Programmierkurs Prolog

SS 2000

Peter Brockhausen

Universitaet Dortmund

nach Joachims 1998

Suche

- Repräsentation von Bäumen
- Repräsentation von Graphen
- Suchstrategien

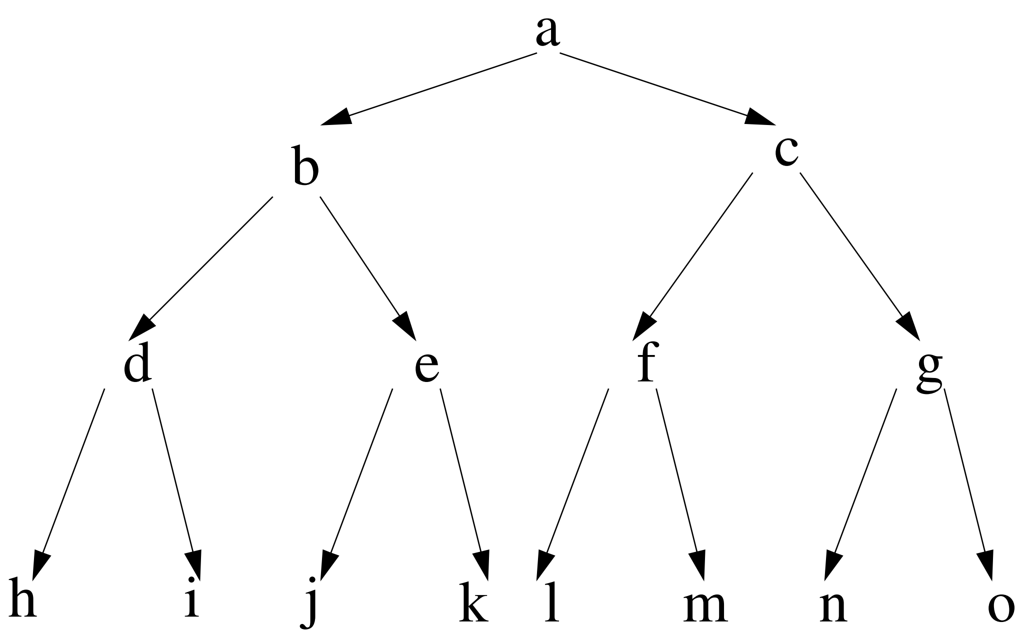
  DFS, BFS, Iterative Deepening, Locale Heuristiken, Globale Heuristiken, A^*
- Spiele

  MiniMax, Alpha/Beta Pruning

Anwendungen fuer Suche

- Spiele
- Planen
- Maschinelles Lernen
- Theorembeweisen
- Text Parsing/Sprachverarbeitung
- Optimierung

Repräsentation von Bäumen



Einbettung in eine Struktur

```
a(b(d(h,i),e(j,k)),
  c(f(l,m),g(n,o)))
```

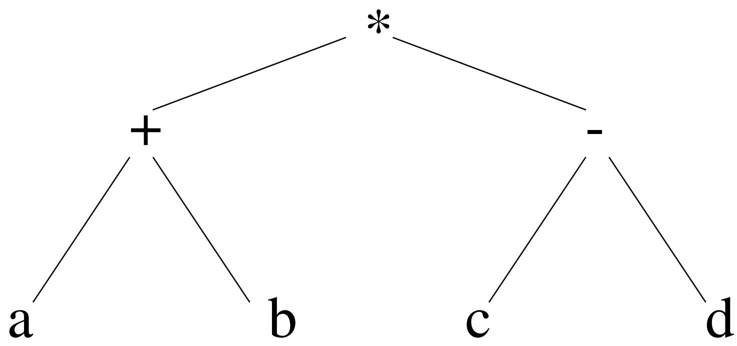
oder

```
[a,[b,[d,[h,[],[]],[i,[],[]]],
      [e,[j,[],[]],[k,[],[]]]],
   [c,[f,[l,[],[]],[m,[],[]]],
      [g,[n,[],[]],[o,[],[]]]]]
```

