

Künstliche Intelligenz - Übung 4

Adrian Neumann (4140810) und Naja von Schmude (4127652)

15. Mai 2009

Aufgabe 1

Mehrere Anfangszustände sind ja nichts anderes als eine Ebene im Suchbaum. Daher könnte man, um wieder auf nur einen Anfangszustand zu kommen, eine Wurzel überhalb all dieser Anfangszustände einfügen. Dann können die selben Verfahren wie bei nur einem Anfangszustand verwendet werden. Man muss natürlich den Wegen von der Wurzel zu den Anfangszuständen ein neutrales Gewicht, zum Beispiel 0, geben, damit kein Anfangszustand bevorzugt wird.

Aufgabe 2

Wenn man annimmt, dass man alle möglichen Stellungen auch irgendwie durch Züge erreichen kann (das konnten wir nicht beweisen), dann gibt es 140 verschiedene Stellungen

```
Prelude Data.List> length nub permutations $ [1,1,1,0,2,2,2]
140
```

In jeder Stellung kann auch noch der jeweils andere Spieler am Zug sein, so dass man das noch mit 2 multiplizieren muss. Insgesamt gibt es also weniger als 280 Zustände in diesem Spiel.

Vielleicht sind es auch ein paar weniger. Es gibt allerdings Schleifen. x sein schwarz _ leer und y weiß.

```
yyy_xxx
yyyx_xx
yy_xyxx
yyx_yxx
yyxy_xx
yy_yxxx
yyy_xxx
```

Als Heuristik könnte man den Hamming Abstand einer Stellung zur nächsten der wenigen Endstellungen hernehmen. Da Züge mindestens Kosten 1 haben, kann der Hamming Abstand nicht größer sein als die Zugkosten. Gleiches gilt für die Monotonie. Sonderlich informiert ist die Heuristik aber nicht, der Hamming Abstand betrachtet ja einfach nur die Vertauschungen, wieviele Züge man aber tatsächlich braucht, um zwei Steine zu vertauschen, fließt ja überhaupt nicht in die Heuristik mit ein. Das können aber sehr viele sein.

Angenommen wir haben einen ganzen Batzen Heuristiken. Dann können wir, wie wir in Mustererkennung gelernt haben, zum Beispiel mit Adaboost eine coole Heuristik bauen, die die einzelnen Heuristiken mit einer passenden Gewichtung benutzt.

Aufgabe 3

Bei dem gegebenen Baum waren verschiedene Suchalgorithmen durchzuführen. Da bei den Durchläufen von links (Abbildung 2) und rechts (Abbildung 3) natürlich die α und β -Werte unterschiedlich zustande kommen, ist natürlich auch die Auswertung verschieden. Da z.B. von rechts dann direkt mit der Hauptvariante begonnen wird, ist es nur logisch, dass auch mehr abgeschnitten werden kann.

