
Algorithmen und Datenstrukturen

Suchraumbeschneidung, Alpha-Beta-Pruning

Prof. Dr. Ralf Möller

Universität zu Lübeck

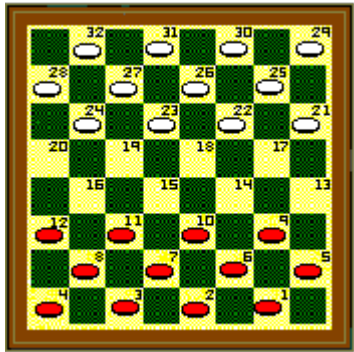
Institut für Informationssysteme

Magnus Bender und Malte Luttermann
(Übungen)

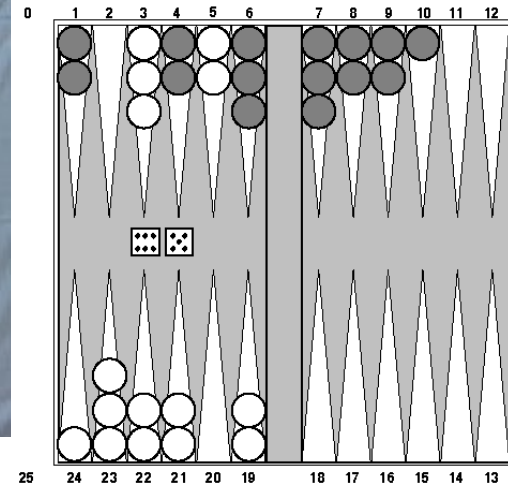
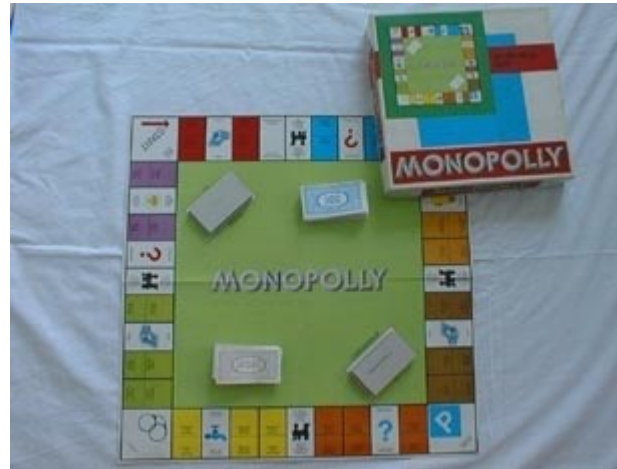
sowie viele Tutoren

Suchgraphen für 2-Personen-Nullsummenspiele

The board set for play



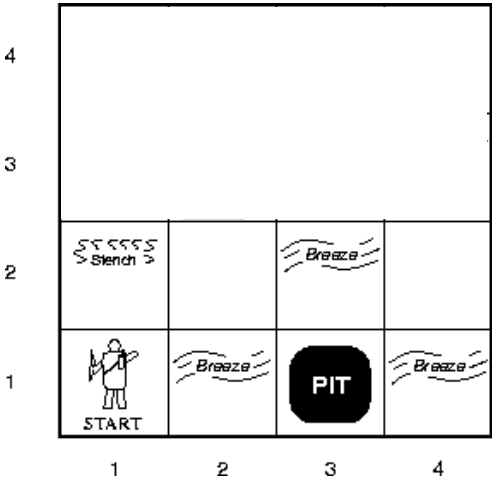
Red to play



Typen von Spielen



	deterministisch	zufällig
Perfekte Information	Schach, Dame, Go, Othello	Backgammon, Monopoly
	Wumpus	Bridge, Poker, Scrabble



Unvollständige Information



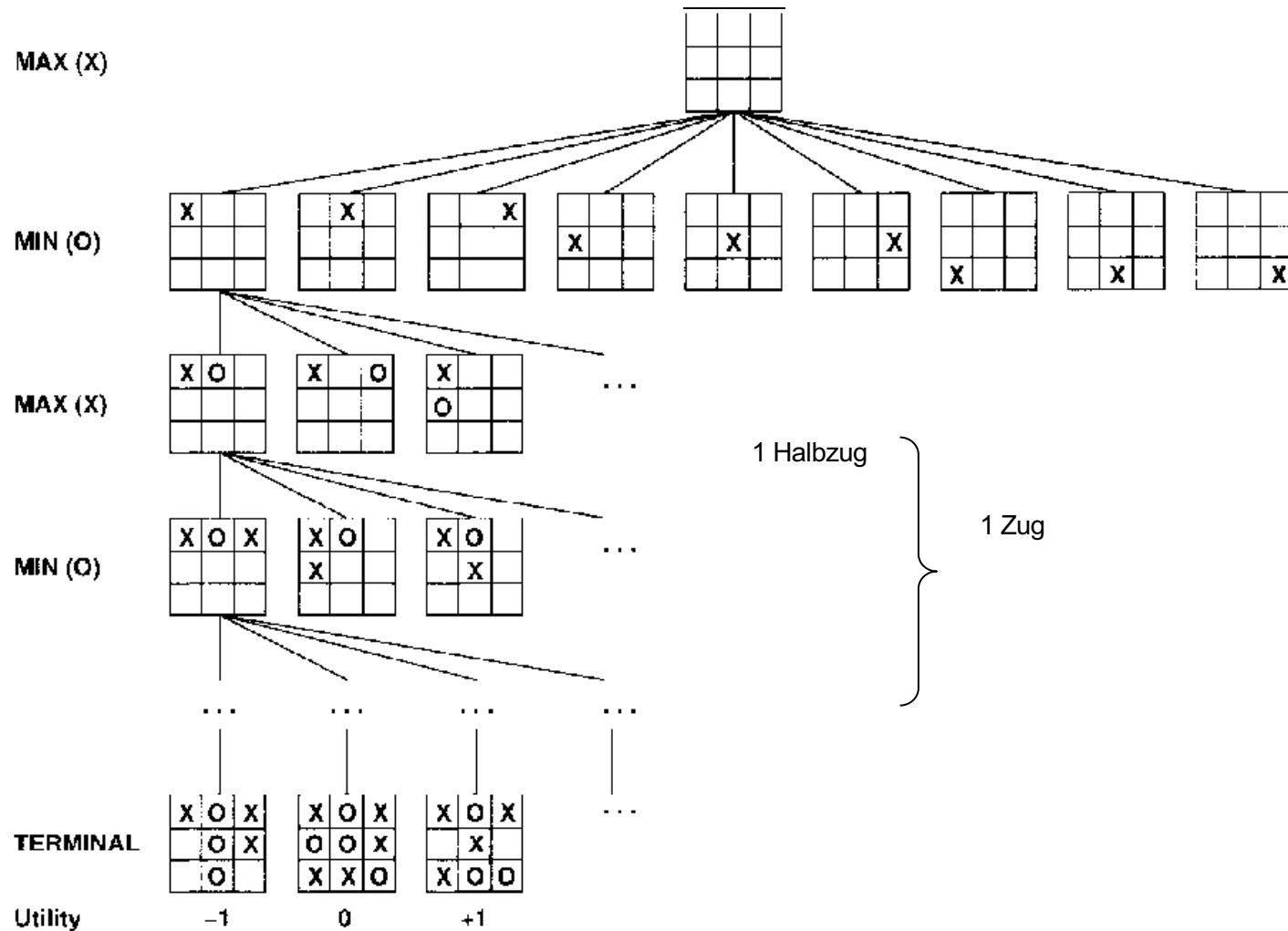
Zweipersonen-Spiele

- Ein Spiel als Suchproblem (z.B. Matt in drei Zügen):
 - Anfangszustand: Brettposition und erster Spieler
 - Operatoren (Züge): In Brettposition legale Züge
 - Endzustand: Bedingungen für Spielende
 - Nützlichkeitsfunktion: Num. Wert für Ausgang des Spiels
-1, 0, 1
für verloren, unentschieden, gewonnen
(Payoff- oder Utility-Funktion)

Entwurfsmuster Suchraumbeschneidung

- Unser Problem: Bestimme besten Zug
- Entscheidung des Problems über Suche in einem Graphen mit der Idee, den...
- ...Suchraum systematisch zu beschneiden...
- ohne die richtige Lösung zu verlieren
- Später:
Suchraumbeschneidung als Approximation
aber mit praktikablem Laufzeitverhalten

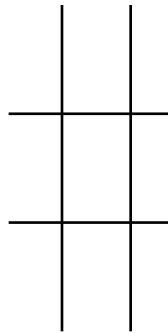
Beispiel: Tic-Tac-Toe



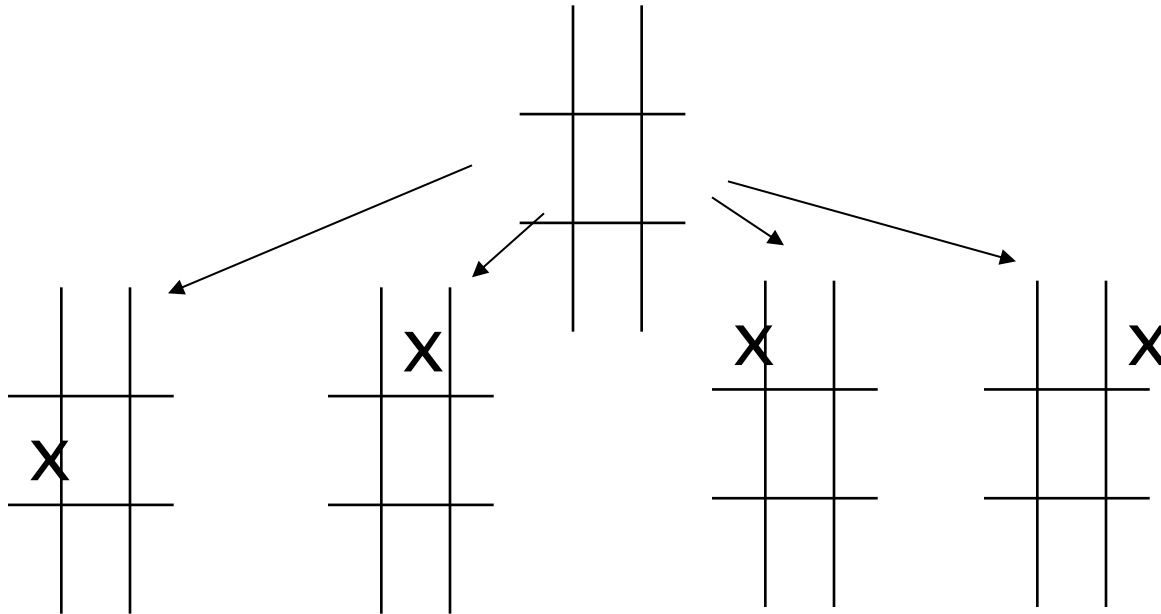
Minimax-Algorithmus

- Optimales Spiel für deterministische Umgebungen und perfekte Info
- **Basisidee:** Wähle Zug mit höchstem Nützlichkeitswert in Relation zum besten Spiel des Gegners
- **Algorithmus:**
 1. Generiere Spielbaum vollständig
 2. Bestimme Nützlichkeit der Endzustände
 3. Propagiere Nützlichkeitswerte im Baum nach oben durch Anwendung der Operatoren MIN und MAX auf die Knoten in der jeweiligen Ebene
 4. An der Wurzel wähle den Zug mit maximalem Nützlichkeitswert
- Schritte 2 und 3 nehmen an, dass der Gegner perfekt spielt.

