$ **ef** gdw. $\in$ Res(**Klauseln**(H))

ist die leere Klausel („Widerspruch“)

Unerfüllbarkeitsnachweis

Beweisidee: Herbrand-Modelle

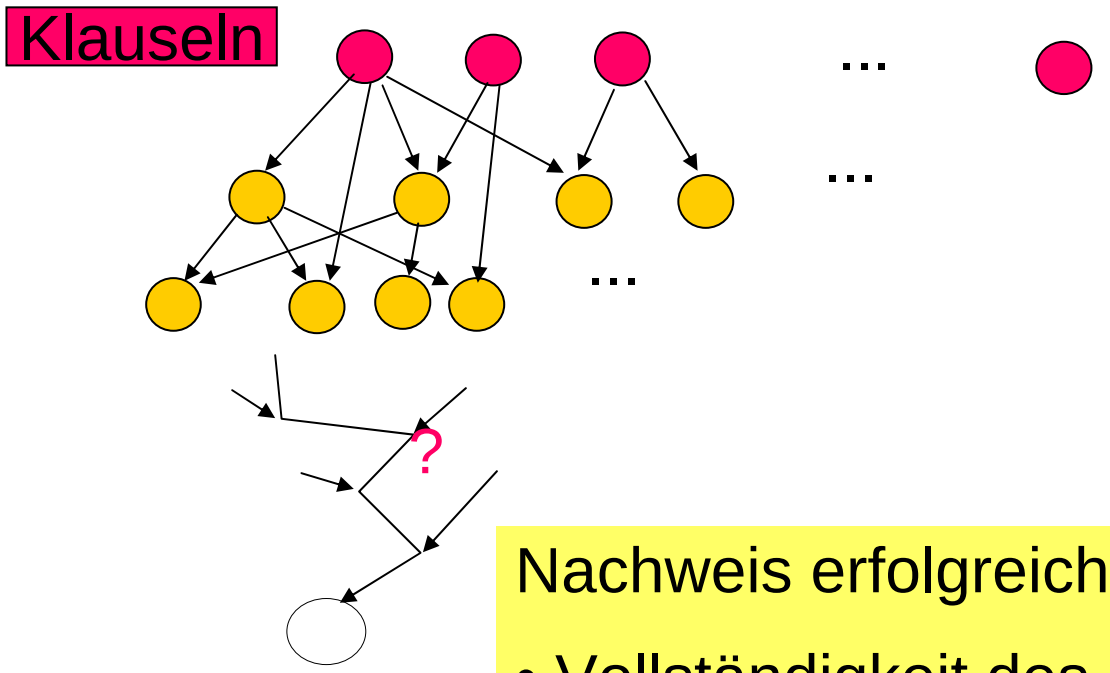
- Satz von Herbrand: Falls Klausel erfüllbar, so schon mit Grundtermen („Herbrand-Modell“: Konstante, Funktionen, keine Variablen) erfüllbar.
- Es reicht, alle durch Grundsubstitutionen (Konstante, Funktionen) erzeugbaren aussagenlogischen Formeln untersuchen (abzählbar viele).
- Resolution für abzählbar viele Formeln durchführen. Dabei Unifikation als verzögerte Substitution (lazy evaluation).

Unerfüllbarkeitsnachweis

Nachweis

der Unerfüllbarkeit ($H \notin$ **ef**) eines Ausdrucks H :

Erfolgreiche Suche in Res(**Klauseln**(H)) nach



Nachweis erfolgreich, wenn gefunden wird

- Vollständigkeit des Suchverfahrens?
- i.a. unendlicher Suchraum

Resolutionsverfahren

Nachweis von $H \in Fl(\mathbf{X})$

Konjunktive Normalform von $(\wedge \mathbf{X} \wedge \neg H)$ bilden.

Darstellung als (erfüllbarkeitsäquivalente) Klauselmenge

$$\mathbf{Kl} = \{ \{ L_{ij} \mid j = 1, \ldots m_i \} \mid i = 1, \ldots, n \}$$

Suchverfahren:

- Wiederholtes Erweitern der Klauselmenge durch neue Resolventen.
- Abbruch, wenn leere Klausel abgeleitet wurde: EXIT($H \in Fl(\mathbf{X})$).
- Wenn keine neuen Klauseln ableitbar:

  EXIT($H \notin Fl(\mathbf{X})$). (i.a. aber unendlicher Suchraum)

Resolutionsverfahren

Nachweis von $H \in Fl(\mathbf{X})$

Resolutionsverfahren liefert Lösung im Fall der Unerfüllbarkeit in endlich vielen Schritten z.B. bei Breite-Zuerst-Verfahren.

Für erfüllbare Formeln dagegen evtl. kein Resultat (unendliches Resolvieren ohne Erreichen von).

Resolutionsverfahren

i.a. unendlicher Suchraum:

Resolutionsverfahren liefert Lösung im Fall von $H \in Fl(\mathbf{X})$

(bzw. Unerfüllbarkeit der entsprechenden Klauselmenge)

in endlich vielen Schritten bei geeignetem Suchverfahren

(z.B. Breite-Zuerst).

Im Fall von $H \notin Fl(\mathbf{X})$

(bzw. Erfüllbarkeit der entsprechenden Klauselmenge)

dagegen evtl. kein Resultat

(unendliches Resolvieren ohne Erreichen von).

Beispiel: Formales Ableiten

Voraussetzungen (Axiome):

A1: $\forall x\ (\neg\ R(x,x)\ )$ (Irreflexivität)

A2: $\forall x\ \forall y\ \forall z\ (\ R(x,y)\ \wedge R(y,z) \rightarrow R(x,z)\ )$ (Transitivität)

Behauptung:

H: $\forall x\ \forall y\ (\ R(x,y) \rightarrow \neg\ R(y,x)\ )$ (Asymmetrie)

Zu zeigen:

$A1 \wedge A2 \rightarrow H$ ist allgemeingültig

bzw.

$\neg\ (A1 \wedge A2 \rightarrow H)$ ist unerfüllbar

d.h.

$(A1 \wedge A2 \wedge \neg\ H)$ ist unerfüllbar

Beispiel: Formales Ableiten (Klauselform)

$(A1 \wedge A2 \wedge \neg H)=$

$\forall x(\neg R(x,x)) \wedge \forall x \forall y \forall z(R(x,y) \wedge R(y,z) \rightarrow R(x,z)) \wedge \neg \forall x \forall y(R(x,y) \rightarrow \neg R(y,x))$

ist semantisch äquivalent zu

$\forall x(\neg R(x,x)) \wedge \forall x \forall y \forall z(R(x,y) \wedge R(y,z) \rightarrow R(x,z)) \wedge \exists x \exists y \neg (R(x,y) \rightarrow \neg R(y,x))$

ist semantisch äquivalent zur pränexen Form

$\forall x \forall u \forall y \forall z \exists v \exists w(\neg R(x,x) \wedge (R(u,y) \wedge R(y,z) \rightarrow R(u,z)) \wedge \neg (R(v,w) \rightarrow \neg R(w,v)))$

ist semantisch äquivalent zur pränexen KNF

$\forall x \forall u \forall y \forall z \exists v \exists w(\neg R(x,x) \wedge (\neg R(u,y) \vee \neg R(y,z) \vee R(u,z)) \wedge R(v,w) \wedge R(w,v))$

ist erfüllbarkeitsäquivalent zur Skolemform

$\forall x \forall u \forall y \forall z(\neg R(x,x) \wedge (\neg R(u,y) \vee \neg R(y,z) \vee R(u,z)) \wedge R(f1(x,u,y,z),f2(x,u,y,z)) \wedge R(f2(x,u,y,z),f1(x,u,y,z)))$

ist erfüllbarkeitsäquivalent zur Klauselform

$\{ \{\neg R(x,x)\},\{\neg R(u,y),\neg R(y,z),R(u,z)\}, \{R(f1(x,u,y,z),f2(x,u,y,z))\}, \{R(f2(x,u,y,z),f1(x,u,y,z))\} \}$

Beispiel: Formales Ableiten (Klauselform)

Ausgehend von der Umstellung

$(A1 \wedge A2 \wedge \neg H) = (\neg H \wedge A1 \wedge A2)$

ergibt sich einfachere Form über

$\neg\forall x\forall y(R(x,y)\rightarrow\neg R(y,x)) \wedge\forall x(\neg R(x,x)) \wedge\forall x\forall y\forall z(R(x,y)\wedge R(y,z)\rightarrow R(x,z))$

ist semantisch äquivalent zu

$\exists x\exists y\neg (R(x,y)\rightarrow\neg R(y,x)) \wedge\forall x(\neg R(x,x)) \wedge\forall x\forall y\forall z(R(x,y)\wedge R(y,z)\rightarrow R(x,z))$

ist semantisch äquivalent zur pränexen KNF

$\exists v\exists w(\forall x\forall u\forall y\forall z (R(v,w) \wedge R(w,v) \wedge \neg R(x,x)\wedge(\neg R(u,y)\vee\neg R(y,z)\vee R(u,z)))$

ist erfüllbarkeitsäquivalent zur Skolemform

$\forall x\forall u\forall y\forall z(R(c,d) \wedge(R(d,c) \wedge \neg R(x,x) \wedge(\neg R(u,y)\vee\neg R(y,z)\vee R(u,z)))$

ist erfüllbarkeitsäquivalent zur Klauselform

$\{ \{\neg R(x,x)\},\{\neg R(u,y),\neg R(y,z),R(u,z) \}, \{R(c,d)\}, \{R(d,c)\} \}$

Beispiel:Formales Ableiten (Resolution)

KA1: $\neg R(x,x)$

KA2: $\neg R(u,y) \vee \neg R(y,z) \vee R(u,z))$

K1: $R(c,d)$

K2: $R(d,c)$

K3 =Res(KA1,KA2,σ): $\neg R(w,y) \vee \neg R(y,w)$

mit $\sigma(u)= \sigma(z)= \sigma(x)= w$,$\sigma(y)= y$

K4 = Res(K1,K3,σ): $\neg R(d,c)$

mit $\sigma(w)=c$ $\sigma(y)=d$

K5 = Res(K2,K4,σ):

Resolutionsverfahren

ist ein Negativer Testkalkül

verwendet

- spezielle Normalformen (Klauseln)
- Resolutionsregel (Unifikation, Resolventen)

widerlegungsvollständig

- wenn H unerfüllbar, so mit Resolution ableitbar

und widerlegungskorrekt

- wenn mit Resolution ableitbar, so H unerfüllbar

H unerfüllbar gdw. mit Resolution ableitbar

Spezielle Resolutionsstrategien

Resolutionsmethode ist Grundlage für

Deduktionssysteme, Theorembeweiser, ...

Problem: Kombinatorische Explosion des Suchraums

Effizienzverbesserungen durch

1) Streichen überflüssiger Klauseln, z.B. :

- tautologische Klauseln K , d.h. $\{A, \neg A\} \subseteq K$
- subsumierte Klauseln: K_1 wird von K_2 subsumiert,
  falls $\sigma(K2) \subseteq K1$ bei einer geeigneten Substitution σ.

2) Heuristiken für Suchstrategie, Klauselgraphen

Problem: Widerspruchsvollständigkeit gewährleisten.

Spezielle Resolutionsstrategien

- Atomformel: *positives Literal*,
  negierte Atomformel: *negatives Literal*.
- K*lausel heißt negativ*, wenn sie nur negative Literale enthält.
- *Klausel heißt definit*, falls sie genau ein positives Literal (und evtl. noch negative Literale) enthält.
- Klausel heißt *HORN-Klausel*, wenn sie höchstens ein positives Literal (und evtl. noch negative Literale) enthält.

Hornklausel: definit (Prolog-Klauseln)

oder negativ (Prolog-Anfrage)

Spezielle Resolutionsstrategien

P-Resolution:
Eine der resolvierenden Klauseln enthält nur positive Literale.

N-Resolution:
Eine der resolvierenden Klauseln enthält nur negative Literale.

Lineare Resolution:
Resolution benutzt die im vorigen Schritt erzeugte Resolvente.

Input-Resolution (Spezialfall der linearen Resolution):
Resolution benutzt die im vorigen Schritt erzeugte Resolvente und eine Klausel der Ausgangsmenge.

Einheitsresolution:
Resolution benutzt mindestens eine ein-elementige Klausel.

Spezielle Resolutionsstrategien

P-Resolution, N-Resolution, lineare Resolution
sind widerlegungsvollständig.

Input-Resolution und Einheitsresolution
sind widerlegungsvollständig für HORN-Klauseln
(aber nicht im allgemeinen Fall).

SLD-Resolution für HORN-Klauseln

S = „Selection Function“

L = lineare Resolution

D = definite Klauseln

- Programm **P** sei eine
  Menge definiter Klauseln.
- Ein SLD-Widerlegungsbeweis
  für eine (positive) Ziel-Klausel G aus dem Programm **P**
  ist SLD-Ableitung von aus **P** $\cup$ {$\neg$G}

- Start mit negativer Klausel $\neg$G .
- In jedem Schritt wird eine negative mit einer definiten Klausel aus dem Programm **P** resolviert.
- Alle Resolventen sind negativ (Spezialfall der **linearen** Resolution, Input-Resolution, N-Resolution).

SLD-Resolution für HORN-Klauseln

Jeder Resolutionsschritt der SLD-Resolution benutzt

- eine negative (vorherige Resolvente) und
- eine **definite** Klausel (aus Programm **P**).

Es sind zwei Entscheidungen zu treffen:

1. Auswahl eines Literals $\neg L$ der negativen Klausel mittels „**selection** function“ (im Prinzip beliebig: *Und-Verzweigung*).
2. Auswahl einer Klausel mit einem positiven Literal M, das mit L unifizierbar ist (*Oder-Verzweigung*).

Suche im Und-Oder-Baum, z.B.

•Breite-Zuerst (- Findet eine existierende Lösung. -) oder

•Tiefe-Zuerst.

SLD-Resolution für HORN-Klauseln

σ_i sei die im i-ten Schritt der SLD-Resolution verwendete Substitution (Unifikator).

Dann ist $\sigma = \sigma_1 * ... * \sigma_n$ die beim Erfolg nach n Schritten erzeugte Antwortsubstitution.

Für jede Antwortsubstitution σ

eines SLD-Widerlegungsbeweises für G aus **P** gilt:

$$\mathbf{P} \models \sigma(G) \ .$$

Falls $\mathbf{P} \models \sigma(G)$,

so existiert ein SLD-Widerlegungsbeweis für G aus **P**

mit einer Antwortsubstitution σ' , die allgemeiner als σ ist.

PROLOG

Logische Programmiersprache.

Algorithmus = Logik + Steuerung

HORN-Klauseln + programmiersprachliche Konstrukte:

- E/A-Funktionen
- meta-logische Funktionen (Programm-Modifikation)
- Eingriff in Suchprozedur (Cut)

Steuerung durch Interpreter mit Auswahlstrategie:

1. links vor rechts
2. oben vor unten

und Suchstrategie:

Tiefe-Zuerst.

Prolog-Interpreter

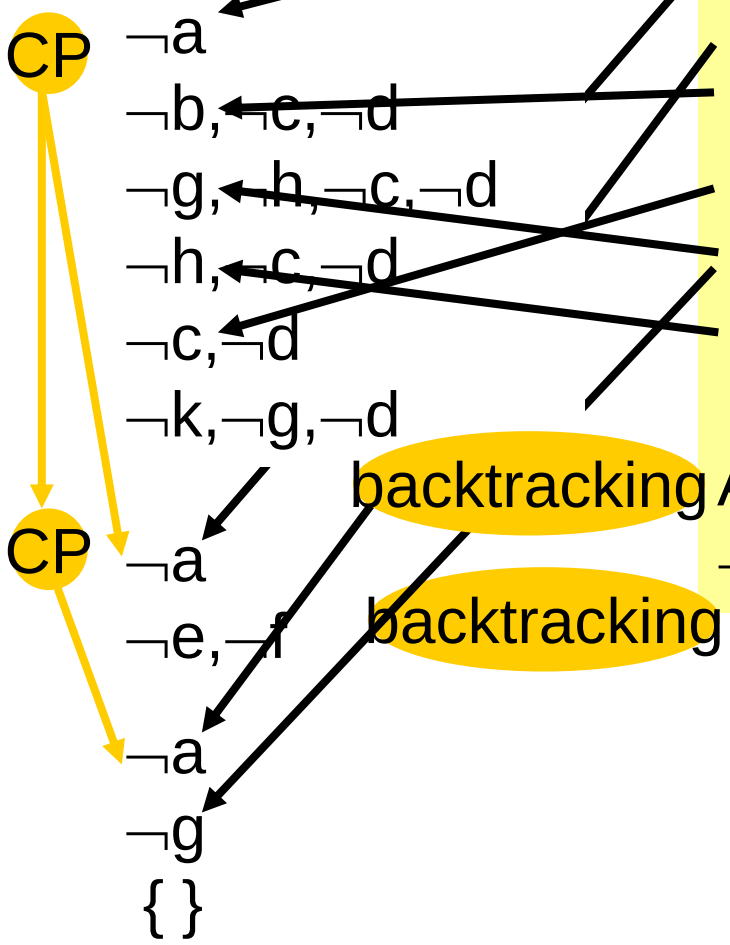
Umsetzung des SLD-Verfahrens als Zustandsraumsuche (Tiefe-Zuerst)

Verwendung des Backtracking-Prinzips (Organisiert durch Prozedurkeller)

Variablenbindungen durch Unifikation

Optimierungen

SLD-Ableitungen



a,¬b,¬c,¬d
a,¬e,¬f
a,¬g
b,¬g,¬h
c,¬k,¬g
g
h

Anfrage:
¬a

a :- b,c,d.
a :- e,f.
a :- g.
b :- g,h.
c :- k,g.
g.
h.

Anfrage:
?-a.

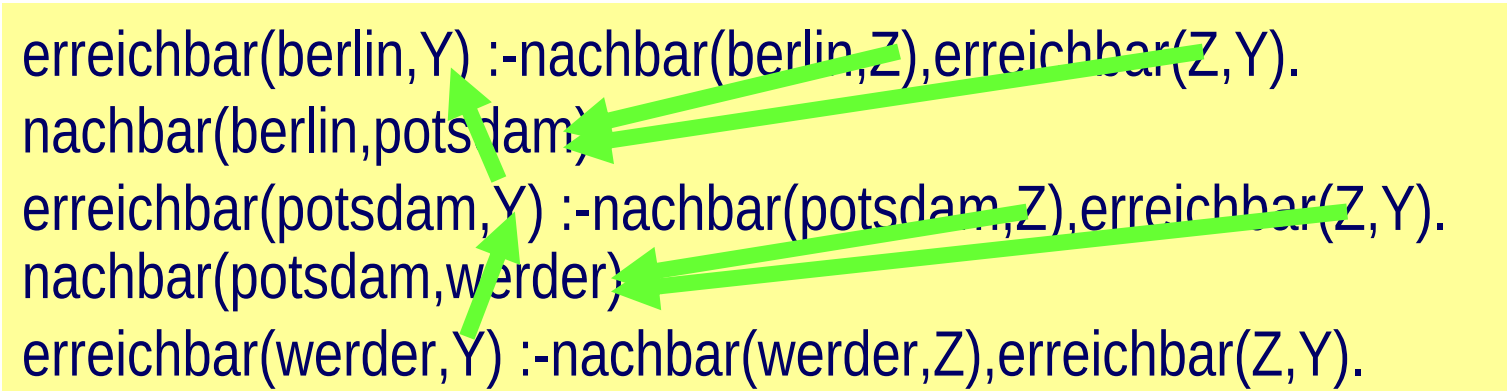
Structure Sharing vs. Structure Copying

```
erreichbar(X,Y)
      :-nachbar(X,Z),erreichbar(Z,Y).
erreichbar(X,X).
nachbar(berlin,potsdam).
nachbar(berlin,adlershof).

nachbar(potsdam,werder).
nachbar(potsdam,lehnin)
...
```

Structure Copying

Für jeden Aufruf einer Klausel wird eine Kopie mit Variablenbindungen angelegt



Structure Sharing vs. Structure Copying

Structure Sharing

Für die Aufruf einer Klausel werden Referenzen auf die Klauselstruktur im Programm angelegt.

```
erreichbar(X,Y)
      :-nachbar(X,Z),erreichbar(Z,Y).
erreichbar(X,X).
nachbar(berlin,potsdam).
nachbar(berlin,adlershof).

nachbar(potsdam,werder).
nachbar(potsdam,lehnin)
...
```

Alle Aufrufe der Klausel
- benutzen die Strukturen des Programms.
- besitzen spezielles Segment für Variablenbindungen.

Environment

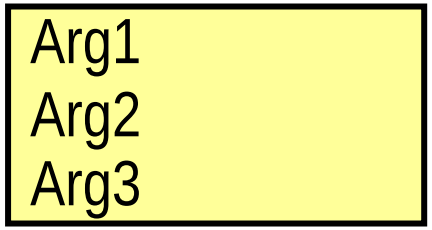
für Argumente einer Klausel werden Speicherbereiche angelegt.

Bindung erfolgt bei Unifikation durch Verweise

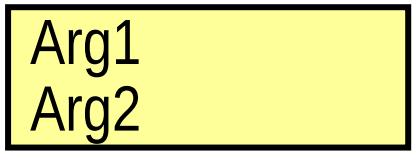
an Argumente von (in der Regel) älteren Klauseln

bzw. an Konstante.

```
erreichbar(X,Y)
    :-nachbar(X,Z),erreichbar(Z,Y).
```

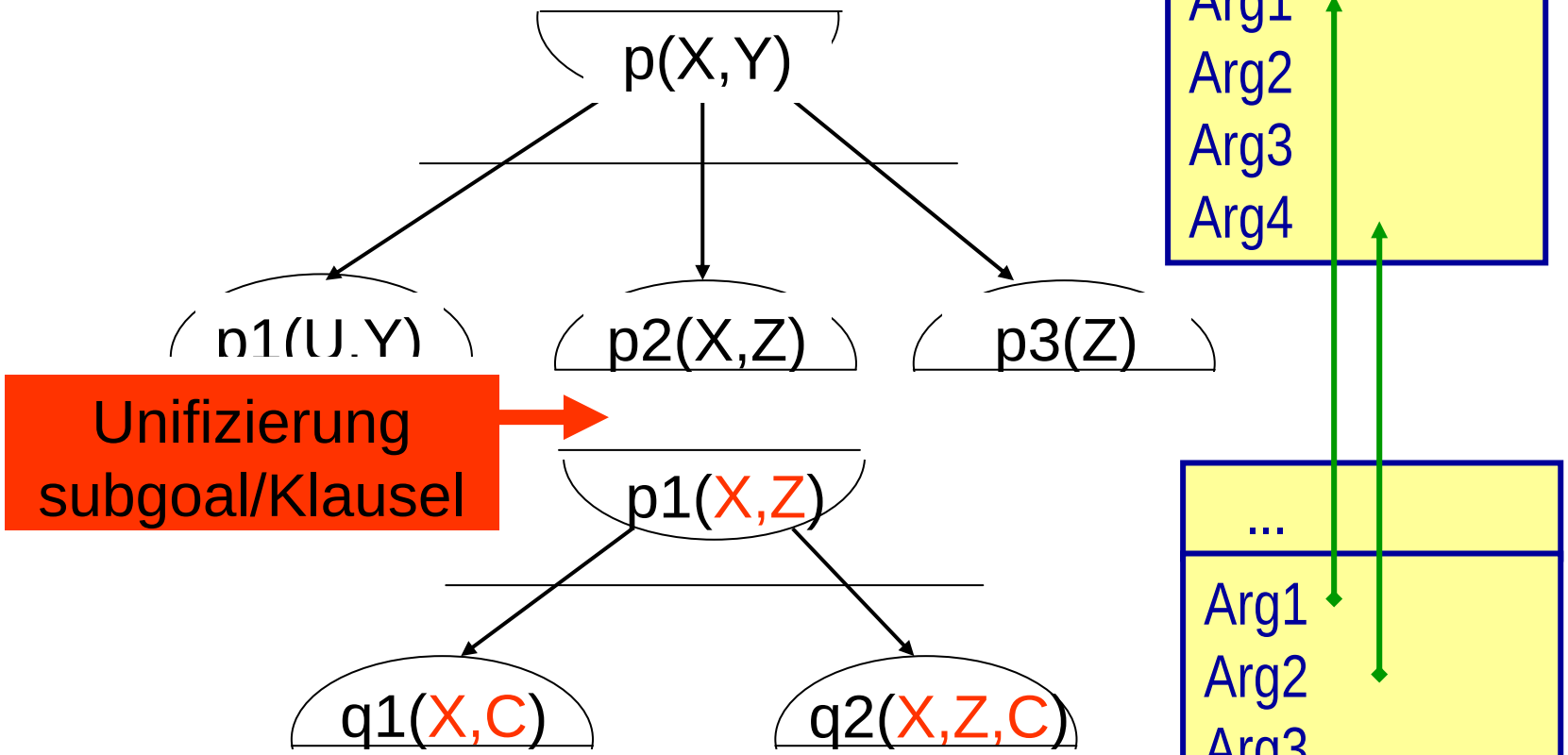



```
erreichbar(X,Y).
```



Bindungen

p(X,Y) :- p1(U,Y), p2(X,Z), p3(Z).



p2(A,B):-q1(A,C),q2(A,B,C).

