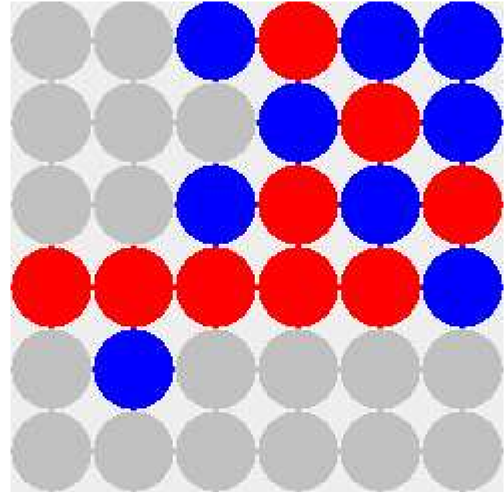
Sind Computer (un)schlagbar?
Computerstrategien für Zweipersonenspiele

Daniel Garmann

Die Entwicklung von Computerstrategien für Zweipersonenspiele mit vollständiger Information wie z. B. Schach, Mühle oder Vier Gewinnt führt zuerst zu dem naiven Ansatz, alle möglichen Spielverläufe durchzuspielen und zu bewerten. Allerdings erkennt man schnell, dass dieses Verfahren an seine Berechnungsgrenzen stößt und dass deshalb Optimierungen notwendig werden. In diesem Aufsatz wird das NegaMax-Suchverfahren und dessen Optimierungen (Alpha-Beta-Suche ohne/mit Zugsortierung und Principal-Variation-Suche) vorgestellt und am Beispiel des Spiels "Fünf in einer Reihe" erläutert.



Steckbrief für den Unterrichteinsatz:

Was die Schülerinnen und Schüler bei diesem Thema lernen:

- NegaMax-Algorithmus und dessen Optimierungen: Alpha-Beta-Suche ohne/mit Zugsortierung, Principal-Variation-Suche, ansatzweise iterative Tiefensuche
- Wiederverwendbarkeit der Algorithmen in unterschiedlichen Strategiespielen durch Klassenbildung
- Laufzeitanalyse von Backtracking-Algorithmen durch Darstellung als Baumstrukturen

Einsatzszenarien

- Unterrichtsreihe im Grund- oder Leistungskurs Informatik in der Jahrgangsstufe 12 oder 13
- Facharbeiten im Informatikunterricht der Sekundarstufe II

Unabdingbare technische Voraussetzungen

- Zugang zu einem PC mit einer objektorientierten Programmiersprache

Vorkenntnisse

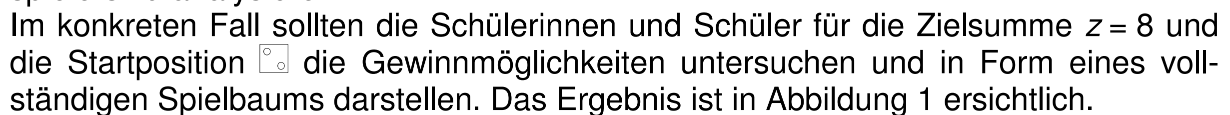
- Grundlagen in der objektorientierten Programmierung
- schnelle Sortieralgorithmen (z. B. Quicksort)
- rekursive Algorithmen und Backtracking-Strategien

Vollständig berechenbare Zweipersonenspiele

Das Spiel Würfelkippen

Den Einstieg in die Thematik der Strategiespiele bildete ein einfaches Zweipersonenspiel, welches ausführlich in Baumann (1994) beschrieben ist: das Spiel Würfelkippen.

Ein Würfel wird geworfen, die oben liegende Augenzahl bildet die Startposition. Beide Spieler einigen sich auf eine zu erreichende Zielsumme *z*, welche genau erreicht werden muss, um das Spiel zu gewinnen. Abwechselnd kippen nun beide Spieler den Würfel über eine der Grundkanten. Die jeweils nach oben gelangte Augenzahl wird nach jedem Schritt aufsummiert. Die Augensumme darf allerdings die zuvor festgelegte Zielsumme *z* nicht überschreiten.
Wer die Zielsumme *z* genau erreicht, gewinnt das Spiel. Gelingt es keinem Spieler, die genaue Punktzahl zu erreichen, so endet das Spiel unentschieden.

Zuerst wurden die Schülerinnen und Schüler dazu angehalten, dieses Spiel mit unterschiedlichen Startpositionen und unterschiedlichen Zielsummen durchzuführen, um einen ersten Einblick in die Thematik Spielstrategien zu erhalten. In einem zweiten Schritt wurden die Schülerinnen und Schüler dann aufgefordert, eine konkrete Spielsituation bis zum Ende durchzuspielen und die Gewinnmöglichkeiten des Startspielers zu analysieren.
Im konkreten Fall sollten die Schülerinnen und Schüler für die Zielsumme $z = 8$ und die Startposition  die Gewinnmöglichkeiten untersuchen und in Form eines vollständigen Spielbaums darstellen. Das Ergebnis ist in Abbildung 1 ersichtlich.

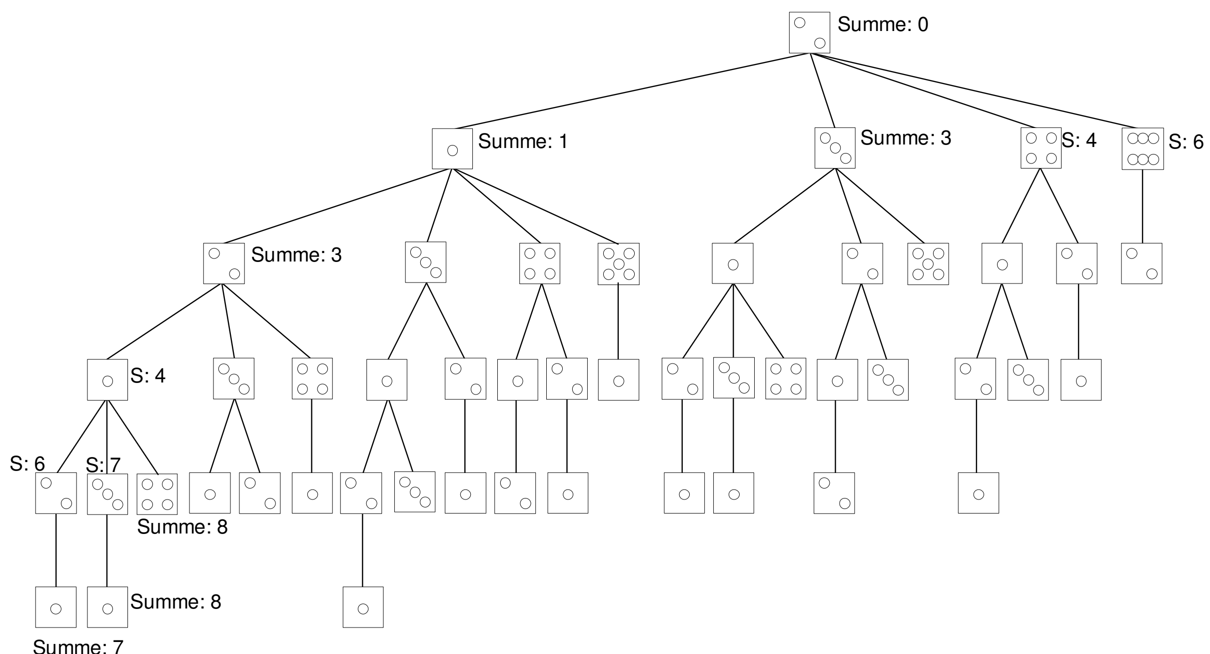


Abbildung 1: Spielbaum zum Spiel Würfelkippen

Bewertung von Spielsituationen – Die NegaMax-Bewertung

Um nun die Frage beantworten zu können, ob der Startspieler eine Gewinnchance hat, wurden allgemeine Überlegungen zu Gewinn-, Remis- und Verlustpositionen der folgenden Art angestellt:

(1) Eine Position ist für den aktuellen Spieler eine Gewinnposition, wenn es mindestens eine Folgeposition gibt, die für den Gegenspieler eine Verlustposition darstellt (denn dann könnte der aktuelle Spieler diese Position anspielen).
(2) Eine Position ist für den aktuellen Spieler eine Verlustposition, wenn alle Folgepositionen für den Gegenspieler Gewinnpositionen darstellen.
(3) In allen anderen Fällen ist die Position für den aktuellen Spieler eine Remisposition.

Diese Fallunterscheidung soll anhand von Teil-Spielbäumen deutlich gemacht werden. Dabei entspricht die Bewertung +1 einer Gewinnposition, –1 einer Verlustposition und 0 einer Remisposition für den aktuellen Spieler.

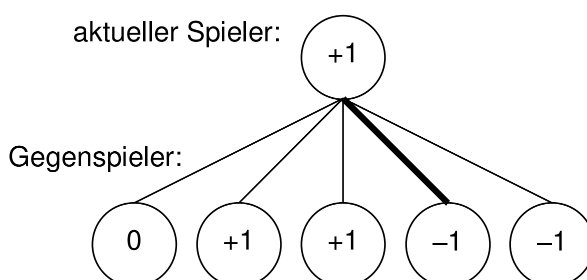


Abbildung 2: Spielbaum für Gewinnposition

Abbildung 2 zeigt eine Gewinnposition für den aktuellen Spieler, da er mit dem markierten Spielzug eine Verlustposition des Gegners erzwingen kann. Die andere mögliche Verlustposition des Gegenspielers bleibt unberücksichtigt, hätte jedoch auch angespielt werden können.

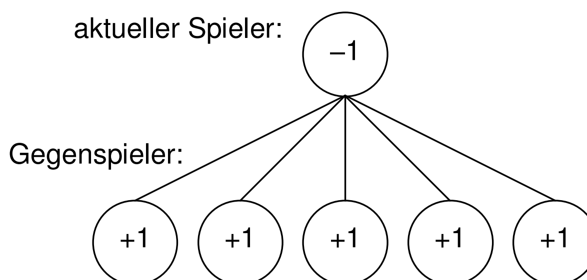


Abbildung 3: Spielbaum für Verlustposition

Abbildung 3 zeigt eine Verlustposition für den aktuellen Spieler, da er mit jedem eingezeichneten Spielzug eine Gewinnposition des Gegners erreicht.

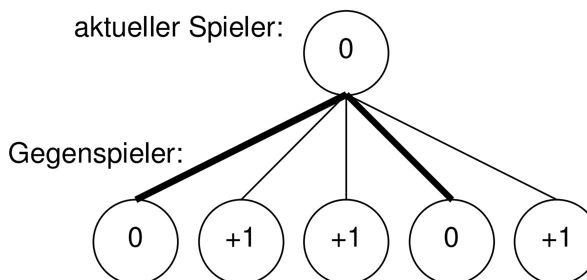


Abbildung 4: Spielbaum für Remisposition

Abbildung 4 zeigt eine Remisposition, da im besten Fall eine Remisposition für den Gegenspieler angespielt werden kann. Die markierten Spielzüge wären damit für den aktuellen Spieler die besten.

Man erkennt sofort, dass zur Bewertung der aktuellen Spielposition im Spielbaum ($bewertung_{akt}$) zuerst alle Folgepositionen bewertet werden müssen ($bewertung_1$ ... $bewertung_k$), um damit die Bewertung der aktuellen Spielposition berechnen zu können. Die Bewertung nach Regeln (1) – (3) erfolgt dann durch:

$$bewertung_{akt} = \max \{ -bewertung_1, -bewertung_2, ..., -bewertung_k \}$$

Alle Blätter des in Abbildung 1 dargestellten Spielbaumes sind für den am Zug befindlichen Spieler Verlustpositionen, wenn die Zielsumme vom Gegenspieler erreicht wurde, bzw. Remispositionen, wenn die Zielsumme noch nicht erreicht wurde aber auch nicht mehr erreicht werden kann. Diese Endpositionsbewertungen sind in Abbildung 5 dargestellt.

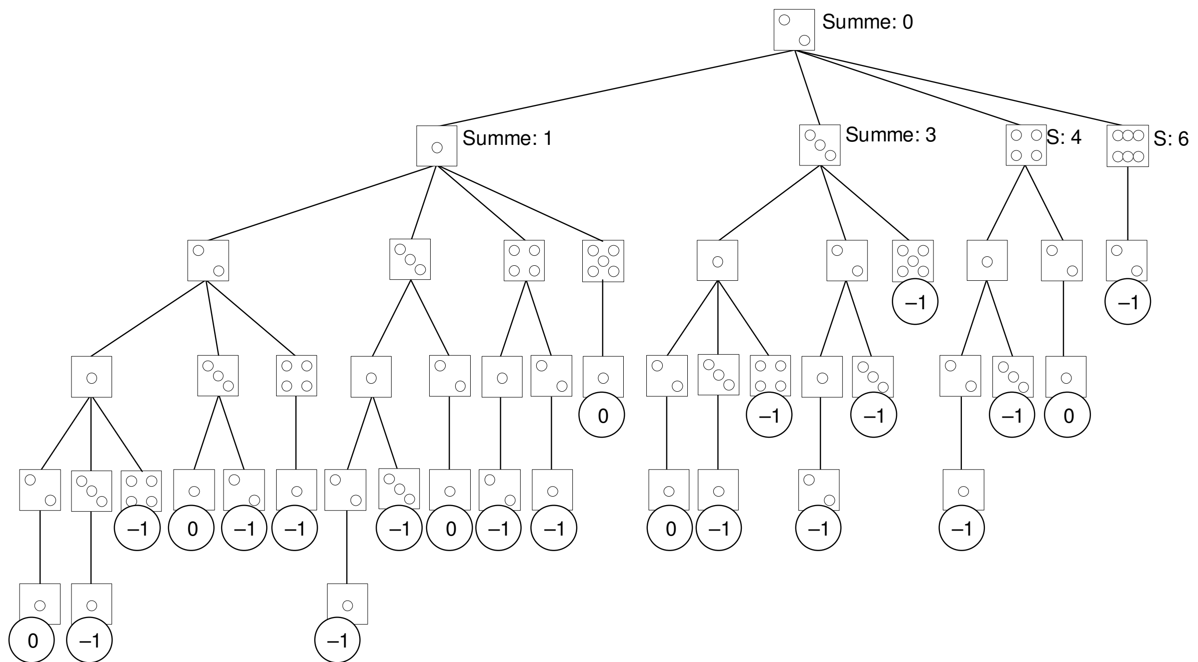


Abbildung 5: Spielbaum zum Spiel Würfelkippen mit Endbewertungen

Um nun entscheiden zu können, ob der Startspieler in einer Gewinnposition ist, oder nicht, müssen die Bewertungen sukzessive bis in die Wurzel des Baumes fortgeführt werden. Abbildung 6 zeigt schließlich, dass sich der Startspieler in einer Remisposition befindet.

