Entwurfstechnik Rekursion

Wer erinnern uns an die Definition der Fakultätsfunktion:

$1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{i=1}^{n} i = n!$

Über die Konstruktion der Funktion können wir auch folgendes ableiten:

$1 \cdot 2 \cdot 3 \cdot \ldots \cdot n = \prod_{i=1}^{n} i = \prod_{i=1}^{n-1} i \cdot n = (n-1)! \cdot n = \prod_{i=1}^{n-2} i \cdot (n-1) \cdot n = (n-2)! \cdot (n-1) \cdot n$

Wir können die Funktion nutzen, um sich selbst zu beschreiben! Das machen wir beispielsweise so:

$0! = 1$

$n! = (n-1)! \cdot n$

Rekursionstyp – Lineare Rekursion

Für eine Funktion f ist ein Funktionsanker $f(0)$ definiert und in jedem Zweig $f(n)$ gibt es höchstens einen rekursiven Aufruf $f(k)$ mit $k < n$.

Beispiel:

$0! = 1$

$n! = (n-1)! \cdot n$

Ein weiteres Beispiel ist die Definition zur Ermittlung des größten gemeinsamen Teilers zweier ganzer Zahlen n und m:

```
ggT(n,n) = n
ggT(n,0) = n
ggt(n,m) = ggT(m, n mod m), mit m>0
```

Rekursionstyp – Kaskadenförmige Rekursion

Es gelten die gleichen Bedingungen, wie bei der linearen Rekursion mit folgender Ausnahme: Sollten mehrere Funktionsaufrufe der Funktion f in einem vorkommen, dann sprechen wir von einer kaskadenförmigen bzw. baumartigen Rekursion. Ein typisches Beispiel dafür ist:

Fibonacci-Folge:

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), für n>1
```

Rekursionstyp – Verschachtelte Rekursion

Bei der verschachtelten Rekursion wird das Argument für den rekursiven Aufruf selbst durch einen rekursiven Aufruf bestimmt. Sie ist von großen theoretischem Interesse.

Die Ackermann-Funktion spielt in der Berechenbarkeitstheorie eine wichtige Rolle:

```
ack(x,0) = x+1
ack(0,y) = ack(1,y-1)          , für y>0
ack(x,y) = ack(ack(x-1,y),y-1) , für x>0 und y>0
```

Diese Funktion ist berechenbar, aber nicht primitiv-rekursiv.

Rekursionstyp – Wechselseitige Rekursion

Eine Funktion ruft eine zweite Funktion auf, die widerum auf die erste verweist.

Schauen wir uns folgendes Beispiel bestehend aus zwei Funktionen an, die prüfen, ob eine Eingabe gerade oder ungerade ist:

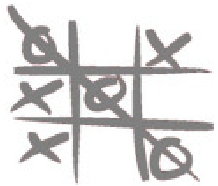
$$istGerade(n) = \begin{cases} true & n = 0 \\ istUngerade(n-1) & n > 0 \end{cases}$$

$$istUngerade(n) = \begin{cases} false & n = 0 \\ istGerade(n-1) & n > 0 \end{cases}$$

Wir machen einen Abstecher in die Spieltheorie

TicTacToe - Eigenschaften

Zunächst betrachten wir die speziellen Eigenschaften und Spielregeln vom Spielklassiker Tic-Tac-Toe.



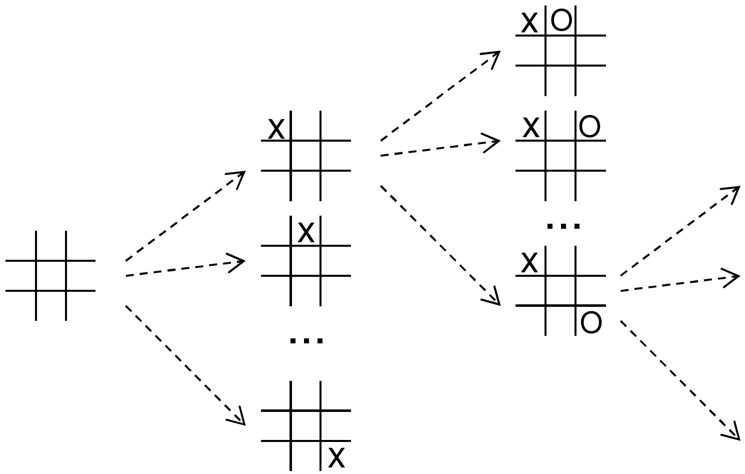
alternierendes Zwei-Spieler-Spiel mit Spieler X und O

vollständige Information

Nullsummenspiel

des einen Gewinn ist des anderen Verlust

Spielablauf



Komplexität

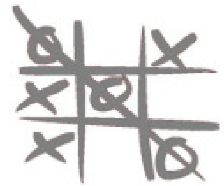
Mögliche Spielverläufe: 9*8*7*...*2*1 = 9!
das entspricht 362.880

Legale Wege gibt es sogar nur 255.168.

Ohne Rotation und Spiegelung gibt es sogar nur 765 unterschiedliche Spielsituationen.

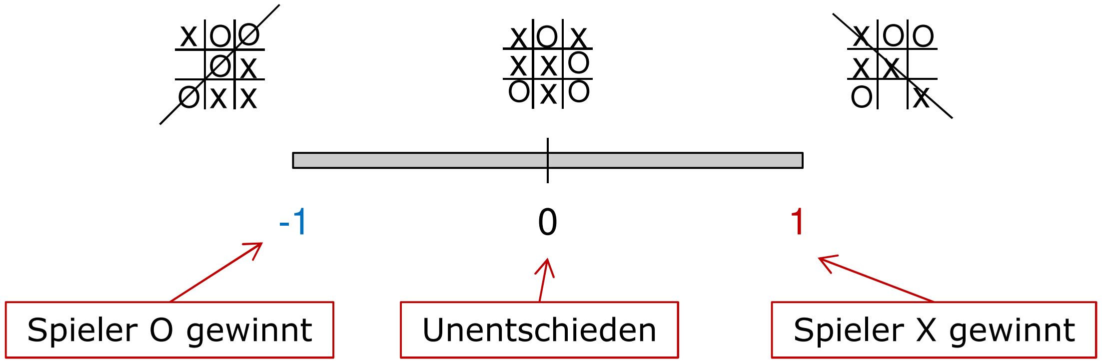
Damit ist die Komplexität von Tic-Tac-Toe relativ klein und die Lösung sehr einfach.

TicTacToe - Bewertungsfunktion

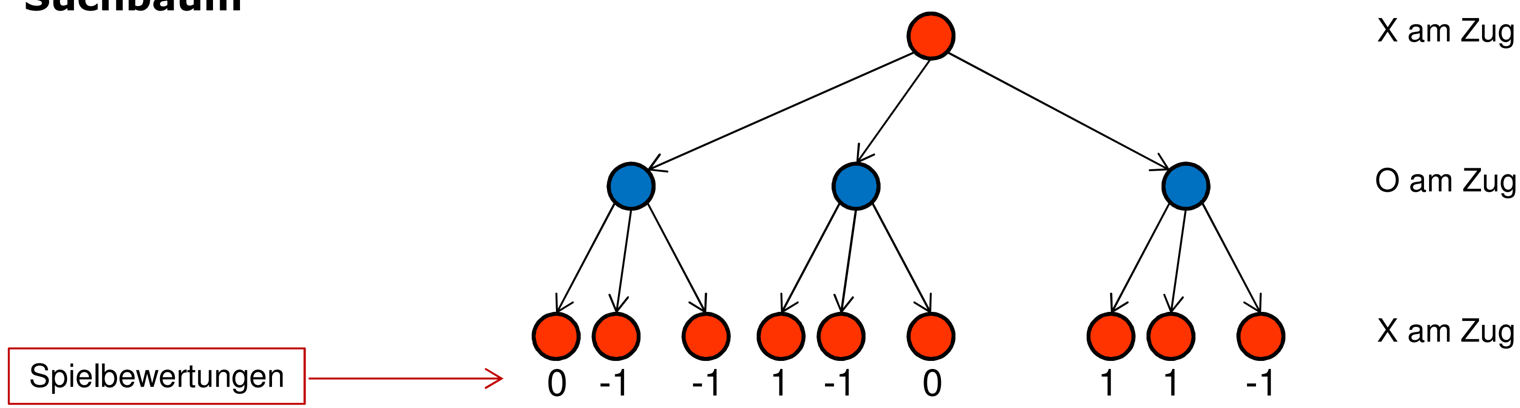