Environment

Bindungen führen in ältere Teile des Kellers

Unifikation durch Ausführen des „matches“ zwischen Aufruf einer Klausel (als subgoal) und Kopf einer Klausel

- Dereferenzieren eines Arguments Arg_i entlang der Bindungen führt zu Deref(Arg_i)

  (kann Variable, Atom oder Struktur sein)
- Unifikation entsprechend Unifikationsregeln für die Dereferenzierten Argumente

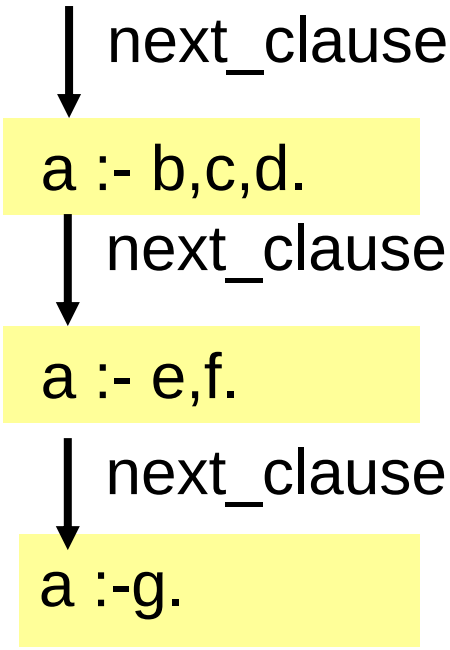
Beim Backtracking entfallen viele Bindungen durch streichen der Segmente

Bindungen, die nicht dadurch entfallen werden im trail protokolliert und beim Backtracking explizit aufgelöst.

Organisation der Prozeduren

Prozedur a

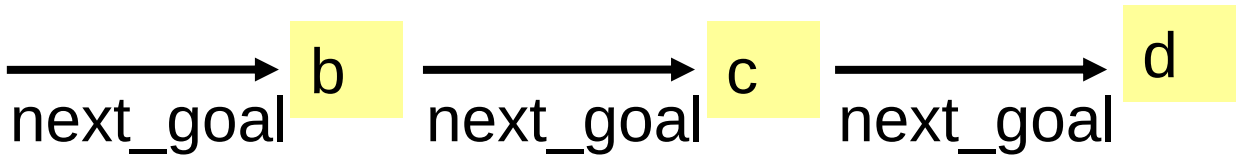
```
a :- b,c,d.
a :- e,f.
a :- g.
```



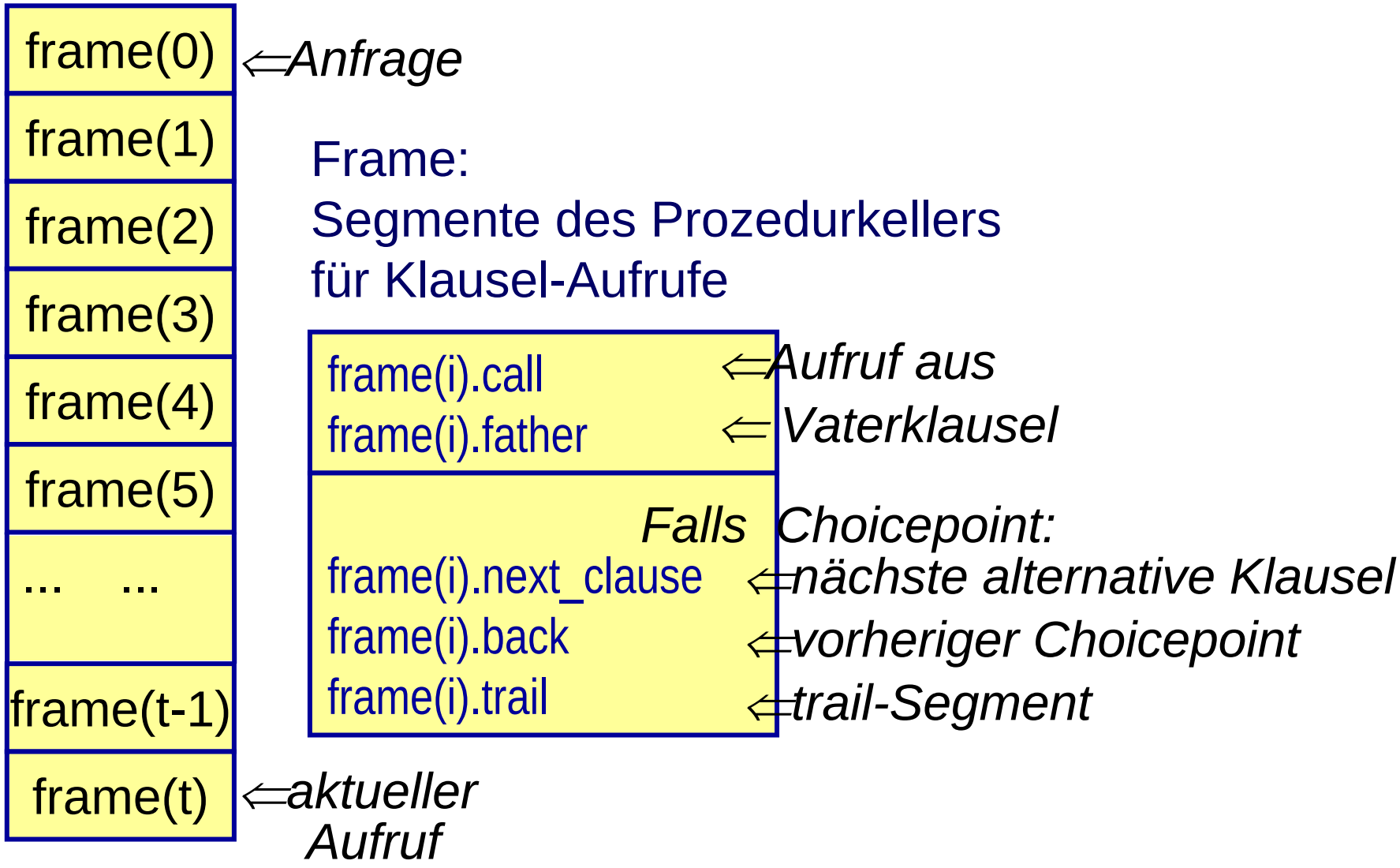
Prozedur als
verkettete Liste der Klauseln

Klausel als Liste von goals

a :- b,c,d.



Frame: Prozedurkeller (local stack)



Frame

Aufbau bei Klauselaufruf

während Unifikation („matching“)

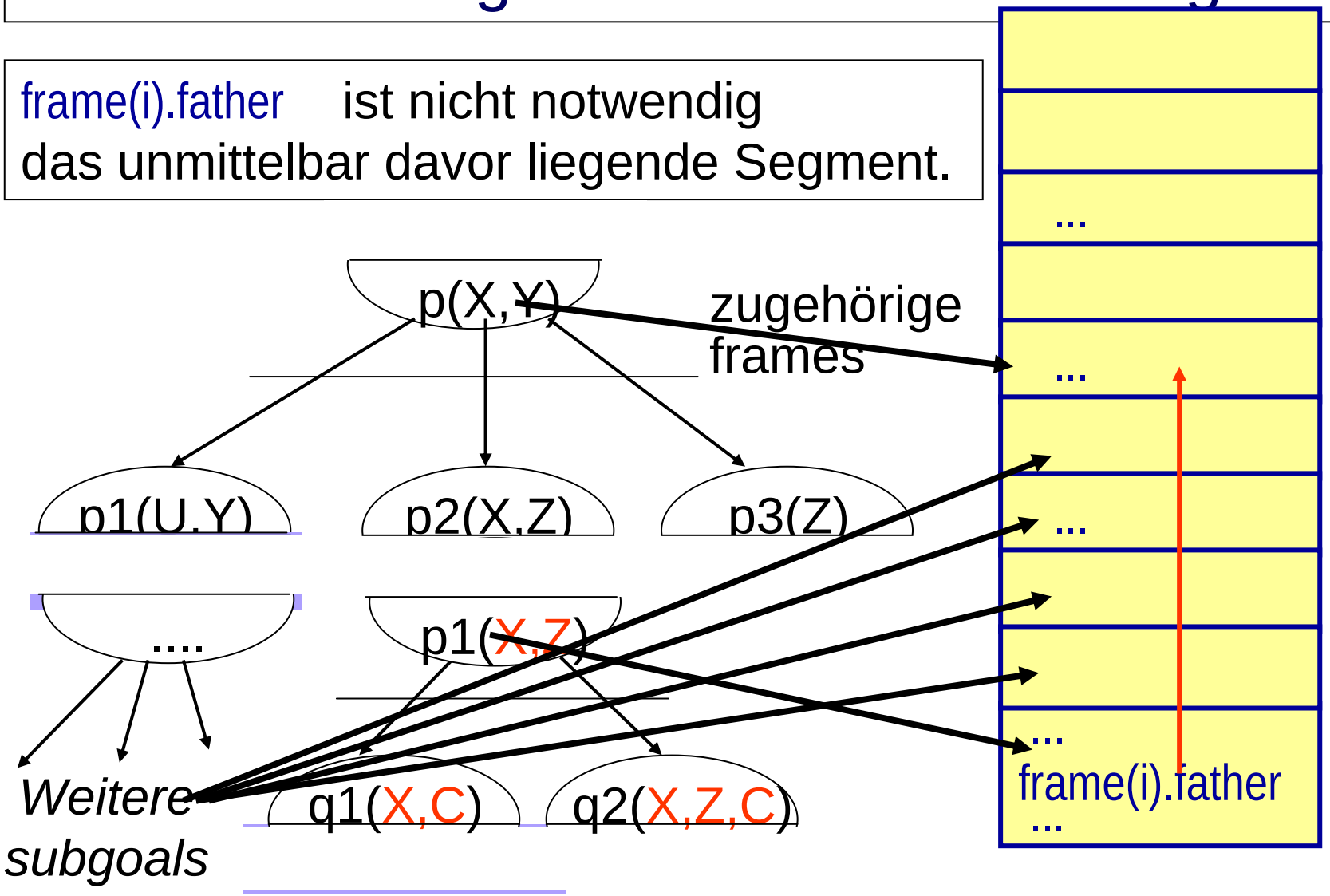
Streichen beim Backtracking

(alle Segmente oberhalb des jüngsten Choice point werden gestrichen)

frame(t).call
frame(t).father
---
frame(t).next_clause
frame(t).back
frame(t).trail

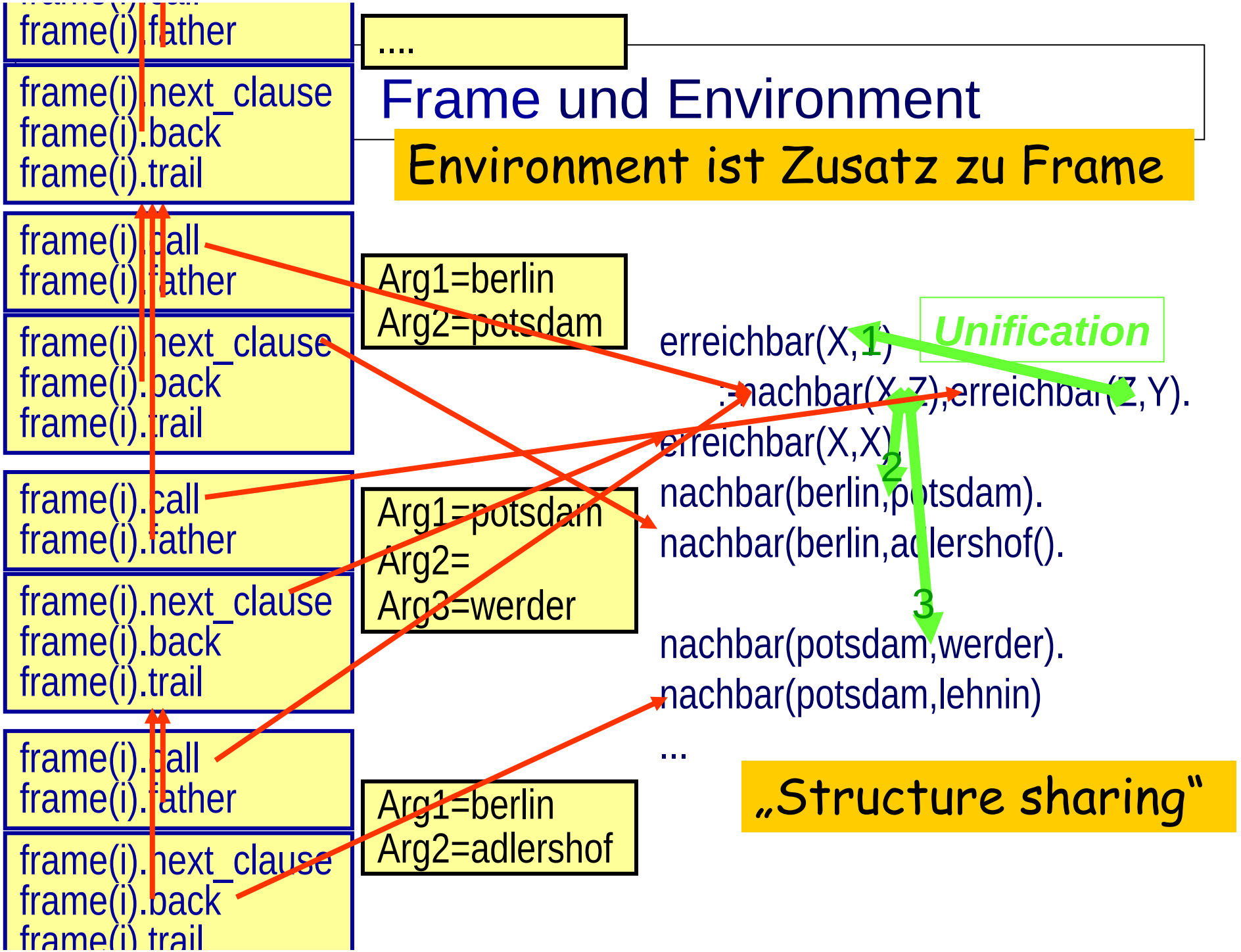
Zusammenhang von Vaterklausel und goal

frame(i).father ist nicht notwendig
das unmittelbar davor liegende Segment.

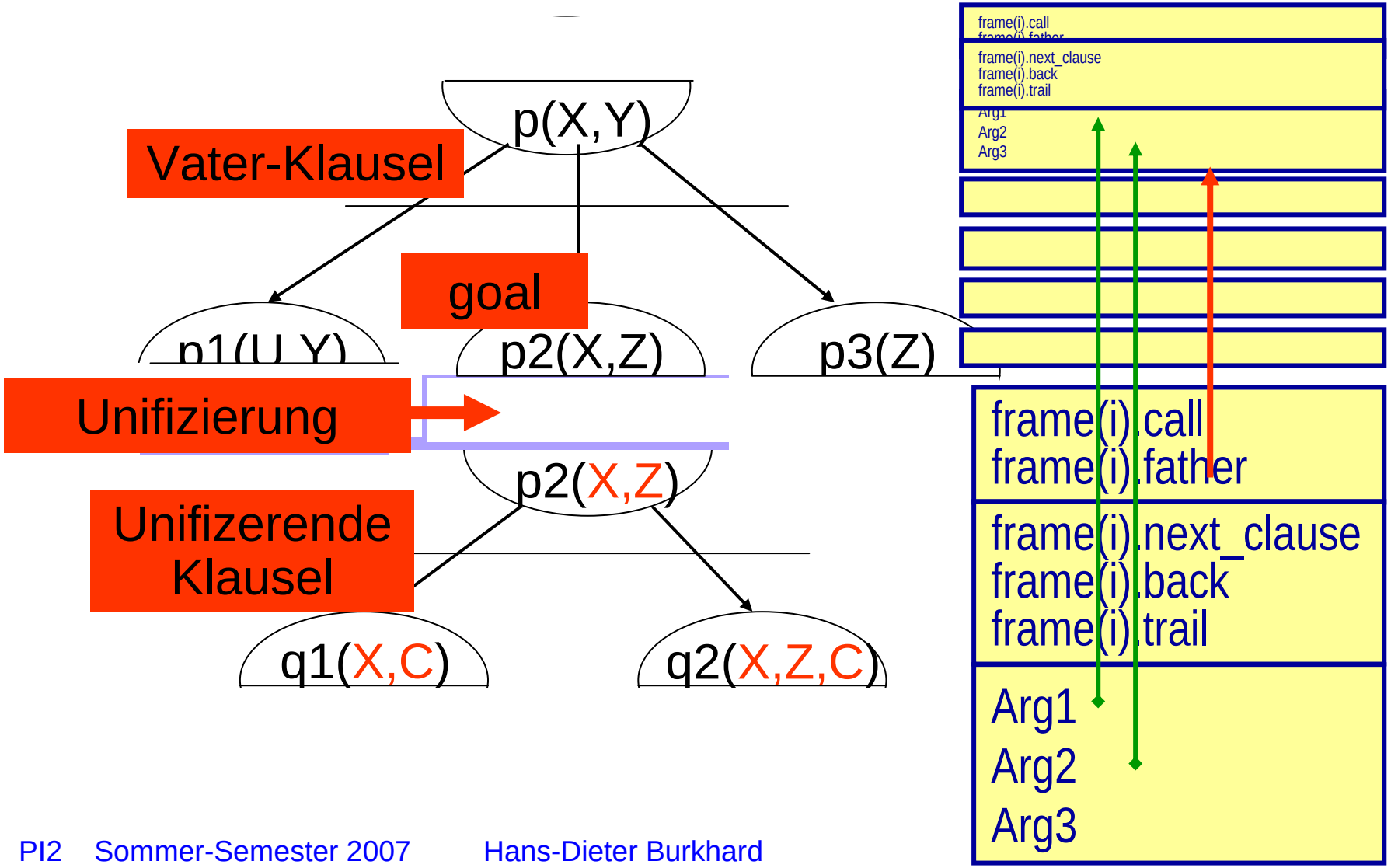


Frame und Environment

Environment ist Zusatz zu Frame

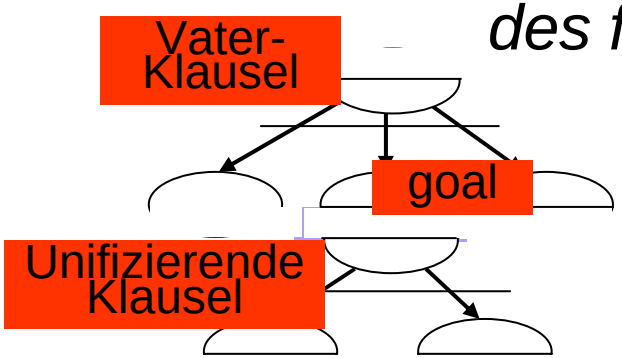


Steuerfluss durch Aufruf eines „goal“ initiiert

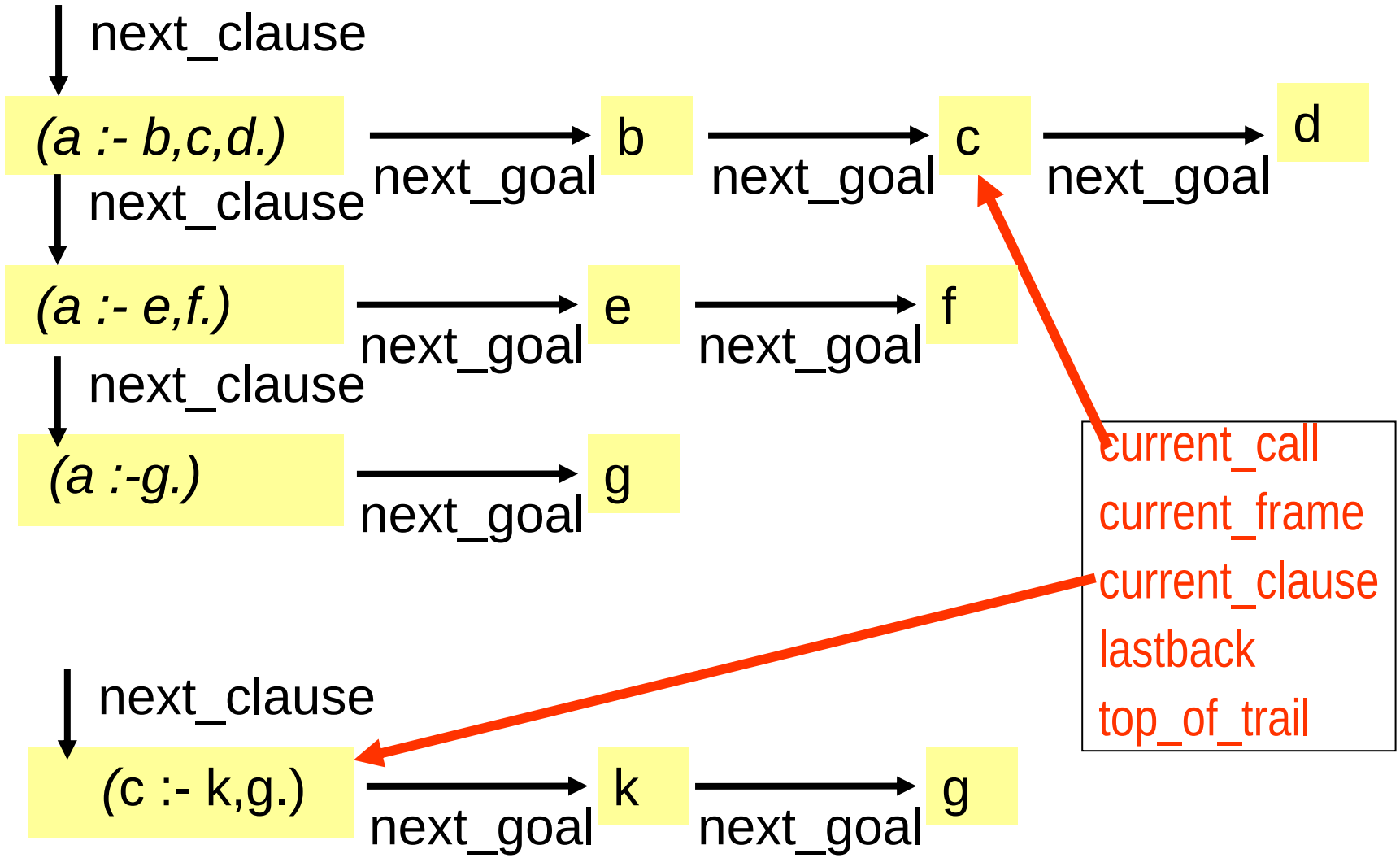


Zustand während Abarbeitung

current_call	⇐*aktuelles (aufrufendes) goal*
current_frame	⇐*frame der Vaterklausel von goal*
current_clause	⇐*mit goal unifizierende Klausel*
lastback	⇐*frame des jüngsten choicepoint*
top_of_trail	⇐*top des trail-stacks (protokolliert Bindungen, die nicht beim Backtracking durch Kontraktion des frame-stacks gelöst werden können)*