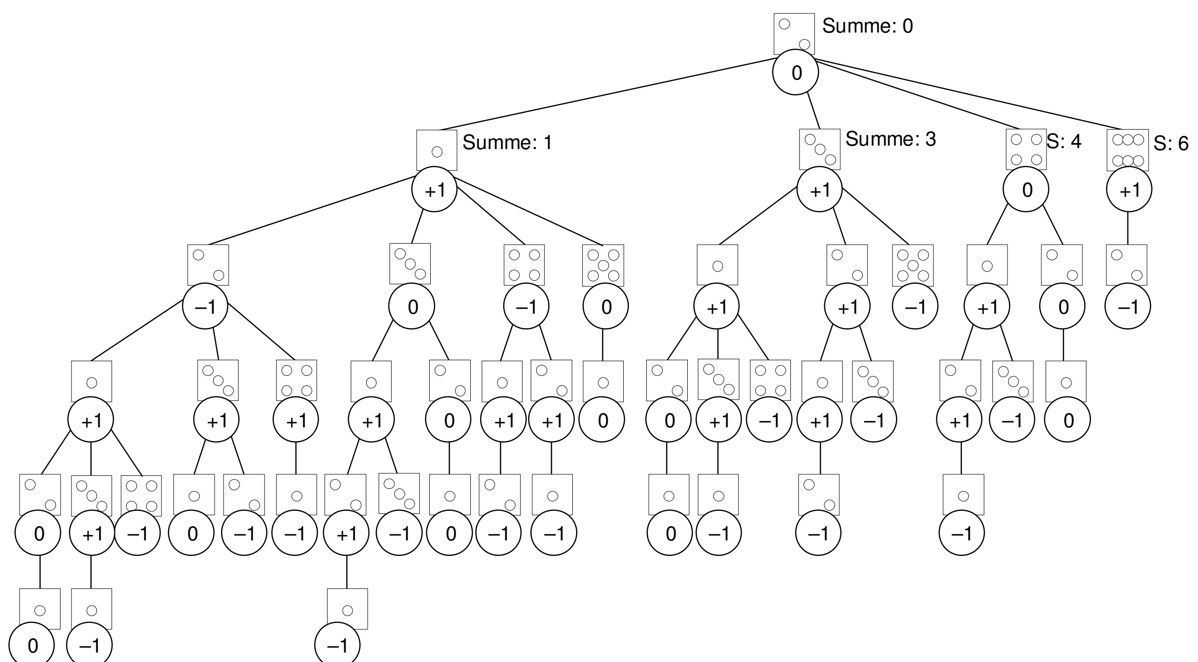


Abbildung 6: Spielbaum zum Spiel Würfelkippen mit vollständiger Bewertung

Der NegaMax-Algorithmus – Implementation

Die Umsetzung des Spiels Würfelkippen erfolgte im Projekt mit Java, kann allerdings mit jeder anderen – wenn möglich objektorientierten – Programmiersprache erfolgen. Zuerst wurde mit den Schülerinnen und Schülern eine geeignete Modellierung für dieses Spiel gesucht, welche in Abbildung 7 dargestellt ist.

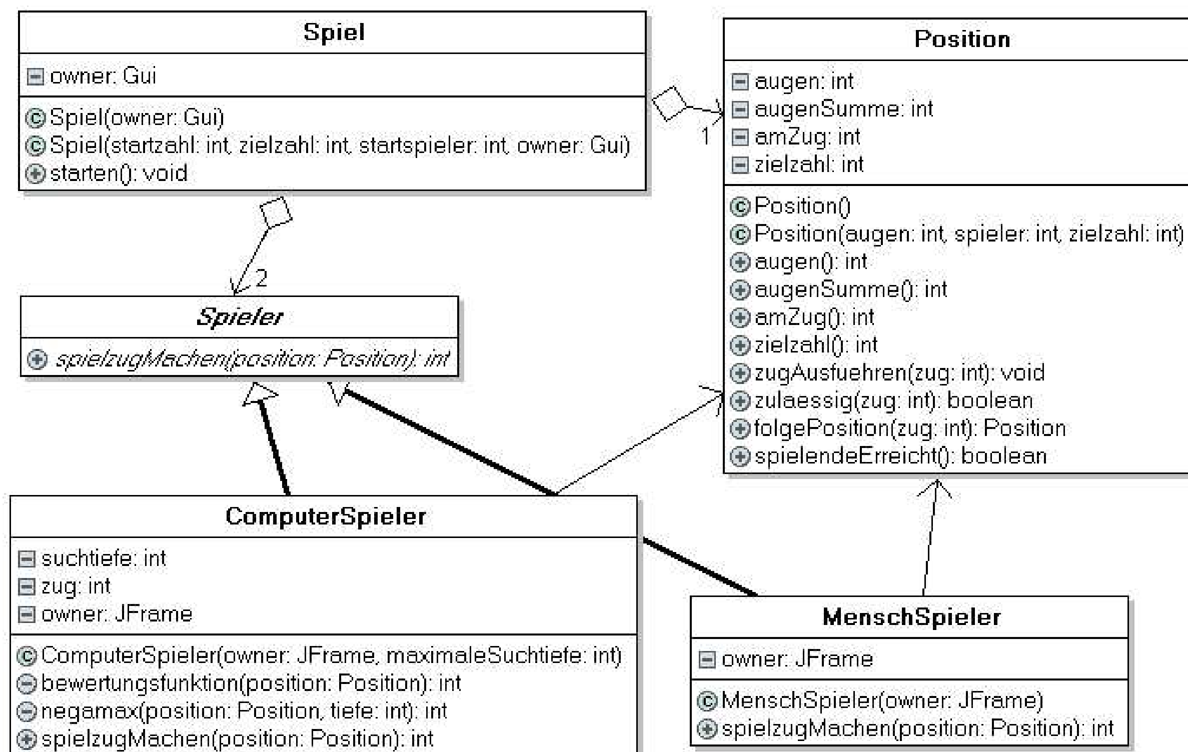


Abbildung 7: UML-Diagramm zum Spiel Würfelkippen

Die grafische Oberfläche (Klasse Gui hier nicht abgebildet) konnte dabei beliebig gestaltet werden, sollte jedoch über eine Methode verfügen, welche eine aktuelle Spielposition darstellen kann, z. B. `public void ausgabe(Position pos)`.

