Dr. Th. Letschert
Softwarekomponenten

THM TECHNISCHE HOCHSCHULE MITTELHESSEN
MNI Mathematik, Naturwissenschaften und Informatik

Aufgabenblatt 2

Aufgabe 1

Die **Alpha-Beta-Suche**[1] ist eine Verbesserung des Minimax-Algorithmus'. Sie optimiert die Suche durch das Auslassen von Unterbäumen, die nichts zum MinMax-Wert der Wurzel beitragen.

Bei der Berechnung von Minimax-Werten wird bei einem Max-Knoten das Maximum seiner Nachfolger gesucht und bei einem Min-Knoten das Minimum seiner Nachfolger. Die Berechnung der Minima und Maxima ist also ineinander verwoben. Das kann man zur Optimierung des Verfahrens ausnutzen.

Nehmen wir an, wir sollten an einer Schule den größten Klassen-Kleinsten suchen. Mit dem Minimax-Verfahren würden alle Schüler gemessen dann wird von jeder Klasse der Kleinste bestimmt und schließlich von den Kleinsten der Größte. Das geht besser: Angenommen wir haben schon die Klassen A und B vermessen und dabei a und b als Klassen-Kleinsten von A und B gefunden. a ist kleiner als b und kann darum bei der Suche nach dem größten der Kleinen vergessen werden. b ist als der bisher gefundene Größte der Kleinen.

Jetzt wird Klasse C nach dem kleinsten durchsucht. Angenommen stoßen dabei auf c und c ist kleiner als b. Die Suche in Klasse C kann jetzt abgebrochen werden. Vielleicht finden wir in C einen der noch kleiner als c ist. Für die Suche nach dem Größten der Kleinen ist das aber unerheblich. Er kann eventuell noch mehr kleiner als c sein und c ist schon kleiner als b und kann so für die Suche nach dem Größten der Kleinen nichts beitragen. (siehe Abb. 1)

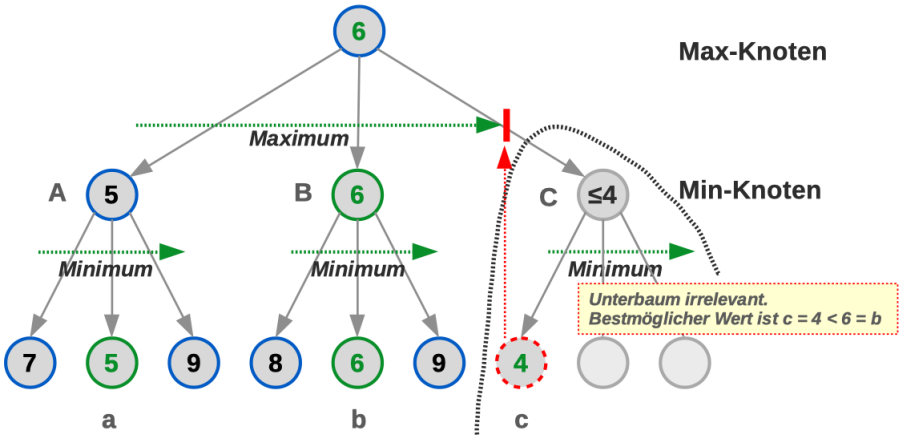

Abbildung 1: Maximium unter Minima suchen

Die Idee wird im Alpha-Beta-Algorithmus rekursiv und im Wechsel zwischen Min- und Max-Knoten umgesetzt: Die Minimax-Berechnung wird mit zwei zusätzlichen Parametern *Alpha* und *Beta* ausgestattet:

[1] https://de.wikipedia.org/wiki/Alpha-Beta-Suche

- **Alpha**: Bester von der Wurzel bis hier gefundener MinMax-Wert für **Max** (Weiss/Wir). Das ist das bis hierher beste Maximum. Initialwert an der Wurzel ist $-\infty$
- **Beta**: Bester von der Wurzel bis hier gefundener MinMax-Wert für **Min** (Schwarz/Gegner). Das bis hierher beste Minimum. Initialwert an der Wurzel $+\infty$

Bei der Berechnung des MinMax-Wertes an einem Min-Knoten s (im Beispiel oben C) wird das Minimum über die MinMax-Werte aller Nachfolger bestimmt. Die Berechnung bei Nachfolger t (im Beispiel oben $t = c$) kann abgebrochen werden, wenn dessen MinMax-Wert kleiner ist, als das aktuelle Alpha (im Beispiel oben 6): Im übergeordneten Max-Knoten wird maximiert über die aktuell berechneten Mimima mit Alpha als bisher gefundenem besten Wert, ein besseres Mimimum – nach dem wir hier suchen – kann zu einem besseren Maximum oben nichts beitragen!

Für Max-Knoten gilt eine komplementäre Argumentation.

Der Alpha-Beta-Algorithmus in **imperativem** Pseudocode:

```
alphaBeta(s, alpha, beta) =
  if s ist Endknoten return score(s)
  else
   if s ist Max-Knoten
     var a = alpha
     for t in Nachfolger(s) :
         a = max(a, alphaBeta(t, a, beta))
         if a >= beta
           // Eine Fortsetzung der Maximum-Berechnung
           // hier kann nur zu einen groesseren Wert führen,
           // der dann bei der Minimum-Berechnung oben
           // ignoriert werden wuerde.
           return beta
     return a
   else // s ist Min-Knoten
     var b = beta
     for t in Nachfolger(s) :
         b = min(b, alphaBeta(t, alpha, b))
         if b <= alpha
           // Eine Fortsetzung der Minimum-Berechnung hier
           // kann nur zu einen kleineren Wert führen,
           // der dann bei der Maximum-Berechnung oben
           // ignoriert werden wuerde.
           return alpha
     return b
```

Transformieren Sie den Algorithmus alphaBeta in eine rein funktionale Form. Also kein return, keine imperativen Schleifen, keine Zuweisungen!

Testen Sie mit einer einfachen Konfiguration für *Tic-Tac-Toe*. Etwa wie folgt:

```
import scala.annotation.tailrec

enum Player {
  case WhitePlayer
  case BlackPlayer
}
import Player._

// Typklasse der Konfigurationen (Spielstellungen)
trait Config[C] {
  extension (c: C) def score: Int
  extension (c: C) def successors(player: Player): Seq[C]
  extension (c: C) def isFinal: Boolean
```

```
}

class AlphaBeta[C: Config] {

  private def alphaBeta (
      config: C,
      player: Player,
      alpha: Int,
      beta: Int): Int = {
      ...
      // Funktionale Version des Alpha-Beta-Algorithmus'
  }

  def bestNextConfig(config: C, player: Player): C = {
    val (nextConfig, _) =
      player match {
        case WhitePlayer =>
          config.successors(WhitePlayer)
            .map(c => (c, alphaBeta(c, BlackPlayer, Int.MinValue, Int.MaxValue)))
            .maxBy {
              case (_, v) => v
            }
        case BlackPlayer =>
          config.successors(BlackPlayer)
            .map(c => (c, alphaBeta(c, WhitePlayer, Int.MinValue, Int.MaxValue)))
            .minBy {
              case (_, v) => v
            }
      }
    nextConfig
  }
}

// Instanz der Typklasse Config
given VectorConfig : Config[Vector[Int]] with {
  ...
}

// Ausgangstellung
val actConfig: Vector[Int] = ...

// Stellung nach optimalem Zug von Weiss
val expectedConfig = ...

val AlphaBetaWithConfig = new AlphaBeta

val nextConfig = AlphaBetaWithConfig.bestNextConfig(actConfig_1, WhitePlayer)
assert(expected == nextConfig)
```