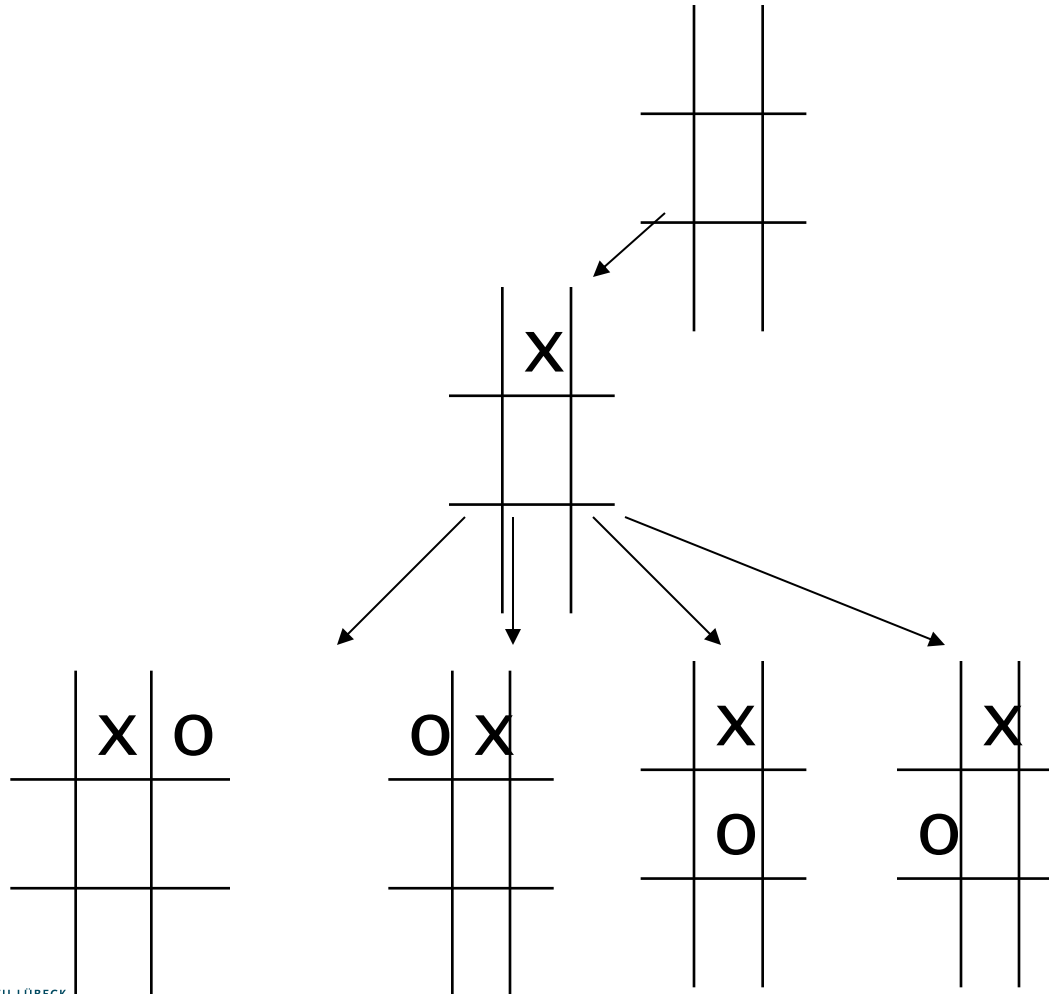
Erzeuge Spielbaum



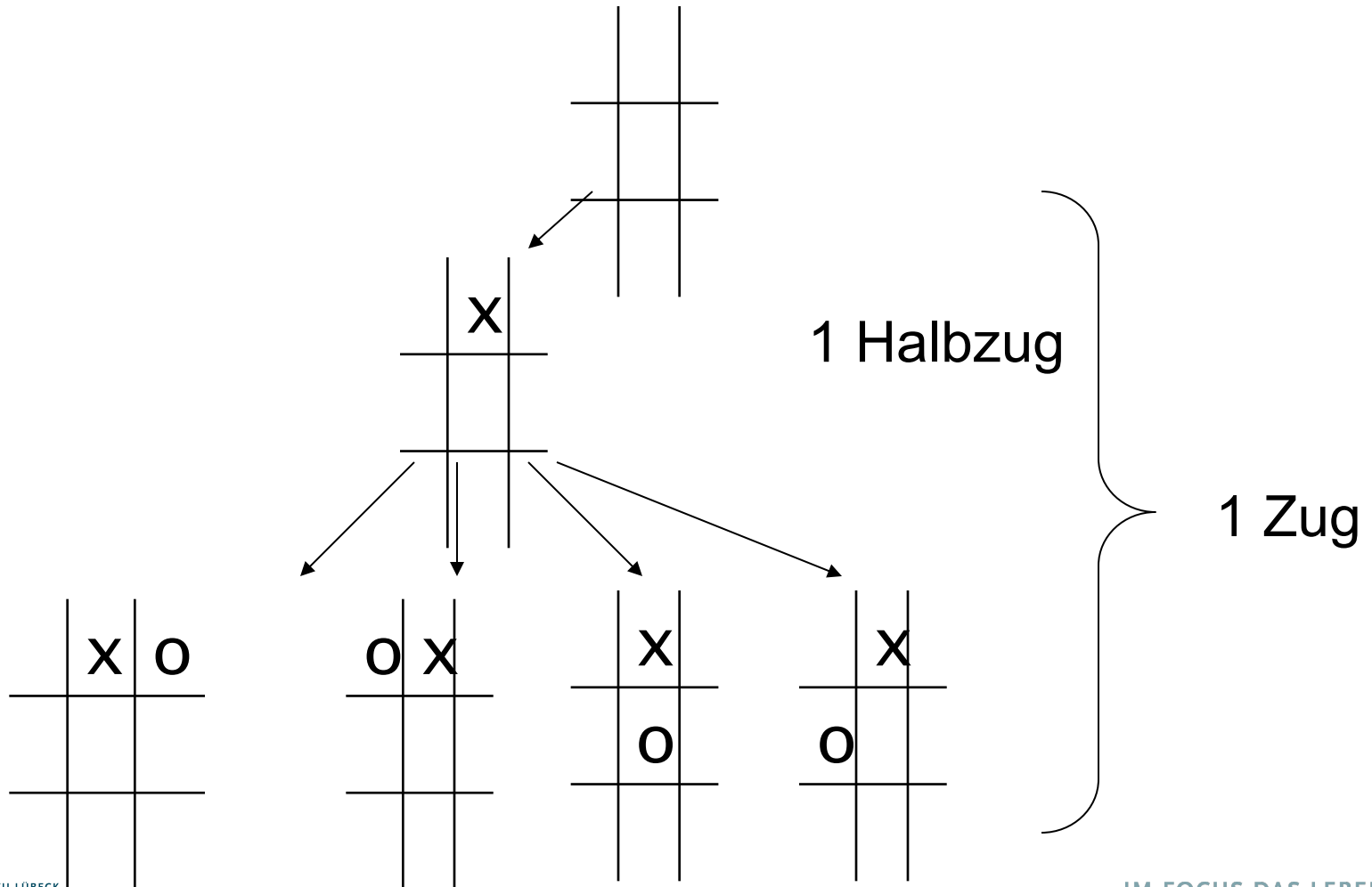
Erzeuge Spielbaum



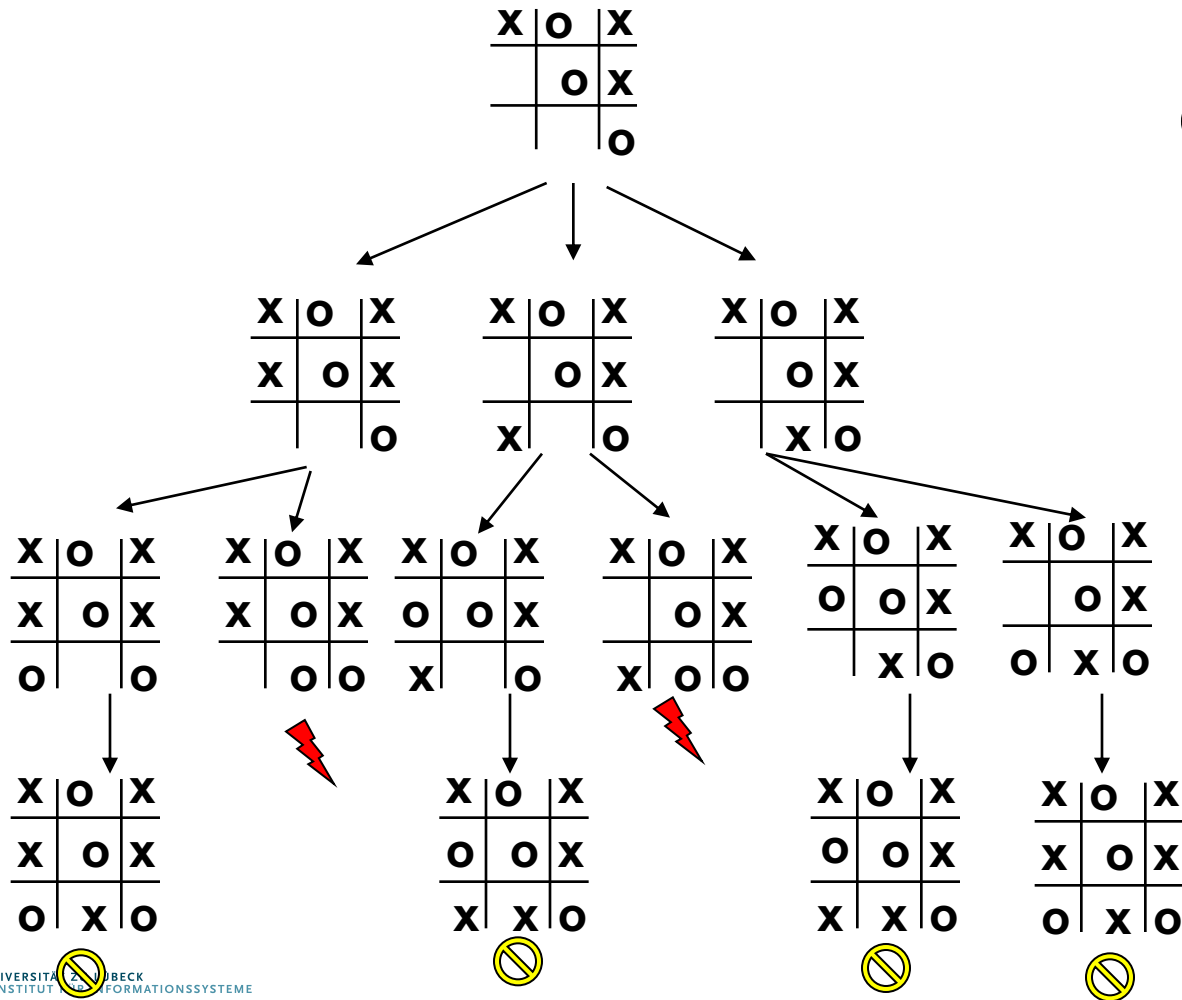
Erzeuge Spielbaum



Erzeuge Spielbaum



Ein Teilbaum



gewonnen

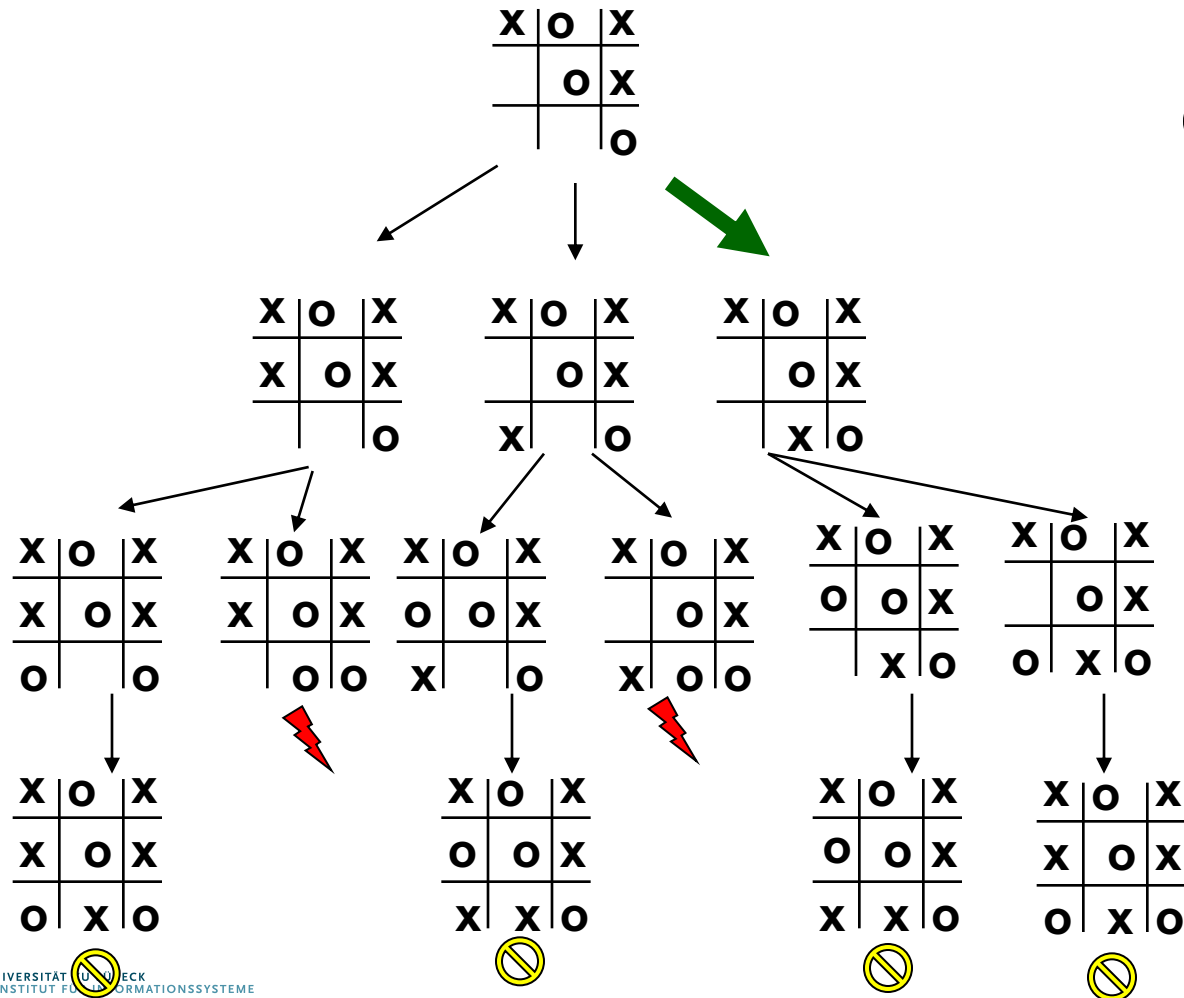


verloren



unentschieden

Was ist ein guter Zug?



gewonnen

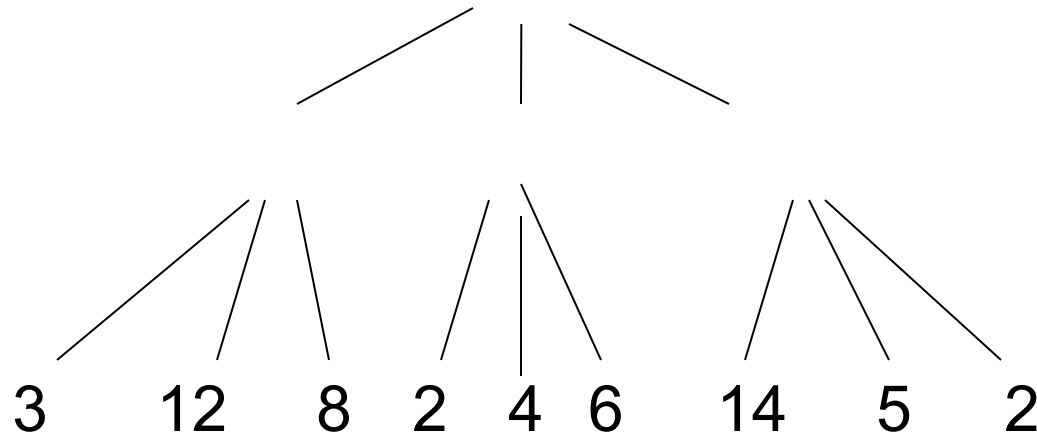


verloren



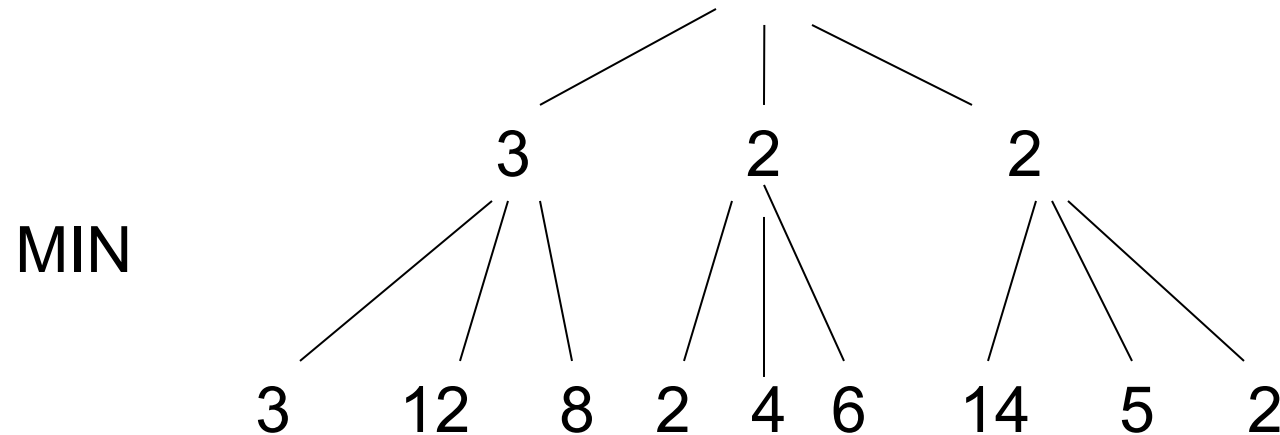
unentschieden

Minimax



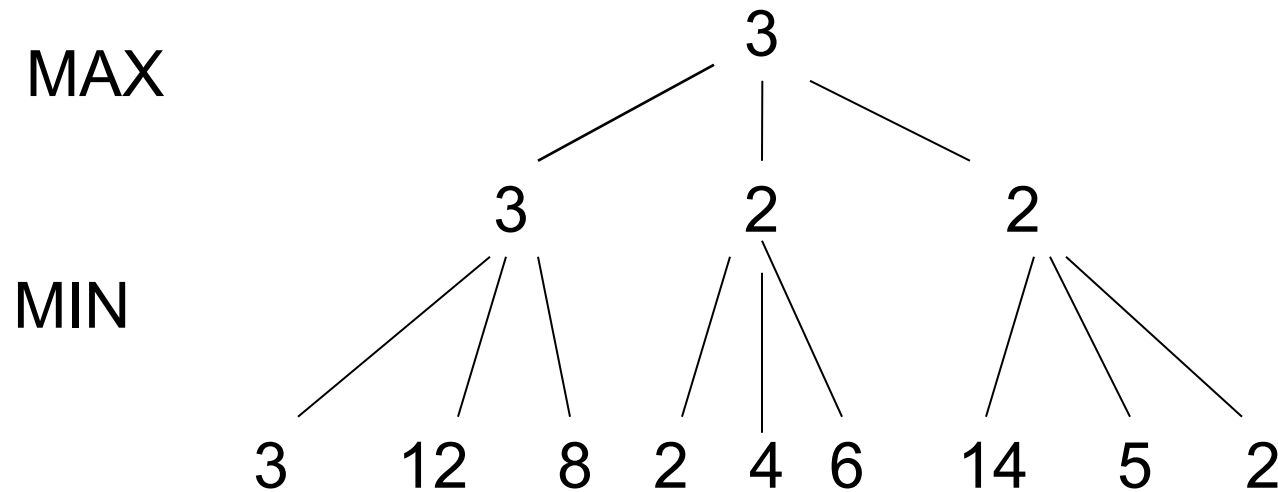
- Minimiere Gewinnmöglichkeiten für den Gegner
- Maximiere eigene Gewinnmöglichkeiten

Minimax



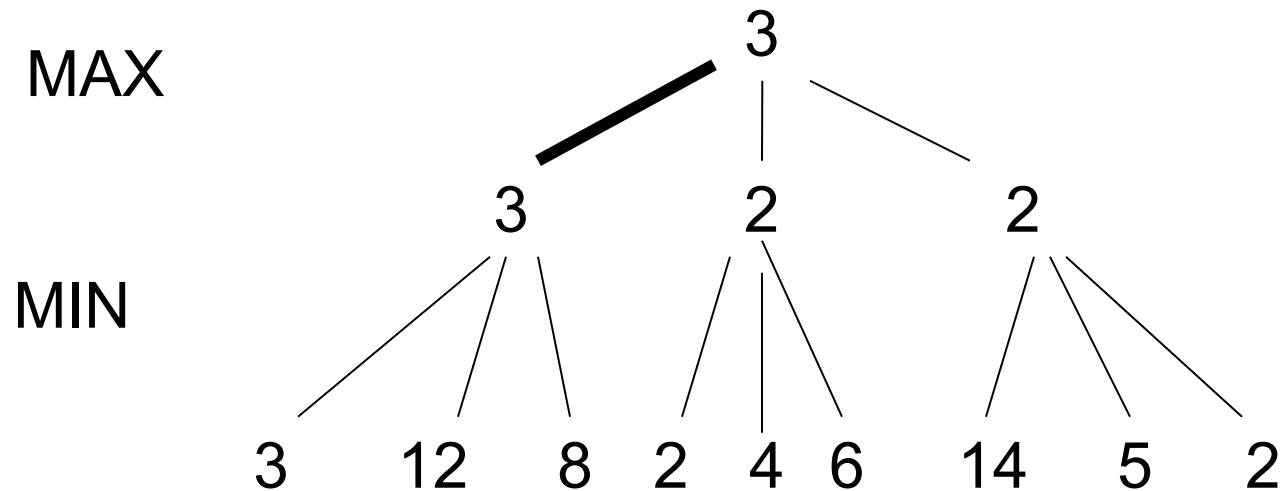
- Minimiere Gewinnmöglichkeiten für den Gegner
- Maximiere eigene Gewinnmöglichkeiten

Minimax



- Minimiere Gewinnmöglichkeiten für den Gegner
- Maximiere eigene Gewinnmöglichkeiten

Minimax



- Minimiere Gewinnmöglichkeiten für den Gegner
- Maximiere eigene Gewinnmöglichkeiten

Minimax: Rekursive Implementierung

Vom aktuellen Zustand
durch eine Aktion
erreichbare Zustände

```
function minmax_decision(t)
    max_value(t.root)      # Berechne alle Werte
                           # Bestimme "besten" nachfolgenden Zustand
    return findmax((s_state) -> s_state.v, t.root.subseq_s)
end
```

```
function min_value(state)
    if is_terminal(state)
        return value(state)
    end
    v = typemax(Int)
    for s_state in state.subseq_s
        v = minimum((v,
                     max_value(s_state)))
    end
    state.v = v
    return v
end
```

```
function max_value(state)
    if is_terminal(state)
        return value(state)
    end
    v = typemin(Int)
    for s_state in state.subseq_s
        v = maximum((v,
                     min_value(s_state)))
    end
    state.v = v
    return v
end
```

Vollständig: ?
Optimal: ?