Wir benötigen eine Bewertungsfunktion, die besagt, welcher Spieler gewonnen hat oder ob eine Partie unentschieden endete.

Bewertungsfunktion



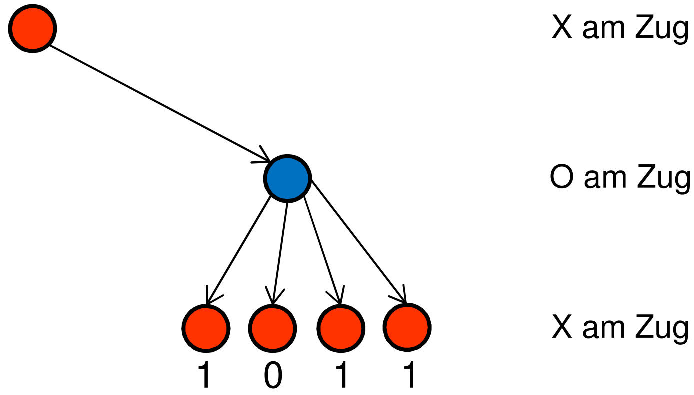
Suchbaum

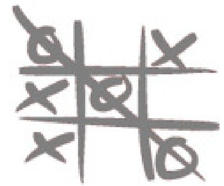


?

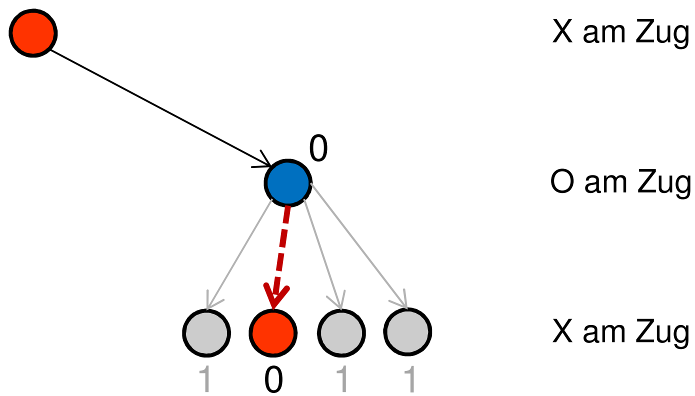
TicTacToe – lokale Betrachtung für Spieler O

Betrachten wir die folgende Spielsituation für den Spieler O



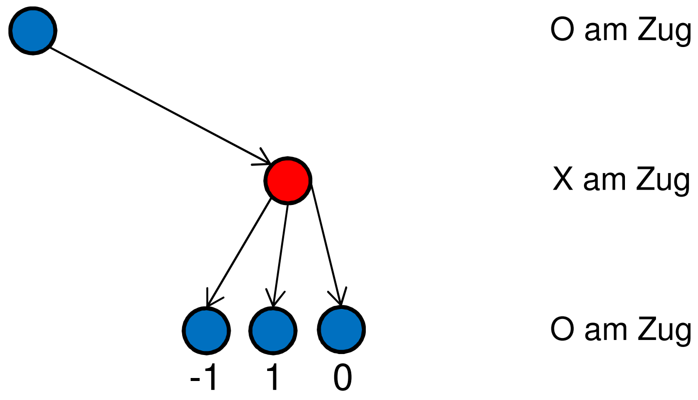


Spieler O wird sich für den vielversprechendsten Zug entscheiden und wählt das **Minimum** der zur Verfügung stehenden Möglichkeiten..

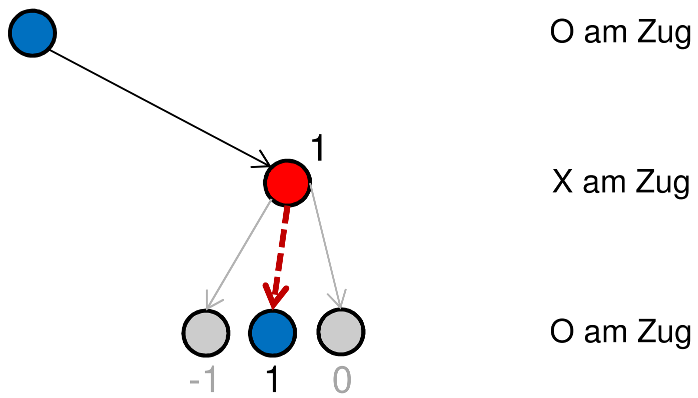


TicTacToe – lokale Betrachtung für Spieler X

Betrachten wir die folgende Spielsituation für den Spieler X

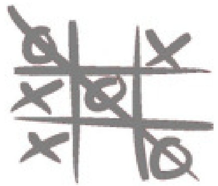


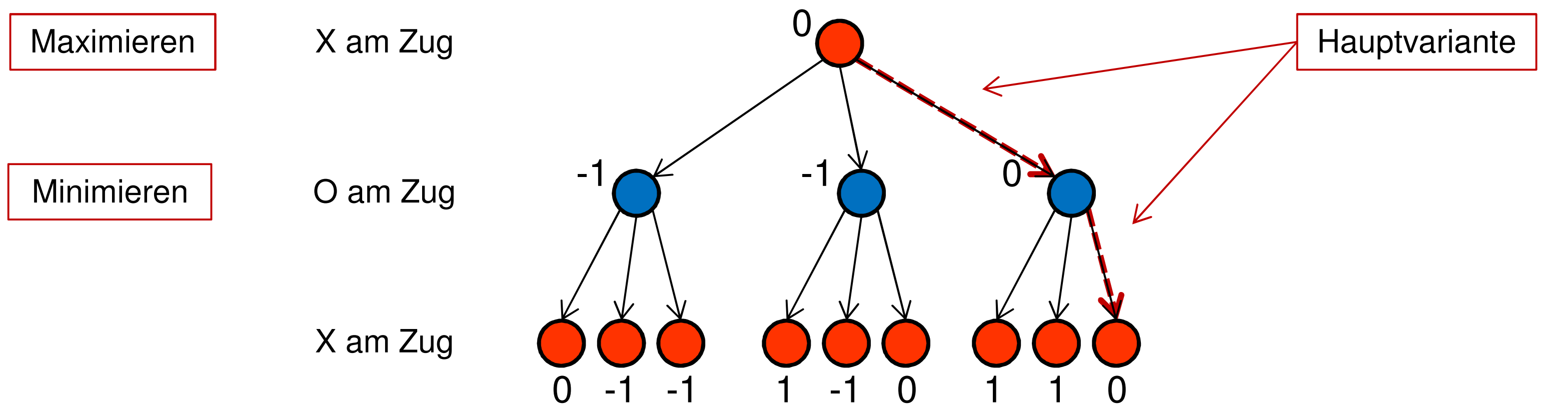
Auch Spieler X wird sich für den vielversprechendsten Zug entscheiden und wählt in diesem Fall das **Maximum** der zur Verfügung stehenden Möglichkeiten..



TicTacToe – MinMax-Strategie

Wenn wir davon ausgehen, dass jeder Spieler immer den für sich aktuell besten Zug spielt, dann können wir die MinMax-Strategie (Claude Elwood Shannon 1950) formulieren:





Formal läßt sich MinMax, wie folgt beschreiben:

$$minmax(n) = \begin{cases} eval(n), & \text{falls } n \text{ terminaler Zustand} \\ \max_{m \in kinder(n)} minmax(m), & \text{falls } n \text{ Max-Knoten} \\ \min_{m \in kinder(n)} minmax(m), & \text{falls } n \text{ Min-Knoten} \end{cases}$$

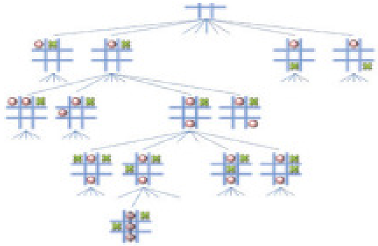
Ein Unbesiegbares Programm!