Baumtraversierung

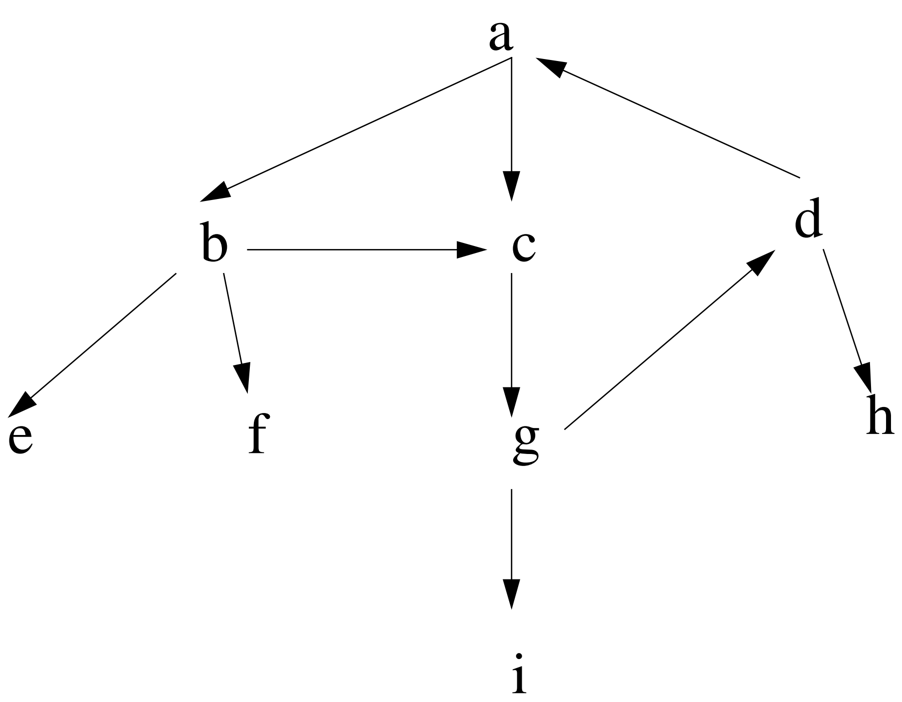
```
operate(X) :- write(X), write(` `).
prefix([]).
prefix([Value,Left,Right]) :-
    operate(Value),
    prefix(Left),
    prefix(Right).
infix([]).
infix([Value,Left,Right]) :-
    infix(Left),
    operate(Value),
    infix(Right).
postfix([]).
postix([Value,Left,Right]) :-
    postfix(Left),
    postfix(Right),
    operate(Value).
traverse(prefix,Tree):-
    prefix(Tree).
traverse(infix,Tree) :-
    infix(Tree).
traverse(postfix,Tree) :-
    postfix(Tree).
```

Beispiel: Baumtraversierung



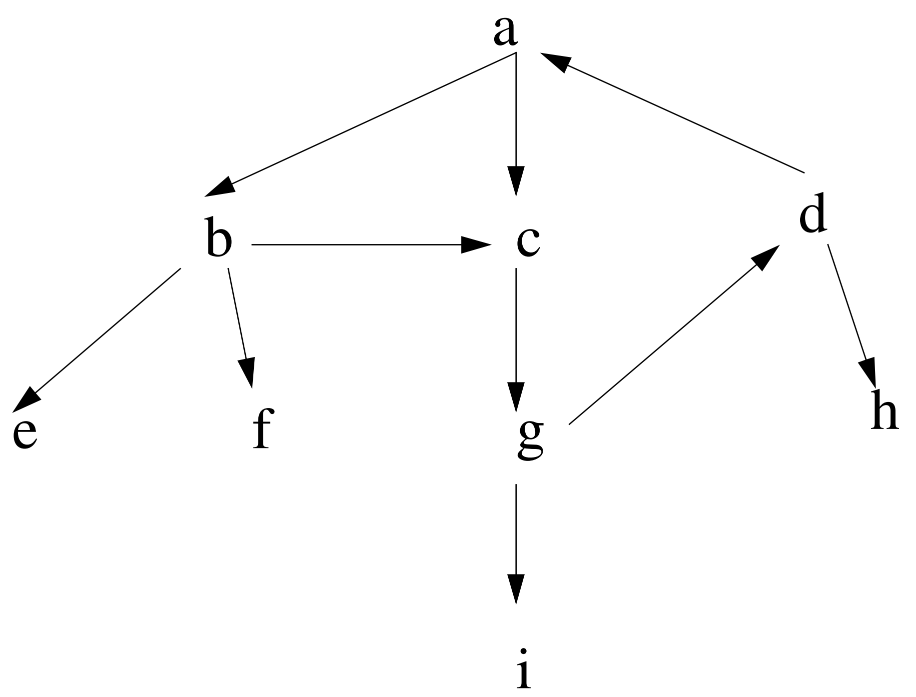

```
tree([*, [+,[a,[],[]],[b,[],[]]],
         [-,[c,[],[]],[d,[],[]]]]).

| ?- tree(X),prefix(X).
* + a b - c d

| ?- tree(X),infix(X).
a + b * c - d

| ?- tree(X),postfix(X).
a b + c d - *
```