Zeitkomplexität: ?
Platzkomplexität: ?

Minimax: Rekursive Implementation

```
function minmax_decision(t)
    max_value(t.root)      # Berechne alle Werte
                           # Bestimme "besten" nachfolgenden Zustand
    return findmax((s_state) -> s_state.v, t.root.subseq_s)
end
```

```
function min_value(state)
    if is_terminal(state)
        return value(state)
    end
    v = typemax(Int)
    for s_state in state.subseq_s
        v = minimum((v,
                     max_value(s_state)))
    end
    state.v = v
    return v
end
```

```
function max_value(state)
    if is_terminal(state)
        return value(state)
    end
    v = typemin(Int)
    for s_state in state.subseq_s
        v = maximum((v,
                     min_value(s_state)))
    end
    state.v = v
    return v
end
```

Vollständig: Ja, für endl. Zust.raum
Optimal: Ja

Zeitkomplexität: $O(bm)$
Platzkomplexität: $O(bm)$ (= DFS)

m = Suchtiefe bis Endzustände erreicht
 b = Verzweigungsfaktor (braching)

Spielen als naive Suche?

- **Komplexität:** Viele Spiele haben einen großen Suchraum
 - **Schach:** $b = 35, m = 100$ $\#Knoten = 35^{100}$
Falls Betrachtung eines Knotens 1 ns benötigt,
braucht jeder Zug **10^{50} Jahrhunderte**
zur Berechnung
- Endzustände können in vertretbarer Zeit
nicht alle generiert werden

Algorithmen für perfektes Spiel: John von Neumann, **1944**

Endlicher Horizont, approximative Zustandsbewertung: Zuse **1945** , Shannon **1950** , Samuel **1952-57**

Suchbaumabschnidungen zur Einsparung von Zeit: McCarthy **1956**

Lösungsansätze

Reduktion des Ressourcenproblems (Zeit, Speicher):

1. **Suchraumbeschneidung (Pruning):** macht Suche schneller durch Entfernen von Teilen des Suchbaums, in denen beweisbar keine bessere als die aktuell beste Lösung gefunden werden kann
2. **Bewertungsfunktion:** Heuristik zur Bewertung der Nützlichkeit eines Spielzustands (Knoten) ohne vollständige Suche

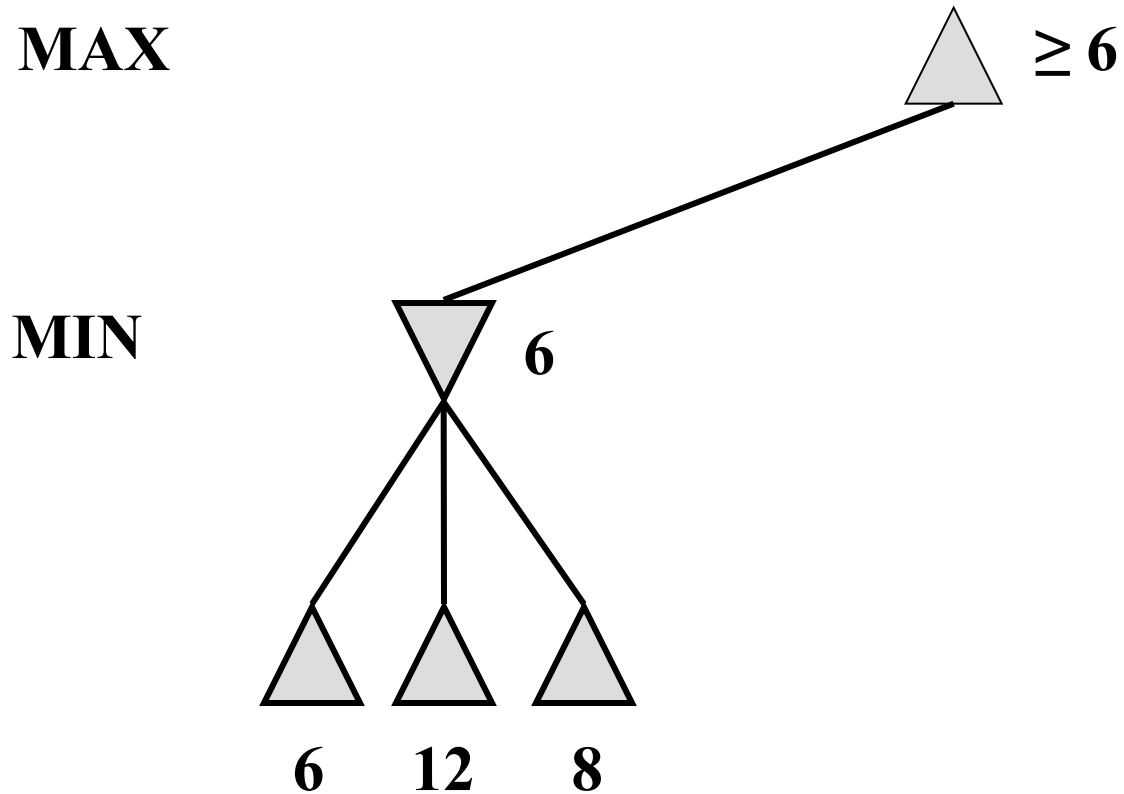
1. α - β -Pruning

- **Pruning:** Elimination eines Zweigs des Suchbaums, ohne vollständige Untersuchung jedes Knotens darunter (vgl. auch A*)
- **α - β -Pruning:** beschneide Teile des Suchbaums, die den Nützlichkeitswerte eines Max- oder Min-Knotens nicht verbessern können

(wobei nur bisher betrachtete Knotennützlichkeitswerte eine Rolle spielen)

- Geht das? Ja (s.u.).
 - Der Verzweigungsfaktor wird in etwa von b auf $\sqrt{b}$ reduziert
 - Die Suchtiefe wird gegenüber Minimax ca. verdoppelt

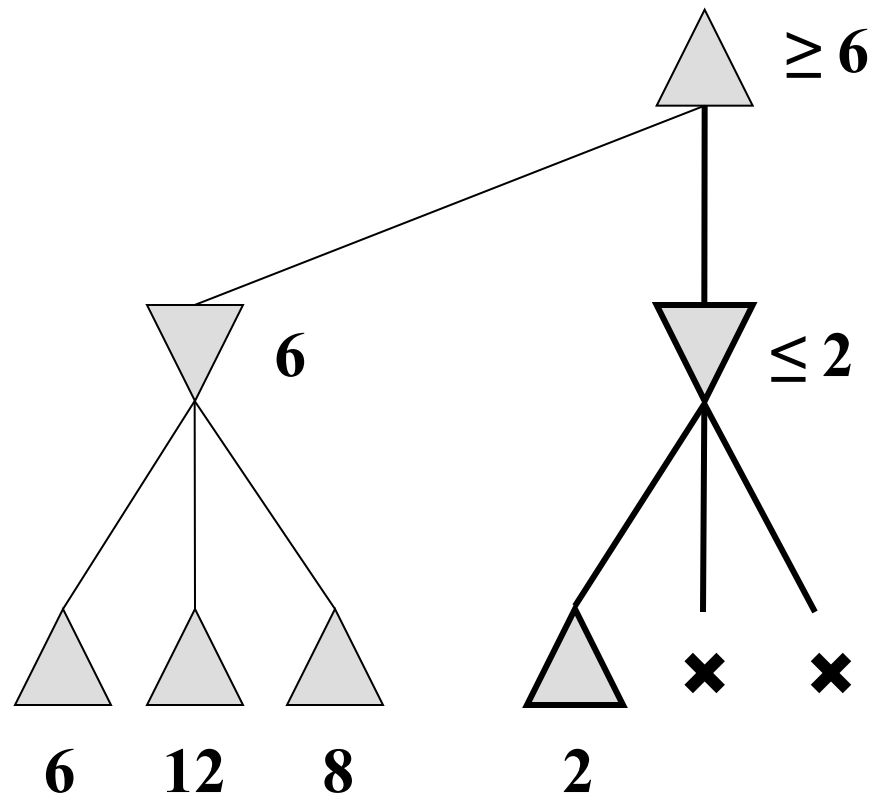
Pruning: Beispiel



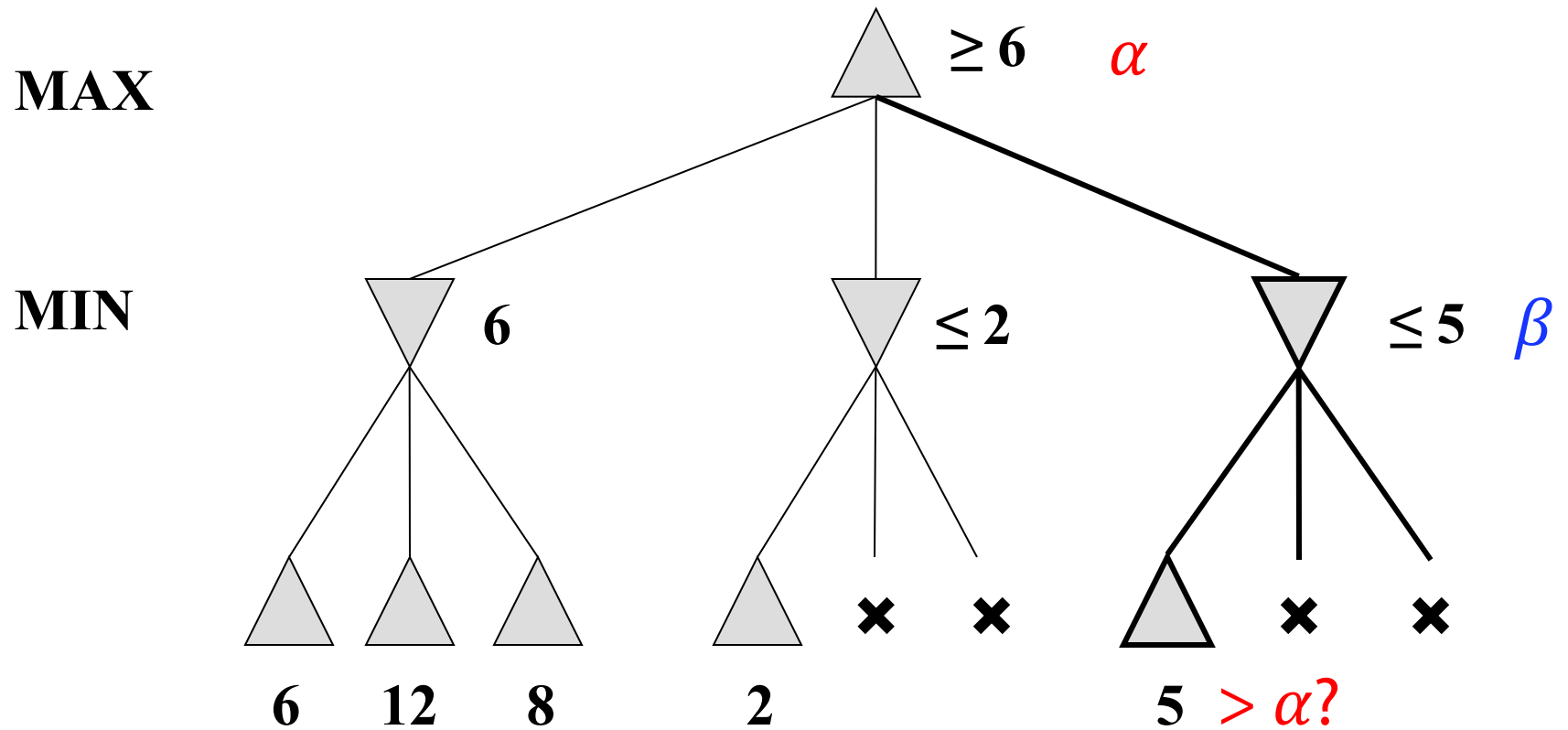
Pruning: Beispiel

MAX

MIN



α - β -Pruning



$$\begin{aligned}
 \text{MINIMAX}(\text{root}) &= \max(\min(6, 12, 8), \min(2, a, b), \min(5, b, d)) \\
 &= \max(6, z, y) \quad \text{wobei } z = \min(2, a, b) \leq 2 \quad \text{und} \quad y = \min(5, b, d) \leq 5 \\
 &= 6
 \end{aligned}$$