Referenzen auf Programm zur Laufzeit



0.Schritt (Aufruf eines goals)

Zustand:
current_call
current_frame
current_clause
lastback
top_of_trail

Aufrufendes goal referenziert durch current_call

Kandidat zur Unifizierung dieses goals ist erste Klausel der Prozedur bzgl. goal-funktor:

current_clause := Referenz auf diese Klausel

current_call
current_frame
current_clause
lastback
top_of_trail

1.Schritt: Frame anlegen

current_call
current_frame
current_clause
lastback
top_of_trail

Frame für Klausel anlegen

frame(t+1).call := current_call

frame(t+1).father := current_frame

current_frame := Referenz auf Frame für Klausel

current_call
current_frame
current_clause
lastback
top_of_trail

1.Schritt: Erweiterung bei Choice-point

Falls (alternative) Klausel existiert
d.h. von aktueller Klausel ausgehende Referenz next_clause ≠ NIL :
Referenzen für Choice-Point anlegen

current_call
current_frame
current_clause
lastback
top_of_trail

frame(t+1).back:=last_back

frame(t+1).next_clause:=next_clause

frame(t+1).trail:=top_of_trail

last_back := current_frame
(Referenz auf Frame für aktuelle Klausel)

current_call
current_frame
current_clause
lastback
top_of_trail

2. Schitt: Unifikationsversuch

Match des aufrufenden goal (current_call)
mit aktueller Klausel (current_clause)

Bindungen von Variablen erfolgen in

frame(t+1).father (environment im Frame für Vaterklausel des aufrufenden goals)

current_frame (dazu neu angelegtes environment im Frame für aktuelle Klausel)

Soweit Bindungen nicht „rückwärts“ angelegt werden können:
In trail protokollieren, top_of_trail weitersetzen

Bei komplexen Argumenten müssen auch die entsprechenden Strukturen angelegt werden.

current_call
current_frame
current_clause
lastback
top_of_trail

Schritt 3a: Falls Unifikation erfolgreich

Nächstes goal bestimmen (für Bearbeitung in frame(t+2))

current_call :=first_call in body of current_clause

(evtl. NIL falls Fakt)

falls current_call ≠ NIL

weiter in Schritt 0 (Anlegen von frame(t+2)

current_call
current_frame
current_clause
lastback
top_of_trail

Schritt 3a (erfolgreiche Unifikation, Fortsetzung)

falls current_call = NIL (d.h. current_clause war ein Fakt):
Nächstes goal ergibt sich aus offenen subgoals in früheren Klauseln

WHILE current_call = NIL DO

IF current_frame = „top_of_frame“ THEN weiter Schritt 4a:ERFOLG

ELSE current_call := next_goal in current_frame.call („rechter Bruder“)

current_frame:= current_frame.father

Fakt bzw. später: Ende der Klausel

current_call
current_frame
current_clause
lastback
top_of_trail

Schritt 3b: Unifikation nicht erfolgreich

Backtracking:

Alternative Klauseln im jüngsten choicepoint anwenden

Falls lastback=NIL : Weiter bei Schritt 4b (FAILURE)

Falls lastback ≠ NIL :

current_call := lastback.call

current_frame = lastback.father

current_clause:= lastback.clause

lastback:=lastback.back

Zurücksetzen des Prodezurkellers:

- Stellt frühere Aufrufsituation her.
- Löscht Bindungen in jüngeren environments.

Bindungen gemäß trail lösen und trail zurücksetzen bis lastback.top_of_trail

top_of_trail := lastback.top_of_trail

current_call
current_frame
current_clause
lastback
top_of_trail

Schritt 4a: ERFOLG

current_frame = „top_of_frame“
(Segment des Aufrufs)

d.h. es gibt keine weiteren unerfüllten subgoals.

Ausgabe:

Bindungen der Variablen in „top_of_frame“

bzw. „yes“, falls Anfrage ohne Variable

Prozedurkeller enthält i.a. weitere frames (mit choice points) für offene alternative Beweisversuche.
Mit Eingabe „ ; “ werden diese aktiviert: weiter bei Schritt 3b

Schritt 4b: MISSERFOLG

lastback=NIL

Es gibt keine alternativen Beweismöglichkeiten für die noch offenen subgoals:

Der Beweisversuch ist fehlgeschlagen.

Ausgabe:

„no“

Implementierung des Cut

`!/0` gelingt stets und löscht Choice-Points für

- –aktuelle Klausel
- –subgoals im Klauselkörper, die vor dem Cut stehen
- –subgoals dieser subgoals usw.

Gefundene Lösung wird ,,eingefroren“
Alternativen für Backtracking entfallen

current_call
current_frame
current_clause
lastback
top_of_trail

⇐*frame des jüngsten choicepoint*

Muss korrigiert werden

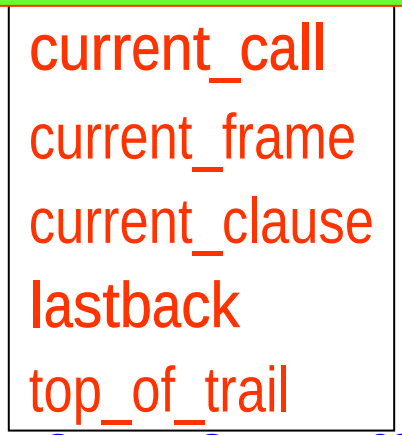
Implementierung des Cut

Reduktion der Abarbeitungsschritte 0-3:

- Kein frame für goal „cut“ anlegen
- Keine Unifizierung
- Falls choicepoint bei Vaterklausel:

  lastback:=current_frame.last_back

  Sonst: lastback:= frame.last_back für jüngsten davorliegenden frame mit Choicepoint
- current_call weitersetzen
- weiter bei Schritt 3a



⇐*frame des jüngsten choicepoint*

Interpreter setzt folgende Strategien um

SLD-Resolution

Structure sharing

Backtrack-Konzept für Tiefe-Zuerst-Suche

Optimierung des Prozedurkellers mittels

- Deterministic call optimization
- Last call optimization

Optimierung des Laufzeitkellers

Bei Beendigung deterministischer Aufrufe

(DCO = deterministic call optimization)

Bei Aufruf des letztes Goals einer Klausel

(LCO = last call optimization)

- Speziell für Rekursion an letzter Stelle

  erreichbar(X,Y):-nachbar(X,Z),erreichbar(Z,Y).

- Voraussetzung: deterministische Aufrufe

Optimierung deterministischer Aufrufe

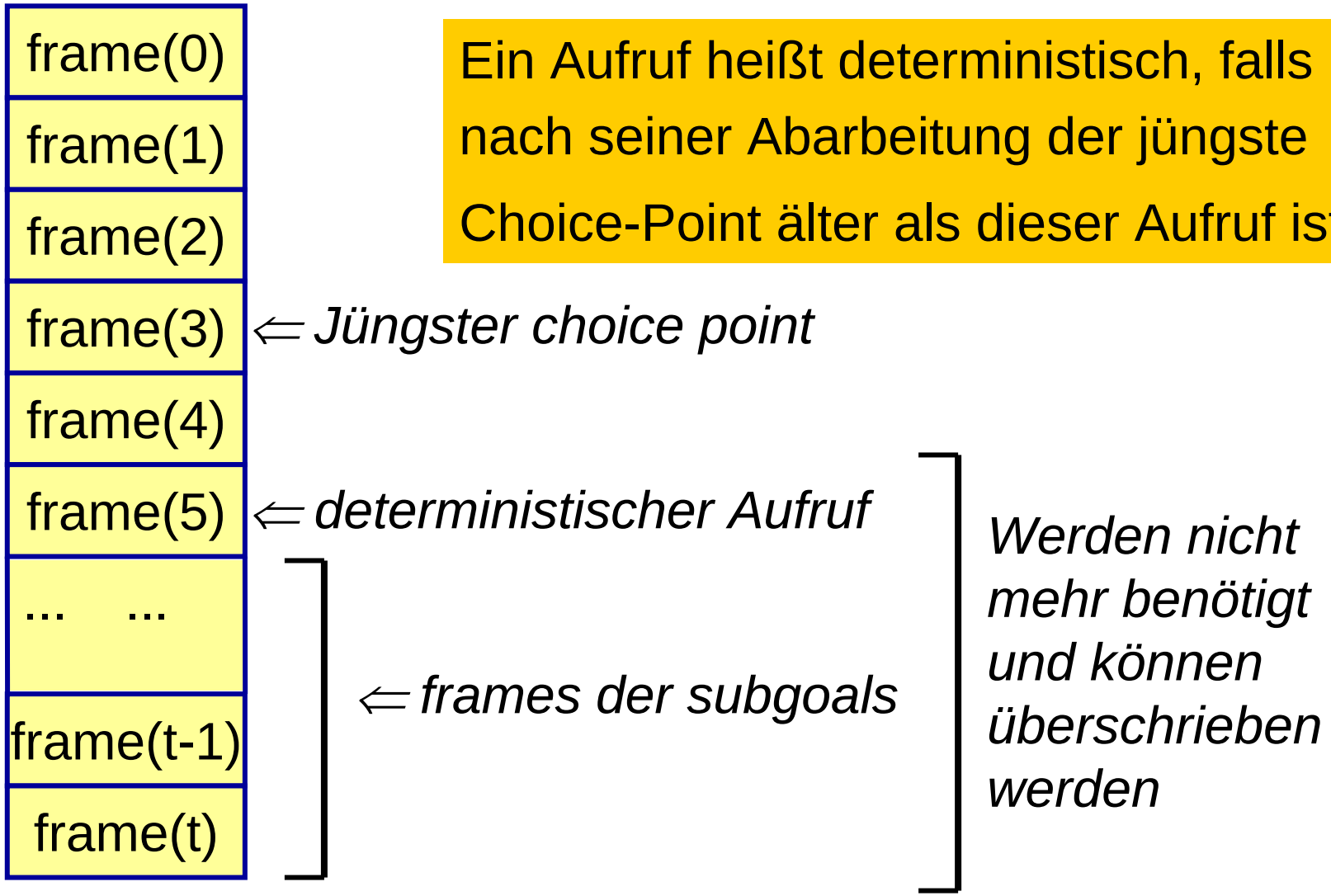
Idee: Frames einsparen, falls nicht mehr benötigt

Prolog-Laufzeitkeller enthält frames für alle Klauseln im aktuellen Beweisbaum für

- Variablenbindungen zwischen Subgoal und Vaterklausel
  - Einsparung möglich, wenn Variablenbindungen an environments älterer Klauseln (am Ende der Derefenzierungskette) erfolgen

- Information zum Backtracking (alternative Klauseln)
  - Einsparung möglich, wenn kein Backtracking mehr erfolgt: „Deterministische Aufruf"

Deterministischer Aufruf



Ein Aufruf heißt deterministisch, falls nach seiner Abarbeitung der jüngste Choice-Point älter als dieser Aufruf ist.

Ergänzung von Schritt 3a für DCO

WHILE current_call = NIL DO

IF current_frame = „top_of_frame“ THEN weiter Schritt 4a:ERFOLG

ELSE current_call := next_goal in current_frame.call („rechter Bruder“)

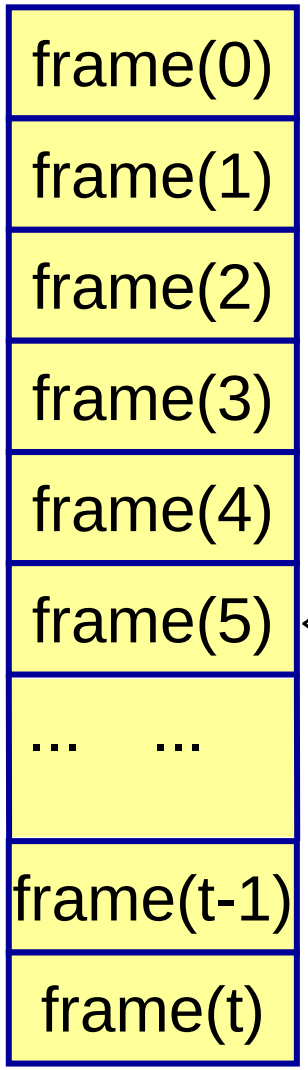
IF lastback älter als current_frame

THEN frame-Keller freigeben

bis einschließlich current_frame

current_frame:=current_frame.father

Deterministischer Aufruf



Ein Aufruf heißt deterministisch, falls nach seiner Abarbeitung der jüngste Choice-Point älter als dieser Aufruf ist.

⇐ *Jüngster choice point*

⇐*Deterministischer Aufruf*

⇐ *frames der subgoals*

Müssen ebenfalls deterministisch sein

(DCO in tieferen Schichten bereits erfolgt)

Unterstützung von DCO

- Cut löscht choice-points und macht dadurch Aufrufe deterministisch
- Indexierung der Klauseln nach:
  - Funktor
  - erstes Argument

Interpreter kann das ausnutzen:
Alternativen (choicepoints)
nur bei Unifizierbarkeit
bzgl. erstem Argument

Programm kann DCO unterstützen:
Durch geschickten Einsatz von cut .
Durch geeignete Wahl des ersten Arguments.

Optimierung des letzten Subgoal-Aufrufs

Idee für LCO:

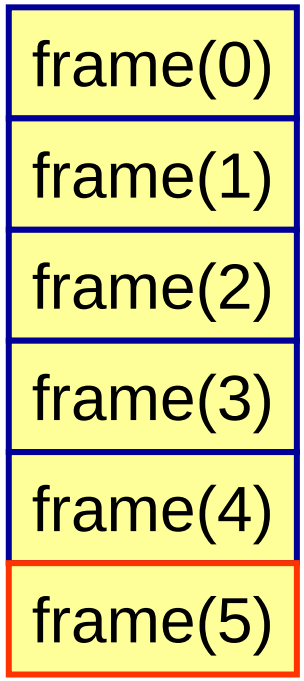
Nach Bildung von frame und environment des letzten Subgoals einer Vater-Klausel werden frame und environment der Vater-Klausel nicht mehr benötigt

Voraussetzungen:

- Geeignete Form der Variablenbindungen (wie DCO)
- Aufruf der Vater-Klausel ist deterministisch

  (jüngster choice point älter als Vaterklausel)

Implementierung von LCO



Gemeinsam mit DCO implementieren.

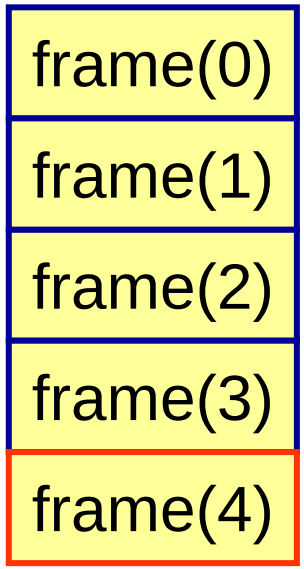
⇐ *Jüngster choice point*

⇐*Deterministischer Aufruf*

⇐*Aufruf des letzten subgoals*

⇐ *frames der weiteren subgoals bereits mit DCO gelöscht*

Implementierung von LCO



Gemeinsam mit DCO implementieren.

⇐ *Jüngster choice point*

⇐*Frame des letzten subgoals*
überschreibt frame der Vaterklausel

Tail-Optimierung

Reduzierung des Speicherbedarfs mittels DCO/LCO:

Rekursiver Aufruf als letztes Teilziel

```
erreichbar(X,Y) :- nachbar(X,Z), erreichbar(Z,Y).
```

Aufrufe deterministisch

- Alternativen ggf. davor

```
erreichbar(X,X).
erreichbar(X,Y) :- nachbar(X,Z), erreichbar(Z,Y).
```

- evtl. cut geeignet einsetzen

Fortsetzung Suchverfahren

Wdh.: Suche in Und-oder-Bäumen

Problemzerlegung

Und-oder-Baum

Lösungsbaum

Beschränkung auf Bäume!

Problemlösung durch Zerlegung

Zerlege ein Problem P in einzelne Probleme $P_1,...,P_n$

Löse jedes Problem P_i

(als einfaches Problem

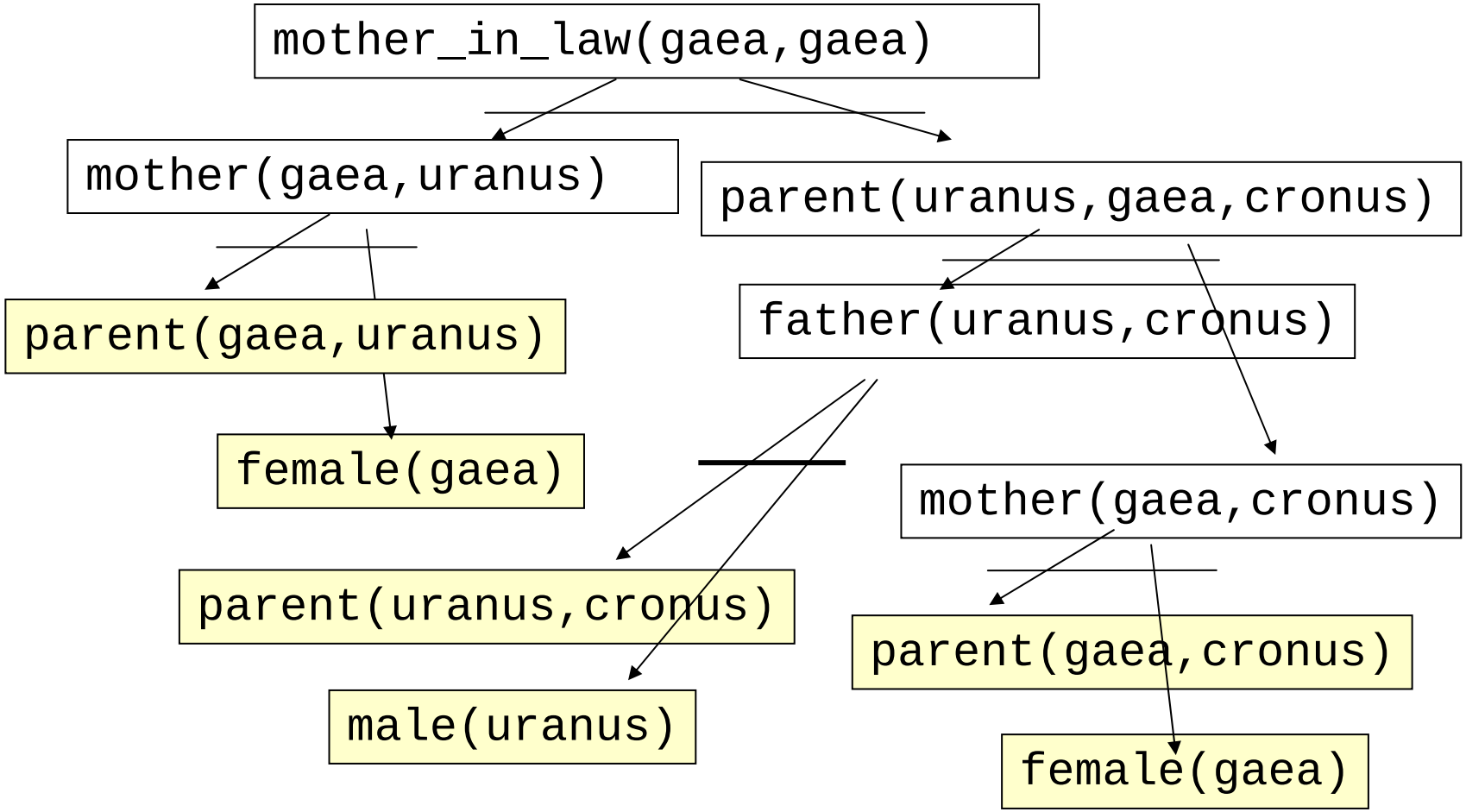
oder durch iterierte Problemzerlegung)

Füge die Lösungen zusammen zu P

Beispiele:

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

Beweisbaum (PROLOG)