Repräsentation von Graphen (1)



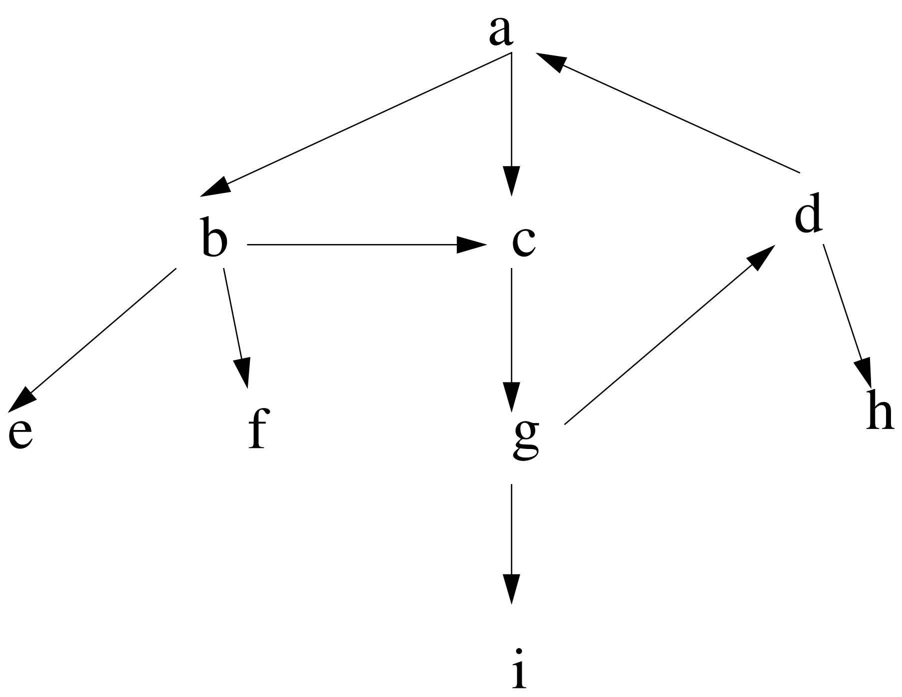
Als Fakten:

Repräsentation von Graphen (2)



Als Knoten/Kantenliste:

Repräsentation von Graphen (3)

