

# Grundlagen der Künstlichen Intelligenz

## **5 Spiele = Adversiale Suche**

Suchstrategien für Spiele, Spiele mit Zufall, Stand der Kunst

*Volker Steinhage*

# Inhalt

---

- Brettspiele
- Minimax-Suche
- Alpha-Beta-Suche
- Spiele mit Zufallsereignissen
- Ausgewählte Ergebnisse

# Spiele?

---

**Einordnung:** Kapitel 2 stellte *Multiagenten-Umgebungen* vor, in denen Agenten ihre Wechselwirkungen berücksichtigen müssen

- **Spiele** sind Multiagenten-Umgebungen
- **Brettspiele** sind eines der ältesten Teilgebiete der KI (Shannon, Turing, 1950)

Warum?

- abstrakte und reine Form des Wettbewerb zw. zwei *Gegnern* (lat. *Adversarius*)
  - erfordern „offensichtlich“ eine Form von Intelligenz
  - i. A. wohl definierte Zustände und Aktionen
- 

Hier

- Realisierung des Spielens als Suchproblem (sog. *adversiale Suche*)
- häufig mit *deterministischen* und *vollständig beobachtbaren Umgebungen*
- aber anspruchsvoll wegen großer Suchbäume und weil *Kontingenzprobleme*, da die Züge des Gegners im voraus nicht bekannt sind

# Probleme

Ein Halbzug ist ein einzelner Zug für jede Seite. Z.B. der Bauernzug von E2 nach E4 ist ein Halbzug

Größe von Suchbäume für Beispiele:

- Schach: ca. 35 mögliche Aktionen in jeder Position und 100 Halbzüge pro Spiel  
→  $35^{100}$  Knoten im Suchbaum (bei „nur“ ca.  $10^{40}$  legalen Schachpositionen)
- Go: durchschnittlich 200 mögliche Aktionen bei ca. 300 Halbzügen  
→  $200^{300}$  Knoten
- Dame: vollständiger Suchbaum umfasst ca.  $10^{40}$  Knoten  
→ ca.  $10^{21}$  Jahrhunderte für Baumaufbau bei 1/3 ns pro Knotengenerierung

---

Eigenschaften guter Spielprogramme:

- Vorausschauen möglichst vieler Halbzüge (engl. *look ahead*)
- gute Evaluierungsfunktion für Zwischenzustände
- Erkennen und Abschneiden (engl. *pruning*) irrelevanter Äste im Spielbaum

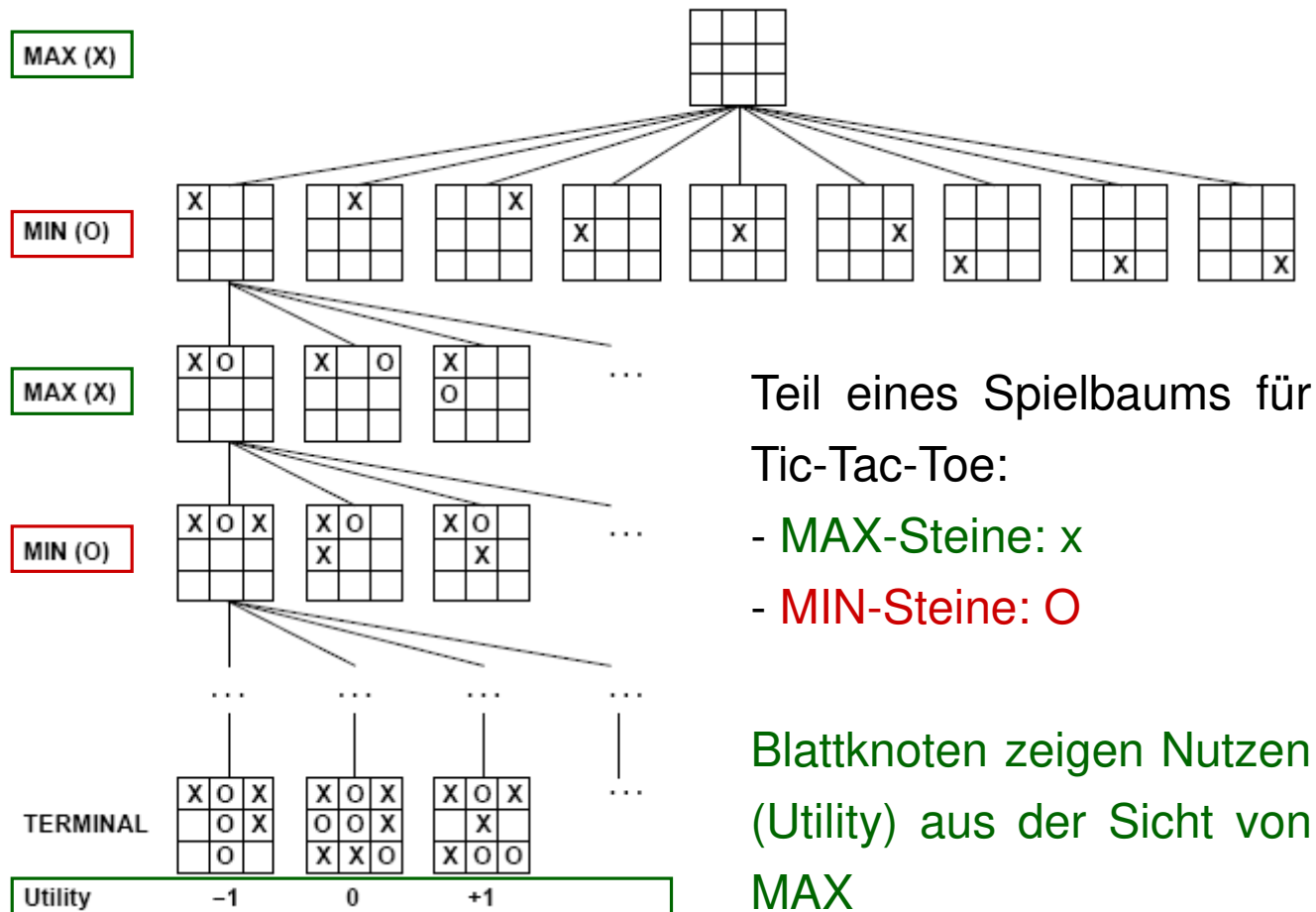
# Terminologie bei Zweipersonen-Brettspielen

- Spieler: **MAX** und **MIN**
  - **MAX** steht für das Spielprogramm (Agent) und beginnt (weil am Zug)
- Anfangszustand: gültige Start- bzw. Ausgangspositionen des jew. Spiels
- Operatoren = legale Züge
- Terminierungstest: wann ist ein Spiel beendet?
  - Terminalzustand = Spielende
- Nutzenfunktion = Bewertung des *Spielausgangs*
  - ggf. einfach: +1 (Sieg), -1 (Niederlage), 0 (unentschieden)
  - ggf. differenziert: bei Backgammon Werte zwischen +192 und -192
- Strategie = *Siegstrategie für MAX* → Ziel: Gewinnmaximierung
  - Strategie, die *unabhängig von MINs Zügen* zum Gewinn führt!
  - korrekte Reaktionen auf *alle Züge von MIN*

Bei großen Spielbäumen:  
Schätzung des Spielausgangs

# Beispiel Tic-Tac-Toe

- Jede Stufe des Suchbaums, auch **Spielbaum** genannt, wird mit dem Namen des Spielers bezeichnet, der am Zug ist (**MAX**- und **MIN**-Stufen)
- Wenn es wie hier möglich ist, den gesamten Spielbaum zu erzeugen, liefert der **Minimax-Algorithmus** eine **optimale Strategie für MAX**

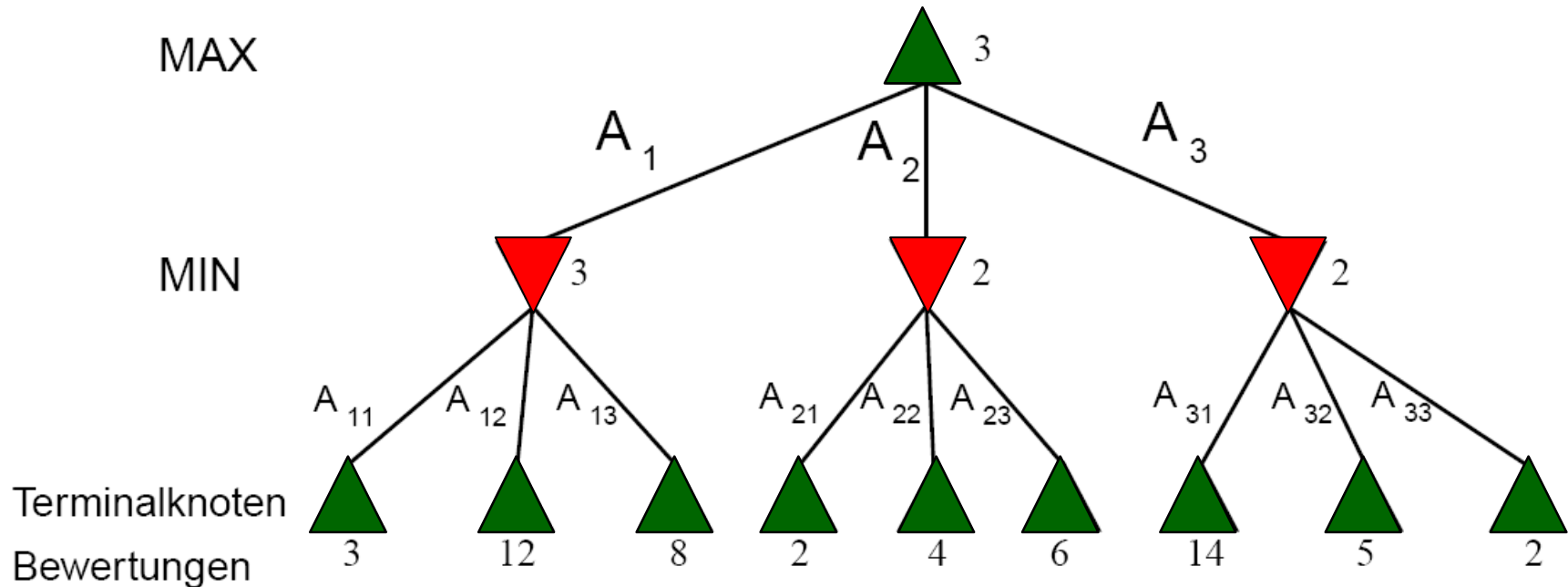


# Minimax

---

1. Erzeuge vollständigen Spielbaum mit **Tiefensuche**
2. Wende die **Nutzenfunktion auf jeden Terminalzustand** an
3. Beginnend bei Terminalzuständen, **berechne Werte der Vorgängerknoten**:
  - **Knoten ist MIN-Knoten**:
    - Wert übernimmt das Minimum der Nachfolgerknoten
  - **Knoten ist MAX-Knoten**:
    - Wert übernimmt das Maximum der Nachfolgerknoten
  - **Minimax-Entscheidung**: MAX wählt im Anfangszustand (Wurzel des Spielbaums) den Zug, der zu dem Nachfolgerknoten mit größtem berechneten Nutzen führt
- **Beachte**: Minimax geht von **perfektem Spiel von MIN** aus.
  - Abweichungen/Fehler von MIN können Ergebnis für MAX nur verbessern

# Beispielsuchbaum für MiniMax



Ein Zwei-Schichten-Spielbaum: Die  $\Delta$ -Knoten sind die MAX-Knoten und die  $\nabla$ -Knoten sind die MIN-Knoten. Die Terminalknoten zeigen die Nutzenwerte für MAX. Alle anderen Knoten zeigen ihre MiniMax-Werte. MAX ist am Zug.