Prolog: Klausel als Problemzerlegung

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

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

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

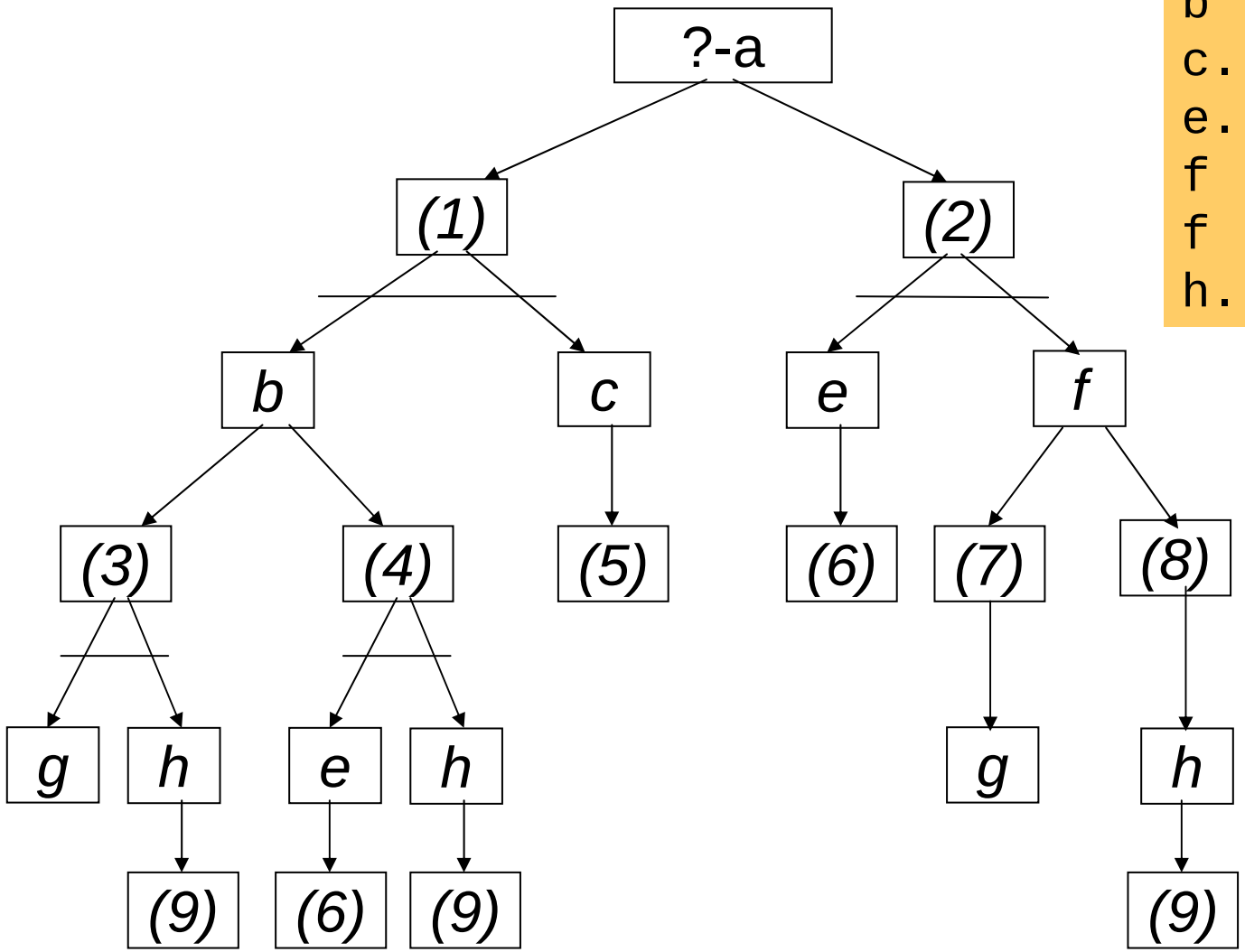
Subgoals einer Klausel

- 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)
```



Suche in Spielbäumen

Spielbäume

2-Personen-Nullsummen-Spiele

Minimax-Strategie

Pruning-Verfahren

Heuristische Verfahren

Spiele mit n Spielern P1,...,Pn

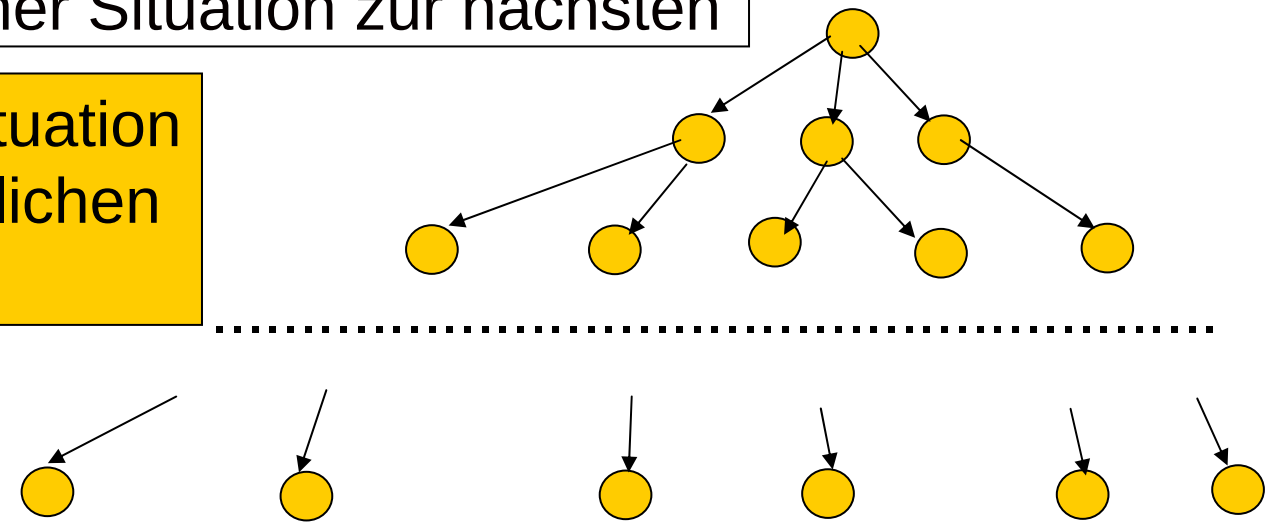
Darstellung des Spiels als Spielbaum:

– Knoten:
  mit Spielsituationen markiert

– Kanten:
  Züge von einer Situation zur nächsten

Graph schwierig zu verarbeiten

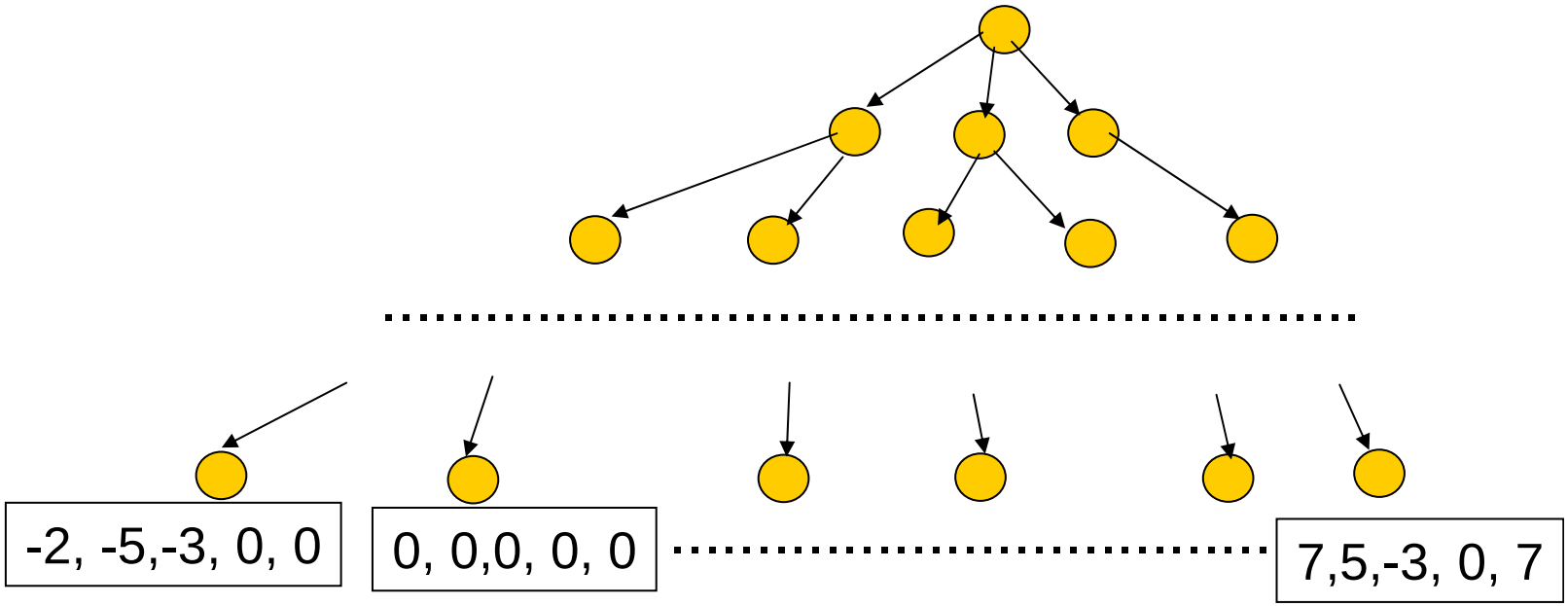
Ggf. gleiche Situation an unterschiedlichen Knoten

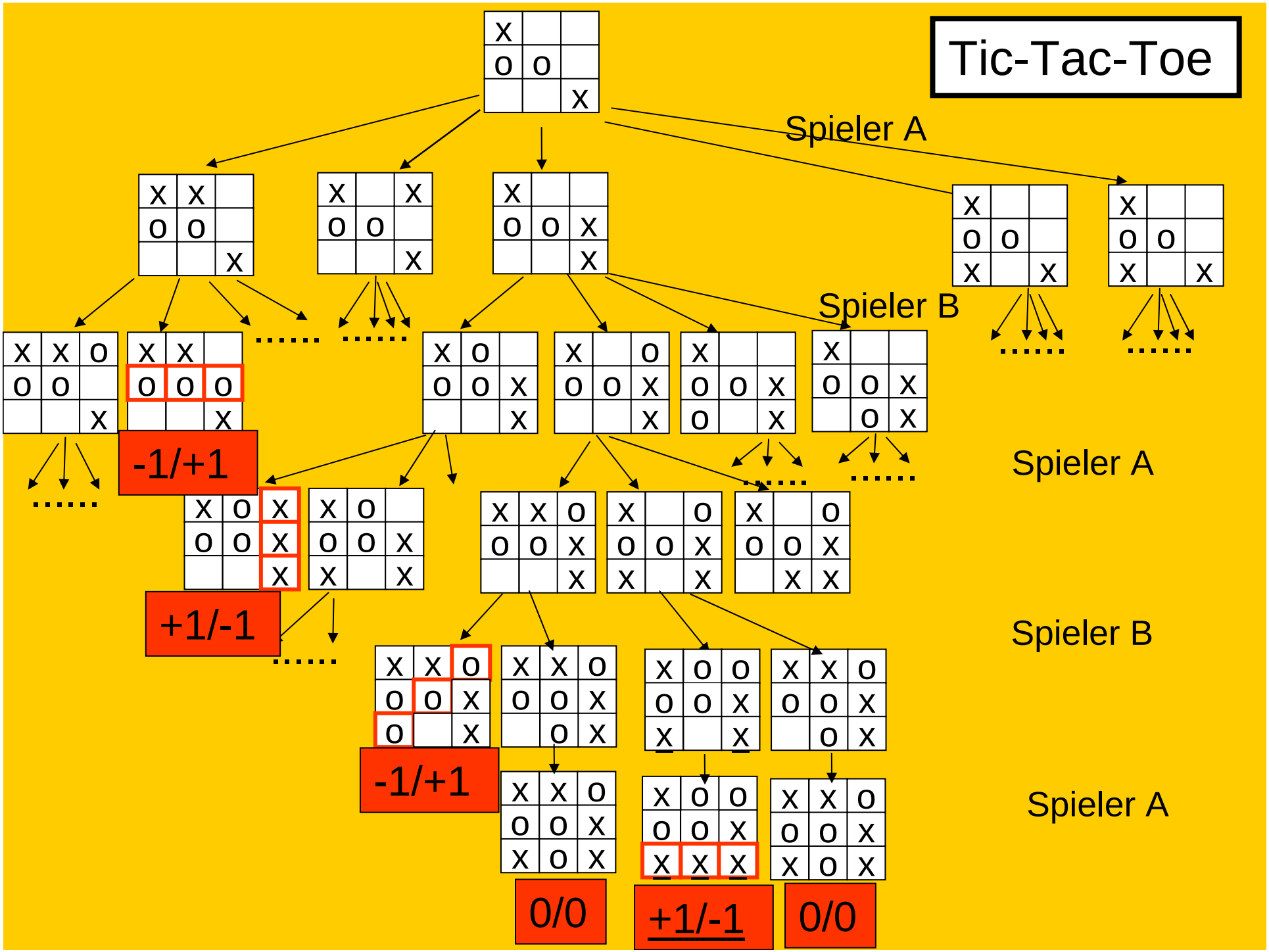


Spiele mit n Spielern P1,...,Pn

–Endknoten mit Bewertung für jeden Spieler markiert

- Positive Zahl: Gewinn
- Negative Zahl: Verlust




Tic-Tac-Toe
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A
-1/+1
+1/-1
-1/+1
0/0
+1/-1
0/0

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: $30^{10} \sim 6*10^{14}$ Nachfolgeknoten

...

10. gegnerischer Zug: $30^{20} \sim 3{,}5*10^{29}$ 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

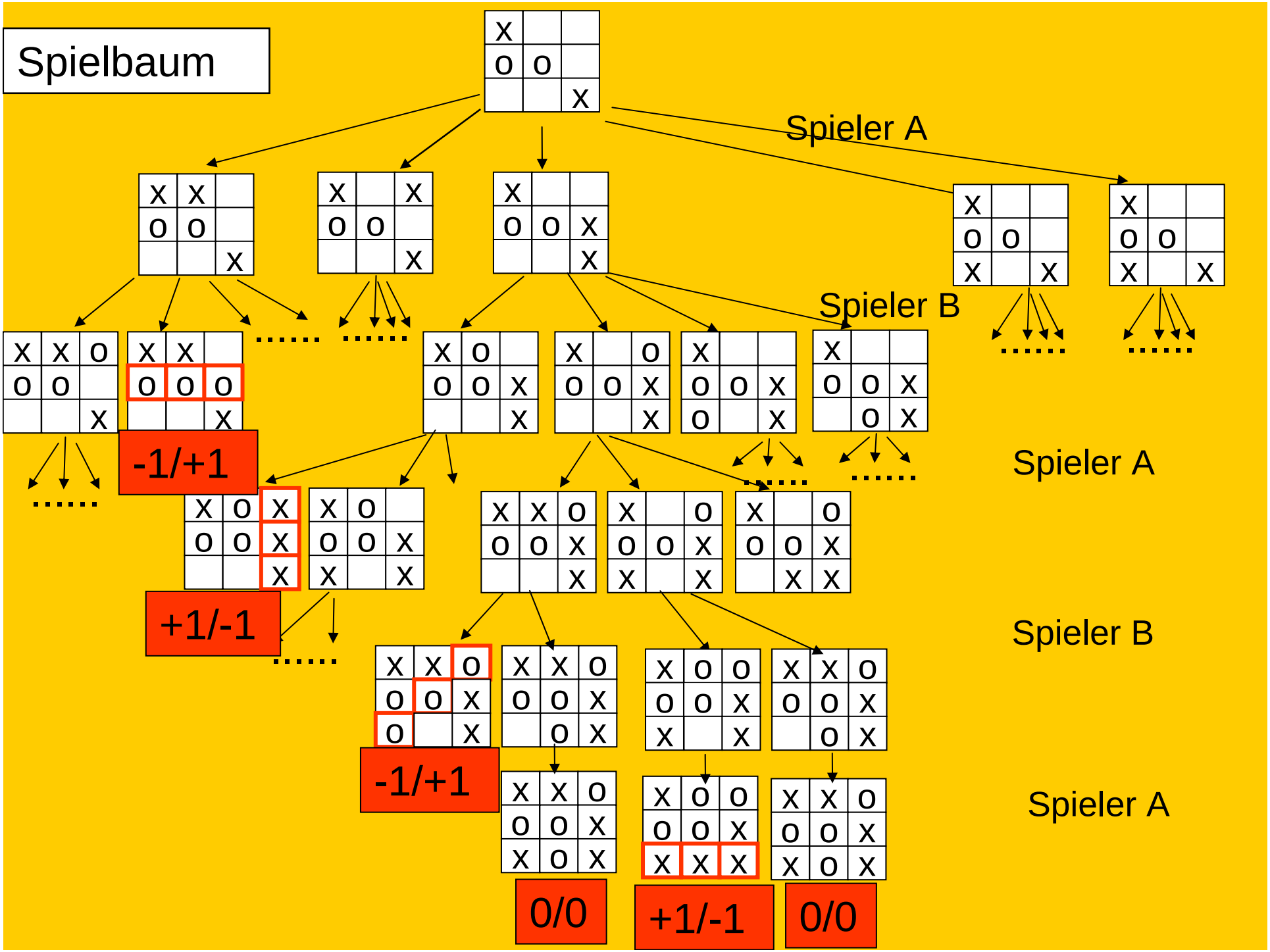
π_i : Spielstrategie (policy) des Spielers P_i

- Vorgabe eines Zuges für jede Situation

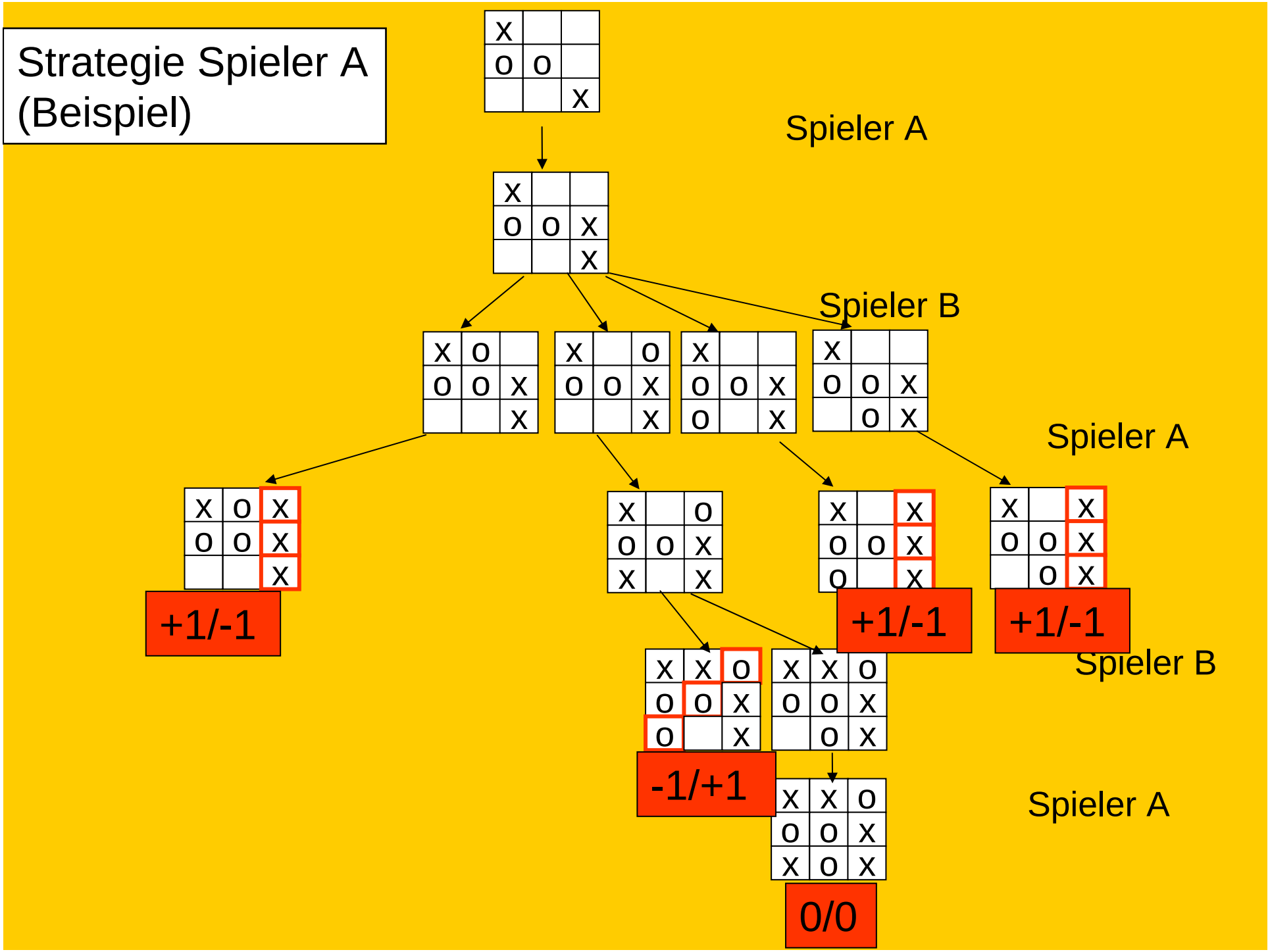
  π_i : Situationen $\rightarrow$ Spielzüge des Spielers P_i

- Entspricht einem Teilbaum des Spielbaums
  - 1 Nachfolger für den eigenen Zug gemäß Strategie
  - k Nachfolger für k mögliche Züge anderer Spieler

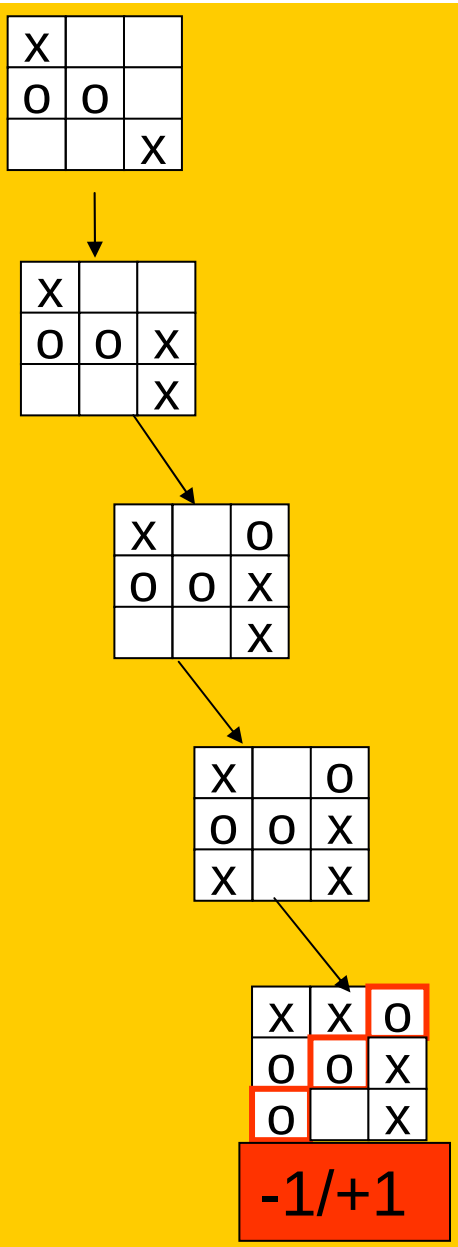
- Spielstrategien $\pi_1,...,\pi_n$ aller Spieler ergeben einen Weg im Baum zu einem Endzustand


Spielbaum
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A
-1/+1
+1/-1
-1/+1
0/0
+1/-1
0/0

Strategie Spieler A (Beispiel)



Zugfolge:
Strategie Spieler A
Strategie Spieler B