α - β -Pruning: Algorithmus

- Weil Minimax Tiefensuche betreibt, betrachten wir die Knoten auf einem gegebenen Pfad im Baum
- Für jeden Pfad, speichern wir:
 - α : Bewertung der bislang besten Wahl für MAX
 - β : Bewertung der bislang besten Wahl für MIN

Suche mit Pfadbeschneidung

```
function alpha_beta_search(t)
  alpha = typemin(Int)
  beta = typemax(Int)

  # Berechne alle Werte (mit Pruning)
  best = max_value(t.root, alpha, beta)
  # Bestimme "besten" nachfolgenden Zustand
  return findmax((s) -> s.v, t.root.subseq_s)
end
```

Von mehreren Forschern **um 1960** entwickelt.

MAX-VALUE und MIN-VALUE mit α - β -Pruning

```
function min_value(state, a, b)
    if is_terminal(state)
        return value(state)
    end

    v = typemax(Int)
    for s in state.subseq_s
        v = minimum((
            v,
            max_value(s, a, b)
        ))
        if v <= a
            break
        end
        b = minimum((b, v))
    end
    state.alpha = alpha
    state.beta = beta
    state.v = v
    return v
end
```

```
function max_value(state, a, b)
    if is_terminal(state)
        return value(state)
    end

    v = typemin(Int)
    for s in state.subseq_s
        v = maximum((
            v,
            min_value(s, a, b)
        ))
        if v >= b
            break
        end
        a = maximum((a, v))
    end
    state.alpha = a
    state.beta = b
    state.v = v
    return v
end
```

α - β -Suchalgorithmus

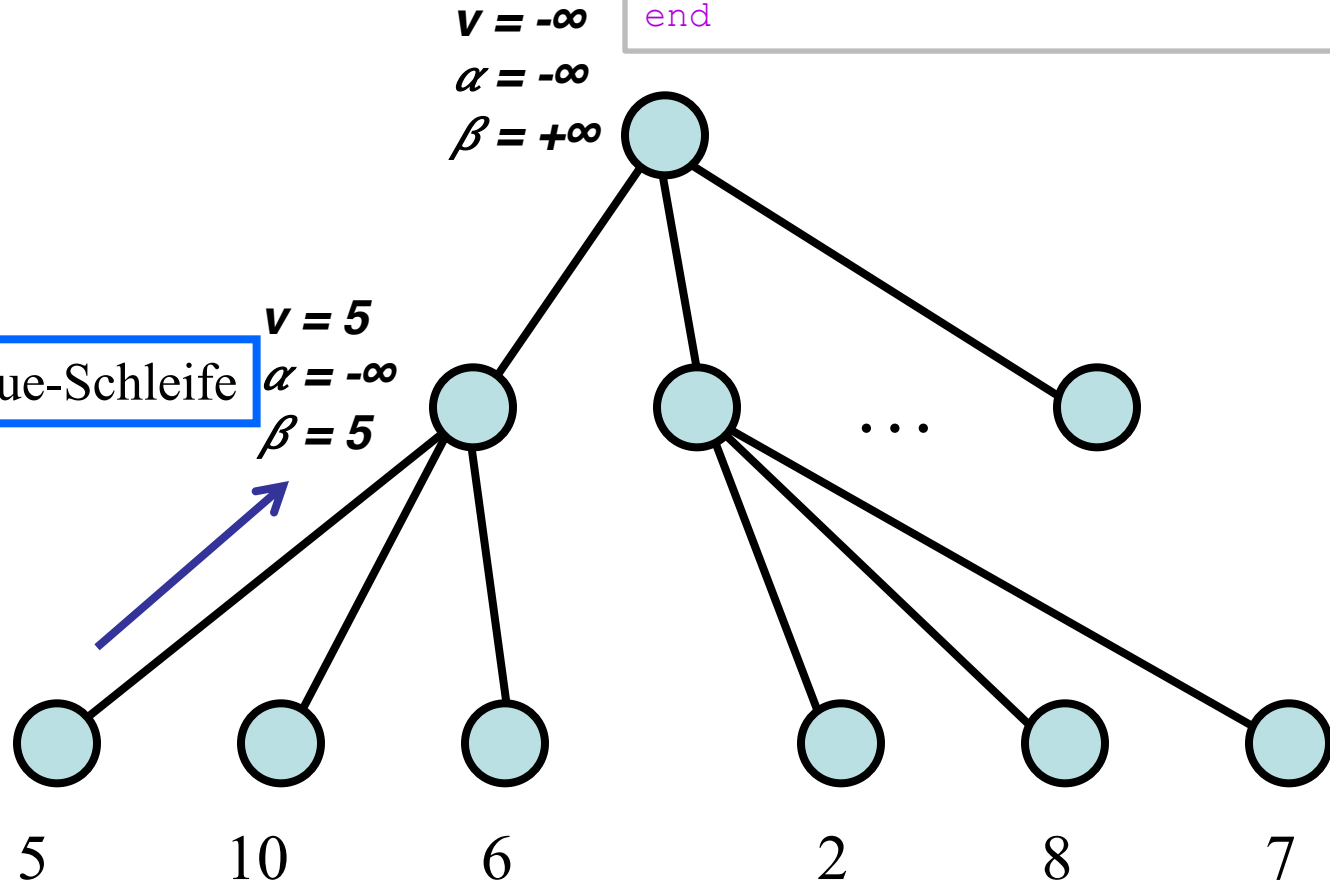
In Min-Value:

```
v = typemax(Int)
for s in state.subseq_s
  v = minimum(( v, max_value(s, a, b)))
  if v <= a    break end
  b = minimum((b, v))
end
```

MAX

MIN Min-Value-Schleife

MAX



α - β -Suchalgorithmus

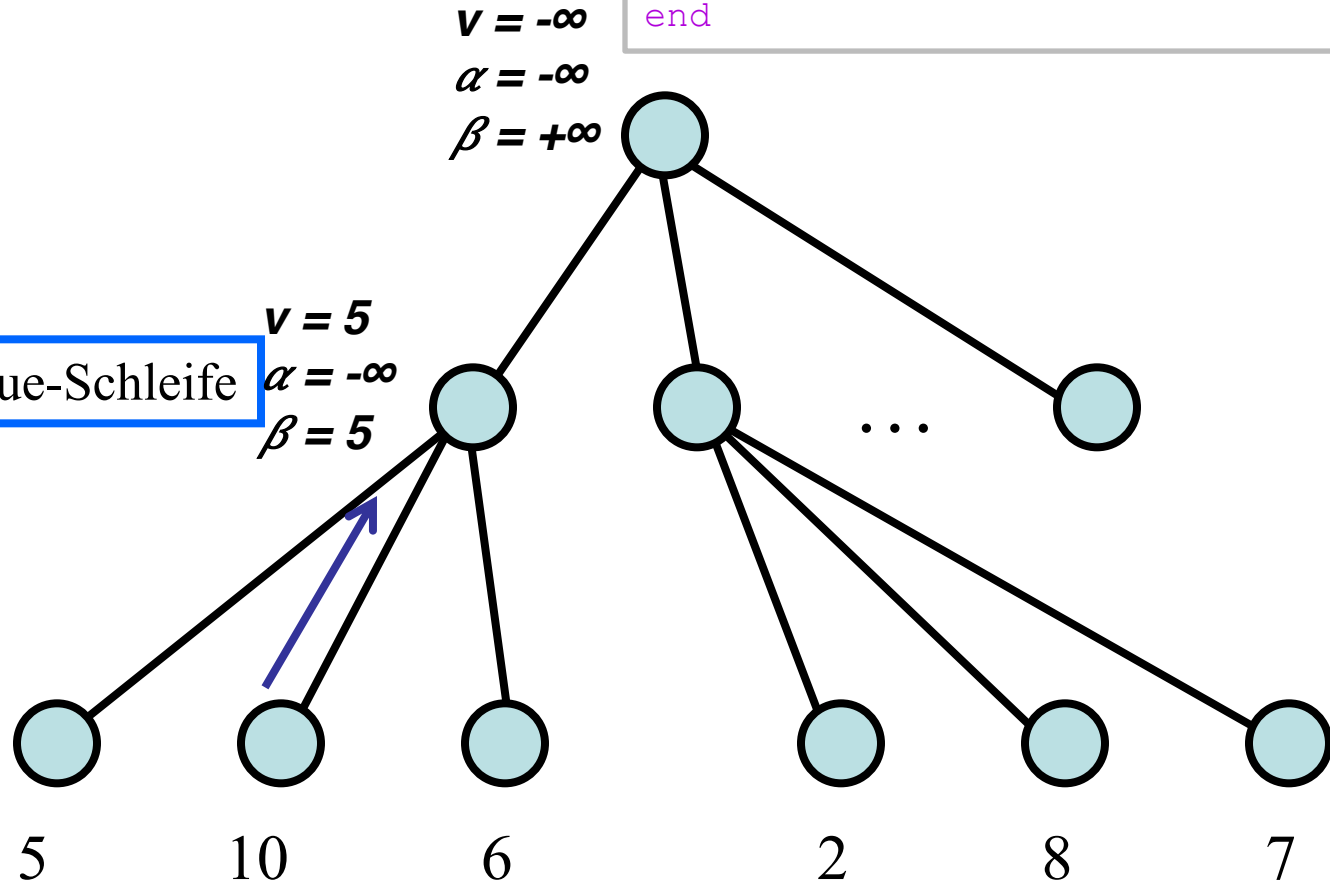
In Min-Value:

```
v = typemax(Int)
for s in state.subseq_s
  v = minimum(( v, max_value(s, a, b)))
  if v <= a    break end
  b = minimum((b, v))
end
```