# Minimax Algorithmus

Berechne rekursiv den besten Zug von der Anfangssituation ausgehend (Tiefensuche):

```
function MINIMAX-DECISION(state) returns an action
```

```
  v ← MAX-VALUE(state)
```

```
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state) returns a utility value
```

```
  if TERMINAL-TEST(state) then return UTILITY(state)
```

```
  v ←  $-\infty$ 
```

```
  for a, s in SUCCESSORS(state) do
```

```
    v ← MAX(v, MIN-VALUE(s))
```

```
  return v
```

```
function MIN-VALUE(state) returns a utility value
```

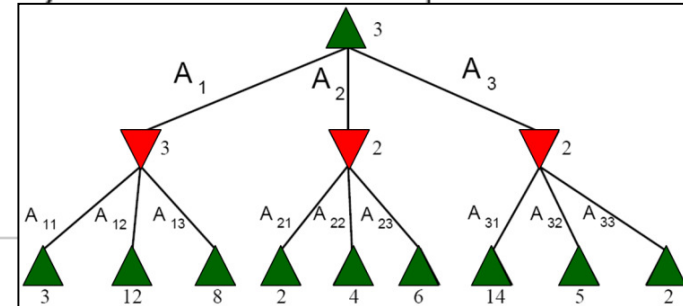
```
  if TERMINAL-TEST(state) then return UTILITY(state)
```

```
  v ←  $\infty$ 
```

```
  for a, s in SUCCESSORS(state) do
```

```
    v ← MIN(v, MAX-VALUE(s))
```

```
  return v
```



# Evaluierungsfunktion

---

Bei zu **großem Suchraum** kann der Spielbaum nur bis zu einer **maximalen Suchtiefe** erzeugt werden.

Die Kunst besteht dann darin, die **Güte der den Blättern entsprechenden Spielpositionen (i.A. keine Endpositionen)** korrekt zu bewerten.

Beispiele einfacher Bewertungskriterien im Schach:

- **Material**bewertung: Bauer = 1, Springer = 3, Läufer = 5, Dame = 9, etc.
- **Positions**bewertung: Sicherheit des Königs bis Positionen der Bauern
- **Faustregeln** wie „3-Punkte Vorsprung = (fast) sicherer Sieg“

Der resultierende Wert einer Spielposition sollte natürlich die Gewinnchancen widerspiegeln, d.h. die Gewinnchancen bei einem Vorteil von +1 sollten geringer sein als die bei einem Vorteil von +3.

# Evaluierungsfunktion - allgemein

---

Bevorzugte Evaluierungsfunktionen sind *gewichtete lineare Funktionen*:

$$w_1 \cdot f_1 + w_2 \cdot f_2 + \dots + w_n \cdot f_n.$$

Hierbei ist  $f_i$  das  $i$ -te Kriterium und  $w_i$  das  $i$ -te Gewicht.

→ Bsp:  $w_1 = 3$ ,  $f_1$  = Zahl der eigenen Springer auf dem Brett.

*Annahme*: Die Kriterien sind unabhängig voneinander.

- Die *Gewichte* können u.U. gelernt werden.
- Die *Kriterien* müssen allerdings vorgegeben werden.

# Wann die Suche beenden? (1)

---

- Einfach: **Tiefenbeschränkte Suche** entsprechend vorgegebenem **Tiefenlimit**
- Robuster: **Iterative Tiefensuche** mit Abbruch beim Erreichen des **Tiefenlimits**  
→ liefert beste Lösung der tiefsten *vollständigen* Suche.

Bedeutet bei großen Spielbäumen i.A.

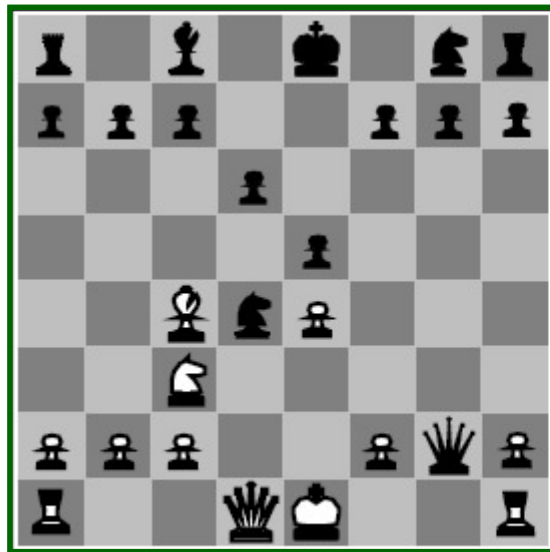
- nicht die optimale Problemlösung bzw. Zielstrategie zum Sieg,
- sondern die *optimale Zwischenlösung* bzw. Strategie zur *besten eigenen Stellung, die bei max. Suchtiefe  $m$  erreichbar ist.*

# Wann die Suche beenden? (2)

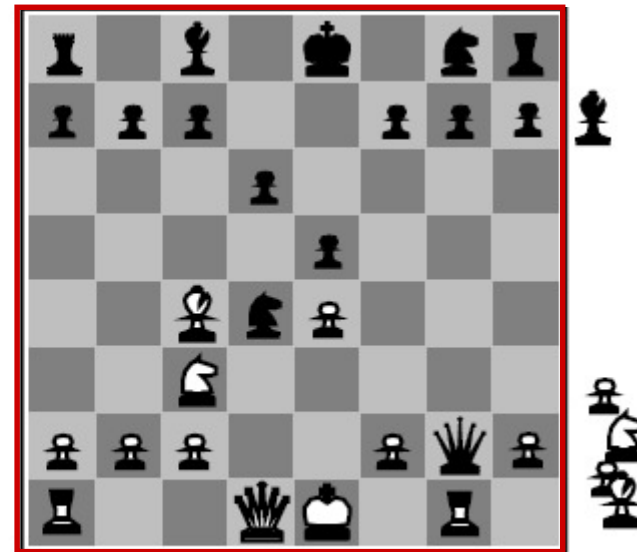
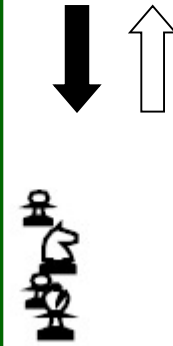
Evaluierung sollte für „**ruhende**“ Positionen, die nicht zu großen Schwankungen bei der Evaluierung in den folgenden Zügen führen, enden.

Bspl.:

- Annahme: alleinige Bewertung durch Materialvorteil und Weiß ist am Zug
- Stellungen (a) und (b) zeigen denselben Materialvorteil für Schwarz (1 Springer, 2 Bauern). Im Ggs. zur „**ruhenden**“ Position in (a) zeigt (b) eine „**nicht ruhende**“ Position, da der nächster Zug von Weiß durch den Turm zum unweigerlichen Damenverlust führt.
- u.U. noch etwas weiter suchen und einen Schlagabtausch zu Ende führen
- Extrasuche bis zu „stabiler“ Bewertungsposition.



(a) White to move



(b) White to move

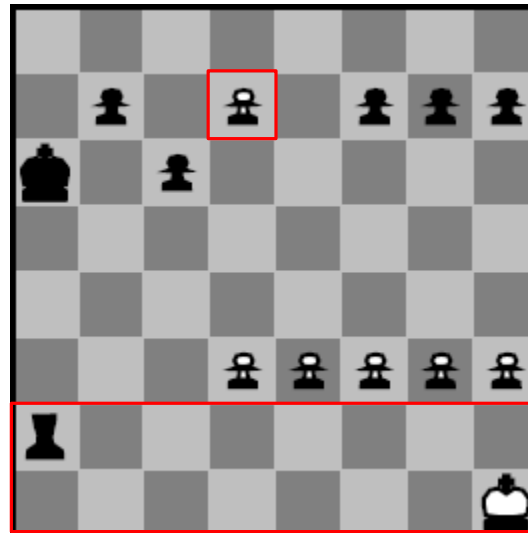


# Wann die Suche beenden? (3)

- **Horizonteffekt**: ein letztlich unvermeidbarer schädigender Gegenzug wird durch eigene verzögernde Züge aus dem durch eine Tiefenschranke begrenzten „sichtbaren“ Suchraum hinaus geschoben.

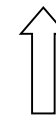


Vermeintliche Lösung: setze mehrmals Schach über Turm und verschiebe den Damenzug über den „Suchhorizont“ hinaus, wo er nicht erkannt wird.



Schwarz mit  
Materialvorteil am Zug

Langfristig wird  
weißer Bauer  
zu Dame



Lösung entweder durch generell größere Suchtiefe oder *singuläre Erweiterung* der Tiefensuche für „deutlich bessere Züge“.

# Komplexität von Minimax $\leadsto$ Alpha-Beta-Kürzung

---

Minimax verwendet **Tiefensuche**:

$\Rightarrow$  Speicherkomplexität  $O(b \cdot m)$  für max. Spielbaumtiefe  $m$  und  $b$  erlaubte Züge

$\Rightarrow$  Speicherkomplexität  $O(m)$  für Backtracking-Variante

$\Rightarrow$  **Zeitkomplexität  $O(b^m)$**

Für reale Spiele ist die Zeitkomplexität völlig impraktikabel!

$\Rightarrow$  Reduktion der Laufzeit durch **Alpha-Beta-Kürzung** (engl. **Pruning**) durch Abschneiden von Knoten und Teilbäumen, die die Entscheidung überhaupt nicht beeinflussen können

$\Rightarrow$  Qualitativ *dasselbe* Ergebnis wie mit Minimax bei besserer Laufzeit!