(Beispiel)



Gewinn/Verlust

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

Gewinn/Verlust des Spielers P_i

wenn jeweils Spieler P_k die Spielstrategie π_k verwendet

Bewertung für alle Spieler als Vektor:

$[G_1(\pi_1,...,\pi_n), G_2(\pi_1,...,\pi_n), ..., G_n(\pi_1,...,\pi_n)]$

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 $\pi_1,...,\pi_n$ gemäß erwarteten Gewinnen $[G_1(\pi_1,...,\pi_n), ..., G_n(\pi_1,...,\pi_n)]$

- und Konflikt (Gegnerschaft)

Probleme:
Welche Werte optimieren?

- Ziele:
  - (Individuellen/Globalen) Gewinn 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)

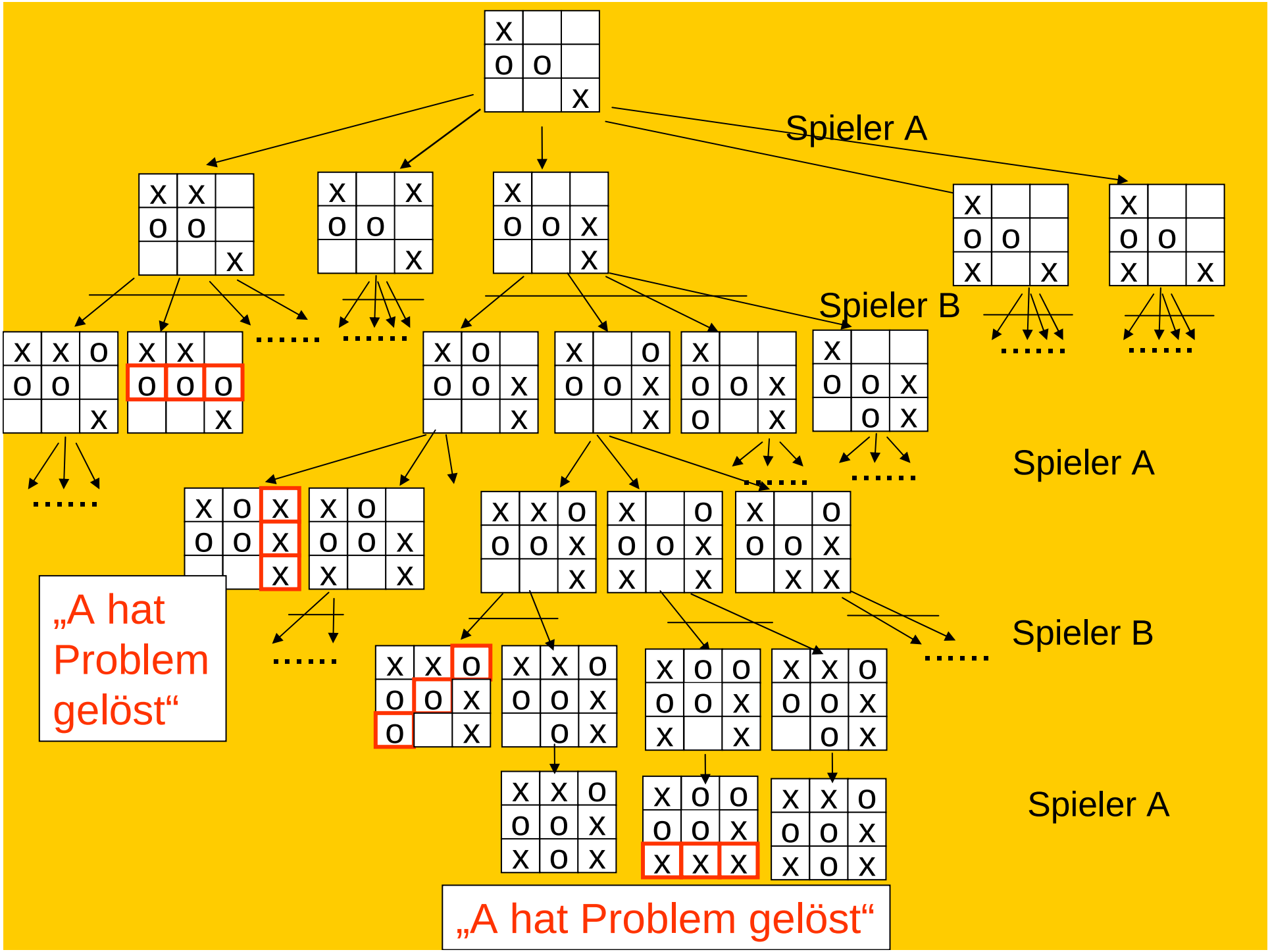
volle Analogie zu Problemzerlegung:

- 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.


Spieler A
Spieler B
Spieler A
Spieler B
Spieler A
„A hat Problem gelöst“
„A hat Problem gelöst“

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 *Maximierer* (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

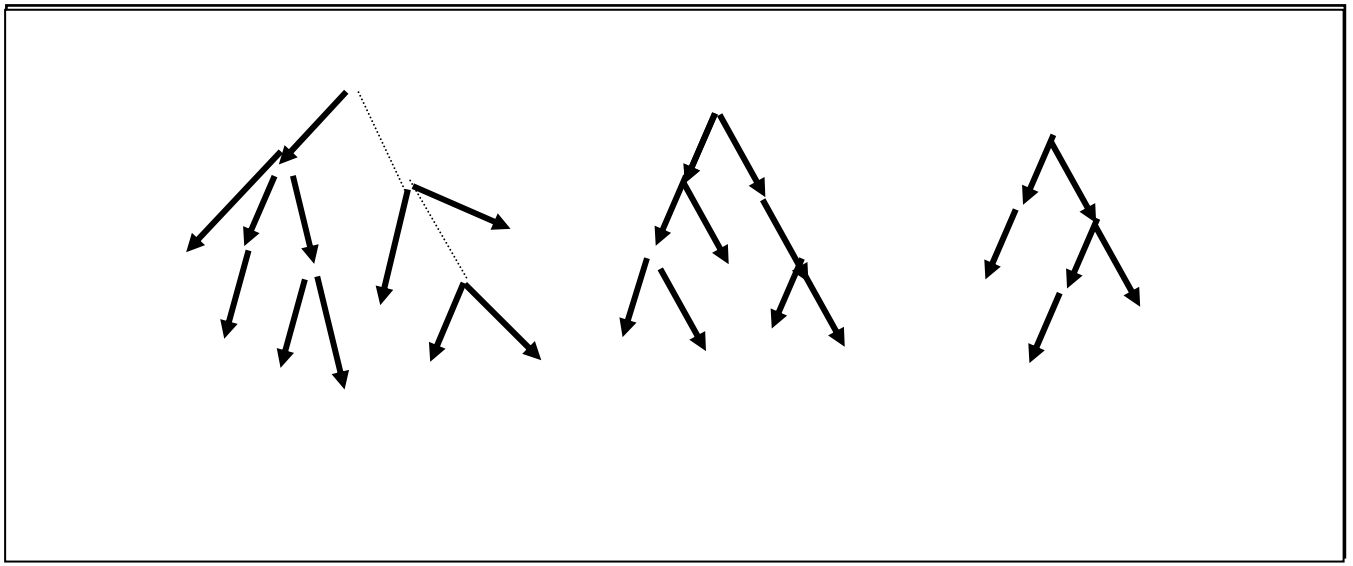
Baumspiel

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

$$G_A(s) \rightarrow \text{Gewinne}$$

$G_A(s)$:= 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:

$G_A(s) = \text{Max} \{G_A(s') \mid s' \text{ Nachfolgesituation von } s \}$

Wenn B in s zieht:

$G_A(s) = \text{Min} \{G_A(s') \mid s' \text{ Nachfolgesituation von } s\}$

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

Minimax-Verfahren

Lernen der Werte von Spielsituationen

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:

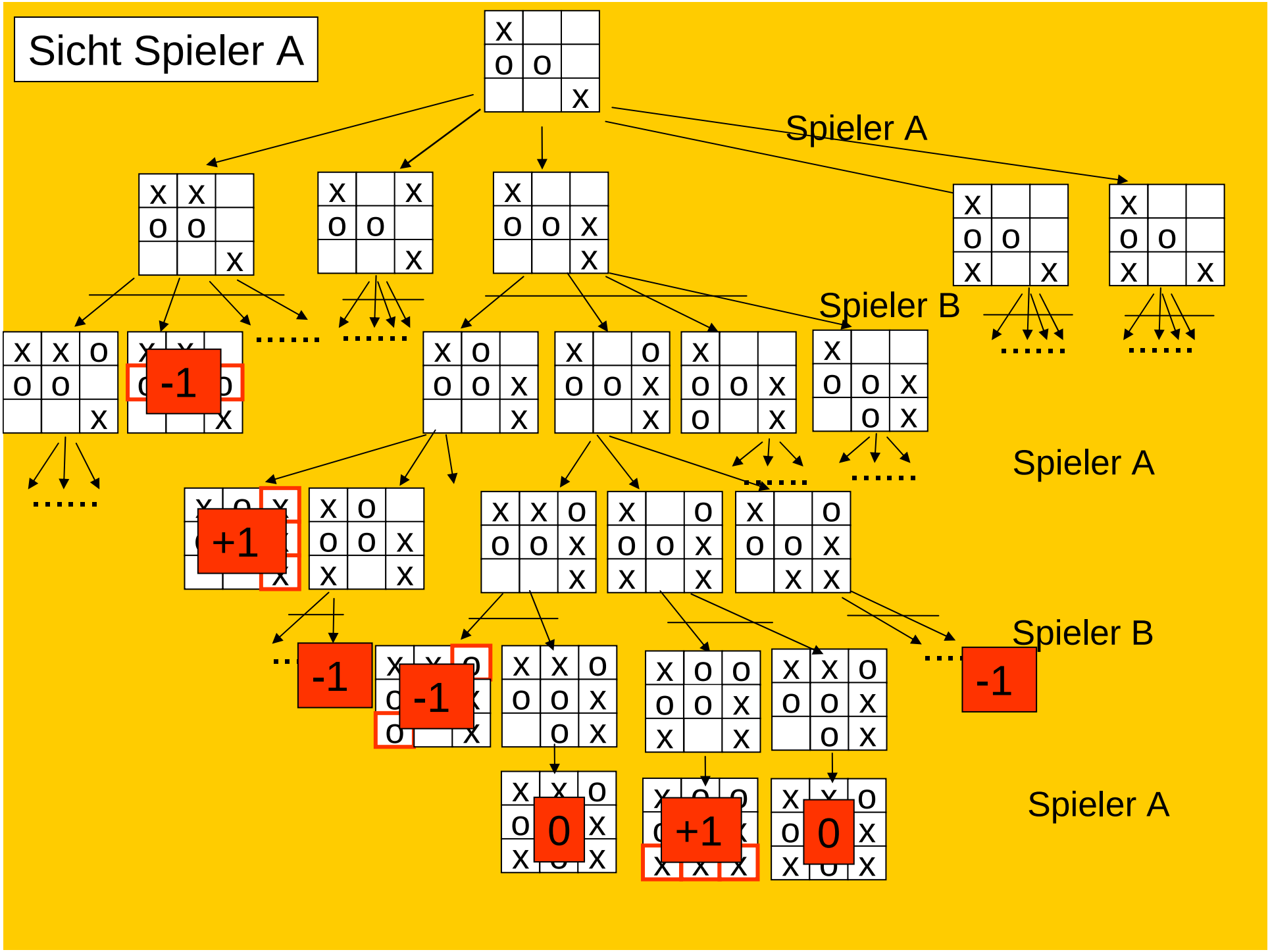
Exit: Wert der optimalen Strategie π_A^* = Wert(Startknoten)

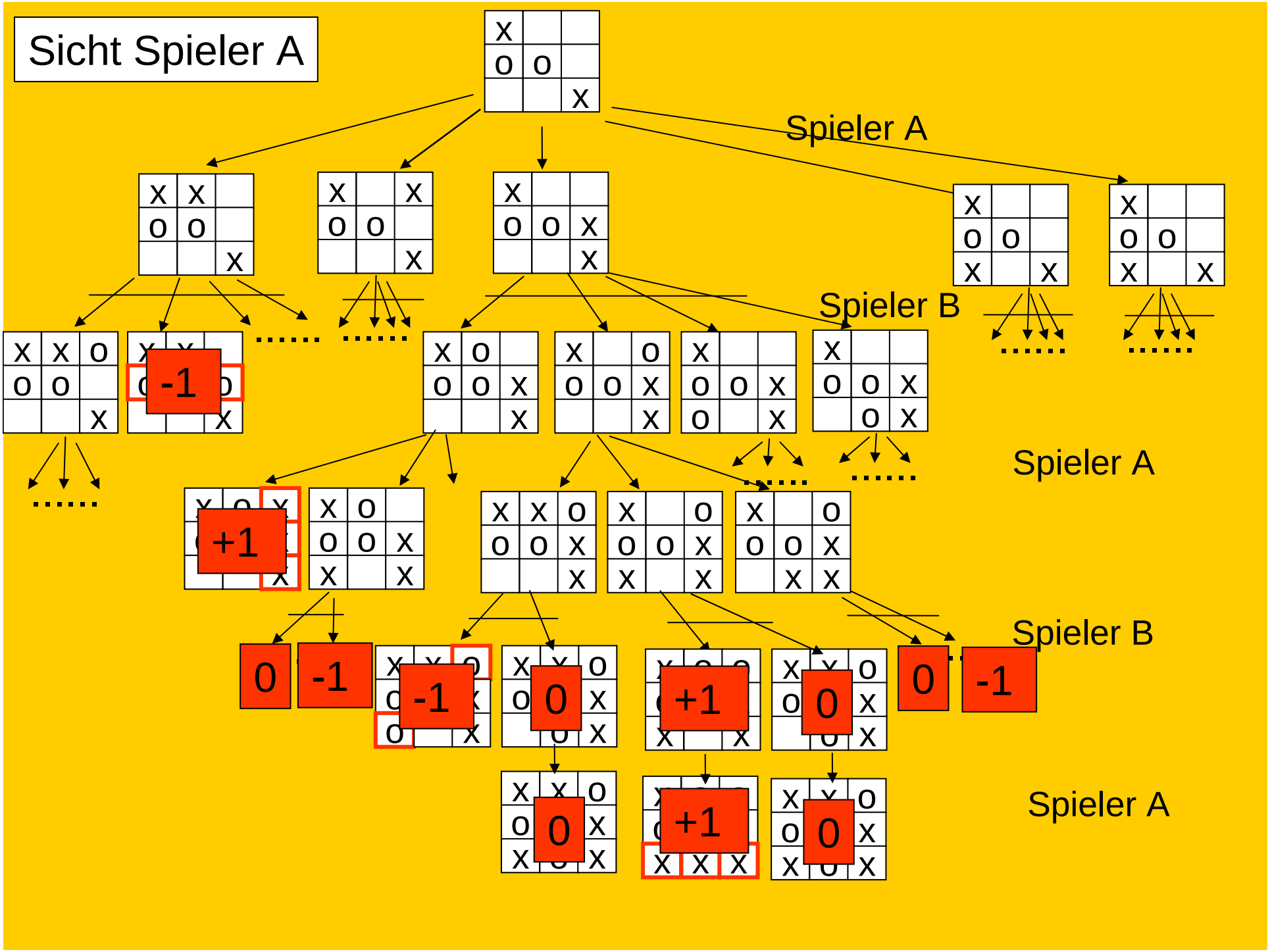
(2)Wähle unmarkierten Knoten k, dessen Nachfolger markiert sind

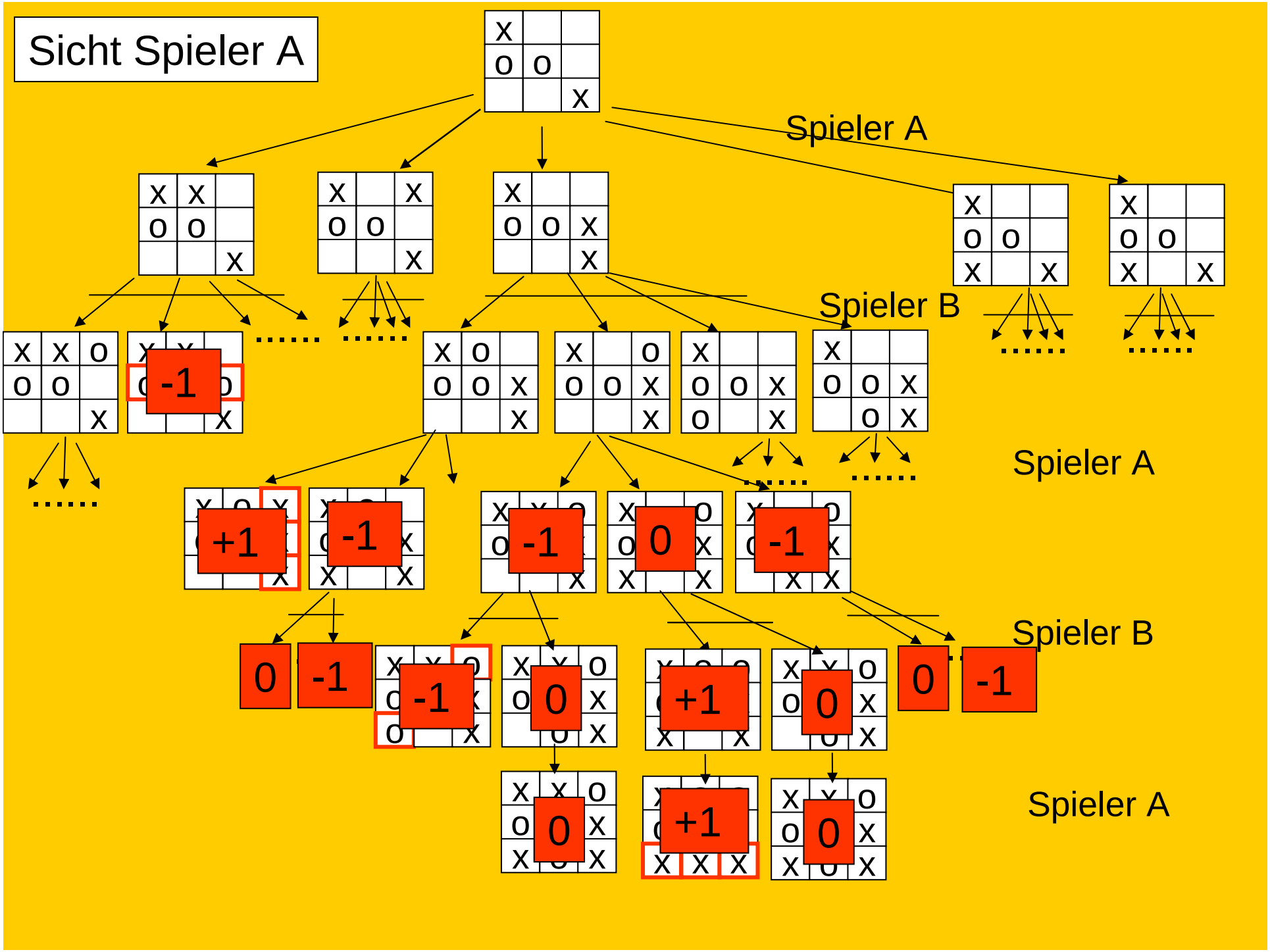
Wenn A in k zieht: Wert(k):=Max{Wert(k´) | k´ Nachfolger von k }

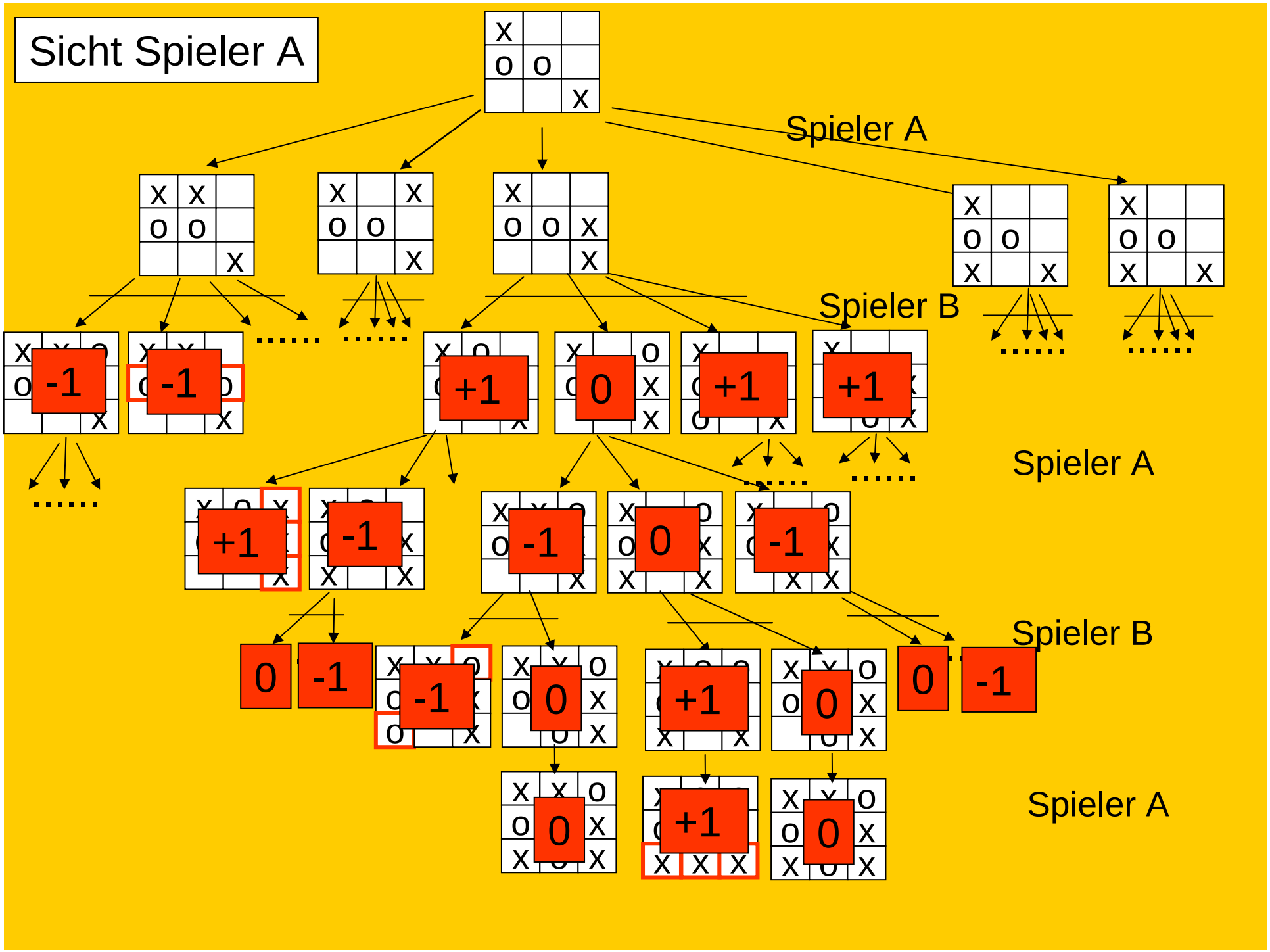
Wenn B in k zieht: Wert(k):=Min{Wert(k´) | k´ Nachfolger von k }

