Praktische Informatik 3 WS 20/21

Christoph Lüth
Thomas Barkowsky
Tobias Brandt
Alexander Krug
Tarek Soliman
Robert Sachtleben

8. Übungsblatt

Ausgabe: 18.01.2021
Abgabe: 01.02.2021

In diesem Übungsblatt wollen wir eine Haskell-Version des bekannten Spieles *Vier gewinnt* (*Connect Four*) implementieren. Als Referenz für die Spielregeln gilt für uns der Wikipedia-Eintrag[1]:

> Das Spiel wird auf einem senkrecht stehenden hohlen Spielbrett gespielt, in das die Spieler abwechselnd ihre Spielsteine fallen lassen. Das Spielbrett besteht aus sieben Spalten (senkrecht) und sechs Reihen (waagerecht). Jeder Spieler besitzt 21 gleichfarbige Spielsteine. Wenn ein Spieler einen Spielstein in eine Spalte fallen lässt, besetzt dieser den untersten freien Platz der Spalte. Gewinner ist der Spieler, der es als erster schafft, vier oder mehr seiner Spielsteine waagerecht, senkrecht oder diagonal in eine Linie zu bringen. Das Spiel endet unentschieden, wenn das Spielbrett komplett gefüllt ist, ohne dass ein Spieler eine Viererlinie gebildet hat.

Die Implementierung gliedert sich in drei Teile:

(1) Modellierung des Spielbretts und der Regeln in Haskell;

(2) Implementierung des Computerspielers;

(3) Implementierung der Nutzerschnittstelle.

8.1 *Spielbrett und Spielregeln* — *7 Punkte*

Die Spieler (Steine auf dem Brett) werden wie folgt modelliert:

```
data Player = X | O    deriving (Eq, Show)
```

Modellieren Sie das Spielbrett als abstrakten Datentyp `Board` und folgenden Operationen:

```
data Board
type Position = (Int, Int)
moves :: Board → [Position]
move  :: Board → Player→ Position→ Board
full  :: Board → Bool
wins  :: Board → Player → Bool
```

Hierbei liefert `moves` alle im nächsten Zug möglichen Positionen zurück, `move` führt einen Zug durch, `full` prüft, ob das Brett voll ist, und `wins` prüft, ob der betreffende Spieler gewonnen hat.

Ferner benötigen wir Funktionen um ein Brett zu erzeugen:

```
emptyBoard  :: Int→ Board
fromString  :: String→ Board
```

`emptyBoard n` erzeugt ein leeres Spielbrett mit `n+1` Spalten und `n` Zeilen. `fromString` erlaubt es, aus einer Zeichenkette schnell ein Spielfeld zu erzeugen. Die Zeichenkette soll aus k Zeilen bestehen, die alle die gleiche Länge haben (die Länge soll mindestens $k + 1$ sein). Hier ein Beispiel, welches das Brett aus Abb. 1 links kodiert:

```
   O
 X O X
OXXOXXO
XOOXOOX
```

[1] http://de.wikipedia.org/wiki/Vier_gewinnt

Leere Zeilen oben können weggelassen werden, sie werden dann aufgefüllt.

Zum Schluss fehlt eine Funktion, welche das Brett ausgibt. Diese soll alle Gewinn-Positionen in dem Brett durch `'*'` kennzeichnen (d.h. die vier Steine in einer Reihe, Zeile, oder Diagonalen durch `'*'` ersetzen):

```
render :: Board→ String
```

8.2 *Künstliche Intelligenz* *6 Punkte*

Um den nächsten Zug des Computers (der immer `X` spielt) zu berechnen, nutzen wir einen klassischen KI-Algorithmus, den Minimax-Algorithmus. Dieser funktioniert wie folgt:

- Ausgehend von der momentanen Position wird rekursiv eine Bewertung aller möglichen Züge berechnet. Für jeden eigenen Zug wird die größten (maximalen) Bewertung ausgewählt; für jeden Zug des Gegners die kleinste (minimale). Da die eigenen Züge und die des Gegners alternieren, wechseln sich Minimum und Maximum ab, daher der Name des Algorithmus.
- Die Rekursion endet, wenn kein weiterer Zug möglich ist, einer der Spieler gewonnen hat, oder eine anderes Kriterium (beispielsweise eine vorgegebene Rekursionstiefe) erreicht wurde.
- Zum Rekursionsende wird das Spielbrett durch eine realwertige Funktion bewertet, die einen Wert zwischen 1.0 und -1.0 zurückgibt (wobei 1.0 bedeutet, dass das Spiel gewonnen wurde, und -1.0, dass verloren wurde).

Am Ende wird der Zug mit der maximalen Bewertung ausgewählt (die Auswahl kann mehrdeutig sein, wenn mehrere Züge die gleiche maximale Bewertung haben, bpsw. wenn mehrere Züge zum Gewinn führen).

Das Lehrbuch [1] oder auch Wikipedia[2] enthalten einen guten Überblick über den Algorithmus.