Mit Hilfe der MinMax-Strategie können wir für Spiele, mit den folgenden Eigenschaften ...

alternierendes Zwei-Spieler-Spiel mit Spieler X und O

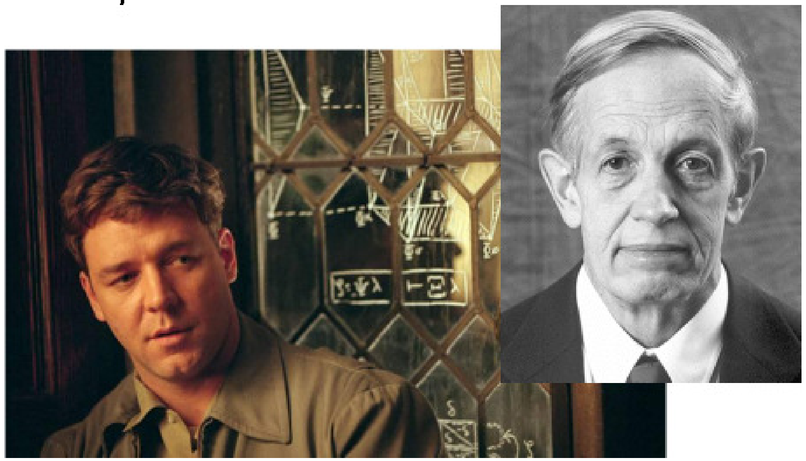
vollständige Information

Nullsummenspiel



Geringe Komplexität

... ein Programm schreiben, das unbesiegbar ist, wenn das Spiel ausgeglichen ist.



John Forbes Nash
1994 Nobelpreis für Wirtschaftswissenschaften

Spieltheorie: Nash-Equilibrium

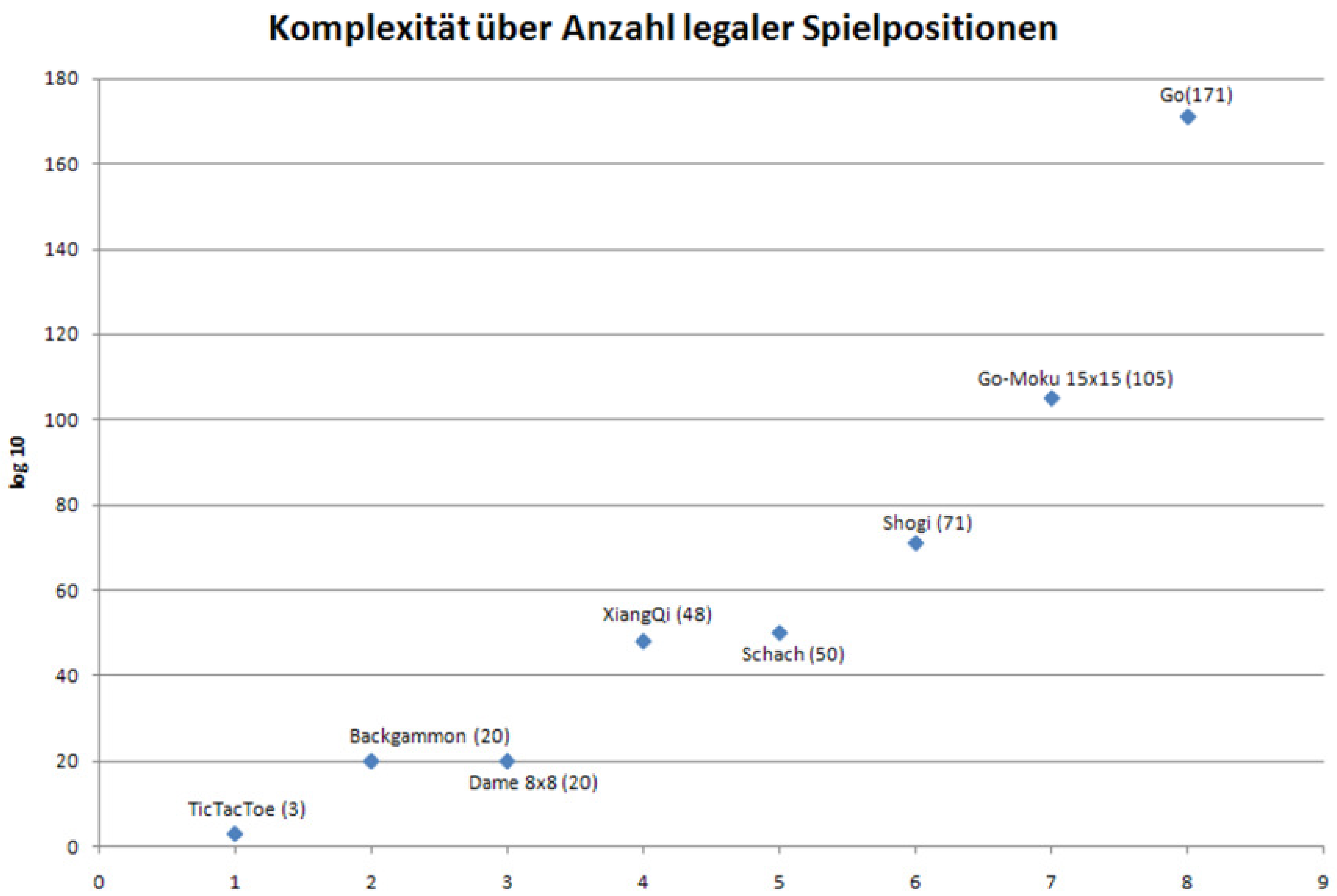
Suchraumkomplexität von Spielen

Die Komplexität von Spielen und die Anwendungen der jeweiligen Lösungsstrategien sind aus akademischer Sicht sehr interessant.

Hier ein paar Informationen zum aktuellen Forschungsstand:

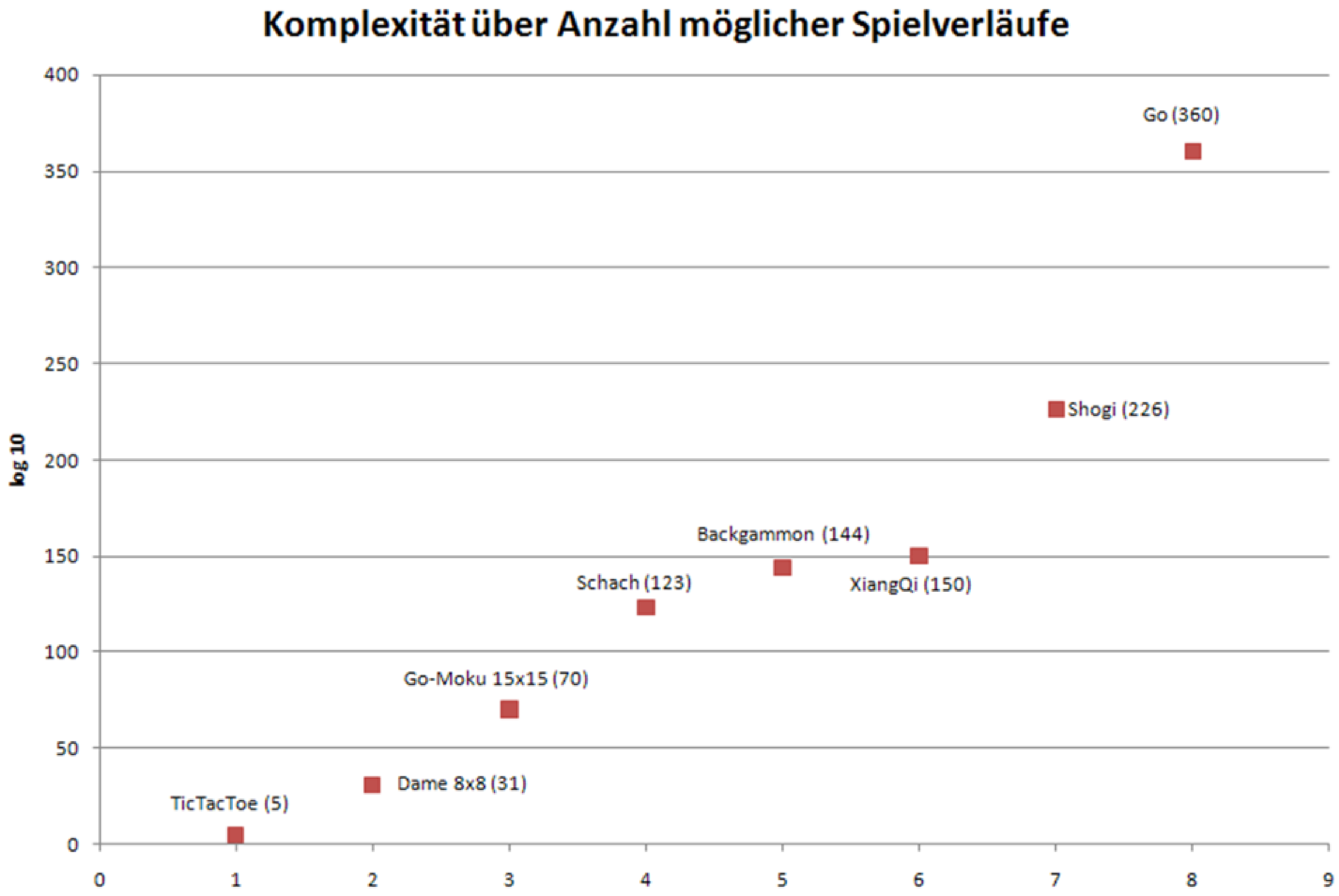
- Das Spiel Dame (checkers) gilt seit 2007 als „weakly solved". **Chinook** ist nicht zu besiegen.
- Im Schach sind Maschinen den menschlichen Großmeistern seit den 1990'ern ebenwürdig. Bestes Programm: **Rybka**
- Go ist die neue *Drosophila melanogaster* der Künstlichen Intelligenz, aktuelle Programme haben aber nur Amateur-Niveau

Komplexität - Spielpositionen



Im Vergleich dazu: Anzahl Atome der Erde (51) und Anzahl Atome im Weltall (78)

Komplexität - Spielverläufe



Komplexität – Was bedeutet eine hohe Komplexität?

Angenommen ein Spiel hat einen konstanten Verzweigungsfaktor von 30 (30 Aktionsmöglichkeiten pro Stellung) und eine durchschnittliche Tiefe von 50 Zügen.

Der entsprechende Suchbaum hätte $\sum_{d=0}^{50} 30^d = \frac{1-30^{51}}{1-30} = 7.4 * 10^{73}$ Knoten.