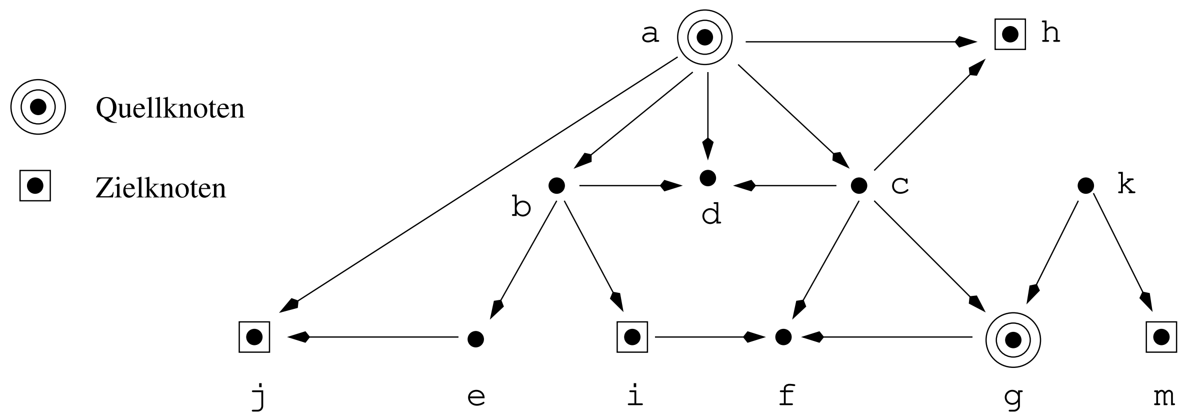


Als Adjazenzlisten:

Suche in Prolog

`start(Knoten)`	Prädikat, das festlegt, welche Knoten gültige Startknoten sind.
`kante(Q,Z)`	Prädikat, das festlegt, ob es eine direkte Kante von `Q` zu `Z` gibt.
`ziel(Knoten)`	Prädikat, das festlegt, welche Knoten gültige Zielknoten sind.

Beispielgraph:




```
start(a). start(g).
kante(a,b). kante(a,c). kante(a,d).
kante(b,d). kante(b,e). kante(b,i).
...
ziel(h). ziel(i). ziel(j). ziel(m).
```

Tiefensuche durch Backtracking

Algorithmus:

```
loesung(Ziel) :-
   start(Start),
   kantenzug(Start,Ziel),
   ziel(Ziel).

kantenzug(Knoten,Knoten).
kantenzug(Quell,Ziel) :-
   kante(Quell,Zwischen),
   kantenzug(Zwischen,Ziel).
```

Beispiel:

Generische Beschreibung von Suchalgorithmen

1. Sei L die Liste der Startknoten für das Problem
2. Ist L leer, so melde einen Fehlschlag. Andernfalls wähle einen Knoten n aus L.
3. Stellt n einen Zielknoten dar, so melde Erfolg und liefere den Pfad vom Startknoten zu n.
4. Anderenfalls ersetze in L den Knoten n durch seine Nachfolgeknoten. Markiere dabei die neuen Knoten mit dem jeweils zugehörigen Pfad vom Startknoten.
5. Weiter mit Schritt 2.

Anmerkung: Knoten werden erst beim Expandieren auf die Zieleigenschaft geprüft.

Die Art der Wahl in Schritt 2 bestimmt die Suchstrategie:

- systematisch
- heuristisch

Tiefensuche mit OFFEN-Liste

Gegeben:

```
nachfolger/2.
```

Algorithmus:

```
tiefensuche(Ziel) :-
   start(Start),
   tiefensuche([Start],Ziel),
   ziel(Ziel).
tiefensuche([X|_],X).
tiefensuche([X|Offen],Y) :-
     nachfolger(X,Nachf),
     append(Nachf,Offen,OffenNeu),
     tiefensuche(OffenNeu,Y).
```

Beispiel:

Ermitteln des Weges zum Ziel

Algorithmus:

```
loesung(Ziel,Weg) :-
   start(Start),
   kantenzug(Start,Ziel,Weg),
   ziel(Ziel).

kantenzug(Knoten,Knoten,[Knoten]).
kantenzug(Quell,Ziel,[Quell|Weg]):-
   nachfolger(Quell,Zwischen),
   kantenzug(Zwischen,Ziel,Weg).
```

Beispiel:

Tiefensuche mit Zyklenerkennung in Graphen

Gegeben:

```
nachfolger/2.
```

Algorithmus:

```
tiefensuche(Ziel) :-
   start(Start),
   tiefensuche([Start],[],Ziel),
   ziel(Ziel).
tiefensuche([X|_],_,X).
tiefensuche([Y|Offen],Geschl,X) :-
   nonmember(Y,Geschl),
   nachfolger(Y,Nachf),
   append(Nachf,Offen,OffenNeu),
   tiefensuche(OffenNeu,[Y|Geschl),X).

tiefensuche([Y|Offen],Geschl,X) :-
     member(Y,Geschl),
     tiefensuche(Offen,Geschl,X).
```