Die abstrakte Klasse Spieler erlaubt eine Gleichbehandlung der Spielzüge des menschlichen und computergesteuerten Spielers in der Klasse Spiel. Sowohl der menschliche als auch der computergesteuerte Spieler erhält als Grundlage für die Durchführung eines Spielzuges den aktuellen Spielstand (`Position`)

Den Schülerinnen und Schülern wurde zur Orientierung die Implementation der Methode `zulaessig` der Klasse `Position` bereitgestellt:

```
public boolean zulaessig(int zug)
{
    if ((augen != zug) && (augen != 7-zug) &&
        (zug >= 1) && (zug <= 6) && (augenSumme + zug <= zielzahl))
      return true;
    else
      return false;
}
```

Alle anderen Methoden der Klasse `Position` wurden im Anschluss von den Schülerinnen und Schülern implementiert, bis man zunächst eine lauffähige Spielleiterversion mit zwei menschlichen Spielern fertiggestellt hatte. Erst danach wurde zusam-

men mit den Schülerinnen und Schülern der Algorithmus für die NegaMax-Bewertung erarbeitet und anschließend implementiert.

```
ALGORITHMUS ComputerZug
Input:    position:    Position
Output:   zug:         Zug        // wie ein Zug aussieht ist vom Spiel abhängig
----------------------------------------------------------------------------
lokal:    besterWert:  {-1,0,+1}
----------------------------------------------------------------------------
• Bewerte aktuelle position mit dem NegaMax-Verfahren:
  besterWert ← NegaMax(position, maximaleSuchtiefe)
• wenn besterWert < 0
        dann  • zug ← zufallszug()   // Wie dieser aussehen könnte ist vom Spiel abhängig.
        sonst • // Zug wurde in NegaMax auf oberster Rekursionsebene gesetzt.
• Zeige den zug an
```

```
ALGORITHMUS NegaMax
Input:    position:    Position
          tiefe:       int
Output:   bewertung:   {-1,0,+1}
----------------------------------------------------------------------------
Lokal:    folgePosition:    Position
          folgeBewertung:   {-1,0,+1}
          probezug:         Zug
----------------------------------------------------------------------------
• wenn position.spielendeErreicht()
  dann • bewertung ← bewertungsfunktion(position)
  sonst• bewertung ← –2 // wird sicher überschrieben, da mindestens –1 erreicht wird.
      • durchlaufe mit probezug alle möglichen Züge und
        tue: • wenn position.zulaessig(probezug)
               dann:• folgePosition ← position.folgePosition(probezug)
                    • folgeBewertung ← –NegaMax(folgePosition, tiefe-1)
                    • wenn folgeBewertung > bewertung
                      dann• bewertung ← folgeBewertung
                          • wenn tiefe = maximaleSuchtiefe // erste Rekursionsebene
                            dann • zug ← probezug          // dann Zug merken
```

```
ALGORITHMUS bewertungsfunktion
Input:    position:    Position
Output:   bewertung:   {-1,0,+1}
----------------------------------------------------------------------------
• wenn position eine Verlustposition ist
        dann   • bewertung ← –1
        sonst  • wenn position eine Remisposition ist
                     dann   • bewertung ← 0
                     sonst  • bewertung ← +1  // hierauf kann verzichtet werden!
```

Andere vollständig berechenbare Zweipersonenspiele

Nach Fertigstellung des Spiels Würfelkippen wandelten die Schülerinnen und Schüler das Programm zu einem NIM-Spiel ab. Hierbei erkannten die Schülerinnen und Schüler die Allgemeingültigkeit des NegaMax-Algorithmus. Lediglich die Methoden der Klasse `Position` mussten angepasst werden, was uns dazu veranlasste, eine allgemeingültige, abstrakte Klasse `Position` zu implementieren, von welcher dann

die Unterklassen `WuerfelPosition` und `NIMPosition` und später auch noch die Unterklasse `TicTacToePosition` abgeleitet wurden.

Nicht vollständig berechenbare Zweipersonenspiele

Bereits bei der Schülerakademie 2007 wurde ein Projekt zum Thema Spielstrategien von Jürgen Zumdick moderiert, welches das Spiel "Mühle" als nicht vollständig berechenbares Spiel analysierte. Besonderer Wert wurde hier auf die Bewertung unterschiedlicher Spielstellungen im Mühlespiel gelegt [Zumdick (2008)], weshalb in diesem Artikel das Augenmerk auf die Optimierung des NegaMax-Algorithmus gelegt werden soll.

Das Spiel "Fünf in einer Reihe"

Allen zuvor behandelten Spielen war gemeinsam, dass sie in vertretbarem Zeitaufwand vollständig bewertet werden konnten. Nun entschied sich die Gruppe dafür, das Spiel "Fünf in einer Reihe" als zunächst nicht vollständig berechenbares Spiel zu implementieren. Bei diesem Spiel geht es darum, auf einem beliebig großen Schachbrett durch wechselseitiges Legen eines Spielsteins insgesamt fünf eigene Spielsteine in einer Reihe zu positionieren. Eine beendete Partie könnte wie in Abbildung 8 dargestellt aussehen.

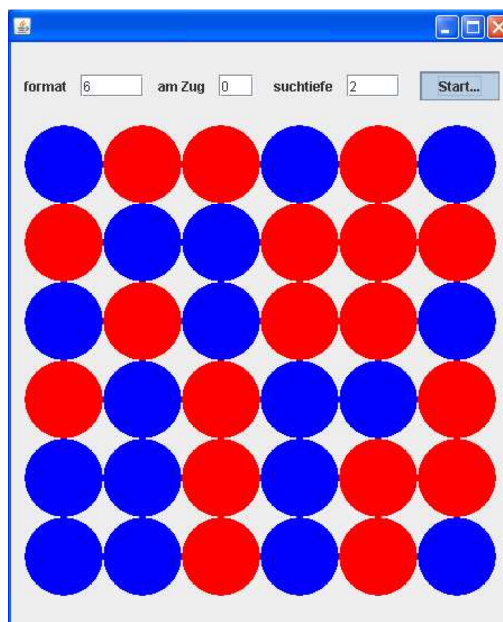


Abbildung 8:
"Fünf in einer Reihe" – Remis

Das Problem dieses Spiels ist, dass die Zahl der Knoten des Spielbaums selbst bei einem so kleinen Spielfeld (hier 7×7) explodiert: Sieht man von frühzeitigen Gewinnpositionen ab, so müssen ca. $6 \cdot 10^{62}$ Stellungen bewertet werden, so dass eine vollständige Berechnung aller Spielpositionen nicht praktikabel ist. Der NegaMax-Algorithmus kann also nicht so lange rekursiv aufgerufen werden, bis er in eine Verlust oder Remisposition gelangt, sondern er darf nur bis zu einer zuvor festgelegten maximalen Suchtiefe rekursiv absteigen. Die Änderungen des Algorithmus sind fett hervorgehoben:

ALGORITHMUS *NegaMax*		
Input:	...	
Output:	**bewertung:**	**int**
Lokal:	folgePosition:	Position
	folgeBewertung:	**int**
	probezug:	Zug
• wenn *position.spielendeErreicht()* **oder** ***tiefe = 0*** dann • bewertung ← *bewertungsfunktion(position)* sonst• **bewertung** ← –∝ *// wird sicher überschrieben (–2 reicht jetzt nicht mehr!)* • durchlaufe ...		

Auf der untersten Rekursionsebene muss nun allerdings eine differenziertere Bewertung der aktuellen Spielposition stattfinden, das heißt die Methode `bewertungsfunktion` muss jeder Position einen Zahlenwert zuordnen, der die Güte der Position für den aktuell am Zug befindlichen Spieler beschreibt.