Angenommen wir hätten 10.000 Computer, die jeweils eine Milliarde Suchschritte pro Sekunde schaffen, und man könnte die Arbeit ohne Verluste auf alle Rechner verteilen, dann beläuft sich die Rechenzeit auf ca.:

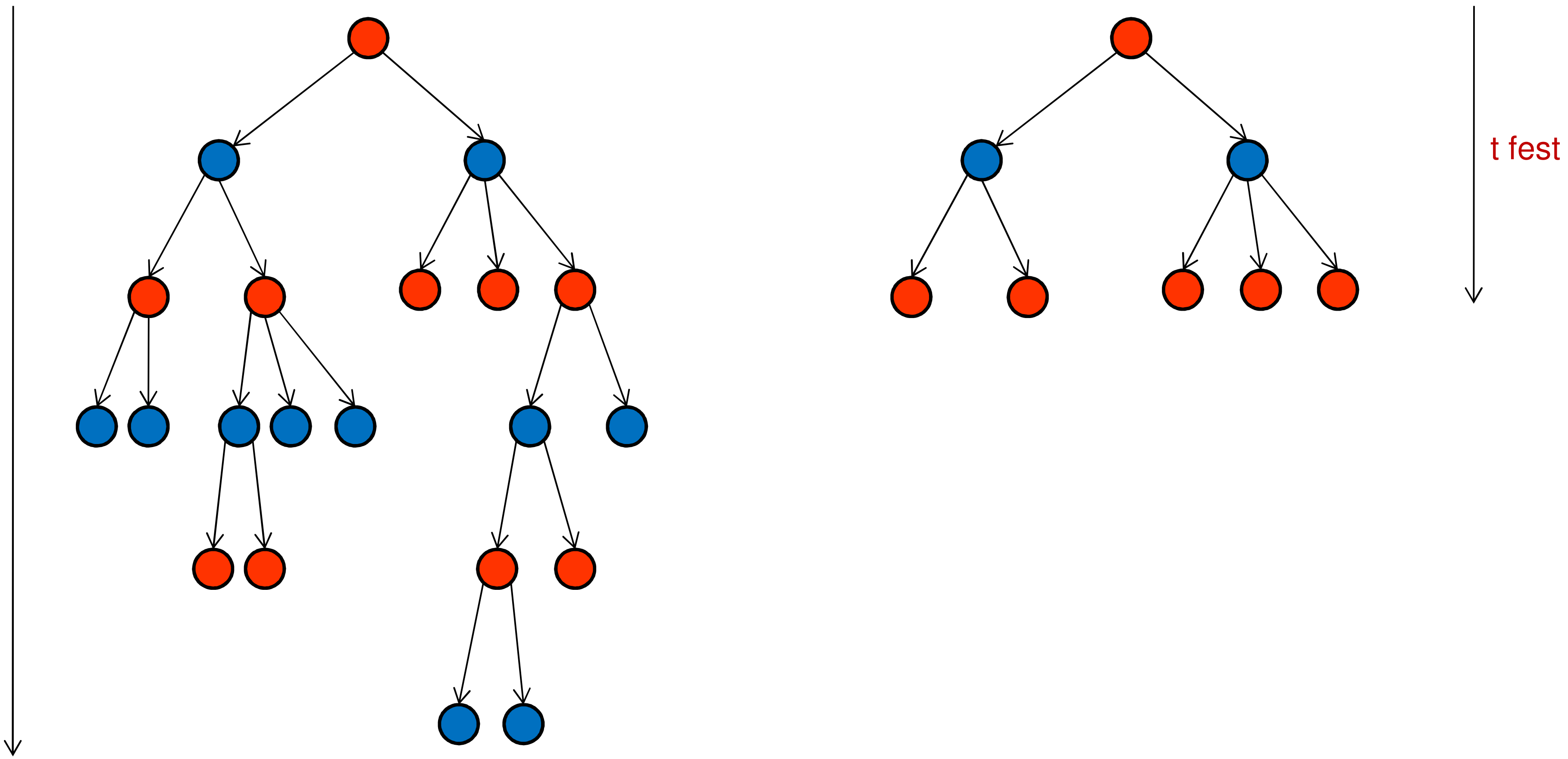
$$\frac{7.4 * 10^{73} \text{ Rechenschritte}}{10000 * 10^{9} \text{ Rechenschritte/sec}} = 7.4 * 10^{60} \text{ sec} = 2.3 * 10^{53} \text{ Jahre}$$

Dies ist „zum Glück“ nur 10^{43} mal so lange wie unser Universum alt ist.

Wie kann es dann trotzdem sein, dass ein Schachcomputer gegen einen menschlichen Weltmeister gewinnt?

Suchtiefe begrenzen

Die einzige Möglichkeit trotz einer hohen Komplexität eine vernünftige Zugauswahl zu treffen, besteht darin, die Suchtiefe zu begrenzen.



Allgemeine Bewertungsfunktion

Es werden unterschiedliche Bewertungskriterien als Funktionen beschrieben und diese gewichtet aussummiert. Das funktioniert auf Grund der Tatsache, da es sich um ein Nullsummenspiel handelt.

$$\sum_{i=1}^{n} \alpha_i f_i(x)$$

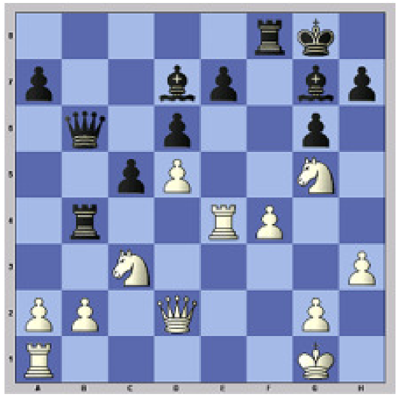
Diese Funktionen werden unabhängig voneinander bestimmt, jeweils für Schwarz und für Weiß. Die Differenz ergibt den Funktionswert.

$$f_1(S) = material(W) - material(S)$$

$$f_2(S) = mobilität(W) - mobilität(S)$$

Bewertungsfunktion im Schach

Aus der Schachliteratur sind viele Muster bekannt, diese können nun lokalisiert und entsprechend bewertet werden.

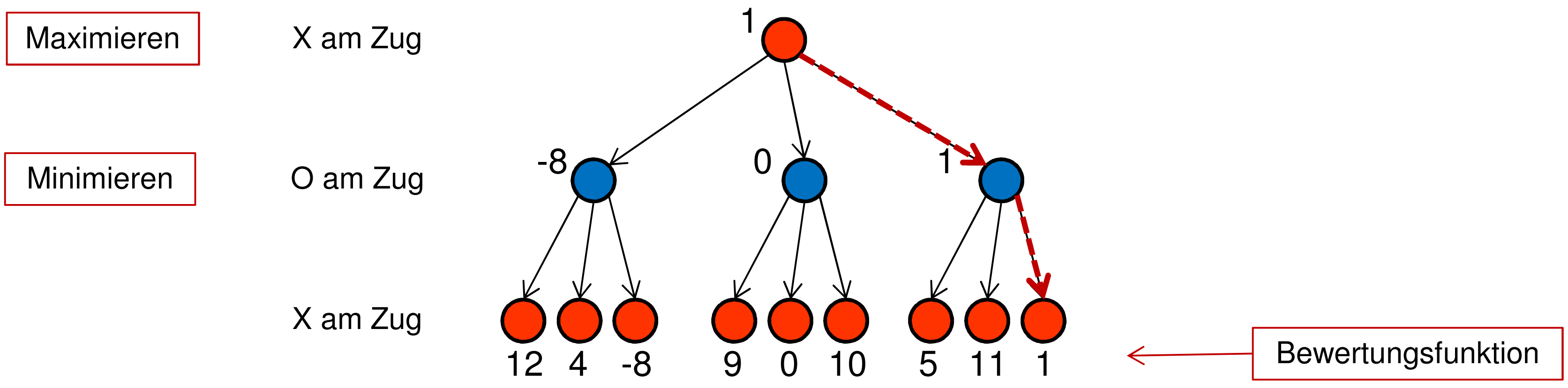