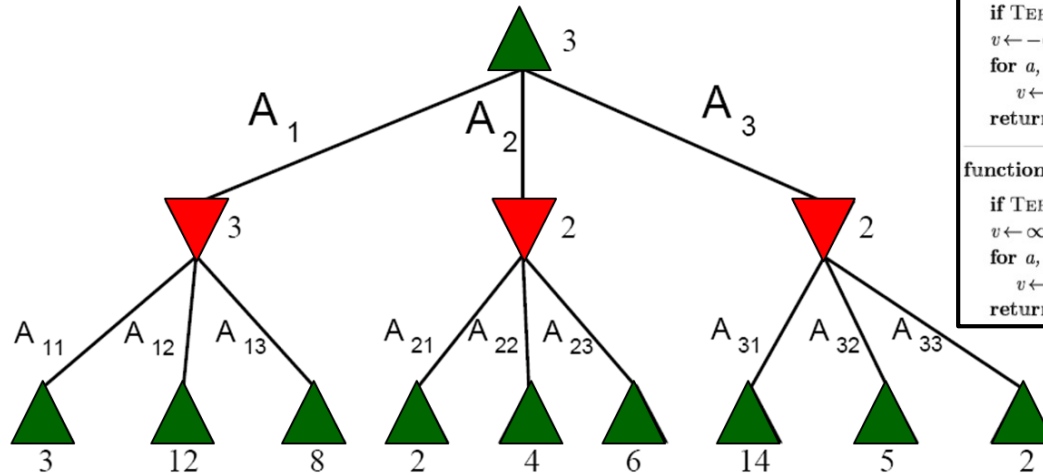
# Alpha-Beta-Kürzung: Beispiel

```

function MINIMAX-DECISION(state) returns an action
   $v \leftarrow \text{MAX-VALUE}(\text{state})$ 
  return the action in SUCCESSORS(state) with value  $v$ 

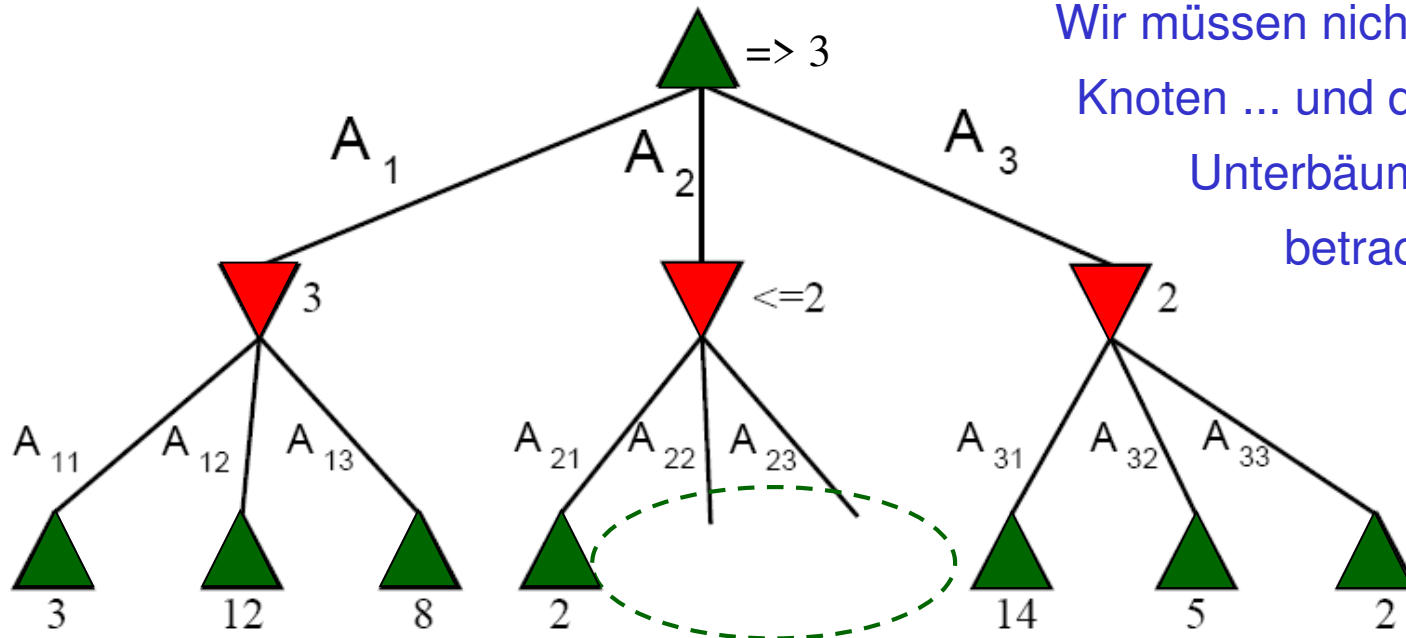
function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow -\infty$ 
  for  $a, s$  in SUCCESSORS(state) do
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$ 
  return  $v$ 

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
   $v \leftarrow \infty$ 
  for  $a, s$  in SUCCESSORS(state) do
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$ 
  return  $v$ 
    
```



MAX

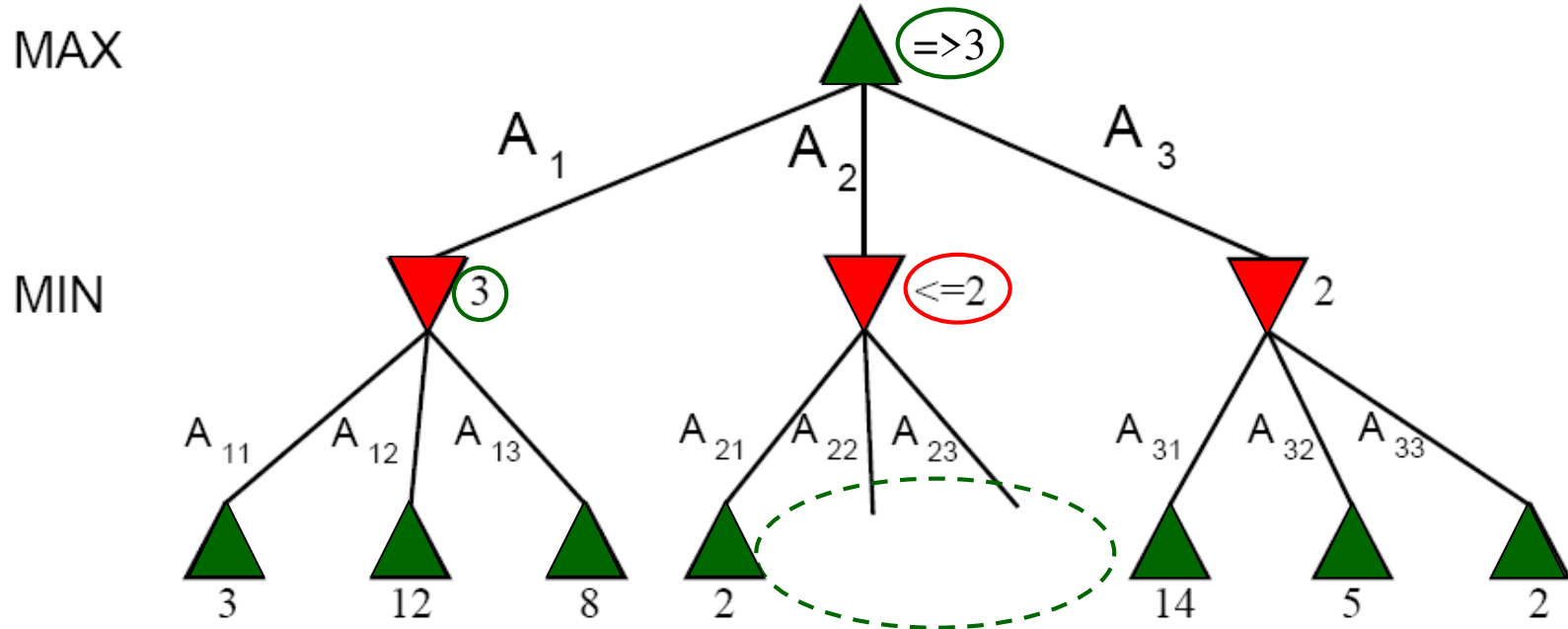
MIN



Wir müssen nicht alle  
Knoten ... und deren  
Unterbäume (!)  
betrachten



# Alpha-Beta-Kürzung: Beispiel



- Falls die Züge  $A_{22}$ ,  $A_{23}$  höhere Bewertungen als  $A_{21}$  erhalten, wird MIN diese verwerfen, da  $A_{21}$  besser ist für MIN.
- Falls Züge  $A_{22}$ ,  $A_{23}$  niedrigere Bewertungen als  $A_{21}$  erhalten, wird MAX diese genauso wie  $A_{21}$  verwerfen, da  $A_1$  besser für MAX ist als das resultierende  $A_2$ .
- Allgemein: sobald für den aktuellen Knoten  $n$  durch Betrachtung einiger Nachfolger fest steht, dass eine bessere Wahl in einem Vorgängerknoten möglich ist, kann  $n$  mit gesamtem Unterbaum verworfen werden.

# Alpha-Beta-Werte

---

Da Minimax-Algorithmus die Tiefensuche einsetzt :

→ immer Betrachtung eines aktiven Pfades.

Für diesen aktiven Pfad werden zwei Werte gehalten:

- $\alpha$  = aktuell bester Wert für MAX auf *aktivem* Pfad
- $\beta$  = aktuell bester Wert für MIN auf *aktivem* Pfad

Initialisierung:  $\alpha = -\infty$ ,  $\beta = +\infty$

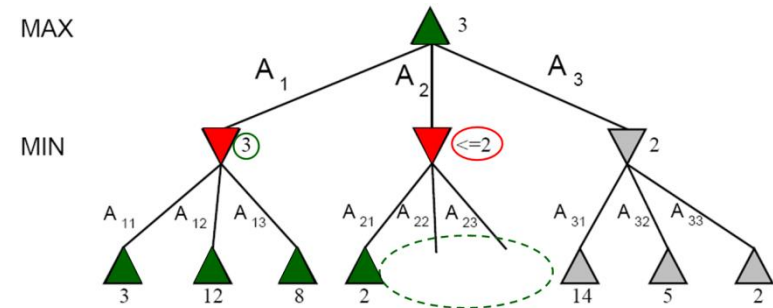
Strategie: Fortlaufende Aktualisierung von  $\alpha$  und  $\beta$  und sowie Verwerfen von Unterbäumen, deren Ergebnisse schlechter sein müssen als die aktuellen Werte von  $\alpha$  und  $\beta$ .