MAX

MIN Min-Value-Schleife

MAX



α - β -Suchalgorithmus

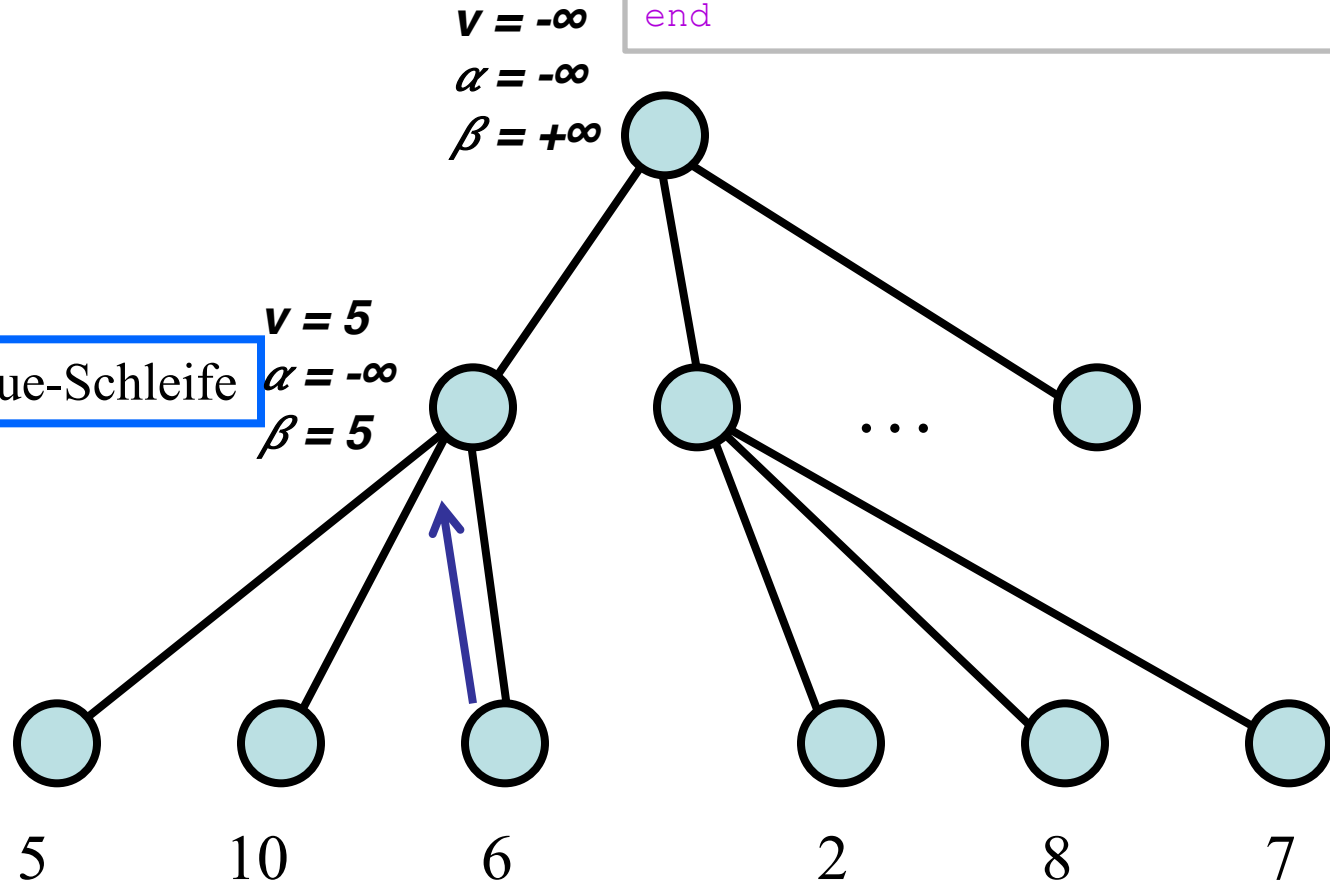
In Min-Value:

```
v = typemax(Int)
for s in state.subseq_s
  v = minimum(( v, max_value(s, a, b)))
  if v <= a    break end
  b = minimum((b, v))
end
```

MAX

MIN Min-Value-Schleife

MAX



α - β -Suchalgorithmus

In Max-Value:

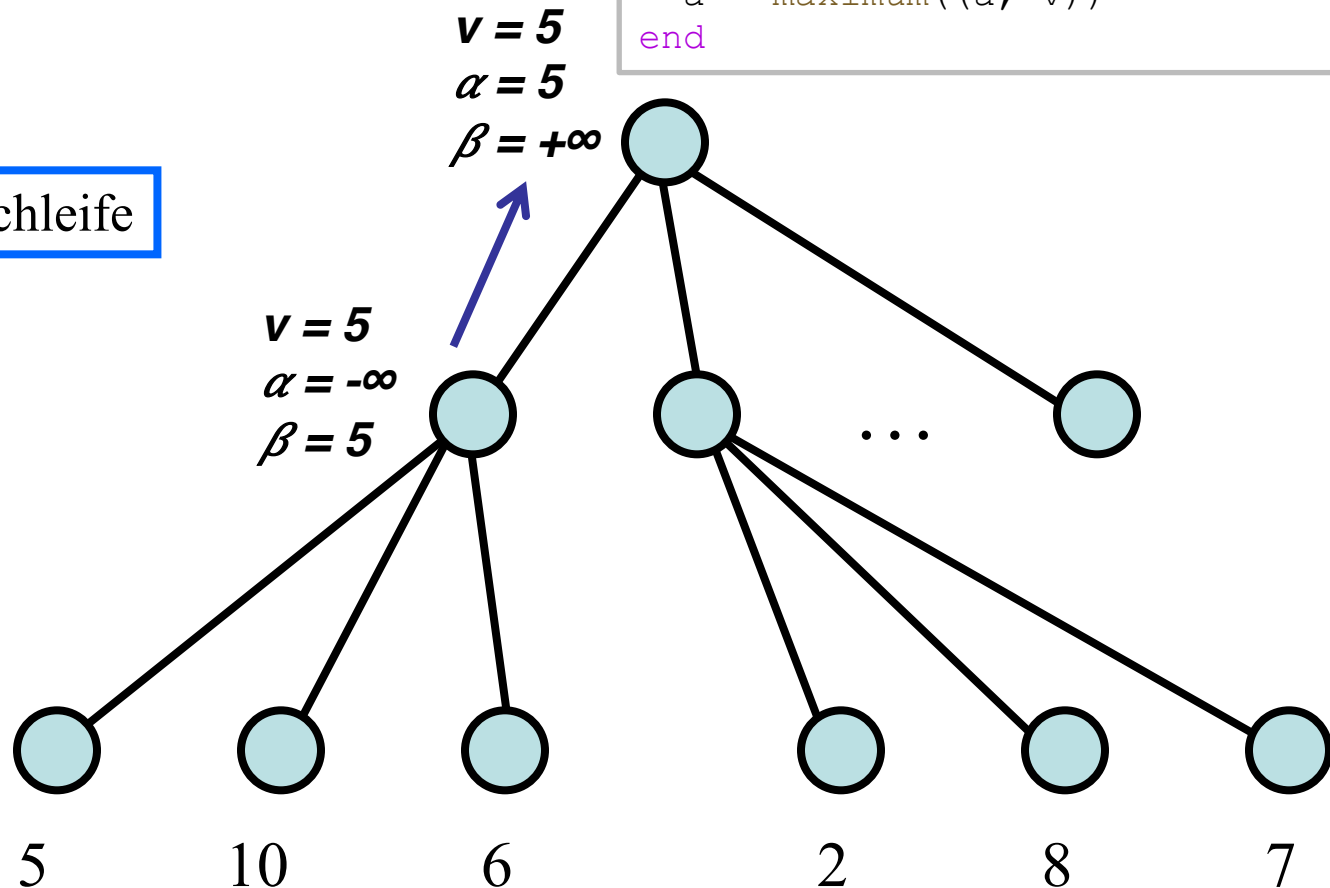
```
v = typemin(Int)
for s in state.subseq_s
  v = maximum(( v, min_value(s, a, b)))
  if v >= b break end
  a = maximum((a, v))
end
```

MAX

Max-Value-Schleife

MIN

MAX



α - β -Suchalgorithmus

In Max-Value:

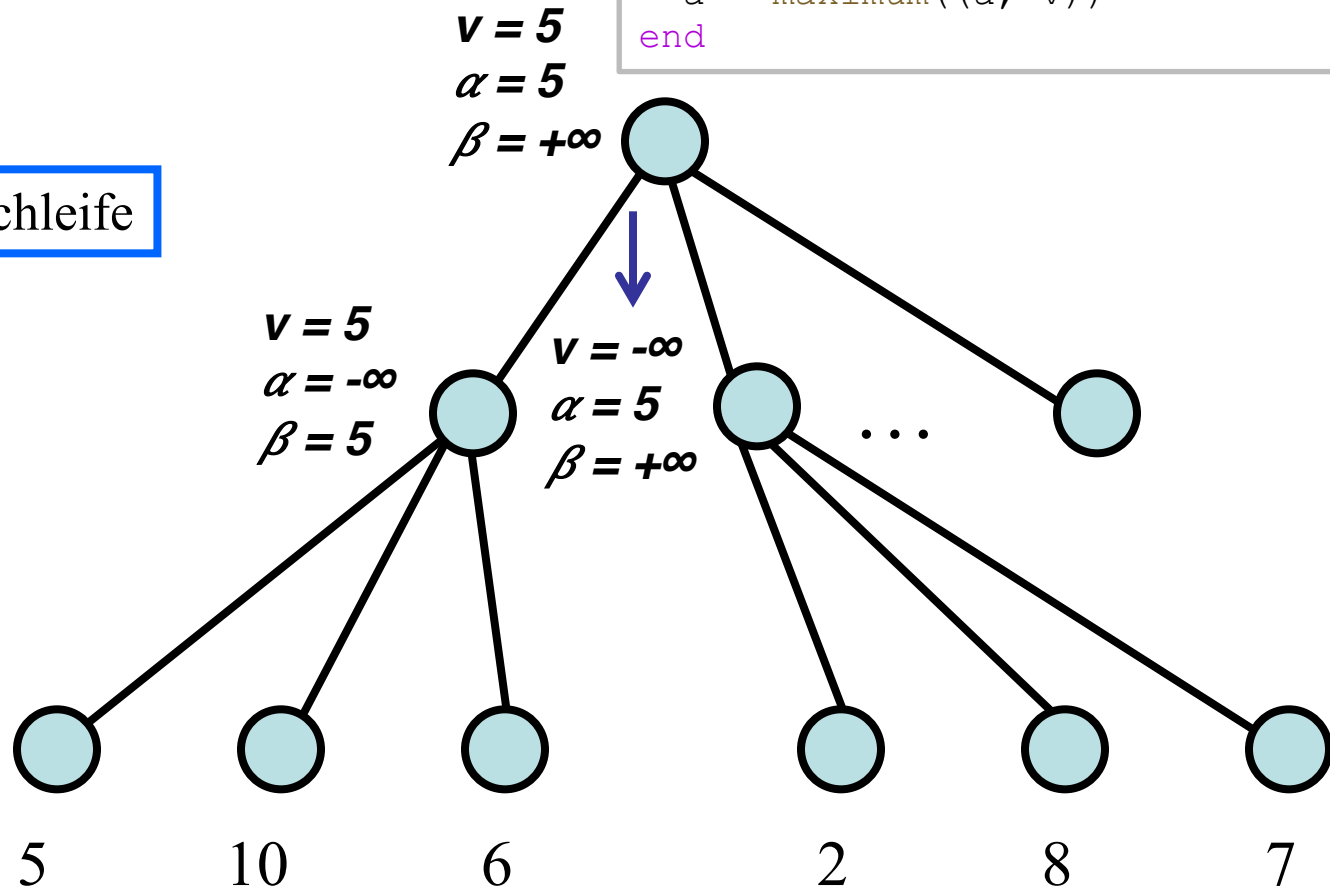
```
v = typemin(Int)
for s in state.subseq_s
  v = maximum(( v, min_value(s, a, b)))
  if v >= b break end
  a = maximum((a, v))
end
```