```
BishopPair
Knight_Outpost
Supported_Knight_Outpost
Connected_Rooks
Opposite_Bishops
Opening_King_Advance
King_Proximity
Blocked_Knight
Draw_Value
No_Material
Bishop_XRay
Rook_Pos
Pos_Base
Pos_Queenside
Bishop_Mobility
Queen_Mobility
Knight_SMobility
Rook_SMobility
King_SMobility
Threat
Overloaded_Penalty
Q_King_Attack_Opponent
NoQ_King_attack_Opponent
NoQueen_File_Safty
Attack_Value
Unsupported_Pawn
Passed_Pawn_Control
Doubled_Pawn
Odd_Bishop_Pawn_Pos
King_Passed_Pawn_Supported
Passed_Pawn_Rook_Supported
Blocked_EPawn
Pawn_Advance
King_Passed_Pawn_Defence
Pawn_Defence
Mega_Weak_Pawn
Castle_Bonus
Bishop_Outpost
Supported_Bishop_Outpost
Seventh_Rank_Rooks
Early_Queen_Movement
Mid_King_Advance
Trapped_Step
Useless_Piece
Near_Draw_Value
Mating_Positions
Ending_King_Pos
Knight_Pos
Pos_Kingside
Knight_Mobility
Rook_Mobility
King_Mobility
Bishop_SMobility
Queen_SMobility
Piece_Values
Opponents_Threat
Q_King_Attack_Computer
NoQ_King_Attack_Computer
Queen_File_Safty
Piece_Trade_Bonus
Pawn_Trade_Bonus
Adjacent_Pawn
Unstoppable_Pawn
Weak_Pawn
Blocked_Pawn
Passed_Pawn_Rook_Attack
Blocked_DPawn
Pawn_Advance
Pawn_Advance2
Pawn_Pos
Isolated_Pawn
Weak_Pawn_Attack_Value
```

... und viele mehr!

MinMax-Strategie mit Bewertungsfunktion

Die MinMax-Strategie lässt sich auch mit einer beschränkten Tiefe und einer Bewertungsfunktion problemlos ausführen, um den besten Zug zu bestimmen.



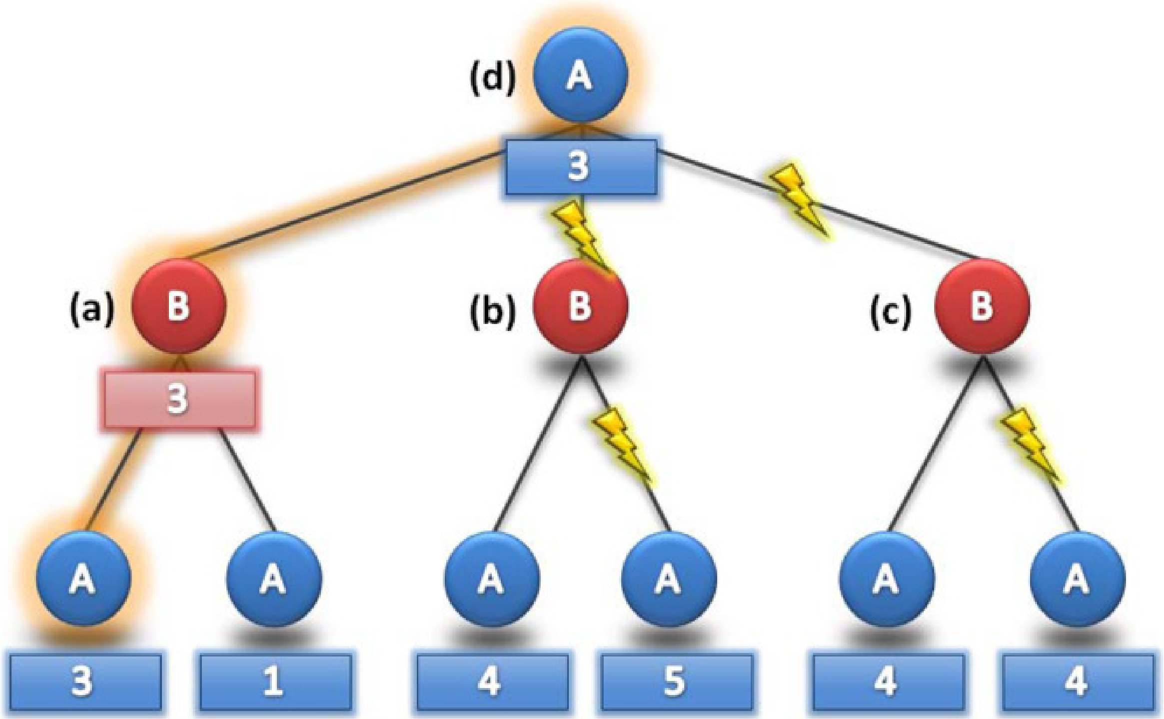
MinMax-Strategie und Alpha-Beta-Pruning

Es gibt jedoch Situationen im Spielbaum, bei denen einzelne Zweige nicht mehr betrachtet werden müssen, da sie ein Maximum/Minimum am Elternknoten nicht mehr verändern.

Das Alpha-Beta-Pruning nutzt dies aus, um unnötige Äste im Baum „abzuschneiden".

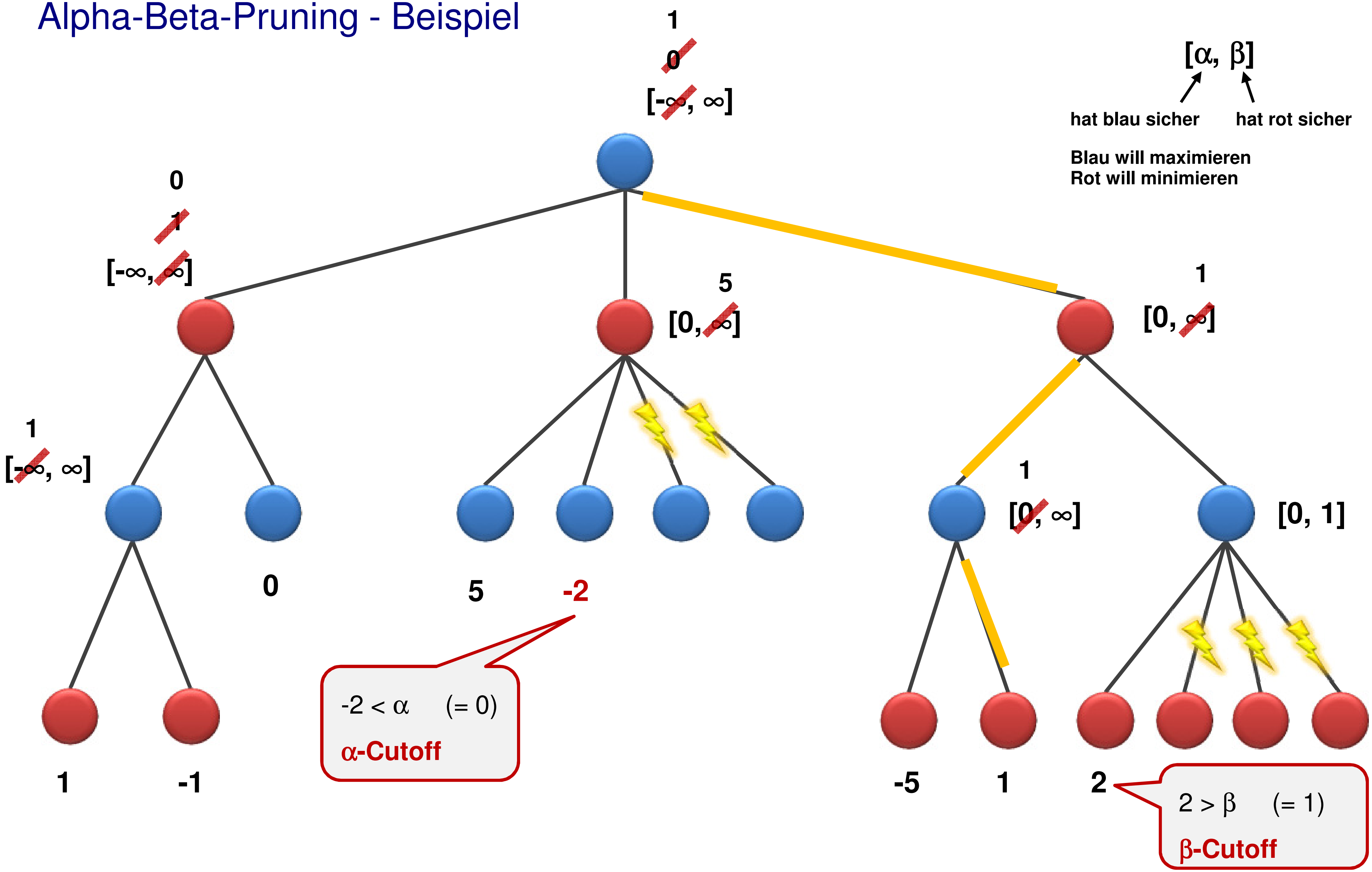
Minimieren

Maximieren



<u>Laufzeit:</u> Im worst case genauso wie Min-Max: **$O(b^t)$**
Im best case (perfekte Zugsortierung): **$O(b^{d/2})$**
Bei zufälliger Zugsortierung: **$O(b^{3d/4})$**

Alpha-Beta-Pruning - Beispiel

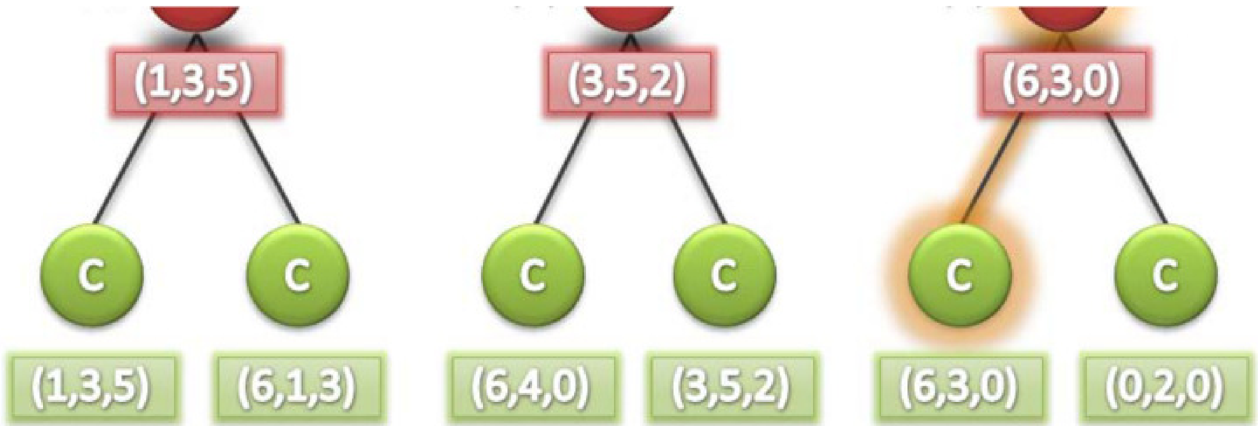


Wie sieht es mit Mehrspielerspielen aus?

Problematik mit Mehrspieler-Spielen

Um Min-Max für mehrere Spieler anwenden zu können, müssen einige Anpassungen vorgenommen werden.

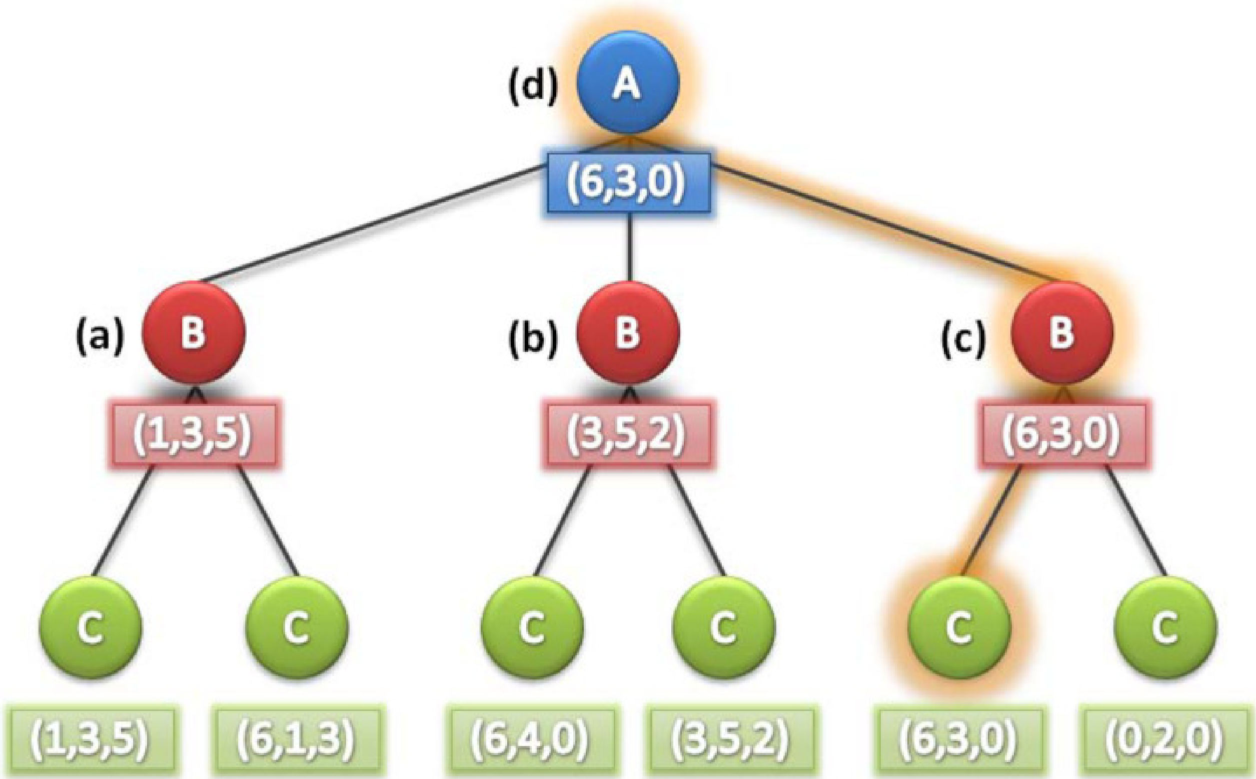