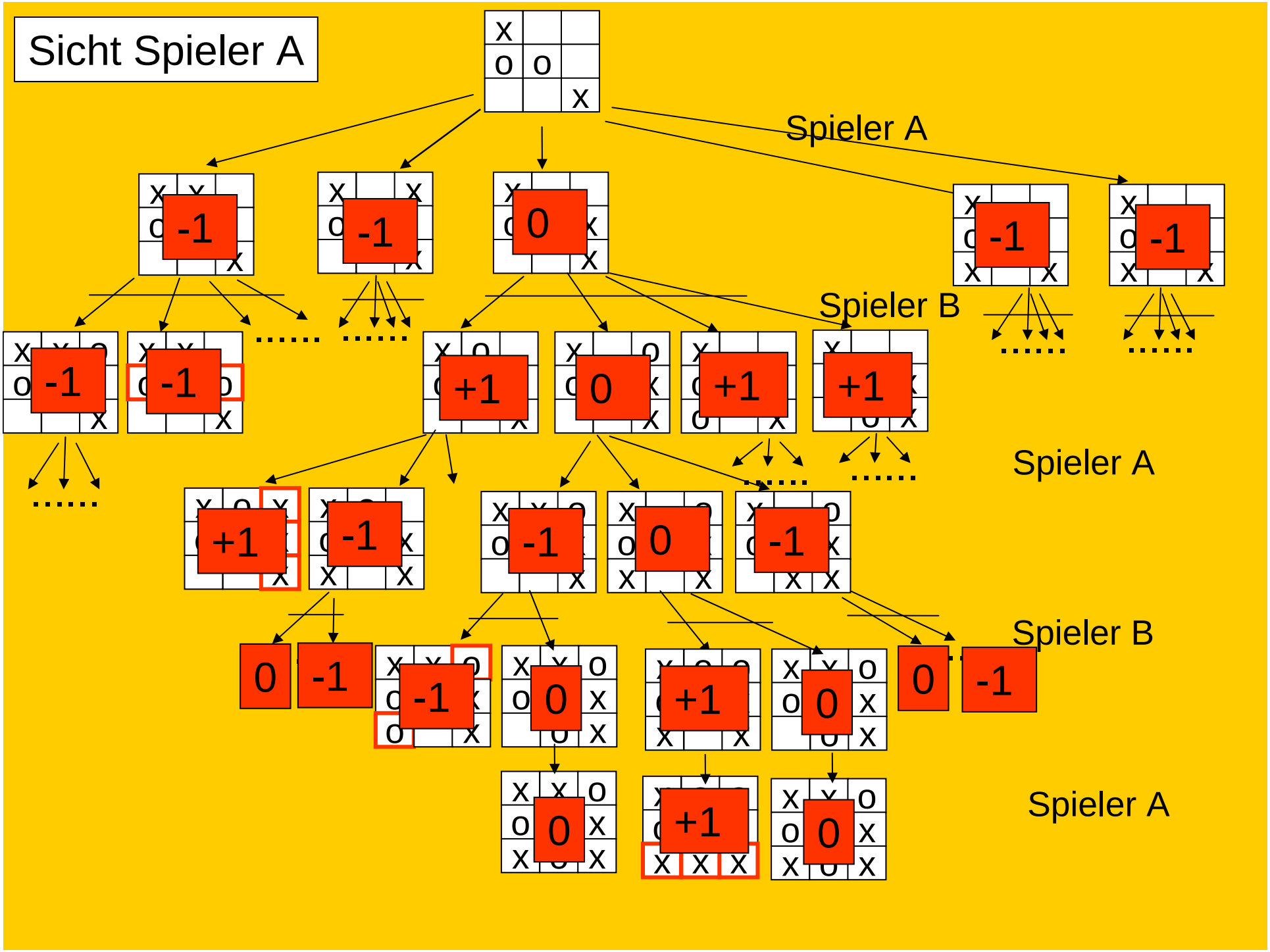
Weiter bei (1).

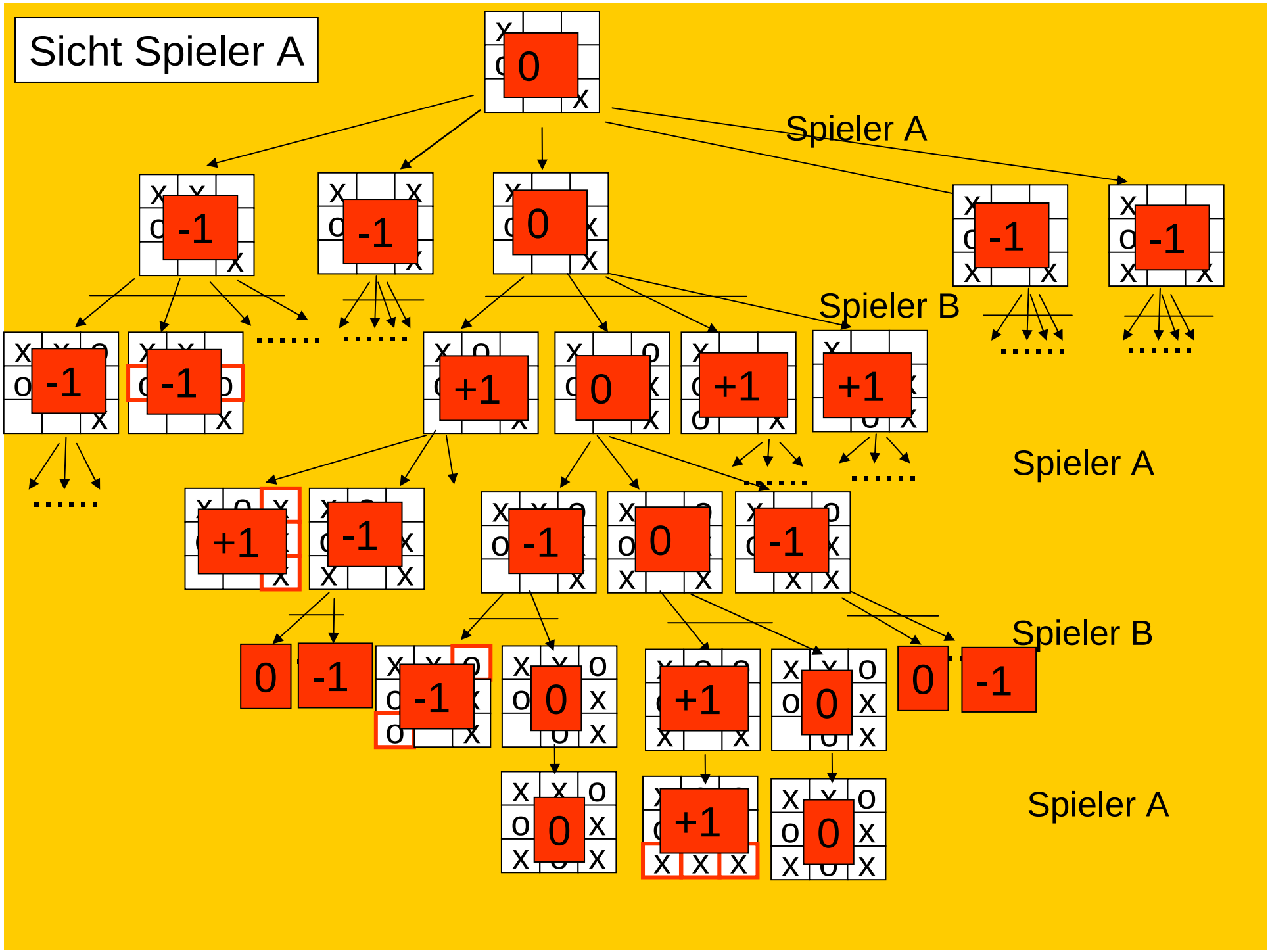

Sicht Spieler A
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A
-1
+1
-1
-1
-1
0
+1
0

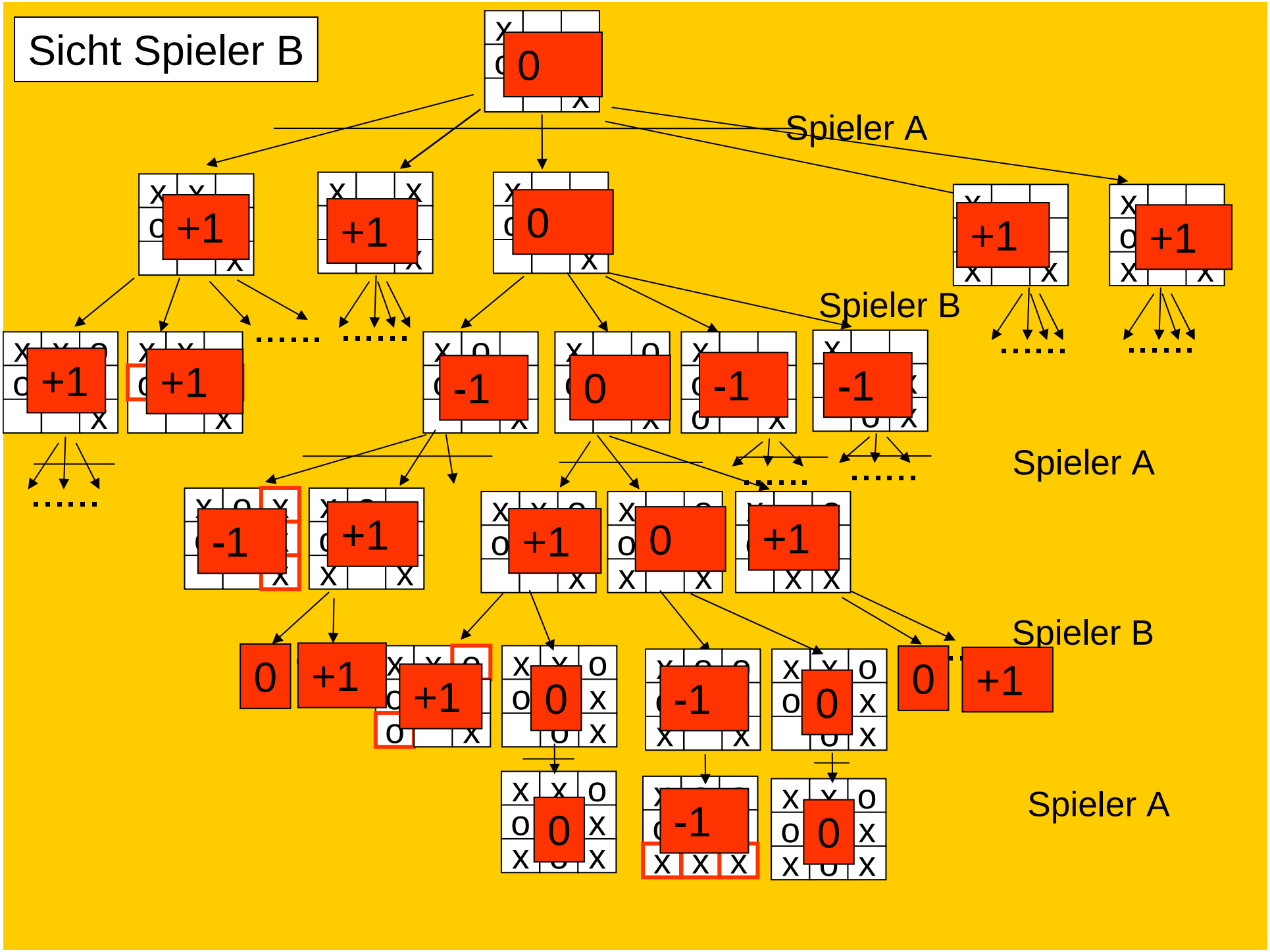

Sicht Spieler A
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A
-1
+1
0
-1
-1
0
+1
0
0
-1
0
+1
0

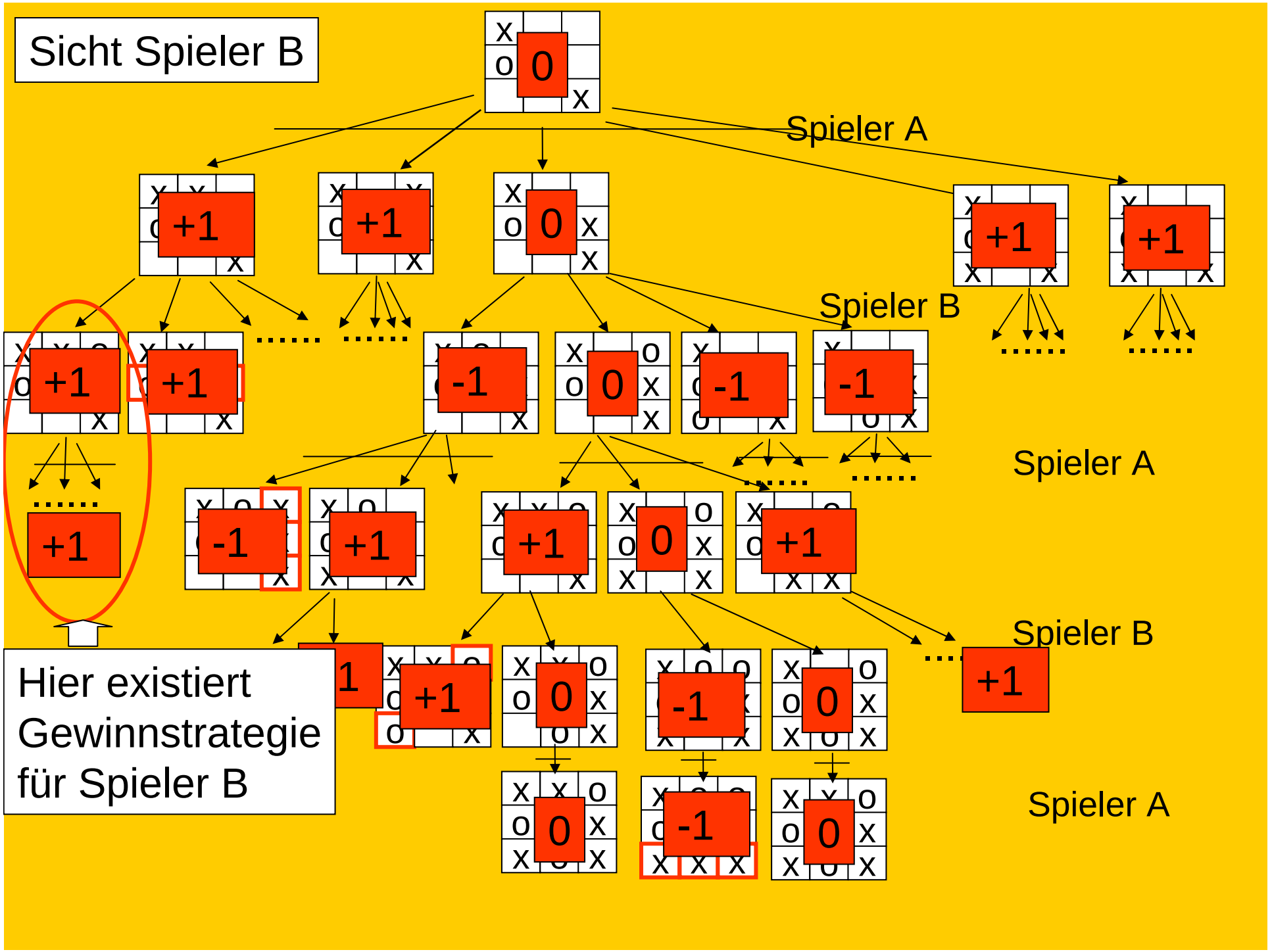

Sicht Spieler A
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A
-1
+1
-1
-1
0
-1
0
-1
-1
0
+1
0
0
-1
0
+1
0


Sicht Spieler A
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A
-1
-1
+1
0
+1
+1
+1
-1
-1
0
-1
0
-1
-1
0
+1
0
0
-1
0
+1
0

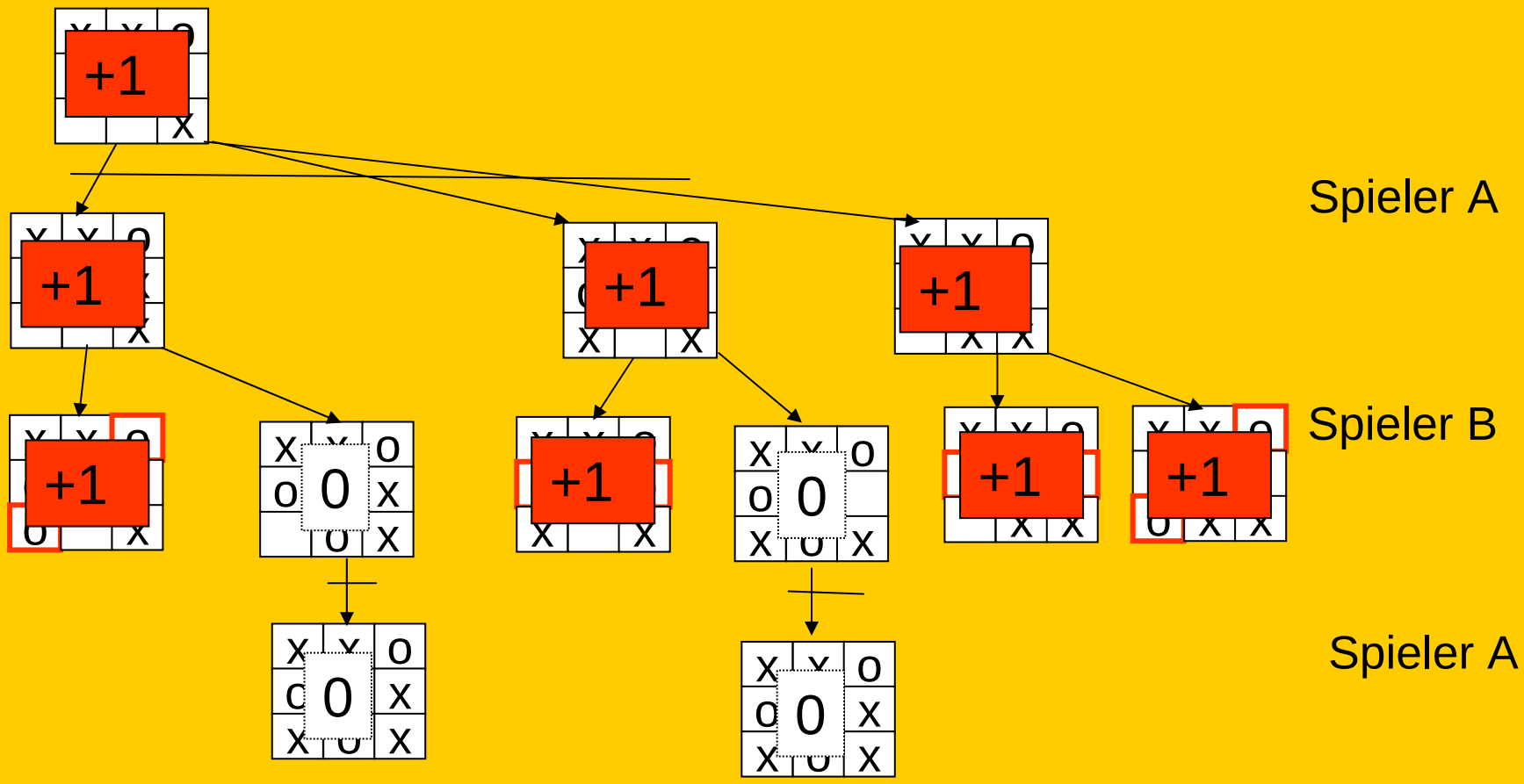

Sicht Spieler A
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A


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

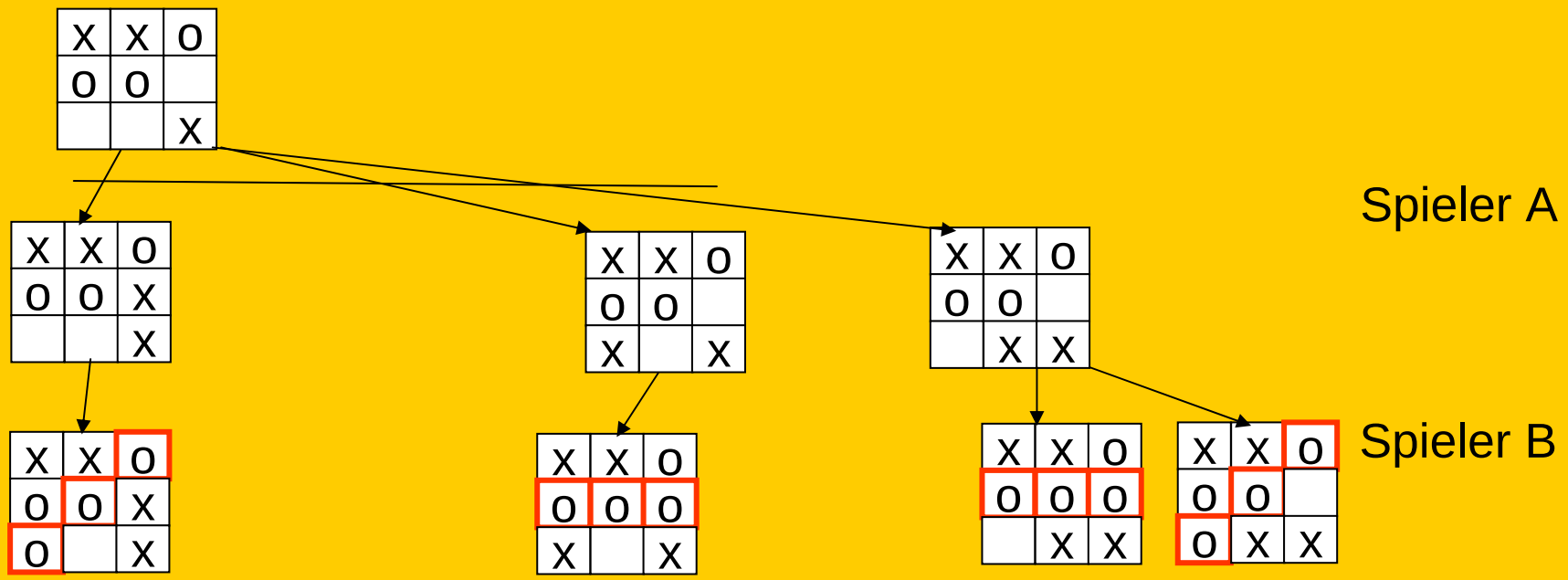

Sicht Spieler B
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A


Sicht Spieler B
Spieler A
Spieler B
Spieler A
Spieler B
Spieler A
Hier existiert Gewinnstrategie für Spieler B

Gewinnstrategie Spieler B

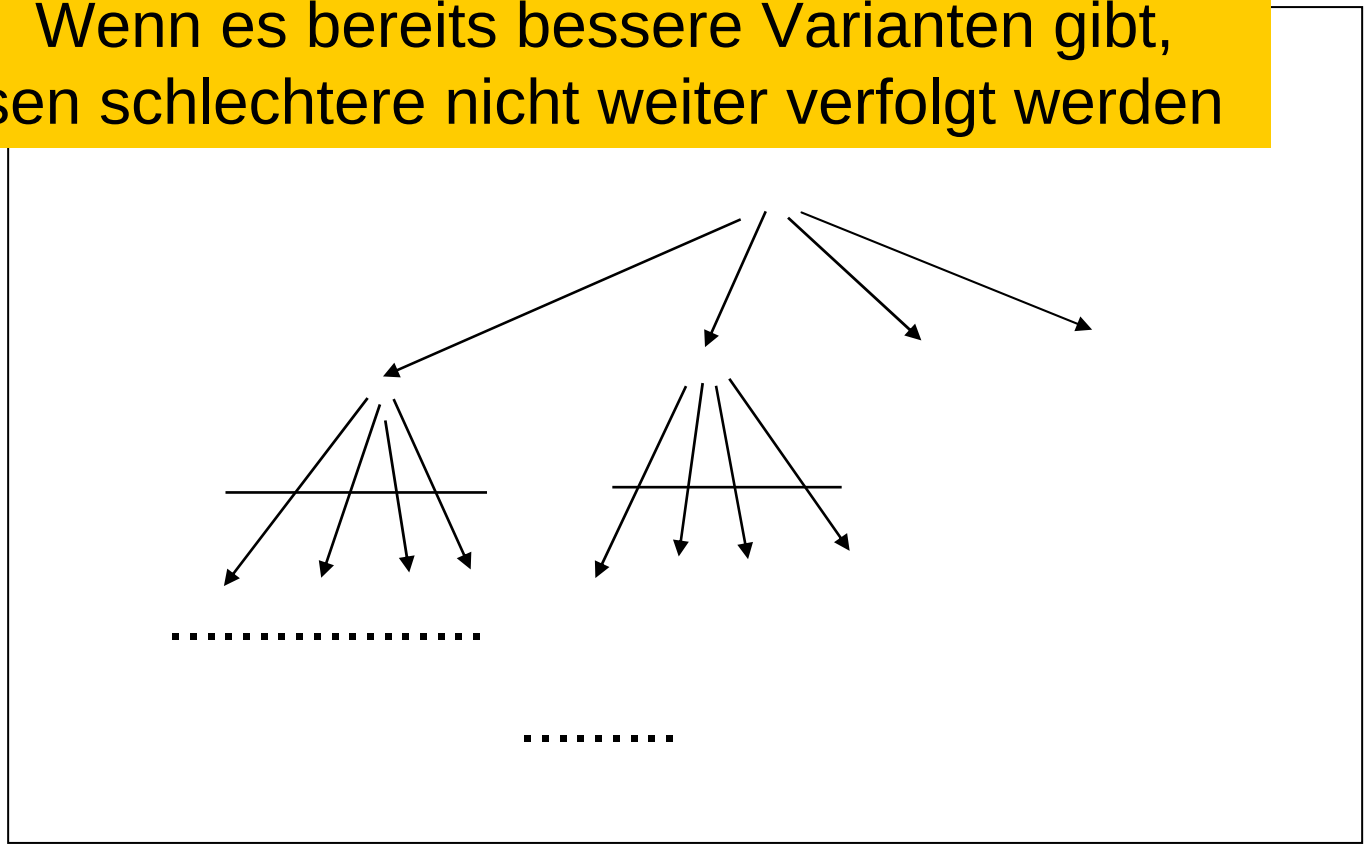


Gewinnstrategie Spieler B

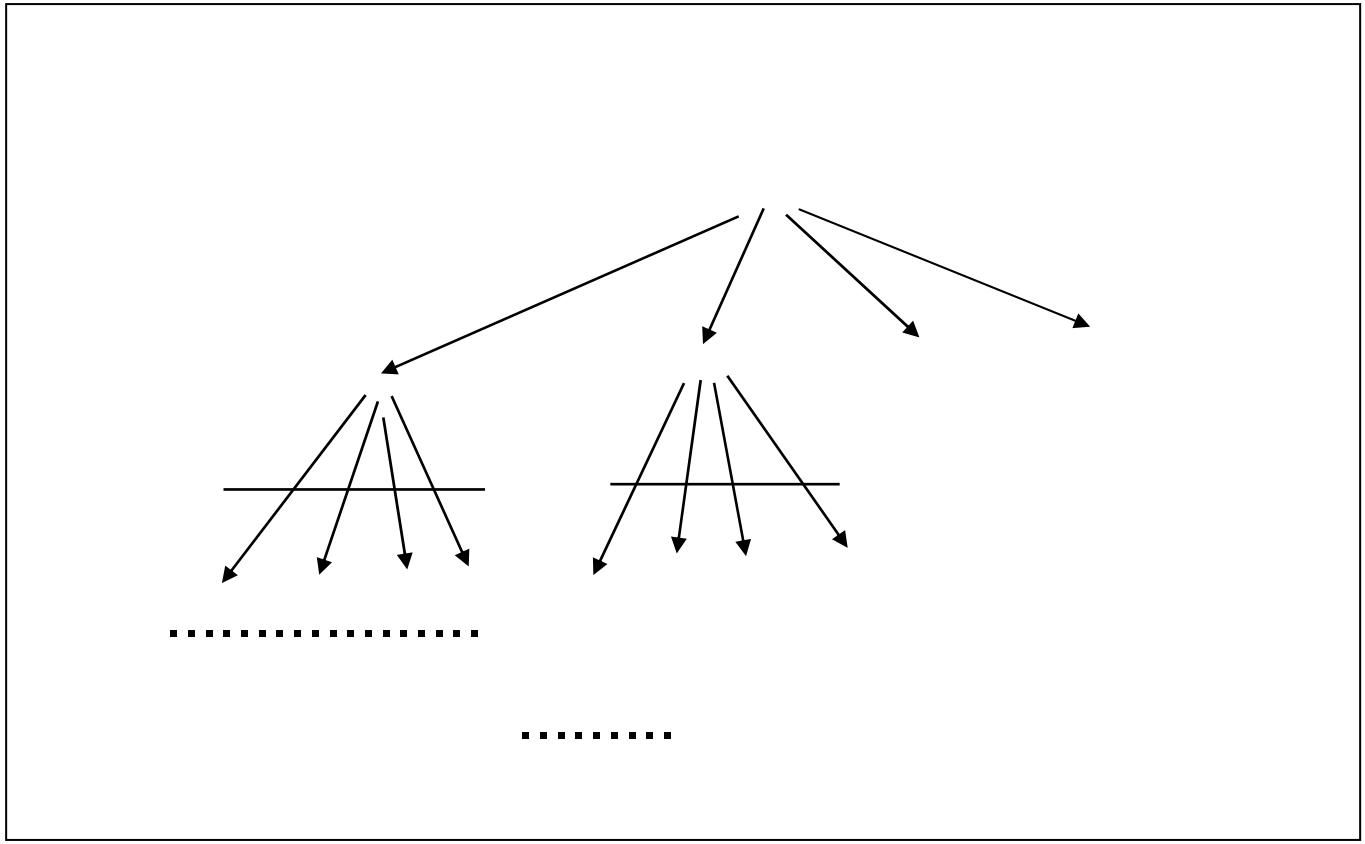


Züge ausschließen: Pruning-Strategien

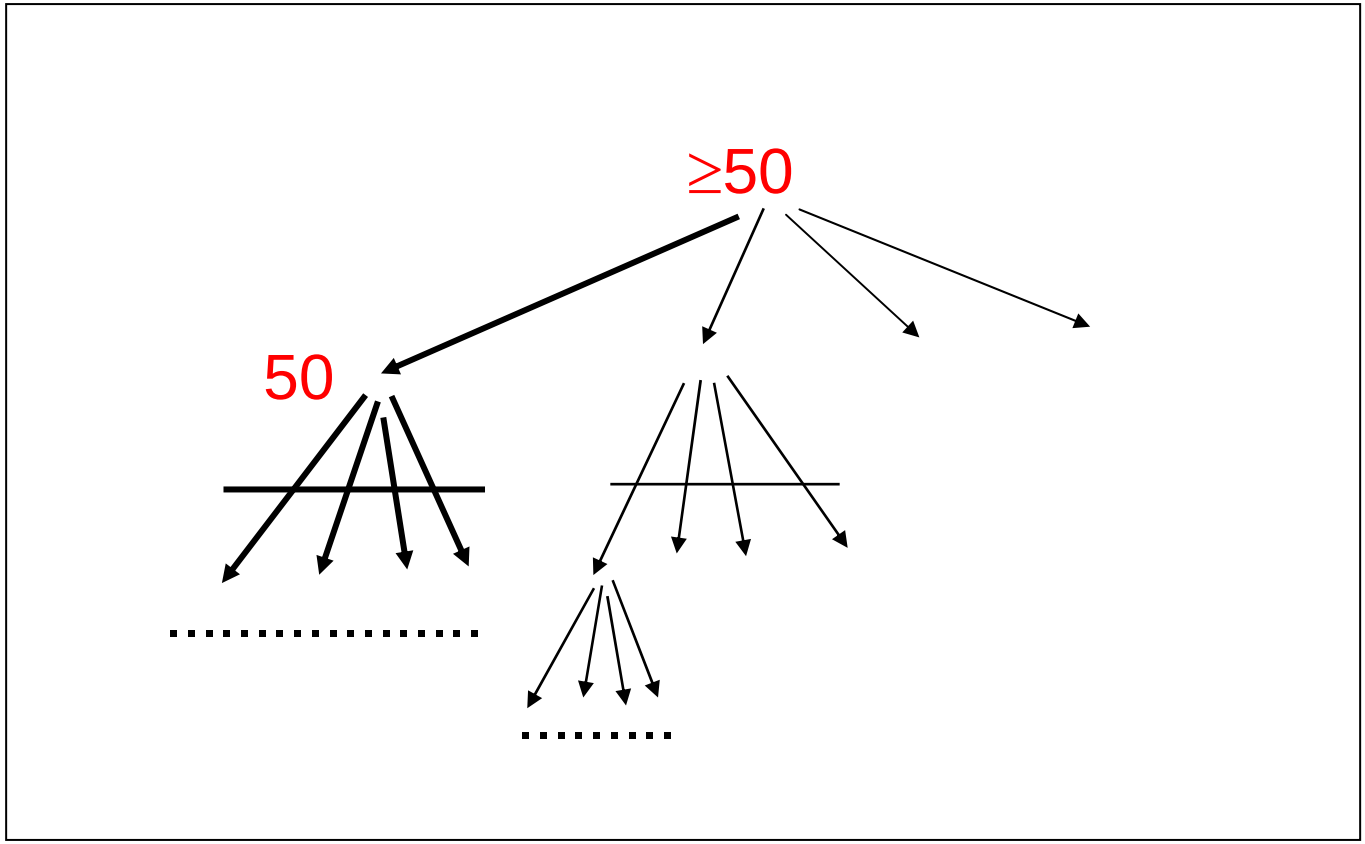
Idee: Wenn es bereits bessere Varianten gibt, müssen schlechtere nicht weiter verfolgt werden



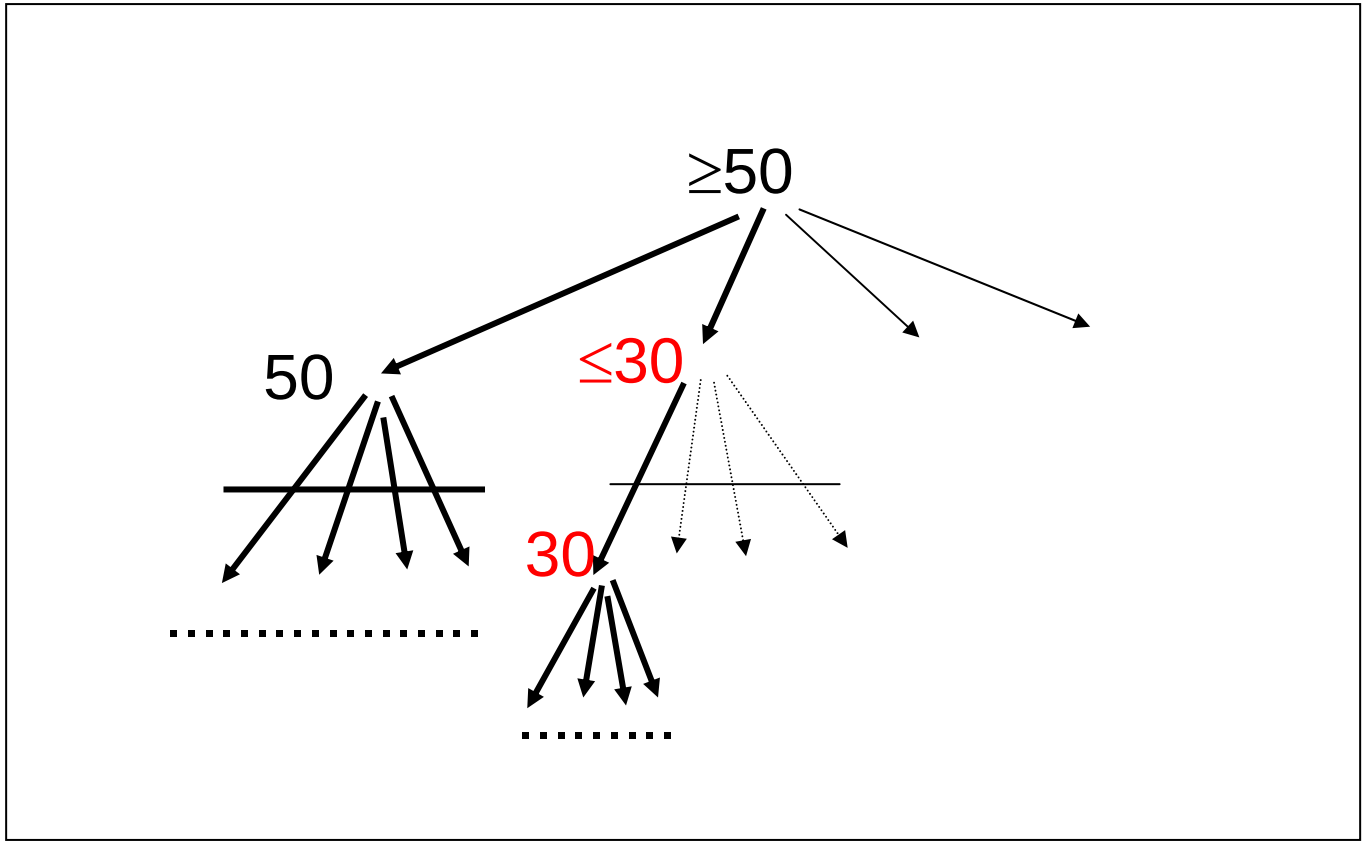
α-pruning (Züge von A ausschließen)



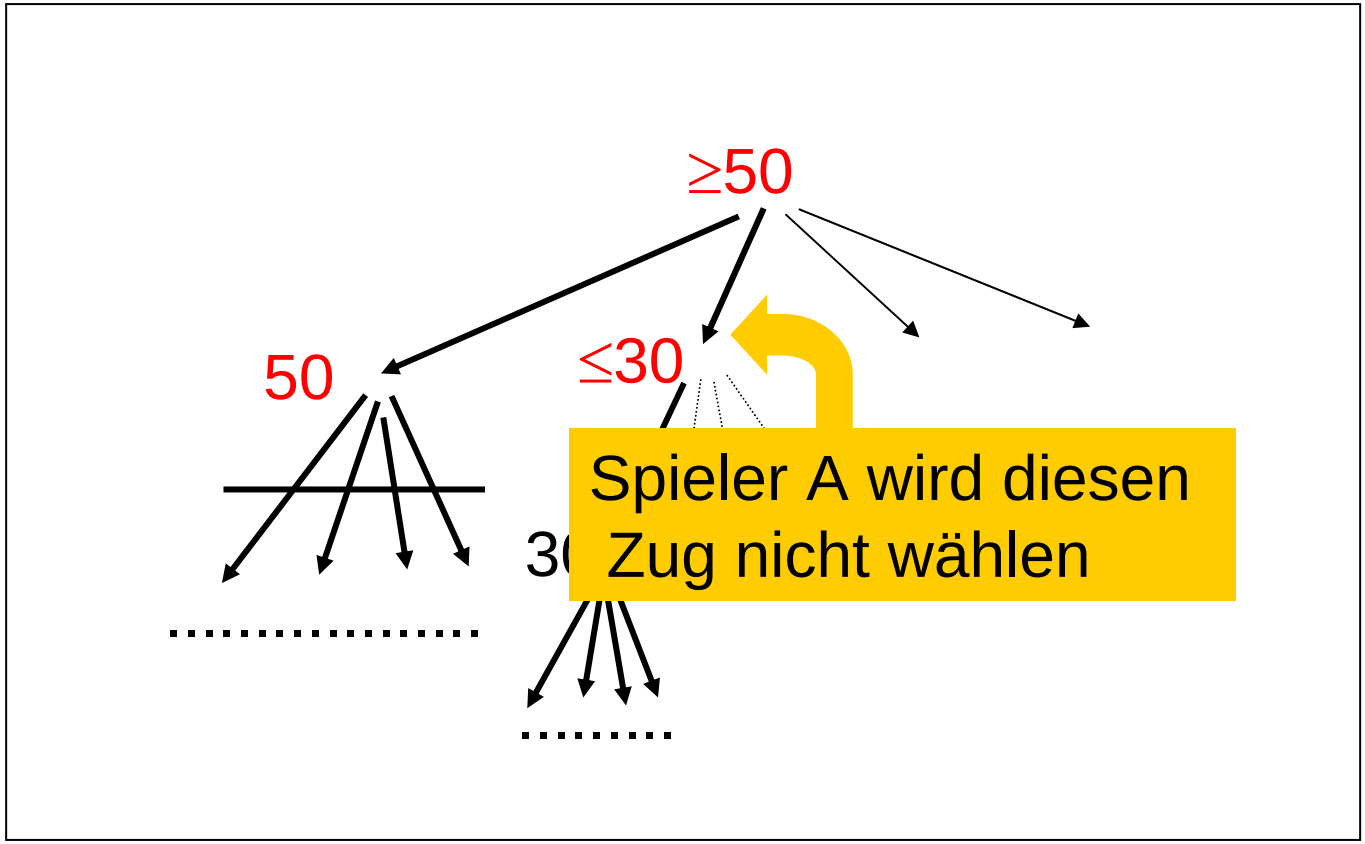
α-pruning (Züge von A ausschließen)



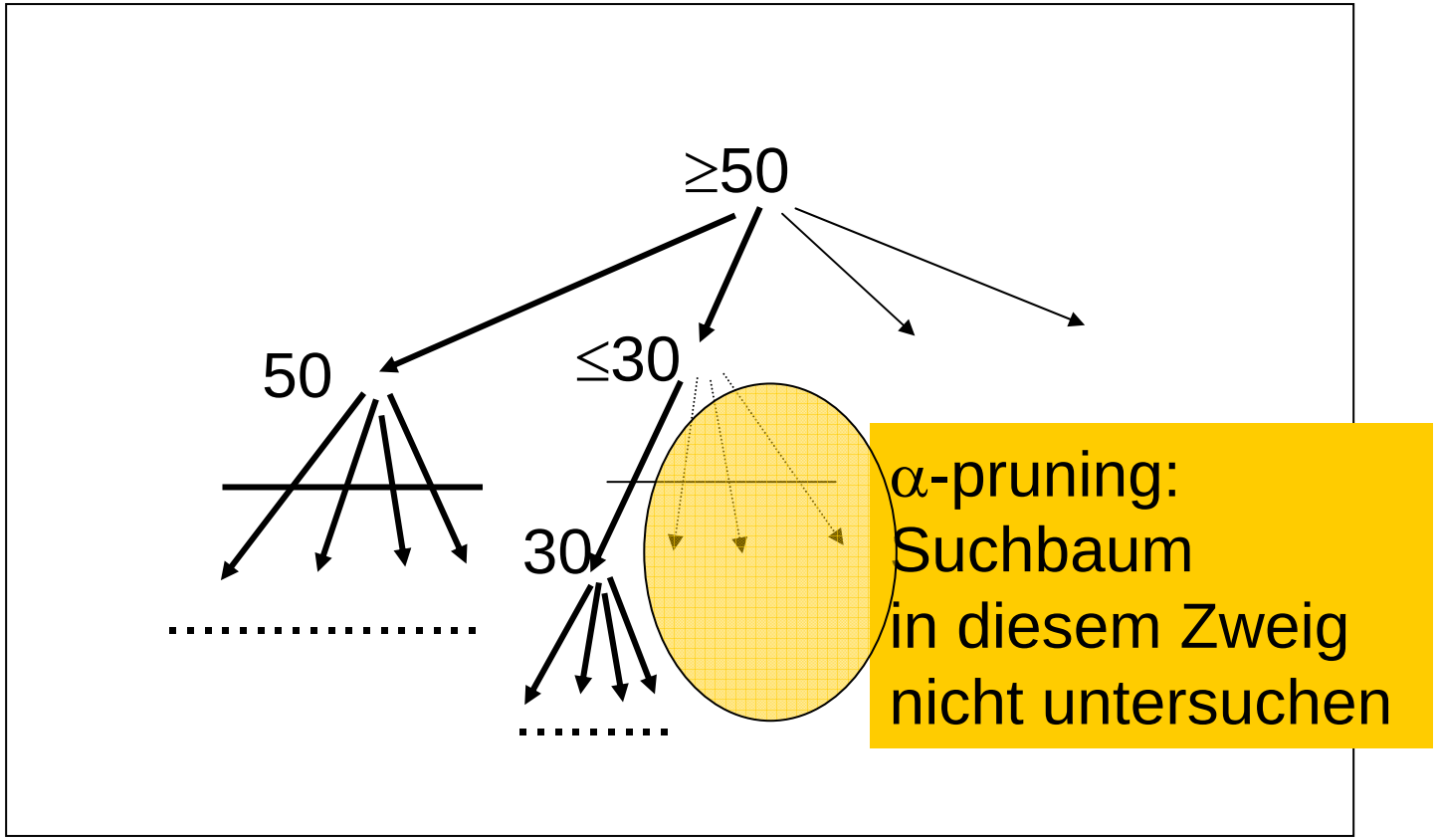
α-pruning (Züge von A ausschließen)



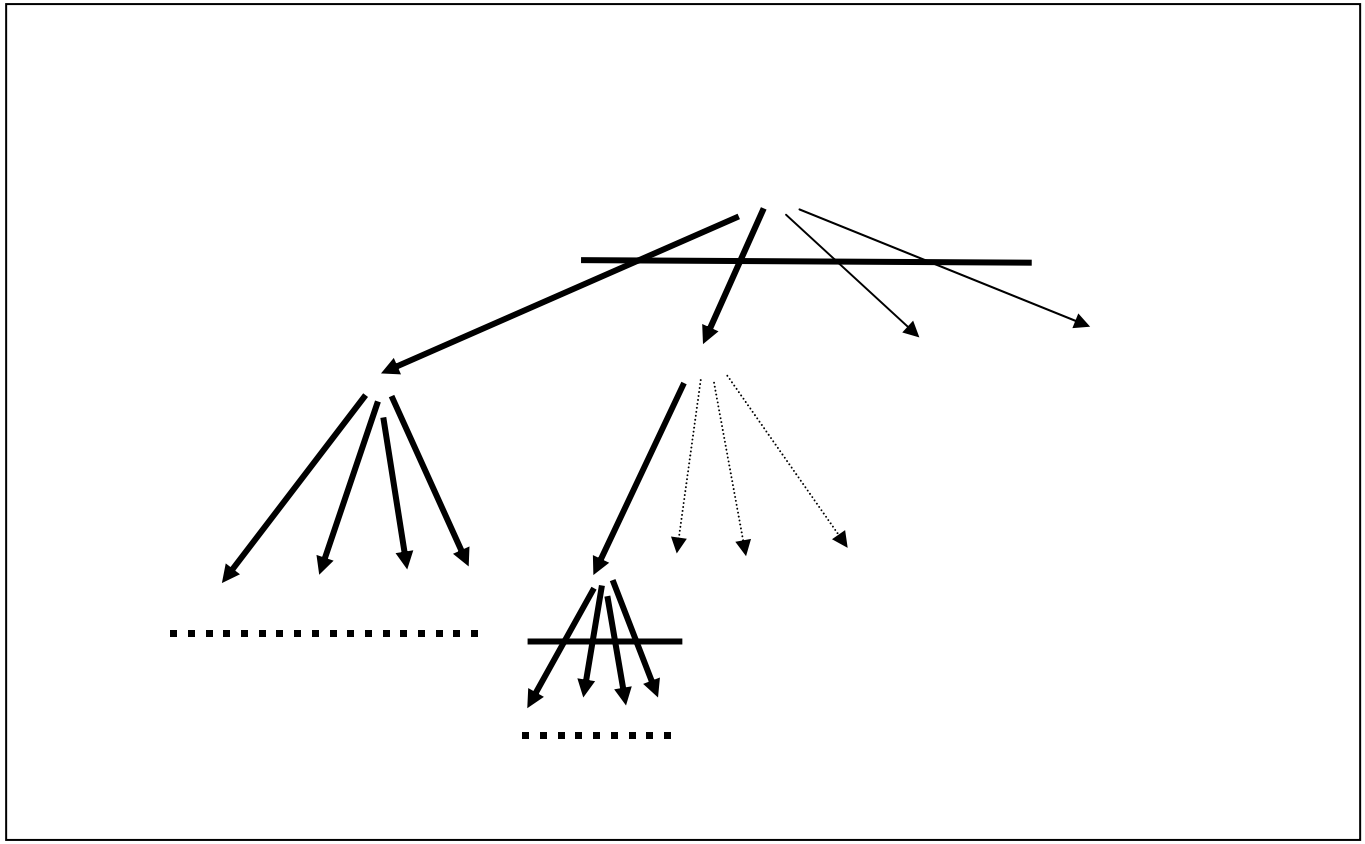
α-pruning (Züge von A ausschließen)



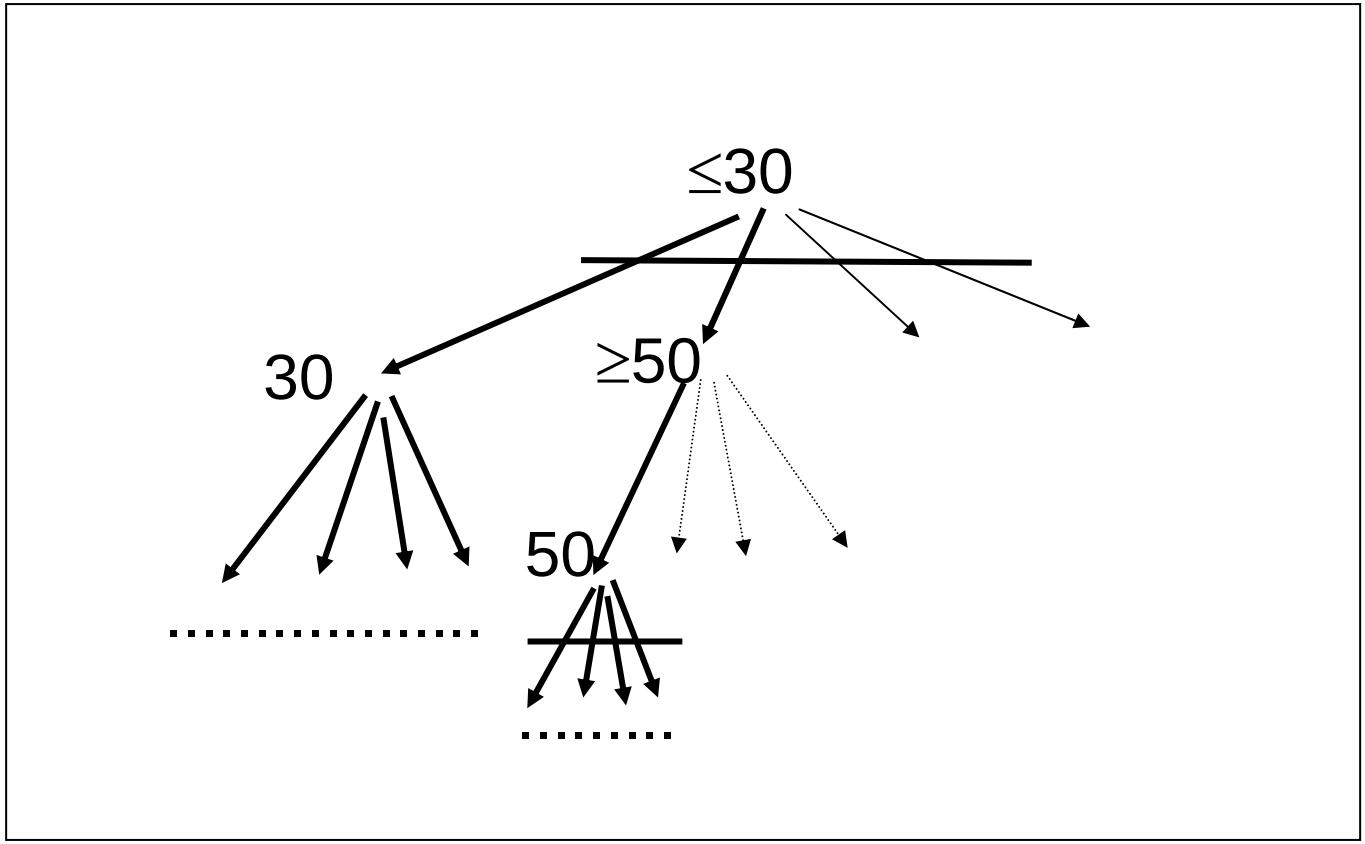
α-pruning (Züge von A ausschließen)



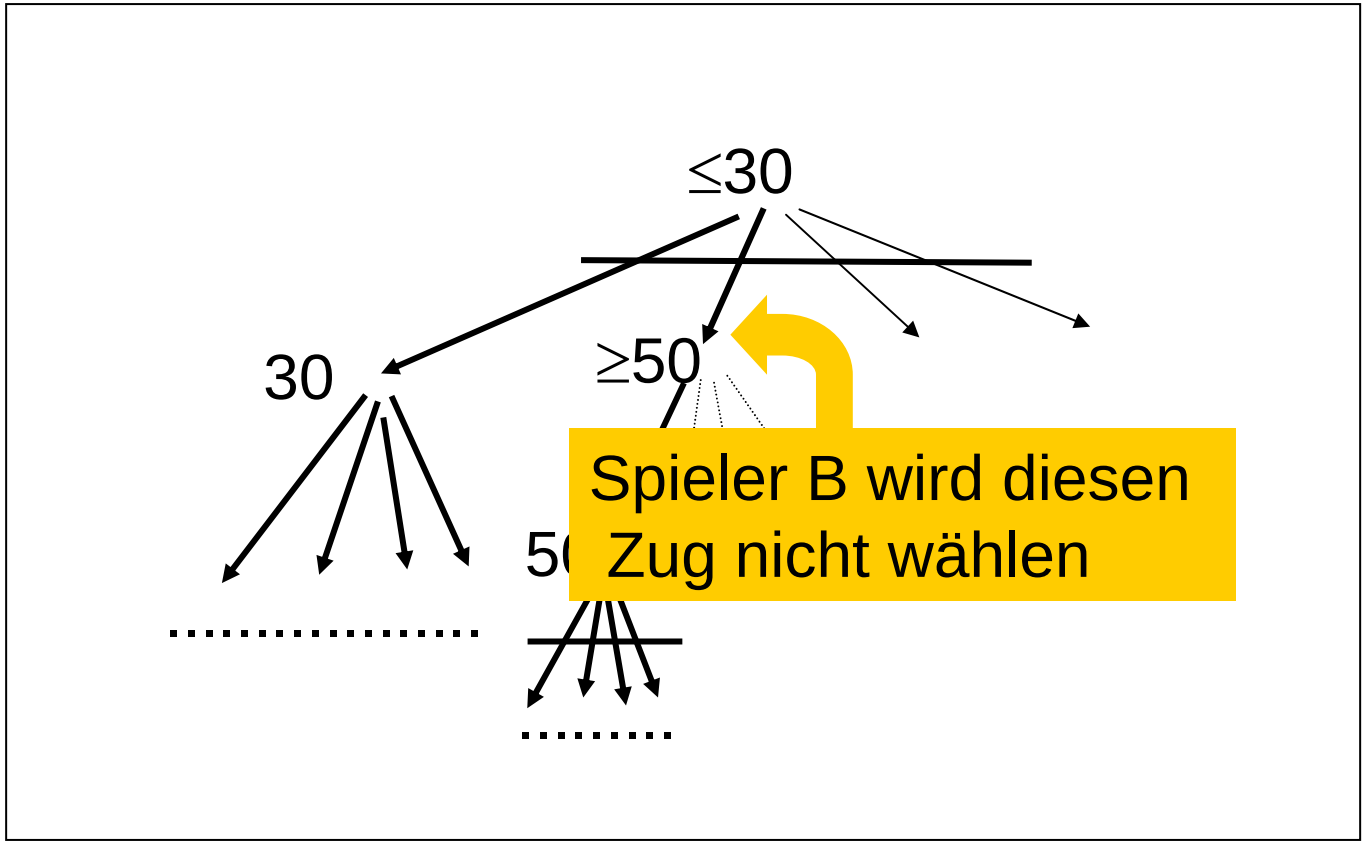
β-pruning (Züge von B ausschließen)



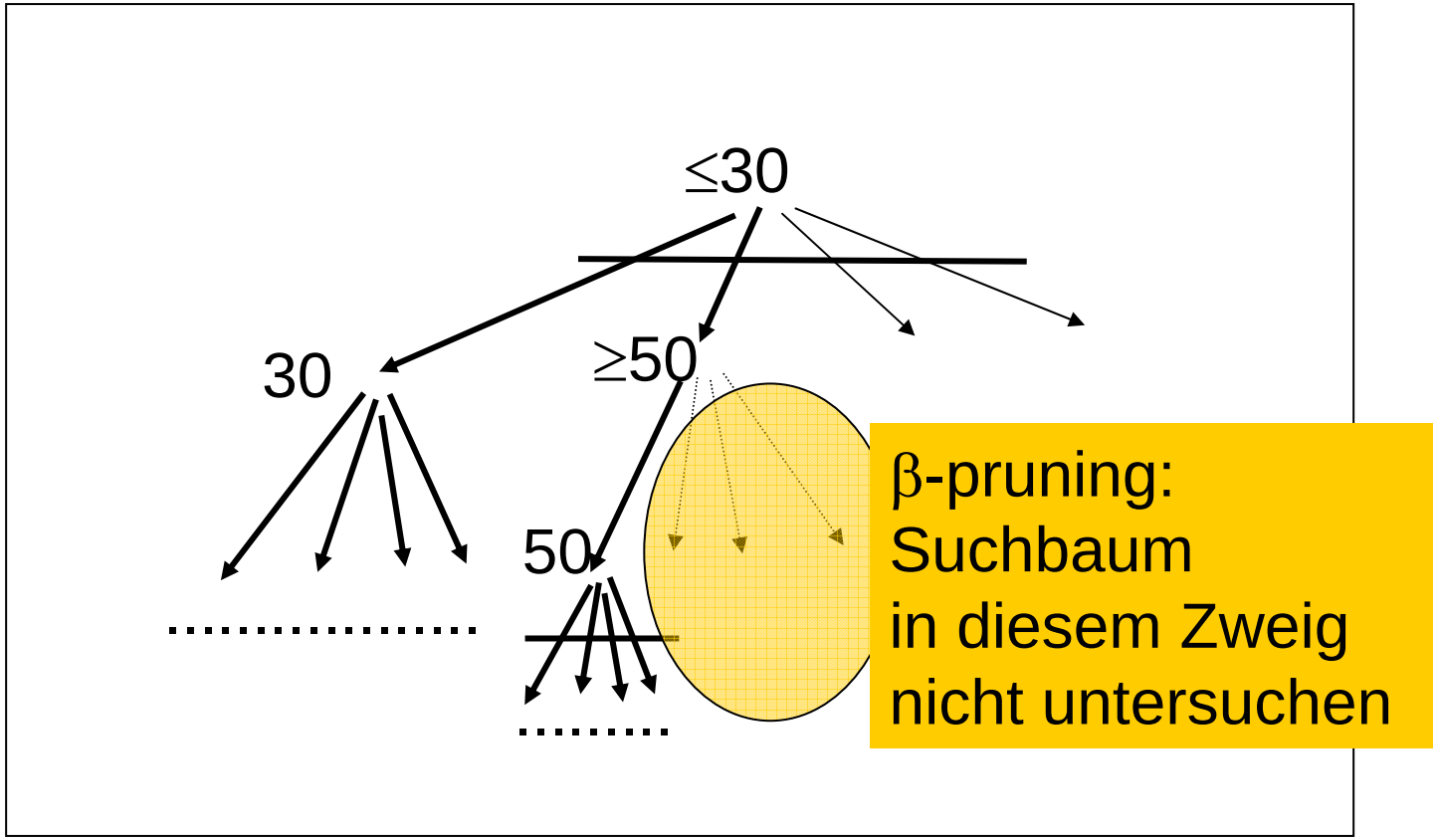
β-pruning (Züge von B ausschließen)



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 $2 * b^{(d/2)}$

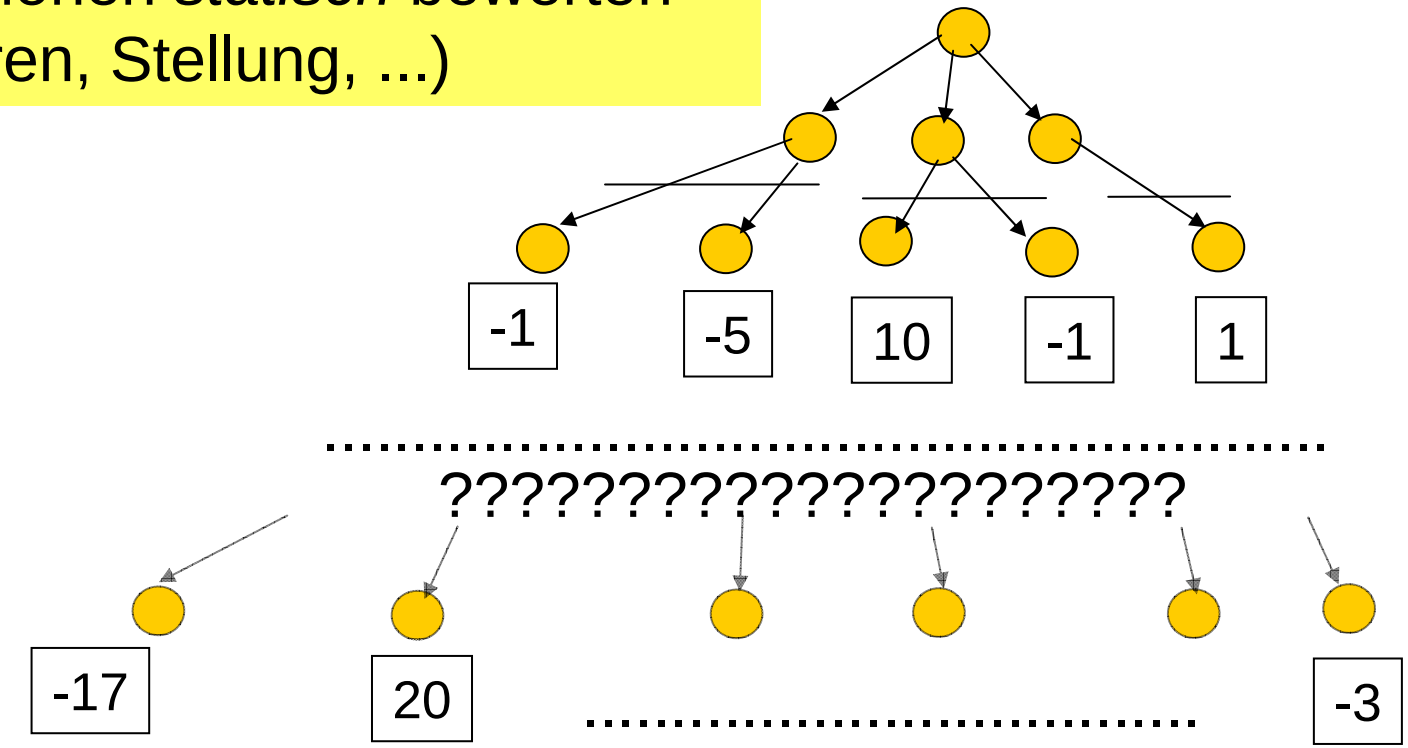