# Wann kann abgeschnitten werden?

Es gilt:

- $\alpha$ -Werte von MAX-Knoten starten mit  $-\infty$  und können nie abnehmen
- $\beta$ -Werte von MIN-Knoten starten mit  $+\infty$  und können nie zunehmen

(1) Abschneiden unterhalb des MIN-Knotens, dessen Wert kleiner oder gleich dem  $\alpha$ -Wert seines MAX-Vorgängerknotens ist.



(2) Abschneiden unterhalb des MAX-Knotens dessen Wert größer oder gleich dem  $\beta$ -Wert seines MIN-Vorgängerknotens ist.

Alpha-Beta-Suche liefert Ergebnisse, die genauso gut sind wie bei vollständiger Minimax-Suche bis zur gleichen Tiefe (weil nur irrelevante Zweige eliminiert werden)!

# Alpha-Beta-Algorithmus

**function** ALPHA-BETA-SEARCH(*state*) **returns** an action  
 $v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$  ←  
**return** the *action* in ACTIONS(*state*) with value *v*

Initialisierung mit Zustand und  
 initialen  $\alpha$ - und  $\beta$ -Werten

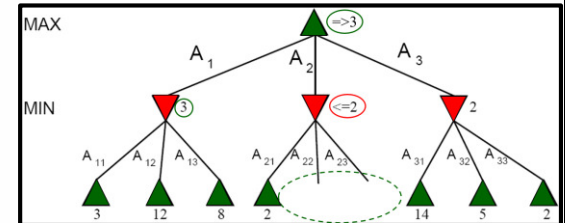
**function** MAX-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value  
**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)  
 $v \leftarrow -\infty$   
**for each** *a* **in** ACTIONS(*state*) **do**  
 $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(\text{state}, a), \alpha, \beta))$   
**if**  $v \geq \beta$  **then return** *v* ← Pruning  
 $\alpha \leftarrow \text{MAX}(\alpha, v)$   
**return** *v*

**function** MINIMAX-DECISION(*state*) **returns** an action  
 $v \leftarrow \text{MAX-VALUE}(\text{state})$   
**return** the *action* in SUCCESSORS(*state*) with value *v*

**function** MAX-VALUE(*state*) **returns** a utility value  
**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)  
 $v \leftarrow -\infty$   
**for** *a, s* **in** SUCCESSORS(*state*) **do**  
 $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(s))$   
**return** *v*

**function** MIN-VALUE(*state*) **returns** a utility value  
**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)  
 $v \leftarrow \infty$   
**for** *a, s* **in** SUCCESSORS(*state*) **do**  
 $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(s))$   
**return** *v*

**function** MIN-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value  
**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)  
 $v \leftarrow +\infty$   
**for each** *a* **in** ACTIONS(*state*) **do**  
 $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(\text{state}, a), \alpha, \beta))$   
**if**  $v \leq \alpha$  **then return** *v* ← Pruning  
 $\beta \leftarrow \text{MIN}(\beta, v)$   
**return** *v*



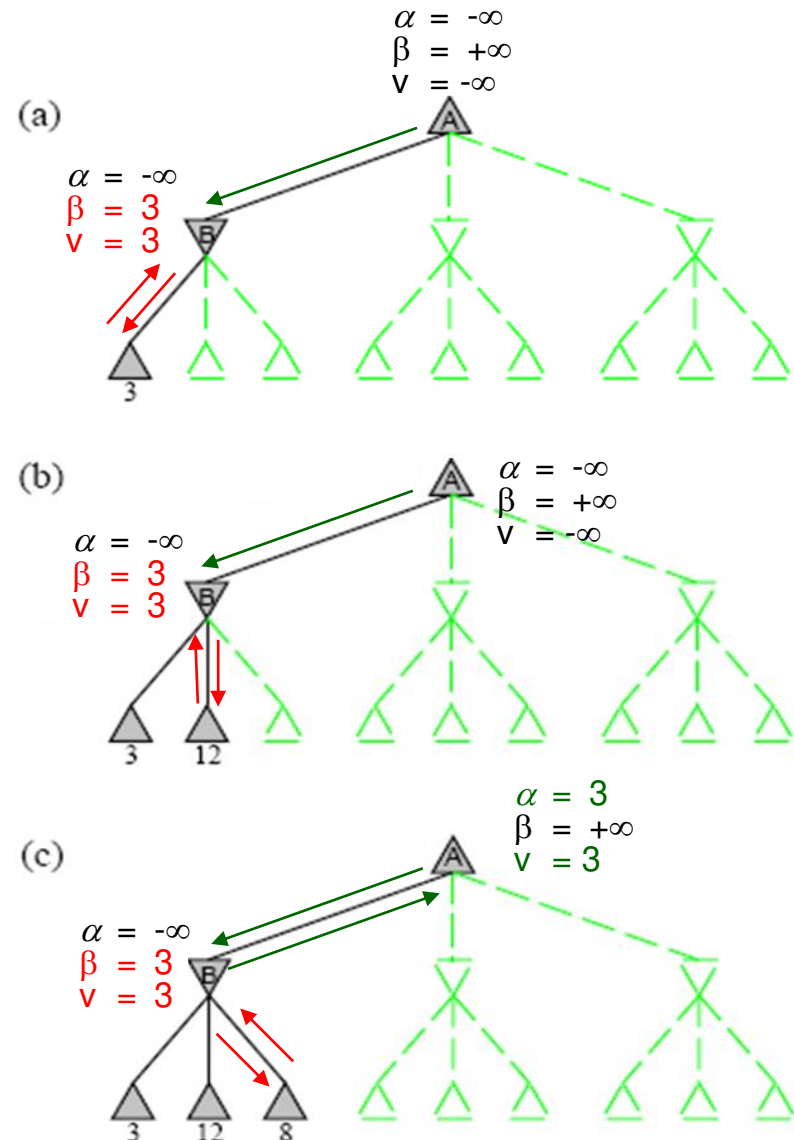
# Alpha-Beta-Algorithmus: Beispiel

Alpha-Beta-Algm. im Zwei-Schichten-Spielbaum:

**function** ALPHA-BETA-SEARCH(*state*) **returns** an action  
 $v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$   
**return** the *action* in  $\text{ACTIONS}(\text{state})$  with value  $v$

**function** MAX-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value  
**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)  
 $v \leftarrow -\infty$   
**for each**  $a$  **in**  $\text{ACTIONS}(\text{state})$  **do**      $v \leftarrow 3$  in (c)  
 $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(\text{state}, a), \alpha, \beta))$   
**if**  $v \geq \beta$  **then return**  $v$   
 $\alpha \leftarrow \text{MAX}(\alpha, v)$       $\alpha \leftarrow 3$  in (c)  
**return**  $v$

**function** MIN-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value  
**if** TERMINAL-TEST(*state*) **then return** UTILITY(*state*)  
 $v \leftarrow +\infty$   
**for each**  $a$  **in**  $\text{ACTIONS}(\text{state})$  **do**      $v \leftarrow 3$  in (a)  
 $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(\text{state}, a), \alpha, \beta))$   
**if**  $v \leq \alpha$  **then return**  $v$       $v = 3$  in (b)  
 $\beta \leftarrow \text{MIN}(\beta, v)$       $\beta \leftarrow 3$  in (a)  
**return**  $v$       $v = 3$  in (c)



# Alpha-Beta-Algorithmus: Forts. Beispiel

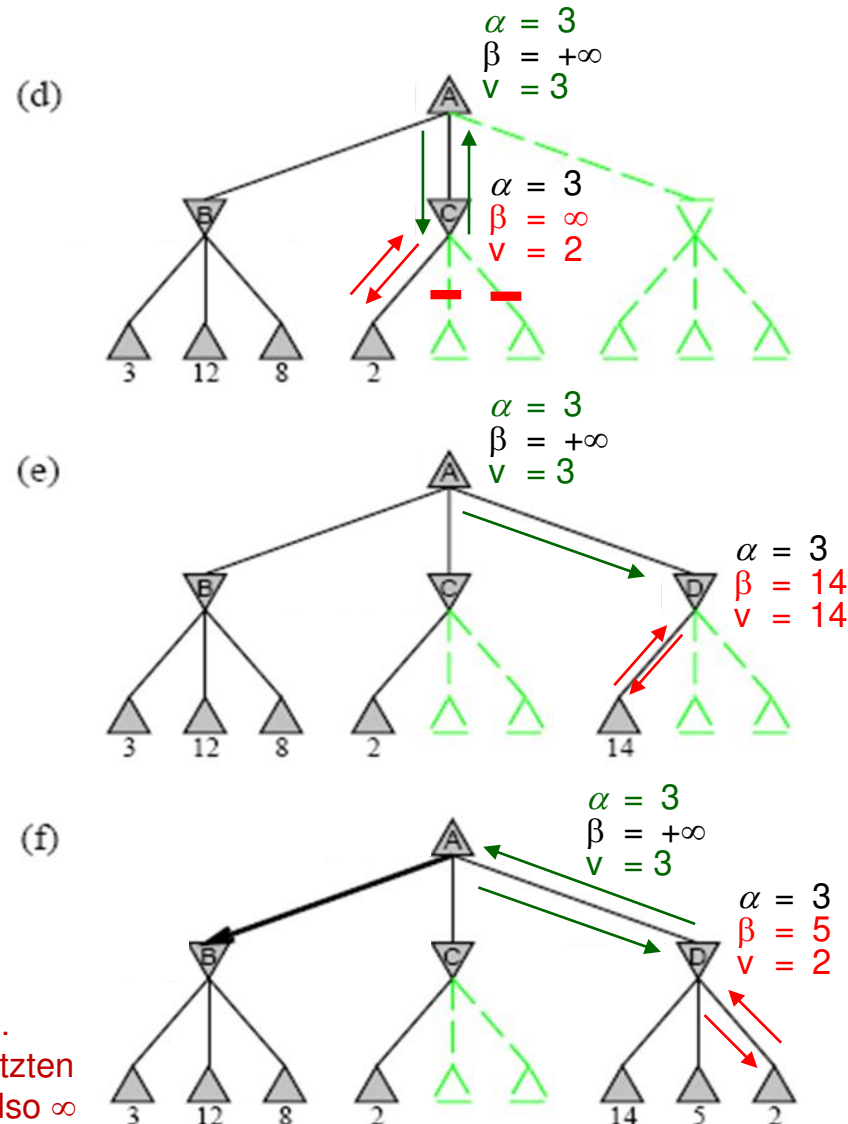
Alpha-Beta-Algm. im Zwei-Schichten-Spielbaum:

**function** ALPHA-BETA-SEARCH(*state*) **returns** an action  
 $v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$   
**return** the *action* in  $\text{ACTIONS}(\text{state})$  with value  $v$