Der Minimax-Algorithmus ist generisch über dem zu Grunde liegenden Spiel, deshalb implementieren wir ihn im Modul `Game` mit einer Typklasse `Game`:

```
class Game b a where
  terminal :: b → Bool
  utility  :: b → Float
  actions  :: b → [a]
  result   :: b → a → b
```

Hierbei modelliert der Typ `b` den Zustand des Spieles (bestehend aus dem Spielbrett, den Spieler am Zug, aktuelle Rekursionstiefe und deren Grenze), und `a` die Züge. `terminal` kennzeichnet das Rekursionsende, `utility` ist die Bewertungsfunktion, `actions` die möglichen Züge, und `result` führt einen Zug aus.

Der Algorithmus wird implementiert in einer Funktion

```
minimax :: Game b a ⇒ b → ([a], Float)
```

welche für ein gegebenes Spiel die bestbewertesteten Züge zusammen mit der Bewertung berechnet.

Die Vorlage enthält eine Modellierung des Spieles Tic-Tac-Toe, welches die Typklasse `Game` instantiiert. Damit können Sie Ihren Minimax-Algorithmus testen, er sollte Gewinn- und Verluststellungen über mehrere Züge hinweg erkennen (es sind einige Unit-Tests in der Vorlage enthalten).

Instantiieren Sie dann den generischen Minimax-Algorithmus mit der konkreten Vier-Gewinnt-Modellierung aus Aufgabe 8.1. Der Zustand des Spieles besteht dabei aus dem Zustand des Spielbrettes, welcher Spieler als nächster zieht, und die noch verbleibende Rekursionstiefe (wir begrenzen die Rekursion, sonst ist das Spiel zu langsam).

Damit implementieren wir jetzt im Modul `ComputerPlayer` eine Funktion, welche für den Computer die bestbewertesten Züge zusammen mit der Bewertung berechnet (wobei das zweite Argument die maximale Tiefe der Vorausberechnung ist):

```
selectMove :: Board → Int→ ([Position], Float)
```

[2] https://en.wikipedia.org/wiki/Minimax

```
+-+-+-+-+-+-+-+          +-+-+-+-+-+-+-+
| | | | | | | |          | | | | | | | |
+-+-+-+-+-+-+-+          +-+-+-+-+-+-+-+
| | | | | | | |          | | | | |X| | |
+-+-+-+-+-+-+-+          +-+-+-+-+-+-+-+
| | | | |O| | |          | | | | |O| | |
+-+-+-+-+-+-+-+          +-+-+-+-+-+-+-+
| | |X| |O| |X|          | | |X| |O|O|X|
+-+-+-+-+-+-+-+          +-+-+-+-+-+-+-+
|O|X|X|O|X|X|O|          |O|X|X|O|X|X|O|
+-+-+-+-+-+-+-+          +-+-+-+-+-+-+-+
|X|O|O|X|O|O|X|          |X|O|O|X|O|O|X|
+-+-+-+-+-+-+-+          +-+-+-+-+-+-+-+
 1 2 3 4 5 6 7            1 2 3 4 5 6 7
Your move? 6
```

Abbildung 1: Ausgabe des Spielfeldes in überzeugendem Retro-Look.

8.3 *Haskell Gewinnt.* *7 Punkte*

In dieser Aufgabe implementieren wir eine Terminal-basierte Nutzerschnittstelle für unser Spiel.

Der Spielverlauf ist wie folgt:

1. Zu Beginn einer Runde wird der aktuelle Stand des Spieles in einer ansprechenden ASCII-Grafik[3] ausgegeben, und der Benutzer um seinen Zug gebeten (Abb. 1 links). Als Zug wird die Nummer der betreffenden Spalte angegeben; dort fällt dann der Stein in die letzte freie Zeile hinein.

   Nach dem Zug des Benutzers rechnet Haskell seinen Zug aus, und eine neue Runde startet. Zur Auswahl des Zuges wird die Funktion `selectMove` genutzt; gibt diese mehr als einen Zug zurück, wird einer zufällig ausgewählt. Wenn beispielsweise der Benutzer als Zug 6 angibt, und Haskell Spalte 5 wählt, ergibt sich im nächsten Zug Abb. 1 rechts.

2. Vor jedem Zug muss geprüft werden, ob das Brett voll ist; in dem Fall wird das Spiel mit einem Unentschieden beendet.

3. Nach jedem Zug muss geprüft werden, ob der ziehende Spieler (Benutzer oder Haskell) gewonnen hat, d.h. vier Steine in einer Zeile, Spalte oder Diagonalen erzielt hat. Ist dies nicht der Fall, geht das Spiel in die nächste Runde.

4. Die Benutzereingabe muss auf Plausibilität geprüft werden; ungültige Eingaben sollten in keinem Fall zum Programmabbruch führen. Die Ausgabe des Programmes sollte ähnlich wie in Abb. 1 aussehen.

Der Spieler sollte auf der Kommandozeile die Größe des Spielfeldes und die Tiefe der Vorausberechnung angeben können (die Defaultwerte sind 7 für die Spielfeldgröße und 4 für die Tiefe).

Literatur

[1] Stuart Russell and Peter Norvig. *Artifical Intelligence: A Modern Approach*. Pearson Publishing, third edition, 2010.

[3] Unicode ist aber erlaubt.