MAX

Max-Value-Schleife

MIN

MAX



α - β -Suchalgorithmus

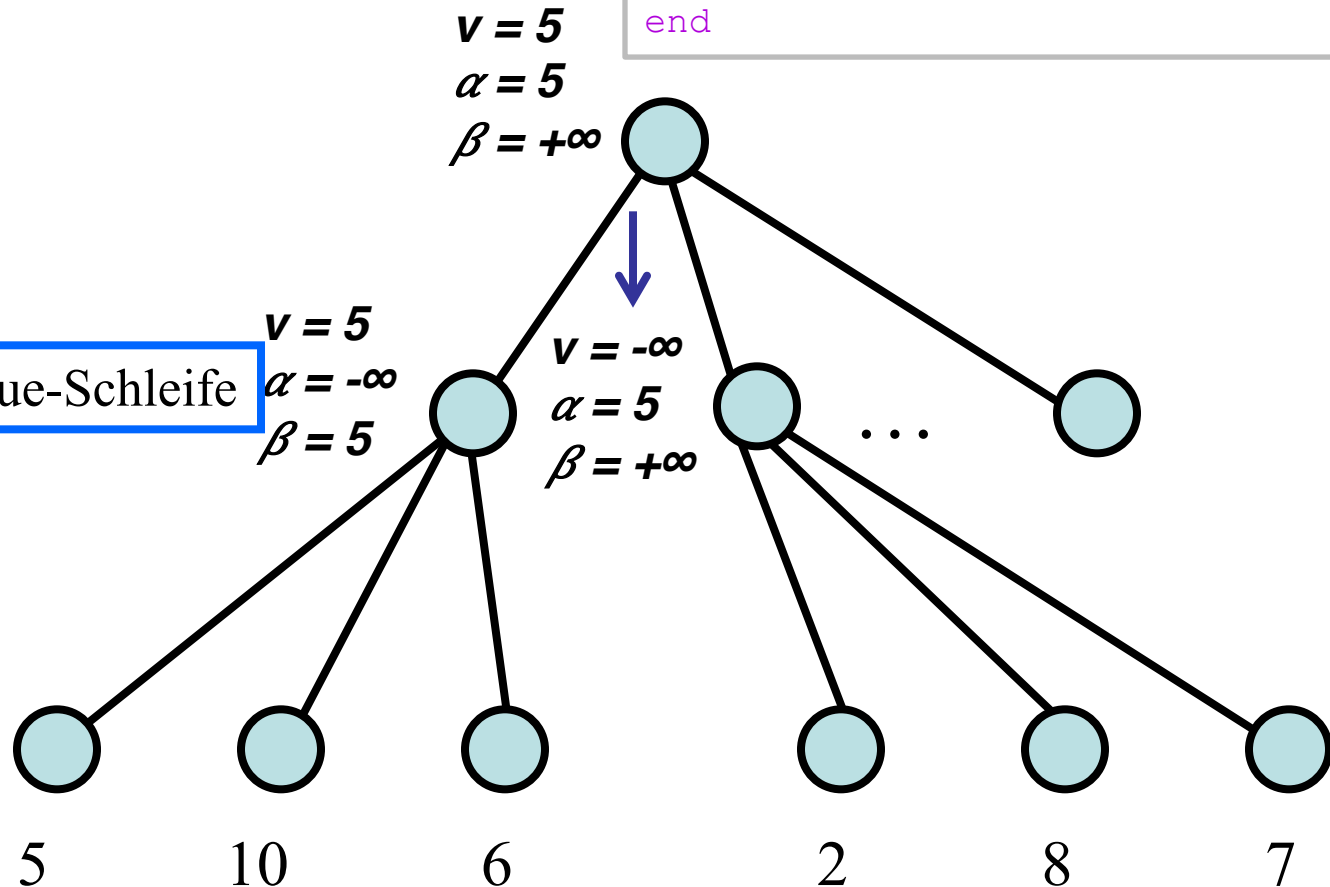
In Min-Value:

```
v = typemax(Int)
for s in state.subseq_s
  v = minimum(( v, max_value(s, a, b)))
  if v <= a    break end
  b = minimum((b, v))
end
```

MAX

MIN Min-Value-Schleife

MAX



α - β -Suchalgorithmus

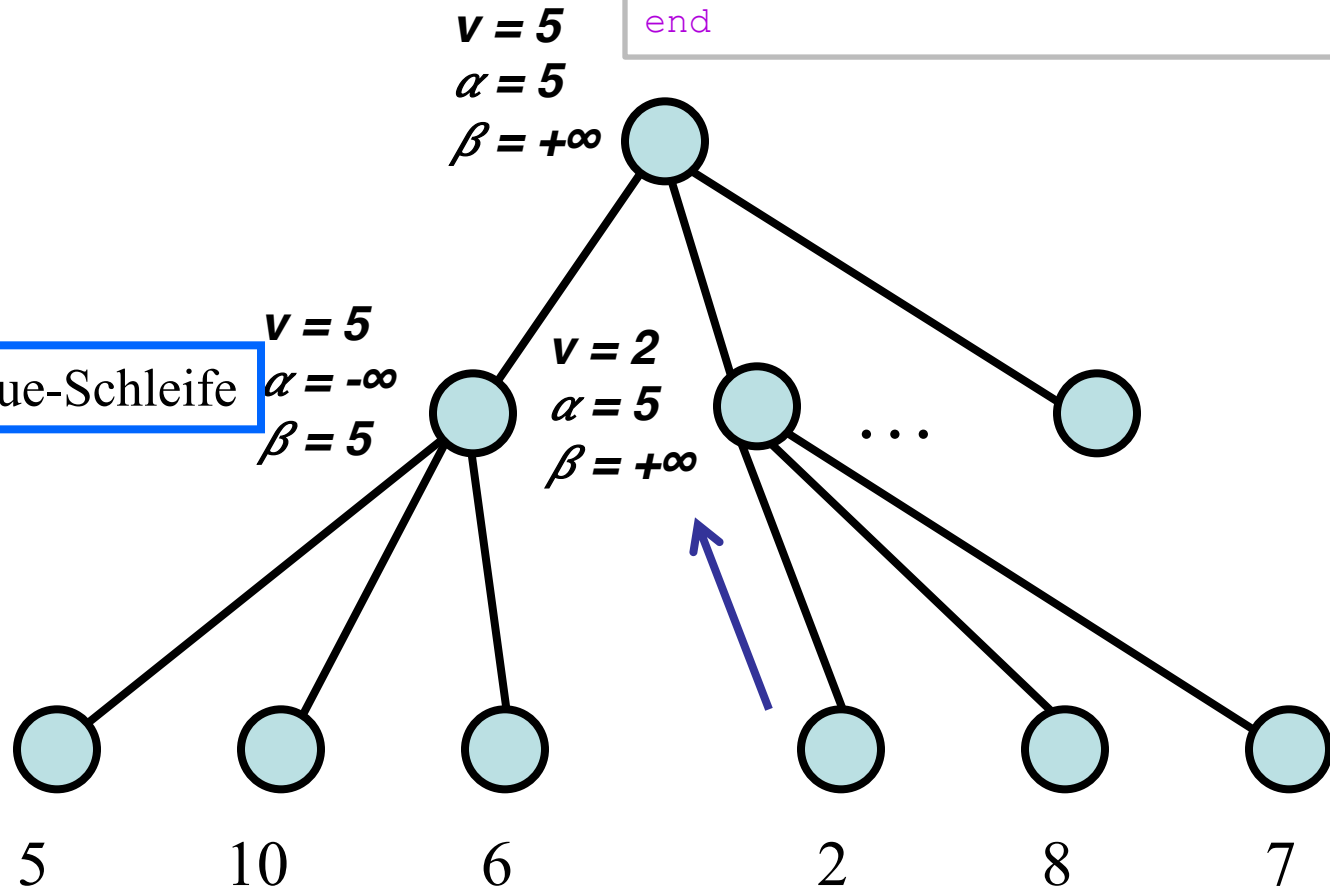
In Min-Value:

```
v = typemax(Int)
for s in state.subseq_s
  v = minimum(( v, max_value(s, a, b)))
  if v <= a    break end
  b = minimum((b, v))
end
```

MAX

MIN Min-Value-Schleife

MAX



α - β -Suchalgorithmus

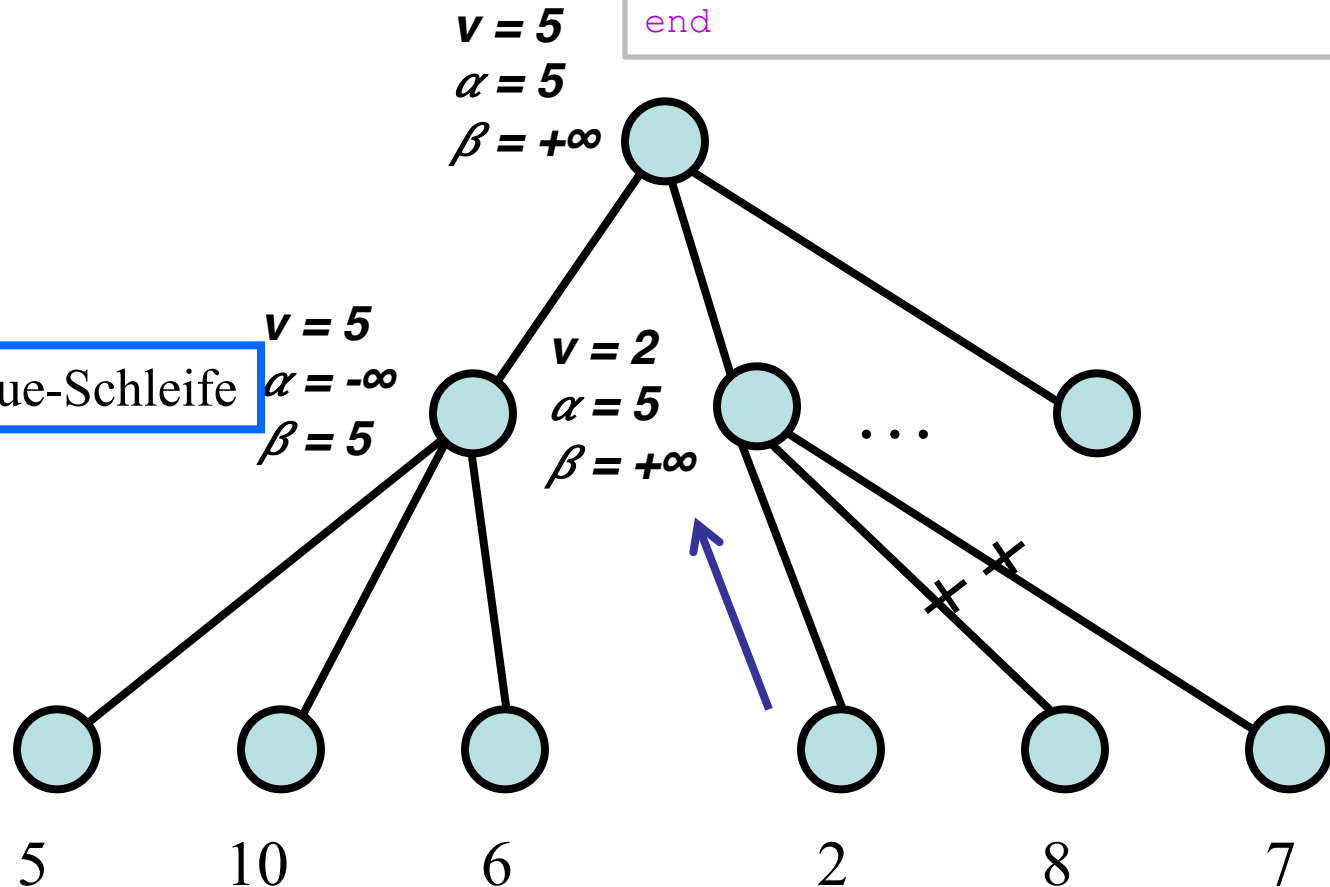
In Min-Value:

```
v = typemax(Int)
for s in state.subseq_s
  v = minimum(( v, max_value(s, a, b)))
  if v <= a    break end
  b = minimum((b, v))
end
```