Hier wurden von den drei Schülergruppen unterschiedliche Strategien verfolgt. Eine Gruppe entschied sich dafür, die Bewertung einer Position von der Anzahl der Fünfer-, Vierer- Dreier- und Zweierreihen abhängig zu machen. Eine andere Gruppe bewertete die Anzahl der freien Felder, die um die eigenen Steine gelegen sind. Die dritte Gruppe bewertete eigene Reihen mit freien Lücken stärker als abgeschlossene eigene Reihen, das heißt diese Art der Bewertung berücksichtigte die Ansätze der beiden anderen Gruppen implizit. Hier ergeben sich auch im Unterricht Möglichkeiten der Differenzierung. Die Bewertung einer Spielposition kann also je nach Leistungsstärke der Schülergruppe variieren, ist allerdings sehr entscheidend für die Spielstärke des Computerspielers.

Trotz unterschiedlicher (und z. T. sehr durchdachter) Bewertungsfunktionen konnten aber die Spielstrategien aller drei Gruppen nicht so richtig überzeugen. Das Problem besteht darin, dass der NegaMax-Algorithmus mit vertretbarem Zeitaufwand maximal vier Halbzüge voraus planen kann. Viele Spielpositionen, die für den menschlichen Spieler auf einen Blick unsinnig erscheinen, werden von dem NegaMax-Algorithmus trotzdem bis in die unterste Rekursionstiefe bewertet. Betrachtet man den zugehörigen Spielbaum, so erkennt man sehr schnell, dass viele Berechnungen überflüssig sind und somit eingespart werden könnten. Der Baum kann also stark reduziert werden.

Der Alpha-Beta-Algorithmus

Um die Arbeitsweise des Alpha-Beta-Algorithmus besser verstehen zu können, betrachten wir zunächst ein Beispiel:

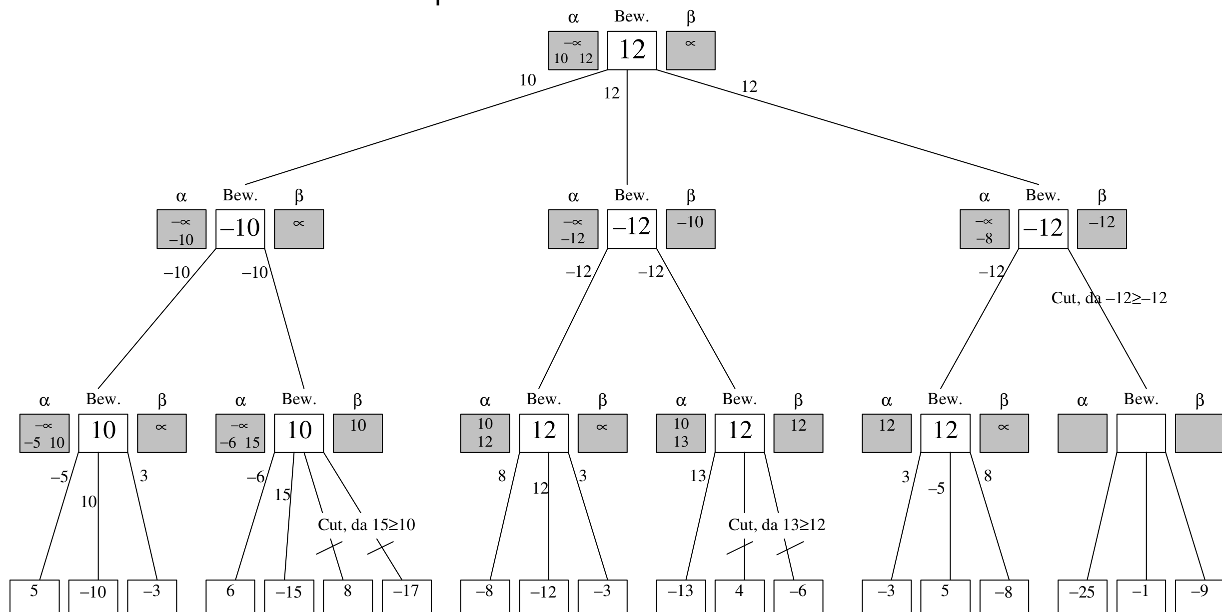


Abbildung 9: Spielbaum mit Alpha-Beta-Baumreduzierung

In den Blättern des Baumes stehen die Bewertungen, welche durch die `bewertungsfunktion` berechnet wurden. Bei jedem Rekursionsaufruf des NegaMax-

Algorithmus werden zwei weitere Parameter mitgegeben: α gibt stets den Wert an, den der am Zug befindliche Spieler in bereits zuvor untersuchten Spielbaum-Ästen mindestens erreichen kann, β gibt den Wert an, den der nicht am Zug befindliche Spieler bereits erreichen kann. α entspricht somit der best-case-Bewertung, β der worst-case-Bewertung des am Zug befindlichen Spielers. α und β schränken also das Suchfenster ein, in dem Spielpositionen überhaupt bewertet werden müssen.

Führt das Anspielen einer Folgepositionen zu einer Bewertung, die kleiner als α ist, so ist *diese eine* Folgeposition für den am Zug befindlichen Spieler uninteressant und kann ignoriert werden.

Führt dagegen das Anspielen einer Folgeposition zu einer Bewertung größer oder gleich β, so muss die *gesamte* aktuelle Position nicht weiter analysiert werden, denn der Gegner könnte in diesem Fall sein bisheriges Ergebnis noch verbessern. *Alle anderen* Folgepositionen des Spielbaums können also ignoriert werden, die aktuelle Position ist mindestens so schlecht, wie bereits zuvor analysierte Spielpositionen, was durch die Rückgabe von β als aktuelle Bewertung deutlich gemacht wird.

Wird dagegen bei der Betrachtung eines Spielknotens eine Bewertung gefunden, die größer als der bisherige Wert von α aber kleiner als der bisherige Wert von β ist, so wurde ein für den am Zug befindlichen Spieler besserer Spielzug gefunden und diese bessere Bewertung kann in α gespeichert werden. Nach Betrachtung aller Folgepositionen wird die aktuelle Spielposition mit dem besten Wert von α bewertet und die Bewertung kann abschließend zurückgegeben werden.

Zu Anfang ist α mit $-\infty$ und β mit $+\infty$ vorinitialisiert. Beim Rekursiven Aufruf des NegaMax-Algorithmus erhält α den Wert von $-\beta$ und β erhält den Wert von $-\alpha$. Dies bedeutet soviel wie: Der Wert α, den Spieler A mindestens erreichen kann ist für Spieler B der Wert, den Spieler A höchstens erreichen darf.

Im obigen Beispiel (Abbildung 9) bedeutet dies folgendes:

(1) Die erste Blattbewertung –5 wird in α des Vaterknotens gespeichert, da $-5 > -\infty$.

(2) Die zweite Blattbewertung 10 wird in α des Vaterknotens gespeichert, da $10 > -5$.

(3) Die dritte Blattbewertung wird ignoriert, da $-3 < 10$.

(4) Der Vaterknoten erhält also die Bewertung $10 = \max \{-5, -(-10), -(-3)\}$

(5) Eine Rekursionsebene darüber wird der bisherige Wert für α mit –10 überschireben, da $-10 > -\infty$ ist.

(6) Der Rekursionsaufruf mit dem rechten Sohn erfolgt nun mit $\alpha = -\infty$ und $\beta = 10$, denn aus Sicht des am Zug befindlichen Spielers kann der Gegenspieler bereits mindestens die Bewertung 10 erreichen.

(7) Die vierte Blattbewertung –6 wird in α des Vaterknotens gespeichert, da $-6 > -\infty$.