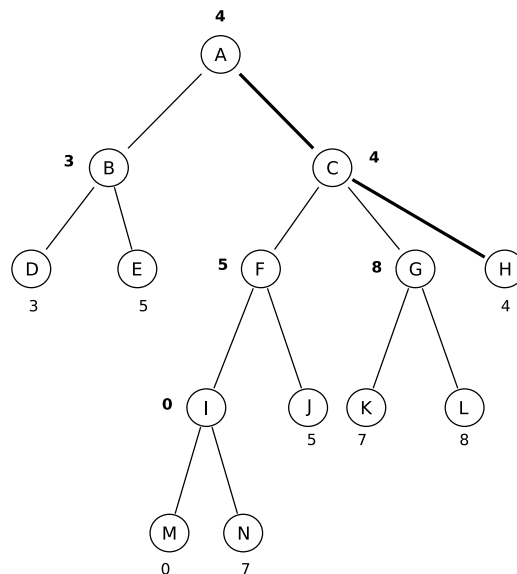


Abbildung 1: Minimax

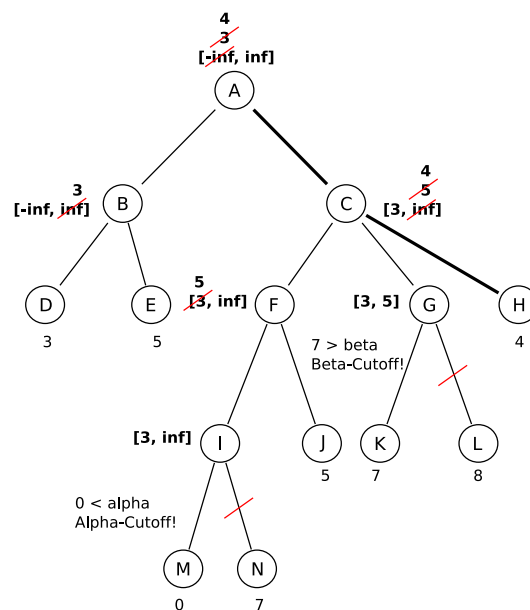


Abbildung 2: Alpha-Beta-Pruning von links

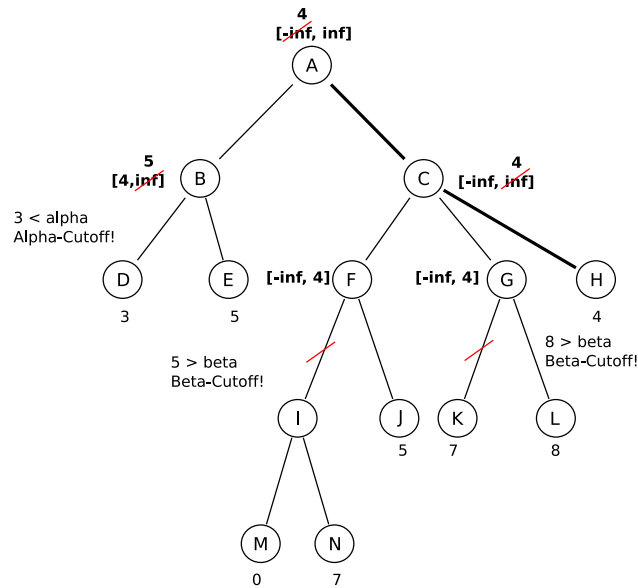


Abbildung 3: Alpha-Beta-Pruning von rechts

Aufgabe 4

Wir haben das Problem nicht in Prolog gelöst, sondern in Haskell. Prolog war uns zu doof. Wir machen eine einfache Breitensuche, deswegen ist der Algorithmus nicht so super schnell. Wenn man zu tief suchen muss, wird auch der Speicherverbrauch enorm. Wir passen auch nicht auf Zyklen auf, es gibt also zahlreiche Äste in unserem Baum, in denen immer von links nach rechts geschoben wird und so. Es wäre schick sich die Felder irgendwie zu merken und diese Pfade abzuschneiden.

```
import Data.Maybe
import Data.List
import System
```

```
-- O(1) Queues fuer BFS
newtype Queue a = Q ([a],[a])
```

```
(.+. ) (Q (h,t)) a = Q (h,a:t)
```

```
-- append many
(++.) q as = foldl (.+. ) q as
```

```
pop (Q ([],t)) = let t' = (reverse t) in case t' of
  [] -> (Nothing, Q ([],[]))
  (a:tt) -> (Just a, Q (tt,[]))
pop (Q (h:t,t1)) = (Just h, Q (t,t1))
```

```
singleton a = Q ([a],[])
```

```
-- Unser Feld
data Zelle = Loch | I Int deriving (Eq,Show, Read)
type Feld = [[Zelle]]
```

```
data Direction = L | R | U | D deriving Show
data Spiel = Done | N [(Direction, Spiel)] deriving Show
```

-- *Der Spielbaum. Der ist unendlich gross, weil wir nicht testen,
 -- ob wir die Stellung schonmal hatten, aber das macht uns nichts, weil wir
 lazy sind.*

```
unfoldSpiel :: Feld -> Spiel
unfoldSpiel feld
  | done feld = Done
  | otherwise = N $
    (map (\(d,x) -> (d,unfoldSpiel x))).catMaybes $
    zipWith ($) movements (repeat feld)
```

-- *Wie koennen wir uns bewegen*

```
movements = (zipWith (.) (map fmap [(,) L, (,) U, (,) R, (,) D]) [left, up,
  right, down])
left = sequence.map moveLeft where
  moveLeft [x,y,Loch] = Just [x,Loch,y] -- Profiprogrammierung
  moveLeft [x,Loch,y] = Just [Loch,x,y]
  moveLeft [Loch,_,_] = Nothing
  moveLeft xs = Just xs
```

```
right = sequence .map moveRight where
  moveRight [x,y,Loch] = Nothing
  moveRight [x,Loch,y] = Just [x,y,Loch]
  moveRight [Loch,x,y] = Just [x,Loch,y]
  moveRight xs = Just xs
```

```
up = fmap transpose.left.transpose
down = fmap transpose.right.transpose
```

-- *BFS auf dem Spielbaum, wenn wir fertig sind, sind wir fertig.*

```
foldSolution :: Spiel -> [Direction]
foldSolution tree = foldSolution' (singleton (tree,[])) where
  foldSolution' queue = let (Just (t,steps), q) = pop queue in -- die
    Schlange kann nicht leer sein
    case t of
      Done -> reverse steps
      N xs -> foldSolution' (q .++. (map (\(d,t') -> (t',d:steps)))
        xs)
```

```
done = (==) [[I 1,I 2, I 3],[I 4,I 5, I 6],[I 7,I 8, Loch]]
```

```
solution = foldSolution.unfoldSpiel
```

```
test = solution [[I 1,I 2, I 3],[Loch,I 5, I 6],[I 4,I 7, I 8]]
```

```
main = do
  [arg] <- getArgs
  let feld = read arg
  print $ solution feld
```

```
>./achtPuzzle "[I 1,Loch, I 3],[I 5,I 2, I 6],[I 4,I 7,I 8]]"
[D,L,D,R,R]
```