Für jede Ebene muss gespeichert werden, welcher Spieler an der Reihe ist und die Bewertungsfunktion darf nicht mehr einen Wert zurückliefern, sondern muss einen Nutzenvektor erstellen, der die Bewertungen für alle Spieler aus ihrer Sicht enthält.



Es existieren nun jedoch **mehrere Möglichkeiten** die beste Alternative zu wählen.

maxN-Variante

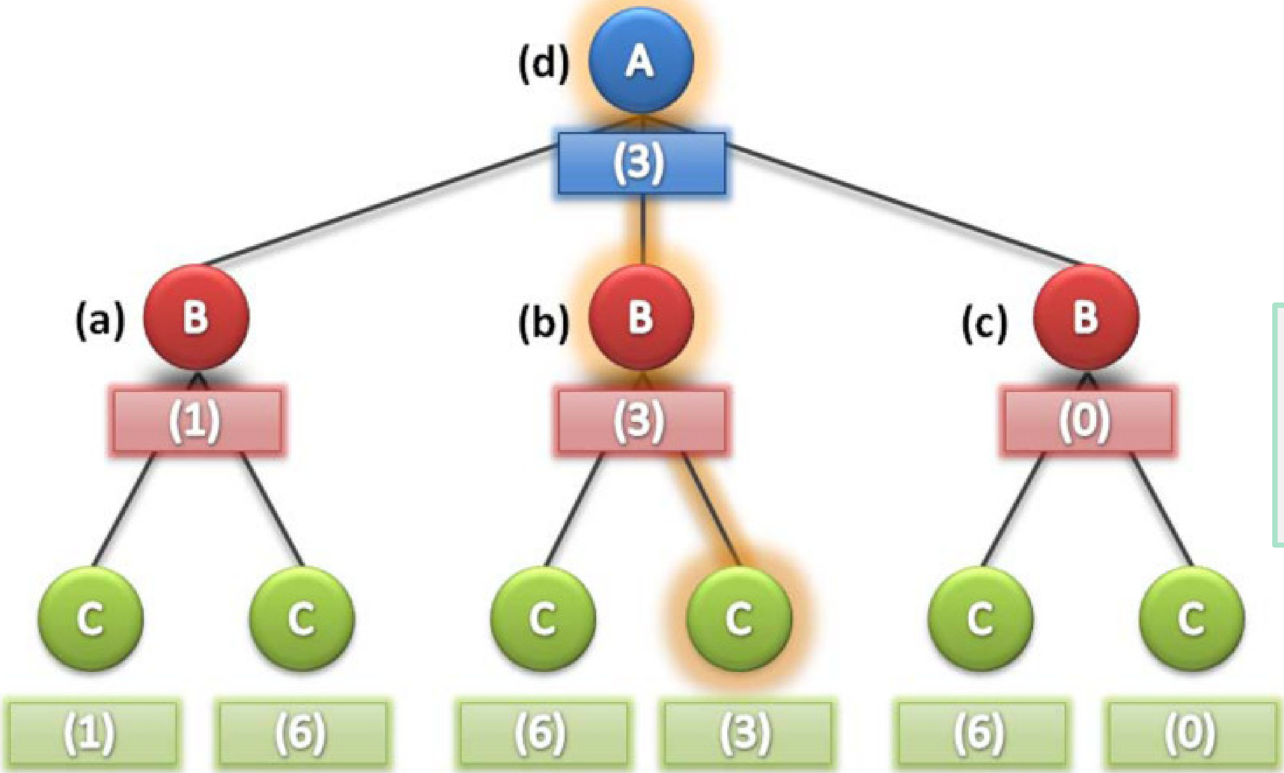
Bei der maxN-Variante, verhalten sich alle Spieler optimistisch und versuchen jeweils ihren eigenen Nutzen im gegebenen Nutzenvektor zu maximieren:



Paranoid-Variante

Im Gegensatz zur maxN-Variante, können sich die Spieler aber auch pessimistisch verhalten und die Annahme treffen, dass sich alle Gegenspieler zu einer Allianz gegen sich selbst geschlossen haben.

Die Annahme der Paranoid-Variante ist also, dass sich die Gegner verbünden und vorhaben den eigenen Nutzen stets zu minimieren:



Selber Suchbaum wie vorhin. Eingetragen ist nur der Nutzenwert vom suchenden Spieler A

Repräsentation und Zuggenerator

Zu einer guten Suche wird eine brauchbare Stellungsrepräsentation mit schnellem Zuggenerator benötigt.

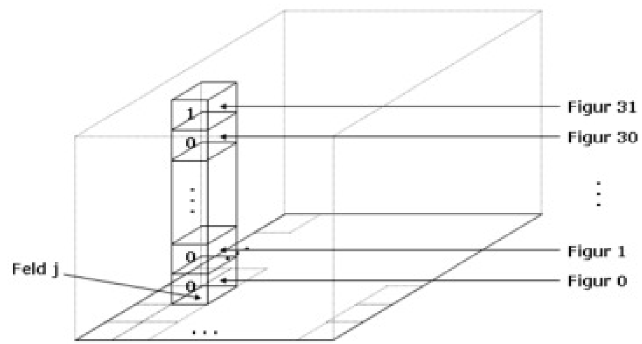
8x8 Spielbrett

-4	-2	-3	-5	-6	-3	-2	-4
-1	-1	-1	-1	-1	-1	-1	-1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
4	2	3	5	6	3	2	4

BitBoards

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0

ToPieces-Board



64 BIT-Wort

0	0	1	1	1	0	...	0	0

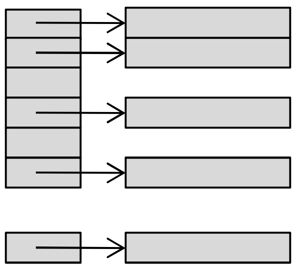
\>> 1 =

0	0	0	1	1	1	...	0	0

Optimierungen der Suche

Es gibt zahlreiche Möglichkeiten, die Suchtiefe von Schachprogrammen zu erhöhen und sie damit entsprechend spielstärker zu machen.

Transpositions-tabellen



r_i XOR (r_j XOR r_k) = (r_i XOR r_j) XOR r_k

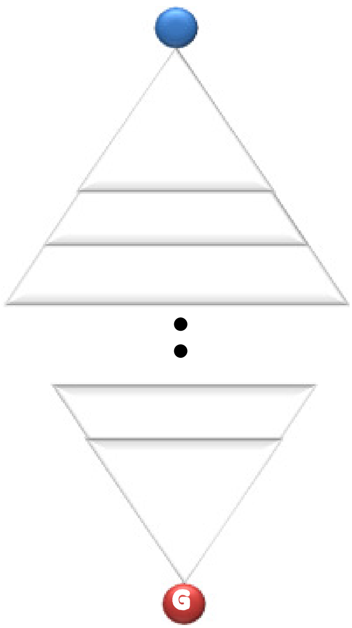
r_i XOR r_j = r_j XOR r_i

r_i XOR r_i = 0

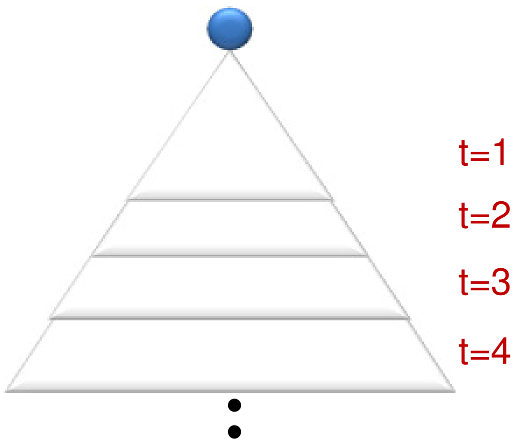
if $s_i = r_1$ XOR r_2 XOR $\cdots$ XOR r_i then $\{s_i\}$

$\{s_i\}$ is uniformly distributed.

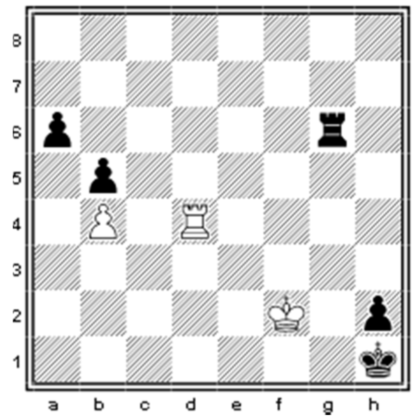
Eröffnungsbuch und Endspieldatenbank



Iterative Tiefensuche



Nullmoves



Desweiteren kommen folgende Optimierungen zum Einsatz: Zugsortierungen, Hauptvarianten, Killer- und Historyheuristik, Ruhesuche, Aspiration-windows, Parameterjustierung, ...

Zobristkey I

Der Zobristkey stellt eine vollständige Brettrepräsentation dar, wie beispielsweise der FEN-Code, und wird binär mit 64-BIT dargestellt.

Alle relevanten Spieleigenschaften werden mit Zufallszahlen gefüllt (Beispiel aus einer Schach-Engine):