Beispiel:

Tiefensuche mit Tiefenbeschraenkung in Graphen

Gegeben:

```
nachfolger/2.
```

Algorithmus:

```
tiefensuche(Ziel,Tiefe) :-
   start(Start),
   tiefensuche_tb(Start,Ziel,Tiefe),
   ziel(Ziel).

tiefensuche_tb(Z,Z,_).

tiefensuche_tb(X,Z,T) :-
     TNeu is T-1,
     TNeu >= 0,
     kante(X,Y),
     tiefensuche_tb(Y,Z,TNeu).
```

Beispiel:

Breitensuche in Graphen

Gegeben:

```
nachfolger/2.
```

Algorithmus:

```
breitensuche(Ziel) :-
   start(Start),
   breitensuche([Start],Ziel),
   ziel(Ziel).
breitensuche([X|_],X).
breitensuche([X|Offen],Y) :-
     nachfolger(X,Nachf),
     append(Offen,Nachf,OffenNeu),
     breitensuche(OffenNeu,Y).
```

Beispiel:

Heuristische Suche

Nutze Wissen über die Problemdomäne!

Eine heuristische Funktion bewertet die als nächstes zu expandierenden Knoten.

- Lokale Heuristik: *Hill Climbing*

- Globale Heuristik: *Best-First-Search*

Hill Climbing

Hill Climbing: Lokale Ordnung durch Heuristik

1. Sei L die Liste der Startknoten für das Problem, *sortiert* nach der Heuristik.
2. Ist L leer, so melde einen Fehlschlag. Andernfalls wähle den *ersten* Knoten n aus L.
3. Stellt n einen Zielknoten dar, so melde Erfolg und liefere den Pfad vom Startknoten zu n.
4. Anderenfalls entferne n aus L und füge seine Nachfolger *sortiert* nach der Heuristik *am Anfang* von L ein. Markiere dabei die neuen Knoten mit dem jeweils zugehörigen Pfad vom Startknoten.
5. Weiter mit Schritt 2.

Best-First-Search

Best-First-Search: Globale Ordnung durch Heuristik

1. Sei L die Liste der Startknoten für das Problem, *sortiert* nach *f*.
2. Ist L leer, so melde einen Fehlschlag. Andernfalls wähle den *ersten* Knoten n aus L.
3. Stellt n einen Zielknoten dar, so melde Erfolg und liefere den Pfad vom Startknoten zu n.
4. Anderenfalls entferne n aus L und füge seine Nachfolger in L ein. Markiere dabei die neuen Knoten mit dem jeweils zugehörigen Pfad vom Startknoten. *Sortiere* L nach der Heuristik *f*.
5. Weiter mit Schritt 2.

Best-First-Search in Prolog

```
:- use_module(library(ordsets)).
:- use_module(library(heaps)).

best_first(Ziel) :-
    start(Start),
    initial_heap(Start,Heap),
    best_first(Heap,[Start],Ziel),
    ziel(Ziel).

initial_heap(Start,Heap) :-
    heuristik(Start,Wert),
    empty_heap(Empty),
    add_to_heap(Empty,Wert,Start,Heap).

best_first(Heap,Closed,Ziel) :-
    get_from_heap(Heap,_,Node,Heap1),
    (
        Ziel=Node
    ;   member(Node,Closed),
        best_first(Heap1,Closed,Ziel)
    ;   nonmember(Node,Closed),
        nachfolger(Node,Nachf),
        add_children(Nachf,Heap1,Heap2),
        best_first(Heap2,[Node|Closed],Ziel)
    ).
```

```
add_children([],Heap,Heap).
add_children([K|Nachf],HeapIn,HeapOut) :-
    heuristik(K,Wert),
    add_to_heap(HeapIn,Wert,K,Heap),
    add_children(Nachf,Heap,HeapOut).
```

wobei:

- `empty_heap(-Heap)`: Erzeugen eines neuen Heaps
- `add_to_heap(+Heap0,+Cost,+Node,-Heap)`: Hinzufügen eines Knoten Node mit Kosten Cost zu Heap0.
- `get_from_heap(+Heap0,?Cost,-Node,-Heap)`: Entfernen des minimalen Knoten aus Heap0.