(8) Die fünfte Blattbewertung 15 führt zu einem Abbruch der Suche (Cut), da $15 \geq 10$ ist. Würde man diese Position anspielen, so könnte der Gegenspieler eine Bewertung von 15 erlangen, also ein für den am Zug befindlichen Spieler schlechteres Ergebnis als bisher. Da hier nicht interessiert, um wie viel schlechter die aktuelle Position ist wird die aktuelle Position mit dem bisher schlechtesten Wert (β) bewertet.

(9) Eine Rekursionsebene darüber ändert sich demnach an der bisherigen besten Bewertung von –10 nichts und kann deshalb als Gesamtbewertung zurückgegeben werden.
(10) In der Wurzel des Baumes wird die Bewertung von 10 in α gespeichert, da $10 > -\infty$.
(11) Die Rekursion wird nun für den mittleren Sohn der Wurzel gestartet, nun allerdings mit $\alpha = -\infty$ und $\beta = -10$.
(12) Beide nachfolgenden Söhne liefern eine Bewertung von 12 (Cut beim zweiten Sohn möglich), so dass die endgültige Gesamtbewertung auf –12 gesetzt wird.
(13) In der Wurzel des Baumes wird die Bewertung von 12 in α gespeichert, da $12 > 10$ ist.
(14) Die Rekursion wird abschließend für den rechten Sohn der Wurzel gestartet, wobei jetzt $\alpha = -\infty$ und $\beta = -12$ ist.
(15) Hier liefert aber die Bewertung des linken Teilbaums keinen besseren Wert, so dass für den dort am Zug befindlichen Spieler eine weitere Betrachtung des Spielbaums keinen Sinn mehr macht. Die Suche wird abgebrochen und der Wert von β (–12) als Gesamtbewertung zurückgegeben.
(16) Die Wurzel erhält somit die Gesamtbewertung 12, das Maximum aller negierten Teilbaumbewertungen.

Ein gutes Applet zur Visualisierung findet sich in Monien, Lorenz, Warner (2006), hier allerdings in der MiniMax-Variante.

Man erkennt an diesem Beispiel sehr deutlich, dass große Teile des Suchbaums abgeschnitten werden können. Je besser die Bewertungen zu Anfang der Suche sind, desto schneller ergibt sich die Möglichkeit eines Cuts. Hätte man z. B. zuerst den mittleren Sohn der Wurzel untersucht, so könnte man im linken Sohn eine Rekursionsebene früher die Suche beenden.

Die algorithmische Beschreibung dieser optimierten NegaMax-Bewertung wurde im Projekt mit den Schülerinnen und Schülern formuliert. Wesentliche Änderungen zum NegaMax-Algorithmus sind fett hervorgehoben:

```
ALGORITHMUS AlphaBeta
Input:    position:      Position
          tiefe:         int
          alpha, beta:   int
Output:   bewertung:     int
-----------------------------------------------
Lokal:    ...
-----------------------------------------------
• wenn position.spielendeErreicht() oder tiefe = 0
  dann • bewertung ← bewertungsfunktion(position)
  sonst• durchlaufe mit probezug alle möglichen Züge und
         tue:  • wenn position.zulaessig(probezug)
                 dann:• folgePosition ← position.folgePosition( probezug)
                      • folgeBewertung ←
                            –AlphaBeta(folgePosition, tiefe-1, -beta, -alpha)
                      • wenn folgeBewertung > alpha
                        dann• alpha ← folgeBewertung
                             • wenn tiefe = maximaleSuchtiefe // erste Rekursionsebene
                               dann      • zug ← probezug
                             • wenn folgeBewertung ≥ beta
```

dann • bewertung ← beta
• BETA-CUT *// Abbruch des aktuellen Rekursionsastes*
• bewertung ← alpha

Optimierung der Alpha-Beta-Suche durch Zugsortierung

Die Alpha-Beta-Suche arbeitet besonders effizient, falls die zu bewertenden Folgepositionen in absteigender Reihenfolge vorliegen. Dies erkennt man an folgendem Beispiel:

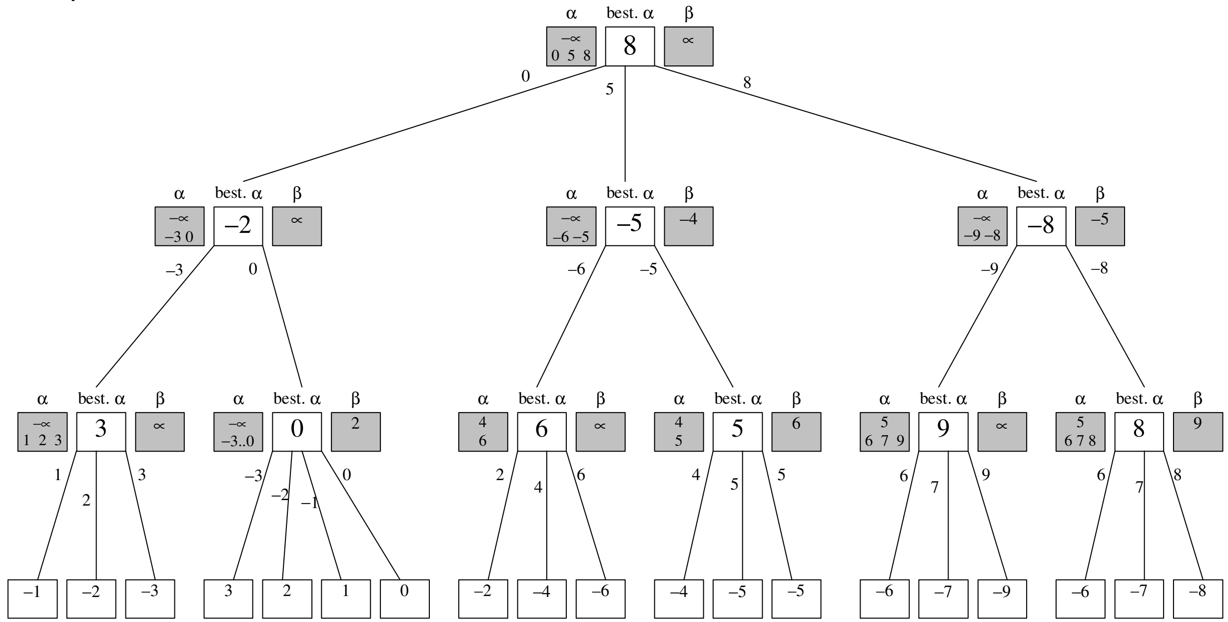


Abbildung 10: schlechte Sortierung, also keine Baumreduzierung

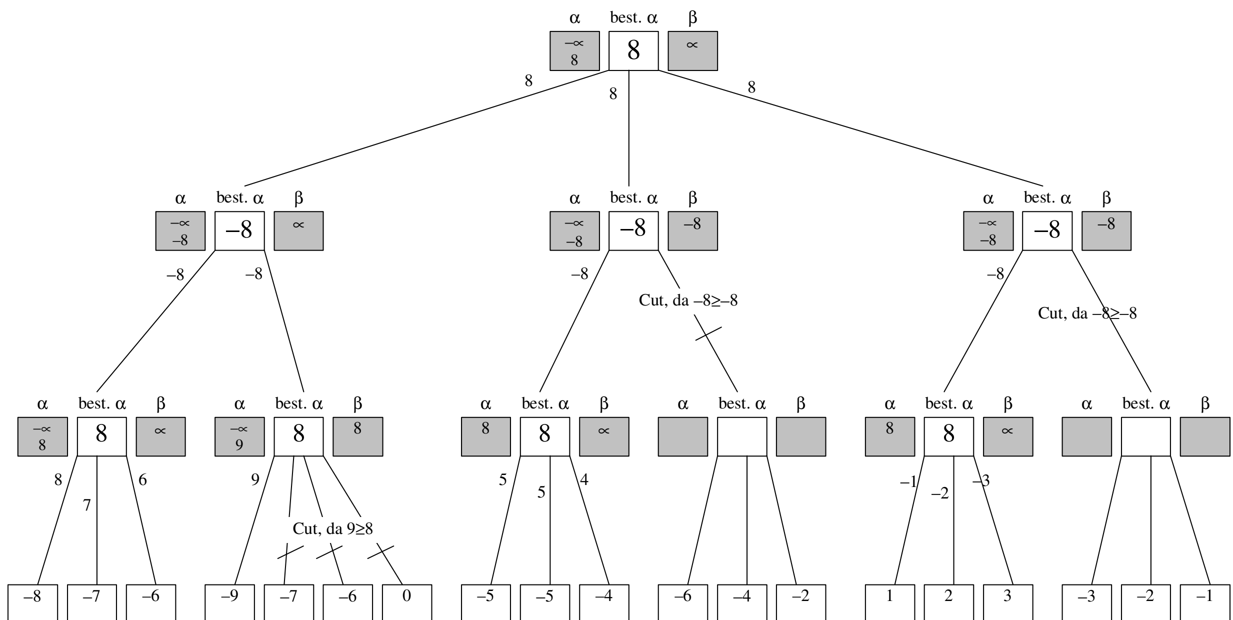


Abbildung 11: gute Sortierung, also maximale Baumreduzierung

Damit diese maximale Anzahl an Cuts erreicht werden kann muss vor dem Schleifendurchlauf aller Probezüge (Zeile 3 des Algorithmus) eine Liste mit allen möglichen Folgezügen erstellt werden und in absteigender Reihenfolge sortiert werden. Die

Frage ist nur, wie man Züge zu den Folgepositionen nach ihrer Bewertung sortieren soll, wenn sie noch gar nicht bewertet wurden?
Die einfachste Möglichkeit besteht darin, alle Folgepositionen einmal anzuspielen, mit der vorhandenen Bewertungsfunktion zu bewerten und anschließend den Zug wieder zurückzunehmen. Hat man auf diese Weise alle möglichen Folgezüge bewertet und in einer Liste gespeichert, so kann man diese Liste sortieren und in absteigender Sortierung durchlaufen.
Mit dieser Zugsortierung ergibt sich ein leicht abgewandelter Algorithmus:

```
ALGORITHMUS AlphaBeta  // mit Zugsortierung
Input:      ...
Output:     bewertung:     int
Lokal:      ...
            liste:         Liste      // über den Datentyp Zug
• wenn position.spielendeErreicht() oder tiefe = 0
  dann • bewertung ← bewertungsfunktion(position)
  sonst• durchlaufe mit probezug alle möglichen Züge und
         tue: • wenn position.zulaessig(probezug)
                dann:• folgePosition ← position.folgePosition( probezug)
                     • folgeBewertung ←  bewertungsfunktion(folgePosition)
                     • probezug.setzeBewertung(folgeBewertung)
                     • liste.einfügen(probezug)
       • liste.sortieren()   // mit Quicksort nach absteigenden Bewertungen
       • durchlaufe mit probezug alle Züge der liste und:
         tue:  :• folgePosition ← position.folgePosition( probezug)
                // ein Test auf Zulässigkeit ist nicht mehr nötig, da nur gültige Züge in der Liste sind.
                • folgeBewertung ←
                       –AlphaBeta(folgePosition, tiefe-1, -beta, -alpha)
                • wenn folgeBewertung > alpha
                  dann• alpha ← folgeBewertung
                      • wenn tiefe = maximaleSuchtiefe // erste Rekursionsebene
                        dann       • zug ← probezug
                      • wenn folgeBewertung ≥ beta
                        dann • bewertung ← beta
                             • BETA-CUT  // Abbruch des aktuellen Rekursionsastes
       • bewertung ← alpha
```

Die Schülerinnen und Schüler haben sowohl das Alpha-Beta-Suchverfahren als auch die Zugsortierung eingesetzt und konnten somit die maximale Suchtiefe von zuvor 4 Halbzügen auf bis zu 7 Halbzüge anheben. Als geschickt hat sich erwiesen, wenn die Zugsortierung auf der untersten Rekursionsebene ausgeschaltet wird, da andernfalls die Endbewertungen überflüssiger Weise doppelt berechnet würden.

Optimierung der Alpha-Beta-Suche durch Principal-Variation-Suche

Eine weitere Optimierungsmöglichkeit besteht in der Suche nach der Principal-Variation-Methode, welche in Baier (2006) sehr gut beschrieben ist: Je kleiner das Suchfenster [α, β] gewählt ist, desto schneller kann ein Cut erzielt werden. Sind die Züge zudem noch gut vorsortiert, so ist meistens schon nach dem ersten Probezug zu erkennen, ob die Suche in diesem Zweig erfolgversprechend ist oder nicht, denn:

a) Ist die Bewertung des ersten Zuges kleiner als α, so ist die aktuelle Spielsituation vermutlich schlechter als vorher getestete Züge (Alpha-Knoten).
b) Ist die Bewertung des ersten Zuges größer als β, so erfolgt ein Cut (Beta-Knoten).
c) Liegt die Bewertung zwischen α und β, so hofft man, einen besseren Spielzug gefunden zu haben (Principal-Variation-Knoten).

Für einen Principal-Variation-Knoten muss man nun nur noch zeigen, dass alle verbleibenden Züge schlechter sind, d. h., man prüft also *hauptsächlich nur die Abweichung* vom besten Wert. Dies erreicht man dadurch, dass man das Suchfenster für alle weiteren möglichen Probezüge auf $[\alpha,\alpha + 1]$ minimiert um möglichst schnelle Cut-Offs zu erreichen, aber dennoch zu zeigen, dass alle anderen möglichen Züge schlechter als der bisherige sind. Sollte im weiteren Verlauf der Wert $\alpha+1$ überschritten werden, so hatte man doch nicht den besten Zug und muss die Suche nochmals, nun aber mit dem originalen Suchfenster $[\alpha,\beta]$ durchführen.

Die Effizienzsteigerung beruht darauf, dass möglichst große Teile des Spielbaumes wegen dem kleinen Suchfenster abgeschnitten werden können. Dies setzt allerdings voraus, dass die Züge gut vorsortiert sind, denn andernfalls müsste die Suche oftmals doppelt durchgeführt werden.