```
long[6][2][120] PIECES     // [piece type][side to move][square]
long[4] W_CASTLING_RIGHTS  // Four different castling setups
long[4] B_CASTLING_RIGHTS  // both ways, short, long and none
long[120] EN_PASSANT       // En passant
long SIDE                  // Used for changing sides

SIDE = rnd.nextLong();
for(int i = 0; i < 4; i++) {
      W_CASTLING_RIGHTS[i] = rnd.nextLong();
      B_CASTLING_RIGHTS[i] = rnd.nextLong();
}
```

Der Trick besteht nun in der Verwendung der XOR-Funktion. Schauen wir uns die Verwendung des Zobristkeys an:

```
long zobristKey = 0;

for(int index = 0; index < 120; index++) {
   int piece = board.boardArray[index];
   if(piece > 0) // White piece
      zobristKey ^= PIECES[Math.abs(piece)-1][0][index];
   else if(piece < 0) // Black piece
      zobristKey ^= PIECES[Math.abs(piece)-1][1][index];
}
```

Zobristkey II

So können wir entsprechend auch die anderen Eigenschaften des aktuellen Spielzustands in den Schlüssel eintragen:

```
zobristKey ^= W_CASTLING_RIGHTS[board.white_castle];
zobristKey ^= B_CASTLING_RIGHTS[board.black_castle];

if (board.enPassant != -1) zobristKey ^= EN_PASSANT[board.enPassant];
if (board.toMove == -1)    zobristKey ^= SIDE;
```

Damit haben wir die aktuelle Stellung also in 64-BIT kodiert. Bei der Suche soll dieser Schlüssel aber nicht immer wieder neu berechnet werden. Deshalb kann bei einem Zug vor und zurück, der millionenfach in der Suche ausgeführt wird, auf folgende Codezeilen zurückgegriffen werden:

```
zobristKey ^= Zobrist.PIECES[3][0][16];
zobristKey ^= Zobrist.PIECES[3][0][32];
```

Die Bedeutung dessen könnte sein: Ziehe weissen Turm von a2 nach a3. In der ersten Zeile löschen wir ihn einfach vom Brett, denn es gilt ja gerade **a xor b=c** und **c xor b=a**. Anschließend setzen wir den Turm in der zweiten Zeile auf das neue Feld.

Endständige Rekursion

Bei der Rekursion ist die letzte Aktion in der Funktion der rekursive Aufruf. So lässt sich z.B. die Summe der Zahlen von 1 bis n endständig rekursiv notieren.

```
sum n = sum‘ n 0
sum‘ 0 a = a
sum‘ n a = sum‘ (n-1) (n+a)
```

Entwurfstechnik – Brute-Force

Mit der Brute-Force-Strategie (Methode der rohen Gewalt) werden einfach alle Möglichkeiten durchprobiert, die für eine Lösung in Frage kommen können.

Beispiel SlowSort:

```
prüfe alle Permutationen, ob diese sortiert sind
```

Laufzeitbetrachtung:

best-case: $\Theta(n)$
worst-case: $\Theta(n!)$
average-case: $\Theta(n!)$

In die gleiche Klasse gehören auch RandomSort, BogoSort, StupidSort, MonkeySort.

Greedy-Strategie

Für ein Problem, das in Teilschritten gelöst werden kann, wird für einen Teilschritt die Lösung ausgewählt, die den aktuell bestmöglichen Gewinn verspricht.

Beispiel Geldrückgabe an der Kasse:

```
gib zunächst den Schein/die Münze zurück mit dem größtmöglichen
Wert, der kleiner oder gleich der Restsumme ist
```

Eine Lösung gilt dabei als optimal, wenn die Anzahl der zurückgegebenen Scheine/Münzen minimal ist.

Beispiel:

```
7,82 Euro
```

```
1*5 Euro
1*2 Euro
1*50 Cent
1*20 Cent
1*10 Cent
1*2 Cent
```

6 Münzen/Scheine

Dynamische Programmierung

Bei der Dynamischen Programmierung wird die optimale Lösung aus optimalen Teillösungen zusammengesetz. Teillösungen werden in einer geeigneten Datenstruktur gespeichert, um die kostspielige Rekursion zu vermeiden.

```
fib(0) = 0
fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), für n>1
```

Daraus ergibt sich die Folge:
0,1,1,2,3,5,8,13,21,34,55, ...

Ein neuer Funktionswert ergibt sich aus der Summe der beiden Vorgänger. Nehmen wir eine Liste als Datenstruktur, dann können wir die Fibonacci-Zahlen effizient ermitteln:

```
fib(n):
   fib[0] = 0
   fib[1] = 1
   for (i:=2 to n)
      fib[i] = fib[i-1] + fib[i-2]
   return fib[n]
```

Dynamische Programmierung mit Memoisierung

Die Memoisierung ist dem Konzept der dynamischen Programmierung ähnlich. Eine Datenstruktur wird dabei verwendet um bereits ermittelte Berechnungen zu speichern und an späterer Stelle (neuer Funktionaufruf) wieder zu verwenden.

```
fib[0] = 0
fib[1] = 1

fib(n):
   if (fib[n] enthält Wert)
      return fib[n]
   else
      fib[n] = fib(n-1) + fib(n-2)
      return fib[n]
   endelse
```