**function** MAX-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value  
**if**  $\text{TERMINAL-TEST}(\text{state})$  **then return**  $\text{UTILITY}(\text{state})$   
 $v \leftarrow -\infty$   
**for each**  $a$  **in**  $\text{ACTIONS}(\text{state})$  **do**  
 $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(\text{state}, a), \alpha, \beta))$   
**if**  $v \geq \beta$  **then return**  $v$   
 $\alpha \leftarrow \text{MAX}(\alpha, v)$   
**return**  $v$

**function** MIN-VALUE(*state*,  $\alpha$ ,  $\beta$ ) **returns** a utility value  
**if**  $\text{TERMINAL-TEST}(\text{state})$  **then return**  $\text{UTILITY}(\text{state})$   
 $v \leftarrow +\infty$   
**for each**  $a$  **in**  $\text{ACTIONS}(\text{state})$  **do**  
 $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(\text{state}, a), \alpha, \beta))$   
**if**  $v \leq \alpha$  **then return**  $v$   
 $\beta \leftarrow \text{MIN}(\beta, v)$   
**return**  $v$

$v = 2$  in (d) und (f) bei  
 $\alpha = 3$  aus (c)  $\rightarrow$  Pruning.  
 $\beta$  bleibt damit auf dem letzten  
Wert vor dem Pruning, also  $\infty$   
in (d) und 5 in (f)



# Alpha-Beta für Tic-Tac-Toe (1)

---

Beispiel für Tic-Tac-Toe auf Zwei-Schichten-Spielbaum (1 Zug MAX, 1 Zug MIN):

Bewertungs- bzw. Evaluierungsfunktion  $e(p)$  eines Spielzustandes  $p$  sei

- wenn Gewinnposition für MAX:  $+\infty$
- wenn Gewinnposition für MIN:  $-\infty$
- sonst:  $\text{Zahl der offenen 3-Ketten für MAX}$   
–  $\text{Zahl der offenen 3-Ketten für MIN}$

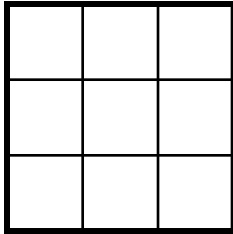
Beispiel:

	X	
	O	

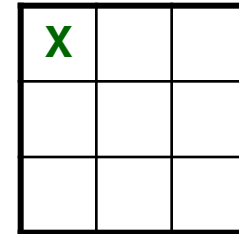
$$e(p) = 6 - 4 = 2$$

# Alpha-Beta für Tic-Tac-Toe (2)

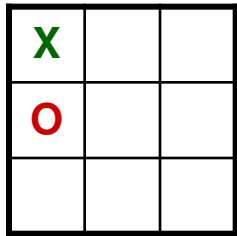
1) Start



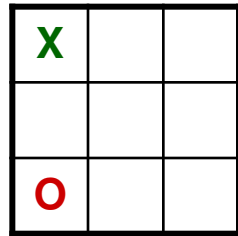
2) erster mögl. Zug MAX



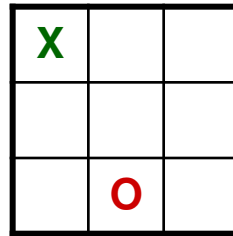
3) Mögliche Reaktionen MIN (unter Ausnutzung der Brettsymmetrie):



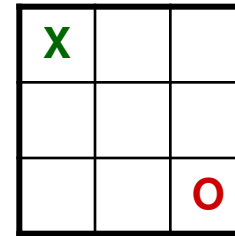
$$6 - 5 = 1$$



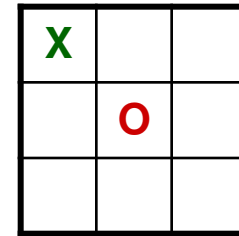
$$5 - 5 = 0$$



$$6 - 5 = 1$$



$$5 - 5 = 0$$



$$4 - 5 = -1$$

MIN wird Aktion zu  $p$  mit  $e(p) = -1$  wählen:

→ MIN wird alternativ nur Zug mit Wert  $w \leq -1$  wählen:

→ obere Schranke für MIN im entspr. Zustand:  $\beta \leftarrow -1$

→ für MAX ist -1 *aktuell* maximale Bewertung eines Nachfolgers von Start,

→ MAX wird in Start als Zug nur einen mit Wert größer oder gleich -1 wählen,

→ untere Schranke für Bewertung des Zuges von Start für MAX:  $\alpha \leftarrow -1$



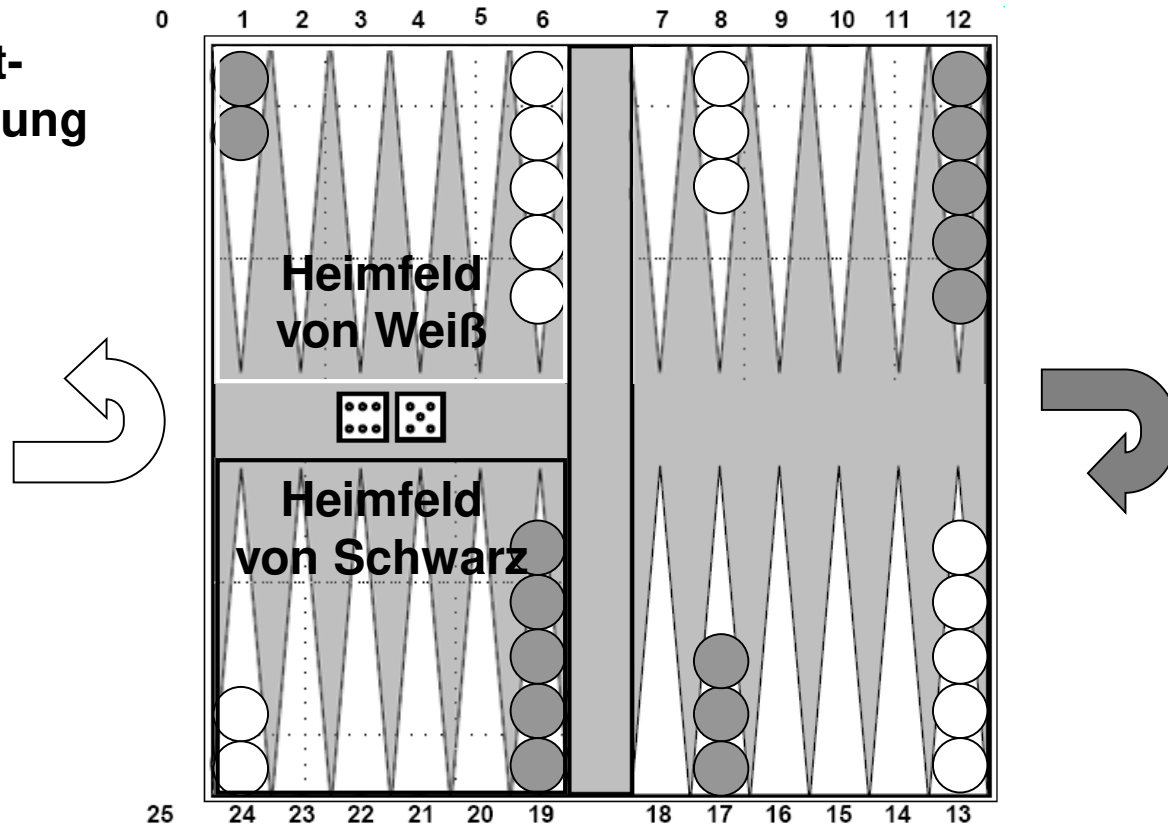
# Evaluierungsfunktion

---

- Alpha-Beta-Suche **schneidet am meisten ab**, wenn jeweils der **beste Zug als erstes** betrachtet wird ( $\rightarrow$  vergl. (e) und (f) im Bspl.).
- **Bester Fall** (immer bester Zug zuerst) reduziert den Suchaufwand auf  $O(b^{m/2})$  und nicht  $O(b^m)$  wie bei Minimax.
- D.h. wir können doppelt so tief in der gleichen Zeit suchen.
- Knuth & Moore (1975): **durchschnittlicher Fall** (zufällig verteilte Züge) reduziert den Suchaufwand auf  $O(b^{3m/4})$ .
- **Praktischer Fall**: Schon mit relativ einfachen *Anordnungsheuristiken* (zuerst Schlagen, dann Drohen, dann Vorwärtsgehen, dann Rückzug) kommt man in die Nähe des besten Falls.

# Spiele mit Zufallsereignissen: Backgammon

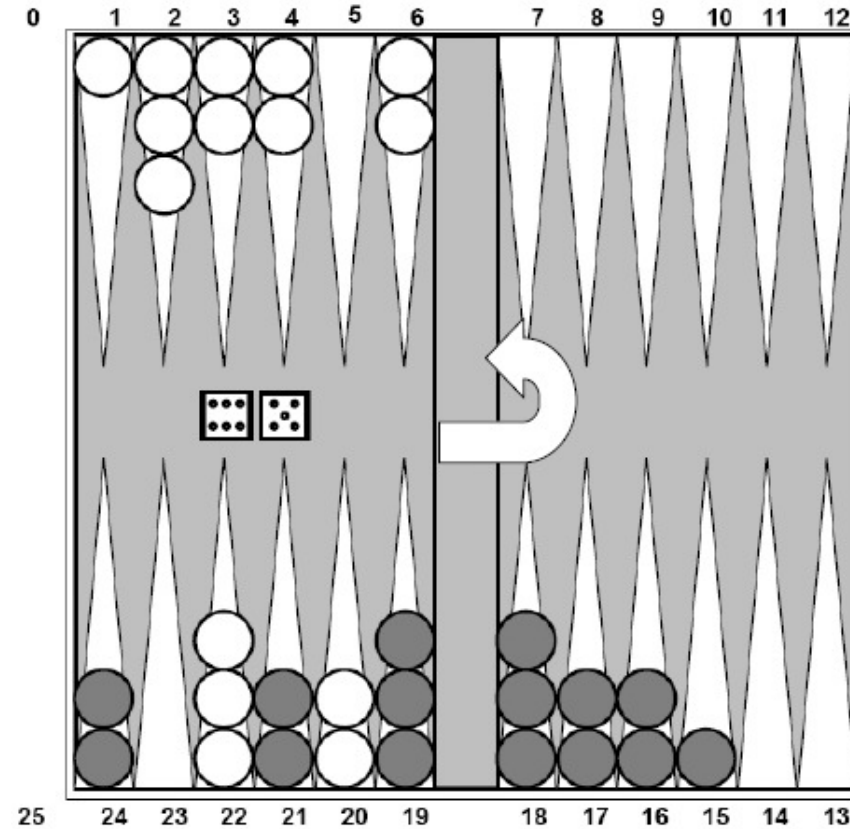
Start-  
aufstellung



Hier relevant:

- **Ziel:** Eigene Steine ins eigene Heimfeld und dann vom Brett bringen
  - Weiß zieht im Gegen-UZS ins Feld 1-6 und dann auf die 0
  - Schwarz zieht im Uhrzeigersinn ins Feld 19-24 und dann auf die 25
- Gültige Züge: dorthin, wo nicht mind. 2 gegnerische Steine sind

# Spiele mit Zufallsereignissen: Backgammon

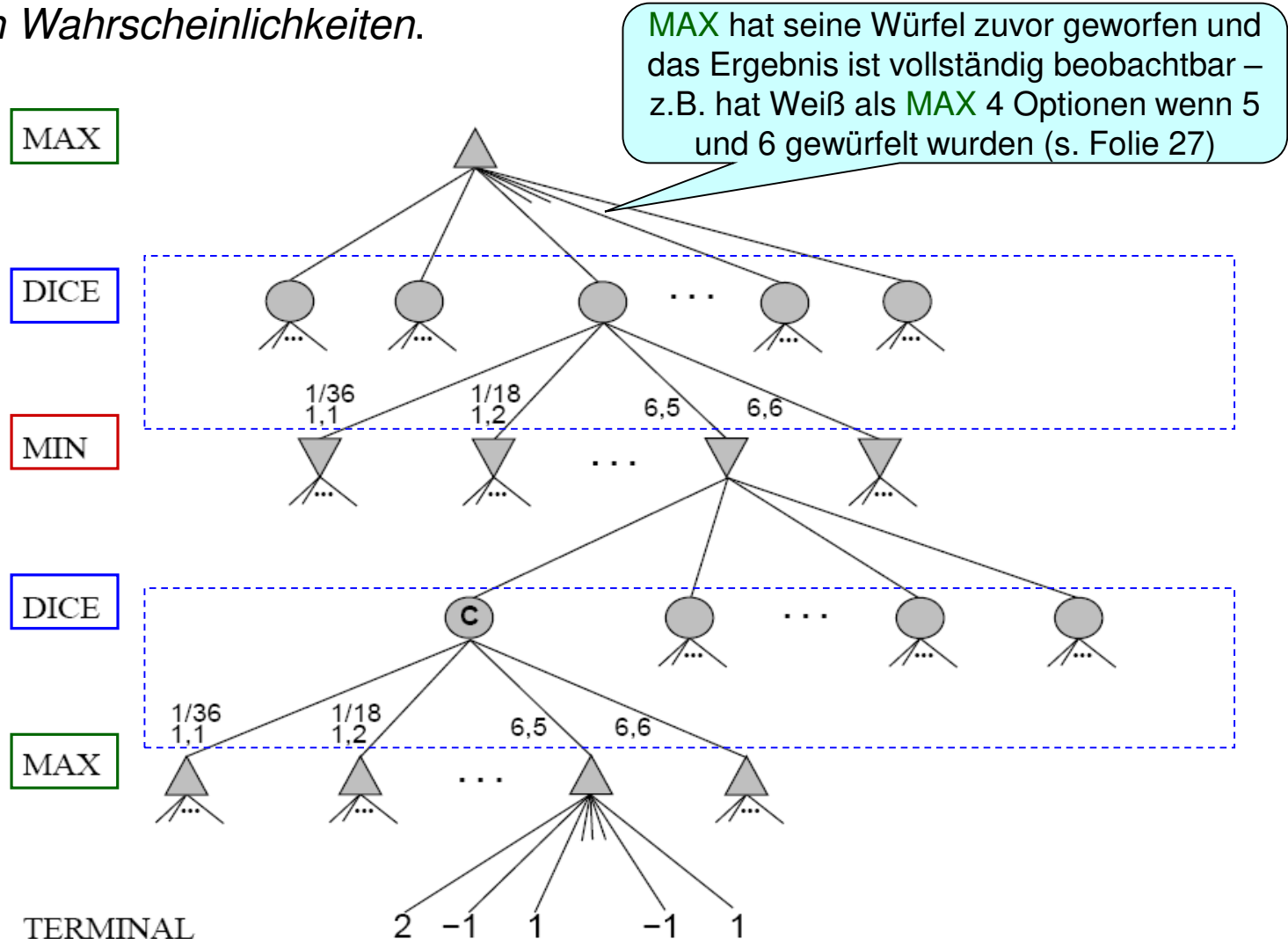


Weiß hat gerade 5 und 6 gewürfelt und hat 4 legale Züge:

$(20 \xrightarrow{6} 14, 20 \xrightarrow{5} 15), (20 \xrightarrow{6} 14, 6 \xrightarrow{5} 1), (20 \xrightarrow{6} 14, 14 \xrightarrow{5} 9), (20 \xrightarrow{5} 15, 15 \xrightarrow{6} 9).$

# Spielbaum für Backgammon

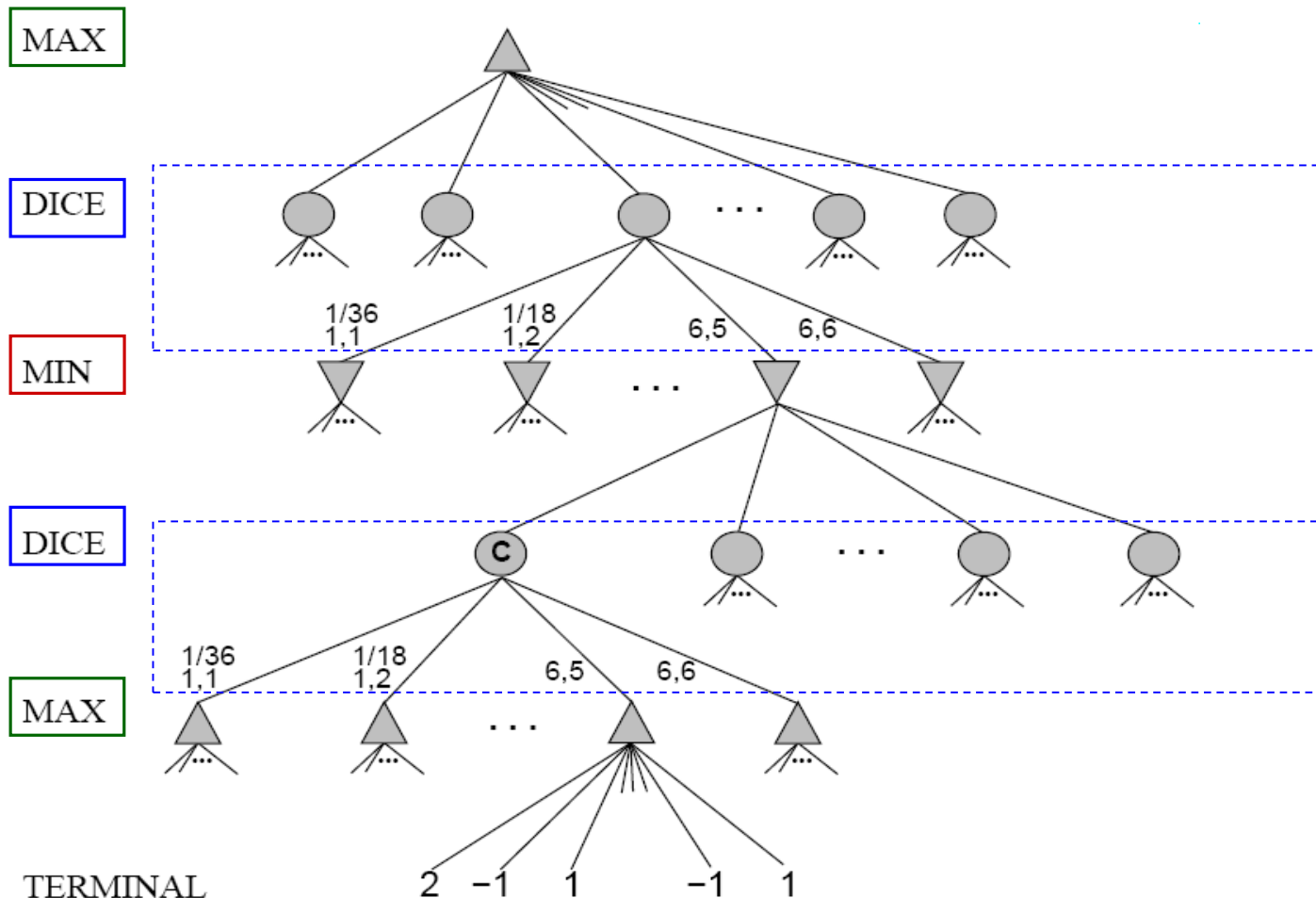
Zusätzlich zu **MAX-Knoten** und **MIN-Knoten** brauchen wir **Würfelknoten** (*chance nodes*). Deren ausgehende Kanten beschreiben *die möglichen Würfelergebnisse und deren Wahrscheinlichkeiten*.



# Berechnung des erwarteten Nutzens

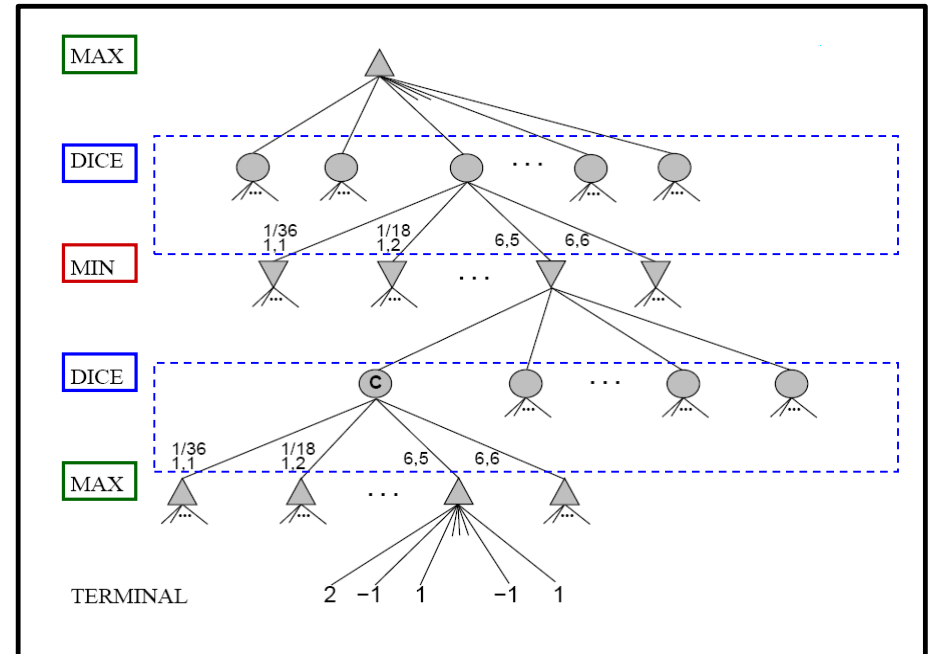
EXPECTIMAX( $n$ ) =

- UTILITY( $n$ ) if  $n$  is a terminal state
- $\max_{s \in \text{Successors}(n)} \text{EXPECTIMAX}(s)$  if  $n$  is a MAX node
- $\min_{s \in \text{Successors}(n)} \text{EXPECTIMAX}(s)$  if  $n$  is a MIN node
- $\sum_{s \in \text{Successors}(n)} P(s) \cdot \text{EXPECTIMAX}(s)$  if  $n$  is a chance node