MAX

MIN Min-Value-Schleife

MAX



α - β -Suchalgorithmus

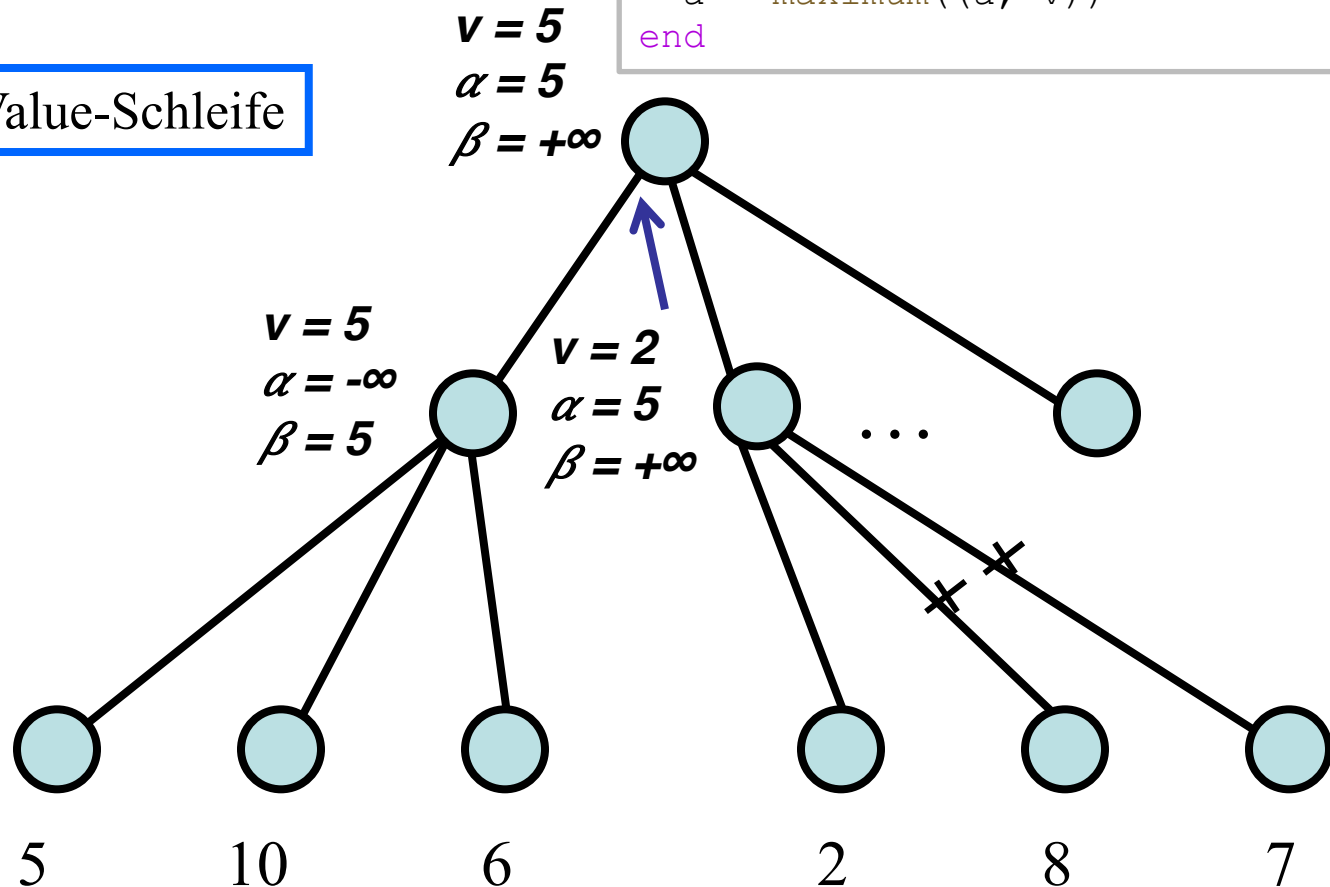
In Max-Value:

```
v = typemin(Int)
for s in state.subseq_s
  v = maximum(( v, min_value(s, a, b)))
  if v >= b break end
  a = maximum((a, v))
end
```

MAX Max-Value-Schleife

MIN

MAX



α - β -Suchalgorithmus: Eigenschaften

- Pruning hat keinen Einfluss auf das Endergebnis!!!
- Eine gute Reihenfolge der Betrachtung möglicher Züge erhöht die Effektivität vom Pruning
- Mit *perfekter Ordnung*, Zeitkomplexität = $O(b^{m/2})$
 - Empirisch ermittelt
 - Verdopplung der Suchtiefe
 - Gute Heuristik zur Ordnung nötig
 - Tiefe 8 erreichbar => guter Schachspieler

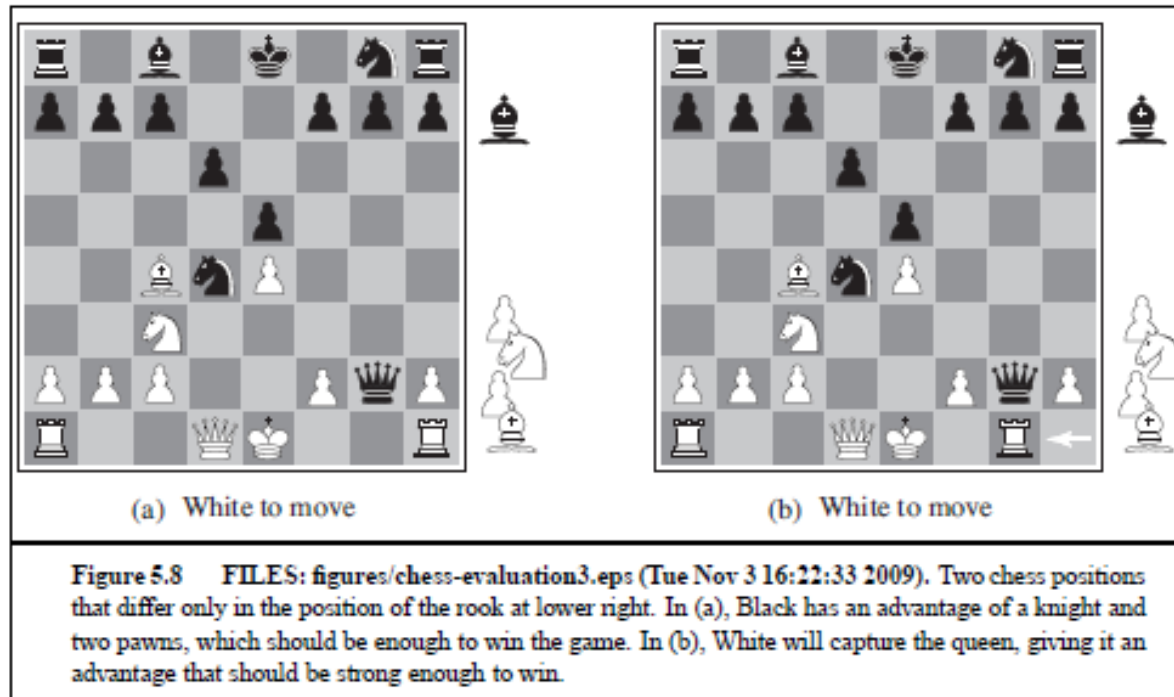
2. Zugbewertung ohne erschöpfende Suche

- Der Minimax-Algorithmus generiert den vollständigen Spielsuchraum, während der α - β -Algorithmus große Teile davon abschneidet (ohne die beste Lösung zu verlieren)
- Trotzdem: Vollständige Suche meist zu aufwendig
- Idee: Propagierung von unten nach oben durch Anwendung einer Bewertungsfunktion ersetzen
- **Bewertungsfunktion:** Evaluiert den Wert von Zuständen durch **Heuristiken** und beschneidet dadurch auch den Suchraum
- **New MINIMAX:**
 - **CUTOFF-TEST:** Ersetzt die Terminierungsbedingung
 - **EVAL:** Bewertungsfunktion als Ersatz für die Nützlichkeitsfunktion

Bewertungsfunktion: Beispiel

- Merkmalsberechnungen (z.B. Anzahl der Bauern, Türme, ...)
- Es ergeben sich Kategorien mit Gewinn, Verlust, Unentschieden
- Für jede Merkmalskombination werden die Gewinnchancen abgeschätzt
- Beispiel für eine Merkmalskombination:
72% Gewinn (+1), 20% Verlust (-1), 8% Unentschieden (0)
- Erwarteter Nutzen:
 $(0,72 \cdot (+1)) + (0,20 \cdot (-1)) + (0,08 \cdot 0) = 0,52$

Bewertungsfunktion: Beispiel



- **Gewichtete Linearkombination:** Kombination von mehreren Heuristiken
$$f = w_1 f_1 + w_2 f_2 + \dots + w_n f_n$$
- Beispiel: f_1 = Bewertung #Bauern, f_2 = Bewertung #Türme, ...
 w_1 = Gewicht #Bauern, w_2 = Gewicht #Türme
(Gewichte können von der Suchtiefe abhängen)

Max-Value mit CUTOFF und EVAL

```
function max_value(state, alpha, beta, depth)
  if cutoff_test(state, depth)
    return eval(state)
  end

  v = typemin(Int)
  for s in state.subseq_s
    v = maximum((v,
                  min_value(s, alpha, beta, depth+1)))
    if v >= beta
      break
    end
    alpha = maximum((alpha, v))
  end
  state.alpha = alpha
  state.beta = beta
  state.v = v
  return v
end
```

Minimax mit CUTOFF: Brauchbarer Algorithmus?

MinimaxCUTOFF ist identisch zu MinimaxVALUE außer

1. IS-TERMINAL wird ersetzt durch CUTOFF-TEST
2. VALUE wird ersetzt durch EVAL

Funktioniert das in der Praxis?

$$b^m = 10^6, b = 35 \rightarrow m = 4$$

Annahme: 100 Sekunden
Rechenzeit werden pro Zug
verwendet und wir können
 10^4 Knoten/s bewerten;
dann können wir 10^6
Knoten/Zug durchrechnen

4 Halbzüge Vorausschau: Anfängerniveau

8 Halbzüge Vorausschau: meisterhaft

12 Halbzüge Vorausschau: großmeisterhaft (~DeepBlue)

Meister durch Computer geschlagen im Jahr...

- Dame: 1994
- Schach: 1997
- Go: 2016

Bemerkung zur Eingabe bei Graphproblemen

- Wir haben z.B. erkannt, dass für einen Graphen $G=(V, E)$ mit $n = |V|$ und $m = |E|$ gilt:
 - $T_{\text{Tiefensuche}} \in O(n+m)$
 - $T_{\text{Dijkstra}} \in O(n \log n + m)$ mit Fibonacci-Heaps
 - Folgerung: Kürzeste-Wege-Optimierungsprobleme sind linear bzw. linear-logarithmisch lösbar (pseudolinear)
- Bei Spielsuchverfahren (oder auch A^*) können wir den Graphen nicht sinnvoll vorher "erzeugen" und als Eingabe verwenden
 - Eingabe ist nur der Startknoten (Startzustand) plus Funktion zur Erzeugung von Knotennachfolgern (nächste Zustände)
 - Vgl. A^* : edges_out muss Kanten dynamisch berechnen
→ n steigt ggf. exponentiell
 - Wir haben Verzweigungen im Suchraum:
Probleme sind mit exponentiellem Aufwand lösbar
 - Folgerung: Grenzen des Einsatzes von Computern erkennbar