$2 * b^{(d/2)} - 1$	für gerade d,
$b^{((d+1)/2)} + b^{((d-1)/2)} - 1$	für ungerade d.

d.h. Doppelte Suchtiefe mit gleichem Aufwand möglich

Weitere Verfahren
zur Auswahl günstiger Expansionsreihenfolge

Heuristische Suche

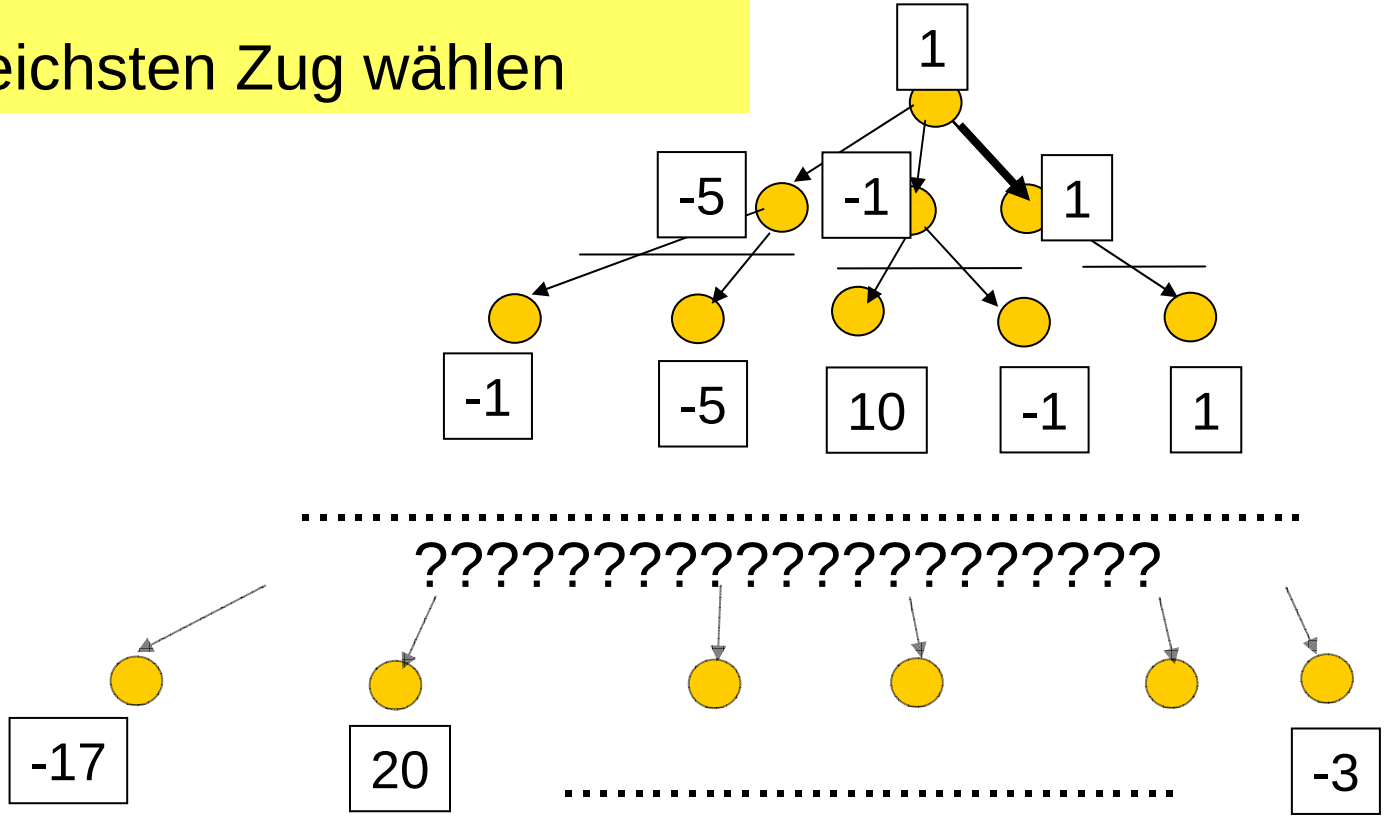
1. Spielbaum teilweise entwickeln von aktueller Situation ausgehend
2. dort Situationen *statisch* bewerten (z.B. Figuren, Stellung, ...)



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*

Suche in Parameter-Räumen

Suche nach optimalen Parametern

Lokale Optimierung/Gradientenverfahren

Evolutionäre Verfahren

Suche in Parameter-Räumen

Parameter: n Variable $x_1,...,x_n$ mit Wertebereichen $W_1,...,W_n$

Optimalitätsfunktion $c(x_1,...,x_n)$

Gesucht:

$w = [w_1,...,w_n] \in W_1 \times ... \times W_n$ mit $c(w)$ minimal

(bzw. maximal)

Lösung durch Extremwertbestimmung:

$$grad(c) = 0$$

Suche in Parameter-Räumen

Such-Verfahren:

Schrittweises Modifizieren der Parameter

Mit 1 Parametersatz:

- Lokale Optimierung
- Gradienten-Verfahren/Steilster Abstieg (bzw. Anstieg)

Mit mehreren Parametersätzen („Individuen“)

- Evolutionäre/Genetische Algorithmen

Lokale Optimierung

Vgl. Bergsteigen:

Zustände: $z = [w_1,...,w_n]$

Die Nachbarschaft der Zustände (succ(z)) ergibt sich aus der Nachbarschaft der Parameterwerte w_i.

Bei differenzierbarer Nutzensfunktion:

Gradientenabstieg anwendbar (Suche in Richtung der größten Änderung der Nutzensfunktion)

Lokale Optimierung

- Vorgebirgsproblem
- Plateau-Problem
- Grat-Problem

Gradienten-Verfahren/Steilster Abstieg

Start: z_0 wählen

Iteration: $z_{t+1} := z_t + \alpha \ \text{grad}(c(z_t))$

mit Schrittlängen $\alpha > 0$

„Simulated Annealing“ bzgl. Schrittlängen:

Schrittlänge α als monoton fallende Funktion der Zeit

Evolutionäre/Genetische Algorithmen

Idee aus der Natur:

Vermehrung „aussichtsreicher“ Lösungskandidaten

Kombination:

- Kreuzung
- Mutation

– Bewertung:

- Fitness

– Auslese:

- gemäß Wahrscheinlichkeit ~ Fitness

Evolutionäre/Genetische Algorithmen

Unterschiedliche Bereiche des Suchraums erfassen

Genetische Algorithmen:
Parameter-Raum = $\{0,1\}^n$

Evolutionäre Algorithmen:
Parameter-Raum = $\mathcal{R}^n$

Population, Individuum

Population: Menge von Individuen

Individuum: Durch Parametersatz beschrieben.

Evolutionäre/Genetische Algorithmen

Mutation:

Veränderung von Werten im Individuum $w \in \text{Population}$

Kombination („cross-over“):

neues Individuum („Kind“) w' aus mehreren Individuen („Eltern“) $w_1, \ldots, w_n \in \text{Population}$

Fitness: Nähe zu Optimalitätskriterium

Auswahl: Wahrscheinlichkeit gemäß Fitness

Evolutionäre/Genetische Algorithmen

Beispiel Kombination („cross-over“):

Aus $w_1 = [w^1_1,...,w^1_n]$ und $w_2 = [w^2_1,...,w^2_n]$

wird $w' = [w'_1,...,w'_n]$ erzeugt,

wobei w'_i mit w^1_i oder w^2_i übereinstimmt

alternativ: w'_i wird aus w^1_i und w^2_i berechnet

Statt 2 können k Eltern beteiligt sein (einschließlich k=1).

Mehrere Varianten innerhalb einer Anwendung möglich

Evolutionäre/Genetische Algorithmen

Grundschema:

E1: (Start) $t:=0$, $Population(0) := \{ w_1(0) ,.., w_k(0) \}$
$Fitness(Population(t)) := \{ Fitness(w_1(t)) ,.., Fitness(w_k(t)) \}$

E2: (Abbruch)
Falls $Fitness(Population(t))$ „gut“: EXIT($Population(t)$)

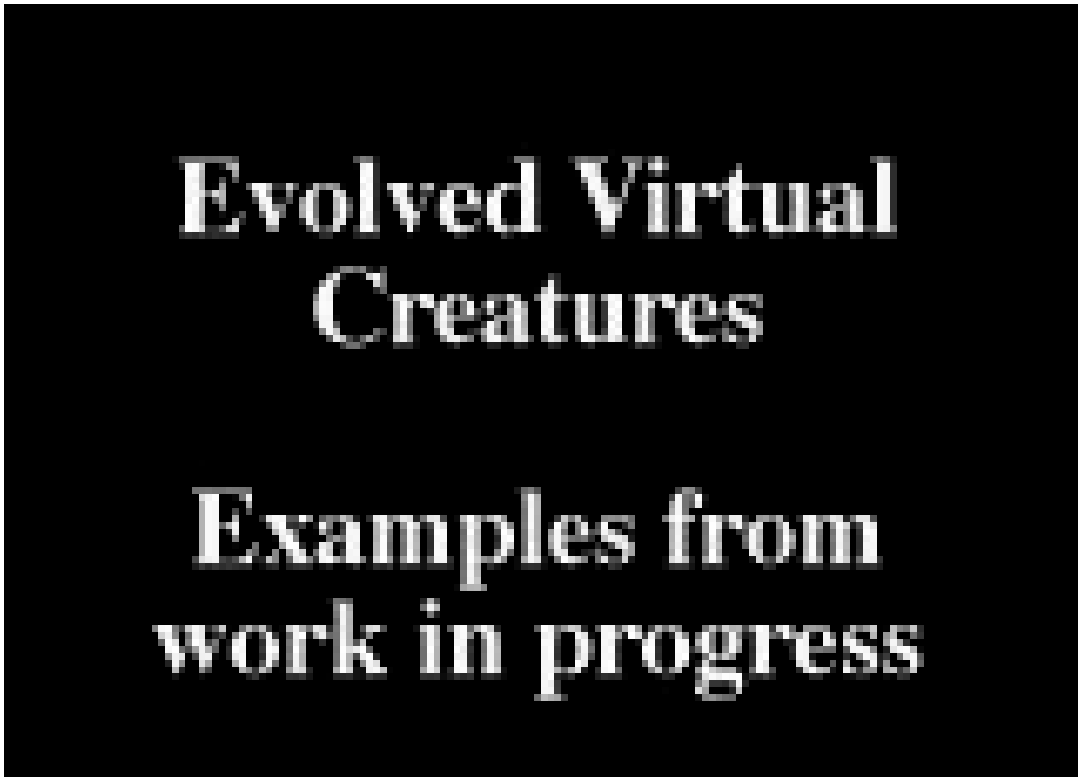
E3: (Kombination, Mutation)
$Population'(t) = \{w'_1(t) ,.., w'_{k'}(t) \}$
$:= mutate (recombine (Population(t))$

E4: (Bewertung)
$Fitness(Population'(t)) := \{ Fitness(w'_1(t)) ,.., Fitness(w'_{k'}(t)) \}$

E5: (Auswahl)
$Population(t+1) = \{w_1(t+1) ,.., w_k(t+1) \}$
$:= select(Population'(t), Fitness(Population'(t)))$
$t := t+1$. Goto E2 .

Evolutionäre/Genetische Algorithmen

Karl Sims -- Virtual Creatures



Evolutionäre/Genetische Algorithmen

Humboldt-Universität:
RoboCup-Projekt, u.a.

- Omnidirektionales Laufen für AIBO
  Diplomarbeit Uwe Düffert
- Simulierter Roboter: Simloid
  Diplomarbeit Daniel Hein