

Künstliche Intelligenz - Übung 5

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

29. Mai 2009

Aufgabe 1

Wir haben uns für das Zweispieler-Spiel *4-Gewinnt* entschieden. Die Regeln sollten ja allgemein Bekannt sein, so dass auf einer Erläuterung verzichtet wird.

Zur Umsetzung in Java haben wir uns zunächst Interfaces überlegt, um unabhängig KI's zu erstellen. Dabei musste man sich auf vier Schnittstellen einigen: Eins für einen Spieler, eins für den Zuggenerator und eins für die Stellungen im Spiel und ein letztes für den Suchbaum. Die Interfaces sind dabei wie folgt definiert:

```
public interface Spieler_I {

    /**
     * Die KI spielt.
     *
     * @param stellung
     *           Ausgangsstellung
     * @return resultierende Stellung nach dem Zug
     */
    public Stellung_I play(Stellung_I stellung);

}

interface ZugGenerator_I {

    /**
     * Berechnet ausgehend von einer Stellung alle m?glichen Z,ge bzw.
     * resultierenden Stellungen
     *
     * @param stellung
     *           Ausgangsstellung
     * @return eine Collection, damit wir zum Beispiel ein TreeSet
     *         nehmen können
     *         und die Z,ge geordnet nach Wahrscheinlichkeit ausgeben
     */
    public Collection<Stellung_I> moves(Stellung_I stellung);

}

public interface Stellung_I extends Cloneable {

    public static final byte EMPTY = -1;

    /**
     * Liefert das aktuelle Spielfeld zur,ck.
     *
     */
}
```

```

* @return Spielfeld. Erster Index gibt Spalte an, zweiter die Zeile
* . Der
*      Wert gibt an, welcher Spieler das Feld besetzt hat bzw.
*      ob das
*      Feld noch unbesetzt ist. 0 = Spieler 1 hat das Feld, 1 =
*      2, -1 =
*      leer
*
*/
public byte[][] getFeld();

/**
* Gibt den aktuellen Spieler zurueck
*
* @return 0 fuer Spieler 1, 1 fuer Spieler 2.
*/
public byte getCurrentPlayer();

/**
* Setzt den aktuellen Spieler auf den uebergebenen Parameter.
* @param player neuer aktueller Spieler. 0 = Spieler 1, 1 = Spieler
*      2
*/
public void setCurrentPlayer(byte player);

/**
* Tu nen Spielstein des aktuellen Spielers in die x-te Spalte (0 -
*      width-1)
*
* @param x
*      Spalte
* @throws IllegalMoveException
*      wenn der Zug ung,ltig ist
*/
public void set(int x) throws IllegalMoveException;

/**
* L?scht den obersten Stein in Spalte x (0 - width -1)
*
* @param x
*      Spalte
*/
public void unset(int x);

/**
* Prueft, ob Spalte x bis obenhin voll ist
*
* @param x
*      Spalte. Index von 0 bis width -1
* @return yes, wenn die Spalte voll ist, false sonst
*/
public boolean isFull(int x);

/**
* Gibt die Breite des Spielfeldes wieder.
* @return

```

```

    */
    public int getWidth();

    /**
     * Gibt die Höhe des Spielfeldes wieder
     * @return
     */
    public int getHeight();
}

public interface SpielBaum_I {

    /**
     * Gibt die Kinder des Knotens zurueck
     *
     * @return Liste der Kinder
     */
    public List<SpielBaum_I> getChildren();

    /**
     * Setzt die Kinder.
     *
     * @param children
     */
    public void setChildren(List<SpielBaum_I> children);

    /**
     * Fuegt dem Knoten ein weiteres Kind hinzu.
     */
    public void addChild(SpielBaum_I child);

    /**
     * Gibt den Vaterknoten zurueck
     *
     * @return Vater oder null, falls kein Vater vorhanden
     */
    public SpielBaum_I getParent();

    /**
     * Setzt den Vaterknoten
     *
     * @param parent
     */
    public void setParent(SpielBaum_I parent);

    /**
     * Setzt die Stellung
     *
     * @param stellung
     */
    public void setStellung(Stellung_I stellung);

    /**
     * Liefert die Stellung zurueck
     *

```

```

        * @return
        */
    public Stellung_I getStellung();

    /**
     * Gibt die Wertung des Knotens zur,ck
     *
     * @return Wertung zwischen -100 und 100
     */
    public int getWertung();

    /**
     * Setzt die Wertung
     *
     * @param wertung
     */
    public void setWertung(int wertung);
}

```

Aus spieltheoretischer Sicht ist 4-Gewinnt ein Zweispieler-Spiel, in dem die Spieler abwechselnd an der Reihe sind und natürlich gegeneinander Spielen. Das Spielbrett ist jederzeit für beide einsehbar und daher haben beide Parteien stets den vollen Informationsstand. Es gibt zudem auch keine Wahrscheinlichkeiten, da nicht gewürfelt oder Karten gezogen werden.

Aufgabe 2

Die Interfaces wurden ja schon kurz vorgestellt, im Prinzip decken sie auch alles gut ab. Die KI implementiert dann das `Spieler_I`-Interface, wo dann der Suchbaum aufgebaut und per Alpha-Beta oder Minmax-Algorithmus durchsucht wird. Da wir zwei funktionierende KIs haben, ist es ungünstig hier diese zu erklären, ein Blick in den Quellcode wäre wohl am aufschlussreichsten.

Doch ein paar Worte noch zu den Heuristiken. Zunächst haben wir die Bewertungsfunktion, die die schwachen Heuristiken entsprechend gewichtet und summiert, selber als Heuristik definiert, so dass sehr transparent gearbeitet werden kann und jederzeit die Bewertungsfunktion ersetzt oder erweitert werden könnte. Entsprechend haben wir diese Heuristik auch die `MasterHeuristik` getauft.

```

public class MasterHeuristik implements Heuristik_I {

    public static Heuristik_I MasterHeuristik() {
        return new MasterHeuristik(new Heuristik_I[] {
            new Gewinnt(), new Verliert(), new
                Threats(), new Lage()
        }, new double[] {
            0.50,
            0.25,
            0.12,
            0.13
        });
    }

    public final Heuristik_I[] weakOnes;
    private final double[] weights;

    /**
     * @see heuristiken.Heuristik_I#bewerte(spiel.Stellung_I)

```

```

    */
    public int bewerte(Stellung_I s) {
        double acc=0;
        for(int i=0; i<weights.length;++i)
            acc+=weak0nes[i].bewerte(s)*weights[i];
        return (int)Math.round(acc);
    }
}

```

Von den schwachen Heuristiken gibt es momentan vier Stück. **Gewinnt**, **Verliert**, **Threats** und **Lage**. Gewinnt und Verliert berechnen dabei die maximale Länge von Steinen, die in einer Reihe (horizontal oder vertikal) bzw. diagonal liegen. Bei vier Steinen in einer Reihe wird die höchste Bewertung (100 oder -100, je nach Spieler) zurückgegeben. Verliert macht das gleiche, nur für den Gegenspieler. Ein Threat prüft auf Dreiergruppen, die durch einen vierten Stein vervollständigt werden können. Die Lage-Heuristik prüft dann noch, wie die Steine auf dem Brett liegen. Die gewählten Gewichtungungen sind mehr oder weniger beliebig und haben sich so als ganz gut rausgestellt. Mit AdaBoost könnte man sie natürlich selber einregeln lassen ...