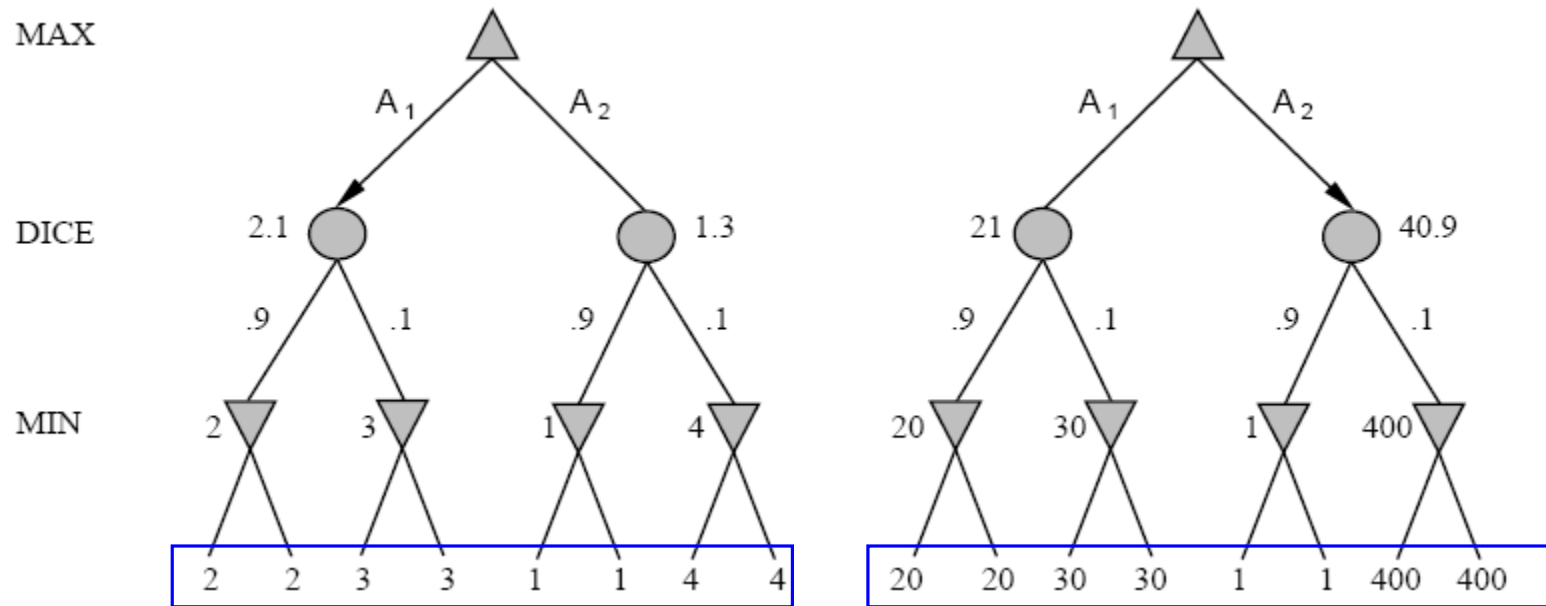
# Bemerkungen zum Backgammon-Spielbaum

- Der Wurzelknoten ist auch hier ein **MAX-Knoten**: **MAX** hat seine Würfel zuvor geworfen und das Ergebnis ist vollständig beobachtbar. Nicht geworfene Alternativen müssen nicht analysiert werden!
- Die **DICE-Knoten** entsprechen den Zuständen, bei denen zu würfeln ist. Man könnte auch personalisiert sagen: die Zustände, die die Würfel „sehen, wenn sie am Zug an sind“.
- Im Ggs. zu **MAX** oder **MIN** ergeben sich die Bewertungen der **DICE-Knoten** aber nicht aus den max. bzw. min. bewerteten Nachfolgeknoten, sondern aus der gewichteten Summe der Bewertungen ihrer Nachfolgeknoten. Die Gewichte entsprechen den Wahrscheinlichkeiten der möglichen Würfelergebnisse.



# Probleme

- Varianten von Evaluierungsfunktionen erhalten i. A. nicht die Ordnung zwischen Zügen:  
Auswahl links:  $A_1$ , Auswahl rechts:  $A_2$ .



Allg.: Bewertungsfunktion muss **positiv lineare Transformation** der Gewinnchancen sein.

- Die Suchkosten steigen:** Statt des  $O(b^m)$  von Minimax haben wir  $O(b^m \cdot n^m)$ , wenn  $n$  die Anzahl möglicher Würfelergebnisse ist. Bei Backgammon ist  $n = 21$  und  $b$  durchschnittlich 20, manchmal aber sogar um 4000. Unter diesen Umständen kann  $m$  maximal 2 sein (also 2 Halbzüge).

# Ausgewählte Ergebnisse (1)

---

**Dame** (*Checkers, Draughts*, d.h. nach internat. Regeln): Das Programm CHINOOK ist amtierender Weltmeister im Mensch-Maschine-Vergleich (anerkannt von ACF (American Checkers Federation) und EDA (English Draughts Ass.)) und höchst bewerteter Spieler.

Am 19. Juli 2007 veröffentlichte die Zeitschrift [Science](#) den Artikel von Schaeffers Team "Checkers is solved", der bewies, dass das beste Resultat eines Gegners von Chinook nur unentschieden sein kann – Chinook also nicht schlagbar ist.

[http://de.wikipedia.org/wiki/Chinook\\_\(Software\)](http://de.wikipedia.org/wiki/Chinook_(Software)) (28.4.17)



# Ausgewählte Ergebnisse (2)

---

**Backgammon:** Das Programm BKG schlug den amtierenden Weltmeister 1980.

**Reversi** (*Othello*): Sehr gut auch auf normalen Computern. Programme werden bei Turnieren nicht zugelassen.

# Schach (1)

---

Schach als „Drosophila“ der KI-Forschung:

- begrenzte Anzahl von Regeln bringt unbegrenzte Zahl von Spielverläufen hervor. Für eine Partie von 40 Zügen gibt es  $1.5 \times 10^{128}$  mögliche Spielverläufe;
- suggeriert Sieg durch Logik, Intuition, Kreativität, Vorwissen;
- erfordert nur spezielle Schachintelligenz, keine „Alltagsintelligenz“.

Spielstärke wird in ELO-Punkten gemessen:

Spielstärken (Stand Juli 1999):

<i>G. Kasparow</i>	2851 ELO
<i>V. Anand</i>	2758 ELO
<i>A. Karpow</i>	2710 ELO
<i>Deep Thought 2</i>	2680 ELO

---

\* Arpad E. Elo, Administrator (1935 - 1937) of the Amer. Chess Federation.

Siehe auch: <http://de.wikipedia.org/wiki/Elo-Zahl> (27.04.2015)

# Schach (2)

---

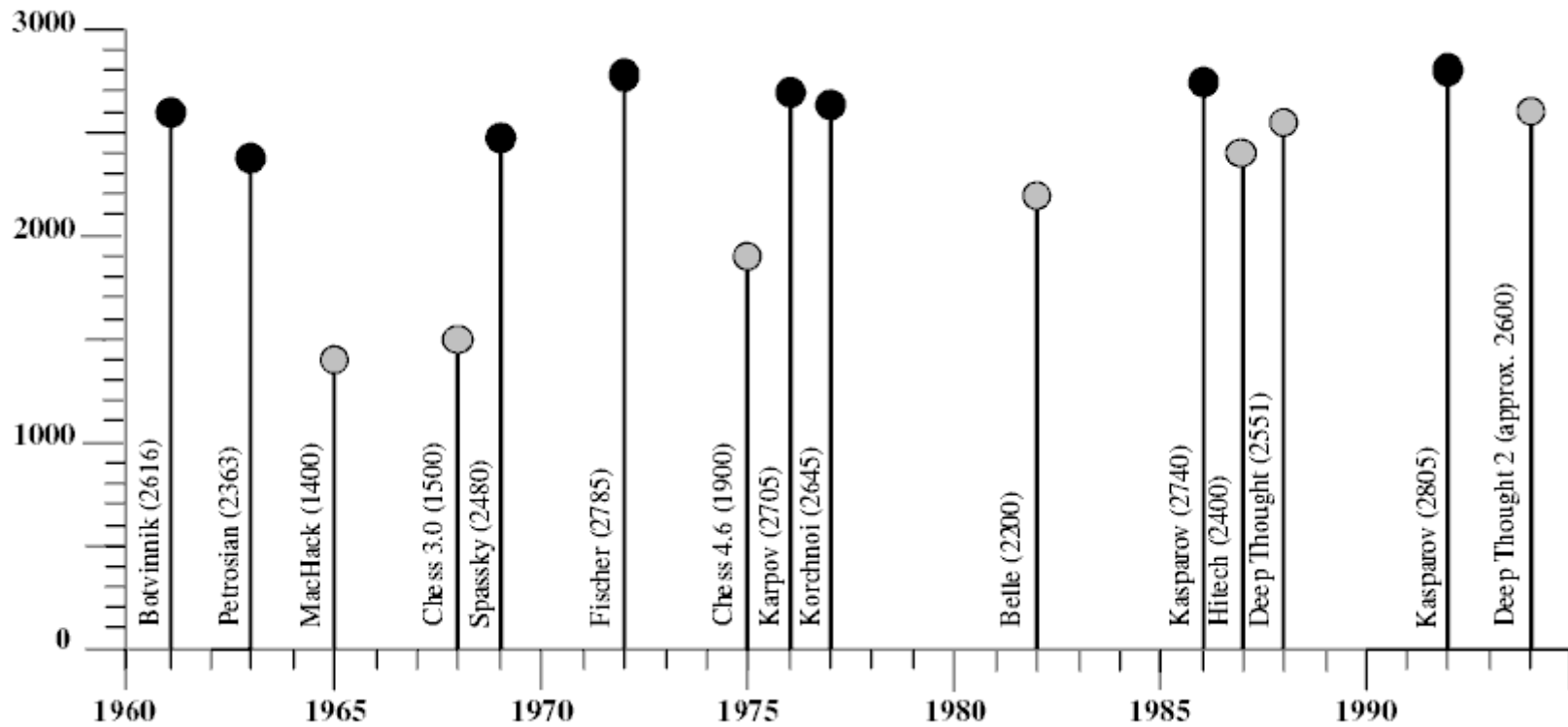
1997 wird der amtierende Weltmeister G. Kasparow erstmals in einem Spiel aus 6 Partien mit 3,5 zu 2,5 von einem Computer geschlagen!