A^*

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

g(x): Kosten, um vom Startknoten zu *x* zu gelangen.

h(x): Geschätzte Kosten von *x* zum Ziel.

- nicht überschätzen => zulässig
- h=0 => Breitensuche

Computerspiele

- Geschicklichkeitsspiele (z. B. Pac-Man, Flipper)
- Glücksspiele (z.B. Backgammon, Skat)
- **Strategiespiele**
    - eine Person (z. B. Puzzles, Solitaire, Rubic's Cube).
    - **zwei Personen**
        - unterschiedliche Information über den Spielstand (z. B. Schiffeversenken)
        - **gleiche Information über den Spielstand** (z. B. Schach, Dame, Go, Tic-Tac-Toe, Nim)
    - mehrere Personen (z. B. Halma)

Zweipersonenspiele

- Zwei Spieler: Maximierer und Minimierer.
- Gleiche Information über den Spielstand
- Spieler wechseln sich gegenseitig ab.
- Des einen Gewinn ist des anderen Verlust (*Nullsummenspiel*)

Beispiel: Nimmspiel

- Zu Beginn gibt es
  - einen Haufen mit n Streichhölzchen
  - zwei Spieler
- Jeder zieht abwechselnd
  - mindestens 1 Streichholz, aber
  - maximal die Hälfte aller Streichhölzchen.
- Wer das letzte Streichholz nimmt hat verloren.

Nimmspiel: Repräsentation

Der Zustandsraum ist als UND/ODER-Graph darstellbar. Eine Stellung führt sicher zum Gewinn (ist gewinnbar), wenn

- sie bereits eine terminale Gewinnstellung ist und der Gegner am Zug wäre, oder
- man selbst am Zug ist und eine (ODER) der zulässigen Folgestellungen gewinnbar ist, odr
- der Gegner am Zug ist und alle (UND) zulässigen Folgestellungen gewinnbar sind.

Feststellung von Gewinnpositionen

`terminalwon(Pos)`	Prädikat, das prüft, ob `Pos` eine Gewinnposition ist.
`terminallost(Pos)`	Prädikat, das prüft, ob `Pos` eine Verlustposition ist.
`move(Pos,Pos1)`	Enumerator aller gültigen Nachfolgepositionen von `Pos`.

Algorithmus:

```
won(Pos) :-
   terminalwon(Pos).
won(Pos) :-
   \+ terminallost(Pos),
   move(Pos,Pos1),
   \+ (move(Pos1,Pos2),
        \+ won(Pos2)).
```

Mehrwertige Spielausgänge

Bei vielen Spielen kann der Spielausgang differenzierter bewertet werden.

Tic-Tac-Toe: gewonnen (+1), verloren (-1), tie (0)

Der Maximierer versucht einen möglichst großen Wert am Spielende zu erreichen, der Minimierer einen möglichst kleinen Wert.

In diesen Fällen kann der beste Zug mit dem MINIMAX-Algorithmus (Shannon 1949) ermittelt werden.

Die Bewertung einen Position ergibt sich

- wenn einen Endposition erreicht ist
  aus der statischen Bewertung dieser Endposition
- wenn der Maximierer am Zug ist
  durch Maximumsbildung der Bewertungen der Nachfolgepositionen
- wenn der Minimierer am Zug ist
  durch Minimumsbildung der Bewertungen der Nachfolgepositionen

MINIMAX in Prolog

```
minimax(Pos,BestPos,Wert) :-
    moves(Pos,PosList), !,
    best(PosList,BestPos,Wert)
    ;
    statwert(Pos,Wert).

best([Pos],Pos,Wert) :-
    minimax(Pos,_,Wert),!.

best([Pos1|PosList],BestPos,BestWert) :-
    minimax(Pos1,_,Wert),
    best(PosList,Pos2,Wert2),
    bester(Pos1,Wert1,Pos2,Wert2,
           BestPos,BestWert).

bester(Pos1,Wert1,Pos2,Wert2,Pos1,Wert1) :-
    min_am_zug(Pos1),
    Wert1 > Wert2, !
    ;
    max_am_zug(Pos1),
    Wert1 < Wert2, !.

bester(Pos1,Wert1,Pos2,Wert2,Pos2,Wert2).
```

Beschränkte Vorausschau

Voraussetzung der bisherigen Verfahren war, daß bis zu den Terminalen des Suchbaumes durchgerechnet werden kann.