Die Änderungen des Algorithmus im Bezug auf den Alpha-Beta-Algorithmus sind fett hervorgehoben:

```
ALGORITHMUS AlphaBeta  // mit Principal-Variation-Suche
Input:    ...
Output:   ...
Lokal:    ...
          pvGefunden:  boolean       // wurde ein Principal-Variation-Knoten gefunden
• wenn position.spielendeErreicht() oder tiefe = 0
  dann  • ...
  sonst • ...
        • liste.sortieren()   // mit Quicksort nach absteigenden Bewertungen
        • pvGefunden ← false
        • durchlaufe mit probezug alle Züge der Liste und:
          tue: • folgePosition ← position.folgePosition( probezug)
               • wenn pvGefunden = true
                 dann• folgeBewertung ←
                           –AlphaBeta(folgePosition, tiefe-1, -alpha-1, -alpha)
                     • wenn folgeBewertung > alpha und folgeBewertung < beta
                       dann • folgeBewertung ←
                                  –AlphaBeta(folgePosition, tiefe-1, -beta, -alpha)
                                   // Suchfenster war wohl zu klein
                 sonst• folgeBewertung ←
                           –AlphaBeta(folgePosition, tiefe-1, -beta, -alpha)
               • wenn folgeBewertung > alpha
                 dann• alpha ← folgeBewertung
                     • pvGefunden ← true // ab jetzt gibt es einen Principal-Variation-Knoten
                     • wenn tiefe = maximaleSuchtiefe // erste Rekursionsebene
                       dann        • zug ← probezug
```

```
                    • wenn folgeBewertung ≥ beta
                      dann • bewertung ← beta
                           • BETA-CUT  // Abbruch des aktuellen Rekursionsastes
        • bewertung ← alpha
```

Im Projekt hat sich allerdings gezeigt, dass diese Optimierung keinen Laufzeitgewinn einbrachte sondern – im Gegenteil – die Laufzeit der Suche negativ beeinflusste. Die Erklärung liegt wohl in der nicht ausgefeilten Zugvorsortierung. Da für die Gewichtung eines Zuges lediglich die Bewertung der direkten Folgeposition gewählt wurde, bekamen Züge mit offensichtlichen Versuchen, eine Fünferreihe zu bilden, höhere Gewichtungen und wurden demnach zuerst angespielt. In der weiteren Analyse stellt sich aber oftmals heraus, dass gerade diese offensichtlichen Züge nicht zum Erfolg führen.

Zusammen mit den Schülerinnen und Schülern wurde nach Möglichkeiten gesucht, bessere Bewertung der einzelnen Züge zu erhalten. Dabei kristallisierte sich als grundlegende Idee heraus, dass die Suche nicht sofort bis in die letzte Rekursionsebene durchgeführt werden sollte, sondern stattdessen sollten alle Ebenen des Spielbaumes nacheinander, also ebenenweise durchsucht werden.

Optimierung der Alpha-Beta-Suche durch Iterative Tiefensuche

Führt man die Tiefensuche im Spielbaum nicht rekursiv sondern iterativ durch, so ist es möglich, die Suchtiefe schrittweise zu erhöhen und die Bewertungen der Knoten zwischenzuspeichern. Diese Bewertungen können dann im nächsten Suchlauf mit der um eins erhöhten Suchtiefe genutzt werden, um die möglichen Folgezüge besser vorzusortieren.
Ein weiterer Vorteil liegt darin, dass die Suche frühzeitig abgebrochen werden kann. Bedeutet ein Abbruch der Suche beim rekursiven Durchlauf, dass eventuell einige Folgepositionen noch gar nicht bewertet und analysiert wurden, so hat die iterative Tiefensuche sichergestellt, dass alle möglichen Spielpositionen gleichermaßen gut analysiert wurden.

Aus Zeitgründen konnte allerdings diese Optimierung nicht mehr programmtechnisch umgesetzt werden, so dass ich hier keine Aussagen über die Effizienzsteigerung machen kann.

Weitere Optimierungsmöglichkeiten wurden im Projekt zwar nicht besprochen, sollen hier allerdings wenigstens ansatzweise erläutert werden:

Aspiration Windows
Wenn man bei der iterativen Tiefensuche noch für jeden weiteren/tieferen Suchlauf das Suchfenster auf ein kleines Fenster um den zuvor berechneten Bewertungswert setzt (Aspiration Window), so kann man ähnlich der Principal-Variation-Suche viele Cutoffs erreichen, weil man die Hoffnung hat, dass der tatsächliche Wert tatsächlich in diesem kleinen Suchfenster liegt. Eventuell muss das Suchfenster aber doch auf $[-\infty;+\infty]$ initialisiert werden, falls eine bessere Bewertung gefunden wird.
Diese Optimierung ist nur sinnvoll, wenn sich Positionsbewertungen auf verschiedenen Ebenen des Suchbaums nur geringfügig unterscheiden.

Killer-Moves

Züge, die einen Cutoff verursacht haben, könnten auch in anderen Teilen des Spielbaums einen Cutoff bewirken und werden deshalb bevorzugt ausprobiert. Diese Strategie bezieht sich also im wesentlichen auf die Sortierung der möglichen Folgezüge.

Search-Extensions
Wenn in Knoten auf höheren Ebenen des Suchbaums die Anzahl der möglichen Folgezüge gering ist, so kann die Suchtiefe für diesen Zweig des Suchbaums erweitert werden. Diese Search-Extension wird erreicht, indem man die Suchtiefe unverändert in die nächste Rekursionsebene weiterreicht.

Quiescent-Search
Die Quiescent-Search hat das Ziel, schwer einschätzbare, unruhige Spielsituationen (z. B. Schlagabtausch beim Schach) tiefer zu durchsuchen als Spielsituationen für welche die Bewertungsfunktion zuverlässige Bewertungen liefert. Die Suchetief wird also an wenigen Endknoten erhöht, falls die Bewertungsfunktion beim Wechsel in die nächste Rekursionsebene stark variiert.

Hash-Tables
Oftmals gelangt man durch unterschiedliche Zugfolgen zu dennoch gleichen Spielsituationen eines Spielbaums. Die Bewertungen dieser Spielsituationen müssen nur einmal bewertet werden und können codiert in einer Hash-Tabelle gespeichert werden.
Schafft man es nun noch, Symmetrien der Spielsituationen einfach zu erkennen (4-Gewinnt, Mühle, Dame, etc.), so könnte man die Zahl der zu untersuchenden Spielsituationen weiter verringern.

Opening-Books
In vielen Strategiespielen gibt es nur wenige, bekannte, sinnvolle Eröffnungszüge, welche in einer Datenbank abgelegt werden. Die wenig erfolgversprechenden Bewertungen von Spielsituationen, welche auf ungeschickten Eröffnungszügen aufbauen, entfallen.

Literatur

Rüdeger Baumann (1994): Arbeitshefte Informatik, Strategiespiele. Klett Schulbuchverlag: Stuttgart
Jürgen Zumdick (2008): Programmierung des Strategiespiels "Mühle", in Gilbert Greefrath, Andreas Pallack (Hrsg.), Materialien für einen projektorientierten Mathematik- und Informatikunterricht Band 5, diverlag franzbecker

Internet-Seiten

Hendrik Baier (2006): Der Alpha-Beta-Algorithmus und Erweiterungen bei Vier Gewinnt", [Stand 06.09.2008, 15:54]
http://www.ke.informatik.tu-darmstadt.de/lehre/arbeiten/bachelor/2006/Baier_Hendrik.pdf

Prof. Dr. Burkhardt Monien, Dr. Ulf Lorenz, Daniel Warner: (2006) "Der Alphabeta-Algorithmus für Spielbaumsuche", [Stand 06.09.2008, 16:03]
http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo19.php