→ Deep Thought 2 (IBM Thomas J. Watson Research Center)

- spezielle Hardware (32 Rechner mit 8 Chips, 2 Mio Berechnungen pro Sekunde)
- heuristische Suche
- fallbasiertes Schließen + Lerntechniken
  - 1996 Wissen aus 600.000 Schachpartien
  - 1997 Wissen aus 2 Mio. Schachpartien
  - Training durch Großmeister

Duell „Maschinenmensch Kasparow - Menschmaschine Deep Thought 2“

# Schach (3)



# Schach (4)

Elo-Zahlen der stärksten  
Schachprogramme<sup>[6]</sup>

Rang	Name	Punkte
1	Stockfish 8.0 x64 12CPU	3543
2	Houdini 5.0 x64 12CPU	3538
3	Komodo 10.3 x64 12CPU	3485
4	Gull 3.0 x64 12CPU	3248
5	Fritz 15 x64 12CPU	3175
6	Deep Shredder 13 x64 1CPU	3153
7	Ginkgo 1.8 x64 4CPU	3150
8	Jonny 8.00 x64 4CPU	3143
9	Rybka 4.1 x64 12CPU	3133
10	Andscacs 0.871 x64 4CPU	3130

## CEGT-Rangliste

Quelle:

<http://de.wikipedia.org/wiki/Schachprogramm>

(Abruf: 28.04.2017)

Zum Vergleich: Ein Großmeister von heute zeigt mind. ELO 2500. Im aktuellem Ranking führt Magnus Carlsen mit ELO 2840 (<https://de.wikipedia.org/wiki/Elo-Zahl>, 15.04.2019). Diese ELO-Zahlen für Schachprogramme sind aber nicht mit denen menschlicher Schachspieler zu vergleichen, da sie überwiegend durch Partien zwischen Computern ermittelt wurden und nicht durch Teilnahme an offiziellen Turnieren.

# Schach (5)

---

Eingesetzte Methoden und Techniken in Schachcomputern:

- Alpha-Beta-Suche
- . . . mit dynamischer Tiefenfestlegung bei unsicheren Positionen,
- gute (aber normalerweise einfache) Evaluierungsfunktionen,
- große Eröffnungsdatenbanken,
- sehr große Endspieldatenbanken (für Dame: alle 8-Steine Situationen),
- und sehr schnelle und parallele Rechner!

# Go (1)

---

## Stand im April 2012:

**Go:** Die besten Programme spielen etwas besser als Anfänger (Verzweigungsfaktor  $b > 300$ ).

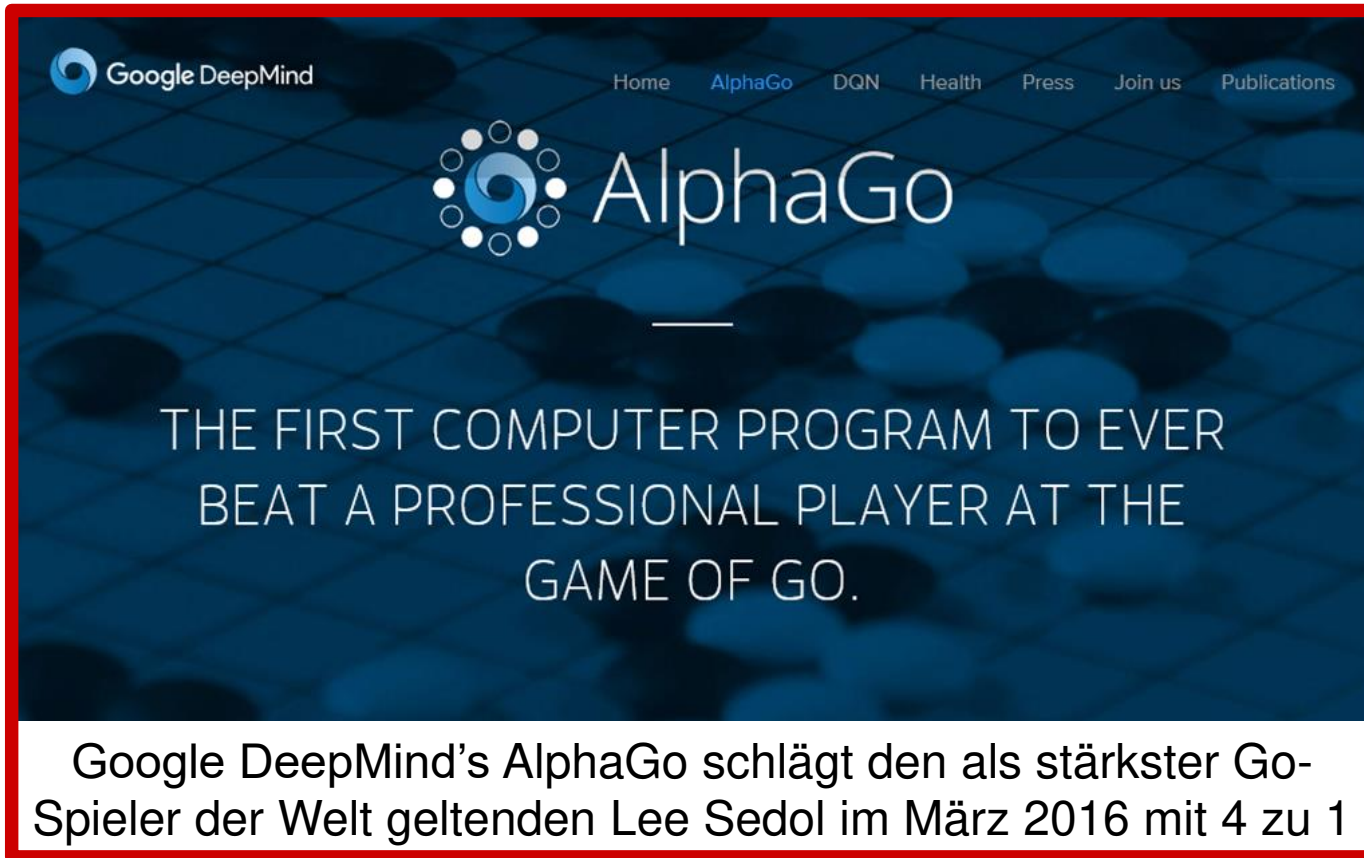
„Erst die Entwicklung von Supercomputern und dafür entwickelten Spezialprogrammen **erlaubte es erstmals im August 2008 gegen einen Go-Profi zu gewinnen, allerdings mit einer Vorgabe von neun Steinen.**

Die geringste Vorgabe, mit der seither (Stand April 2009) ein professioneller Spieler gegen ein Goprogramm verloren hat, beträgt sechs Steine.“

Zitat aus [http://de.wikipedia.org/wiki/Go\\_\(Spiel\)](http://de.wikipedia.org/wiki/Go_(Spiel)) (19.04.2012)

# Go (2)

Stand im März 2016:



The image is a screenshot of the AlphaGo website homepage. At the top left is the Google DeepMind logo. To its right is a navigation bar with links: Home, AlphaGo, DQN, Health, Press, Join us, and Publications. The main heading features the AlphaGo logo, which consists of a blue circular icon with white dots around it, followed by the text "AlphaGo". Below this is a horizontal line. The central text reads: "THE FIRST COMPUTER PROGRAM TO EVER BEAT A PROFESSIONAL PLAYER AT THE GAME OF GO." At the bottom of the screenshot, there is a white text box with a red border containing the German text: "Google DeepMind's AlphaGo schlägt den als stärkster Go-Spieler der Welt geltenden Lee Sedol im März 2016 mit 4 zu 1".

Google DeepMind's AlphaGo schlägt den als stärkster Go-Spieler der Welt geltenden Lee Sedol im März 2016 mit 4 zu 1



# Go (3)

---

## Stand im April 2016:

„**AlphaGo** [...] wurde von [Google DeepMind](#) entwickelt. Im Januar 2016 wurde bekannt, dass AlphaGo bereits im Oktober 2015 den mehrfachen Europameister [Fan Hui](#) ([2. Dan](#)) besiegt hatte. Damit ist es das erste Programm, das unter Turnierbedingungen ohne [Vorgabe](#) (Handicap) auf einem 19×19-Brett einen professionellen Go-Spieler schlagen konnte.<sup>[1]</sup> Im März 2016 schlug AlphaGo den Südkoreaner [Lee Sedol](#) [Anm.: 9. Dan], der als einer der weltbesten Profispieler angesehen wird ([AlphaGo gegen Lee Sedol](#)).“

Quelle: <https://de.wikipedia.org/wiki/AlphaGo> (28.04.2017)

“AlphaGo's algorithm uses a [Monte Carlo tree search](#) to find its moves based on knowledge previously "learned" by [machine learning](#), specifically by an [artificial neural network](#) (a [deep learning](#) method) by extensive training, both from human and computer play.”

Quelle: <https://en.wikipedia.org/wiki/AlphaGo> (28.04.2017)

# Zusammenfassung

---

- Ein **Spiel** ist i. A. ein Suchproblem in einer **Multiagenten-Umgebung**.
- Ein Spiel kann durch Angabe der **Zustandsmenge**, des **Anfangszustands**, der **Operatoren** (legale Züge), eines **Terminaltests** und einer **Nutzenfunktion** (Ausgang des Spiels) beschrieben werden.
- In 2-Personen-Brettspielen kann der **Minimax-Algorithmus** den besten Zug bestimmen, indem er den ganzen Spielbaum aufbaut.
- Der **Alpha-Beta-Algorithmus** liefert das gleiche Ergebnis, ist jedoch effizienter, da er überflüssige Zweige abschneidet.
- Normalerweise kann nicht der ganze Spielbaum aufgebaut werden, so dass man Zwischenzustände mit einer **Evaluierungsfunktion** bewerten muss.
- **Spiele mit Zufallselement** kann man mit einer **Erweiterung** des **Minimax** – und auch des **Alpha-Beta-Algorithmus** behandeln.