Problem: fast immer sehr großer Suchraum

- Dame: ca. 10^{40} Knoten
- Schach: ca. 10^{120} Knoten

Lösung: Tiefenbeschneidung und Einführung einer statischen Bewertungsfunktion für nicht-terminale Positionen.

Die Bewertung einen Position ergibt sich

- wenn die maximale Suchtiefe erreicht ist
  aus der statischen Bewertung dieser Position
- wenn der Maximierer am Zug ist
  durch Maximumsbildung der Bewertungen der Nachfolgepositionen
- wenn der Minimierer am Zug ist
  durch Minimumsbildung der Bewertungen der Nachfolgepositionen

Bewertungsfunktionen

Tic-Tac-Toe:

- Bestimmung der Anzahl der potentiellen Dreierketten. Bildung der Differenz zwischen eigener Zahl und der Zahl des Gegeners.

Schach:

- Sicherheit des Königs
- Materialgewichte
- Zentrumskontrolle
- Beweglichkeit

=> Bewertungsfunktion ist durch dynamische Programmierung oder Reinforcement Learning lernbar.

Probleme und Verfeinerungen

- Verbesserung der statischen Bewertungsfunktion
- Verbesserung der Suchraumbeschneidung
  - Effiziente Implementierung des MINIMAX-Algorithmus -> *Alpha/Beta-Pruning*
- Sonderbehandlung “analytischer Positionen”
  - Schach Endspiel (siehe Bratko)

Alpha/Beta-Pruning

Idee: (nur terminaler Fall)

Wenn ein Zweig bereits nachweislich ein Gewinnzug ist, brauchen die anderen nicht mehr untersucht zu werden.

Allgemein: (auch nichtterminale Fälle)

1. Maximierer am Zug (Alpha-Pruning)
   Wenn die bisher untersuchten Zweige dem Maximierer bereits einen Wert von mindestens α garantieren, brauchen diejenigen Zweige, die nachweislich nur einen kleineren Wert erreichen, nicht genau untersucht zu werden. Letzteres für einen Zweig der Fall, sobald ein Folgezug für den Minimierer mit weniger als α bewertet wird.

2. Minimierer am Zug (Beta-Pruning)
   Wenn die bisher untersuchten Zweige dem Minimierer bereits einen Wert von höchstens β garantieren, brauchen diejenigen Zweige, die nachweislich nur einen größeren Wert erreichen, nicht genau untersucht zu werden...

MINIMAX mit Alpha/Beta-Pruning

```
minimax(Pos,A,B,BPos,BVal) :-
    moves(Pos,PosL),
    boundedbest(PosL,A,B,BPos,BVal)
    ;
    statwert(Pos,BVal).

boundedbest([Pos|PosL],A,B,BPos,BVal) :-
    minimax(Pos,A,B,_,Val),
    goodenough(PosL,A,B,Pos,Val,BPos,BVal).

goodenough([],_,_,Pos,Val,Pos,Val).

goodenough(_,A,B,Pos,Val,Pos,Val) :-
    min_to_move(Pos), Val > B, !
    ;
    max_to_move(Pos), Val < A, !.

goodenough(PosL,A,B,Pos,Val,BPos,BVal) :-
    newbounds(A,B,Pos,Val,NewA,NewB),
    boundedbest(PosL,NewA,NewB,Pos1,Val1),
    bester(Pos,Val,Pos1,Val1,BPos,Bval).

newbounds(A,B,Pos,Val,Val,B) :-
    min_to_move(Pos), Val > A, !.
newbounds(A,B,Pos,Val,A,Val) :-
    max_to_move(Pos), Val < B, !.
newbounds(A,B,_,_,A,B).
```

Komplexität der Alpha/Beta-Suche