Karlsruher Institut für Technologie

Programmieren – Wintersemester 2013/14
Forschungsgruppe Verifikation trifft Algorithmik
`http://baldur.iti.kit.edu/programmieren`
Dr. Carsten Sinz · Markus Iser · Florian Merz

Abschlussaufgabe 1

Ausgabe: 17.02.2014 – 13:00
Abgabe: 17.03.2014 – 13:00

Allgemeine Hinweise

- Achten Sie darauf nicht zu lange Zeilen, Methoden und Dateien zu erstellen[1]
- Programmcode muss in englischer Sprache verfasst sein
- Kommentieren Sie Ihren Code angemessen: So viel wie nötig, so wenig wie möglich
- Wählen Sie geeignete Sichtbarkeiten für Ihre Klassen, Methoden und Attribute
- Achten Sie auf fehlerfrei kompilierenden Programmcode[1]
- Halten Sie alle whitespace Regeln ein[1]
- Halten Sie die Regeln zu Variablen-, Methoden und Paketbenennung ein und wählen Sie aussagekräftige Namen[1]
- Halten Sie die Regeln zu Javadoc Dokumentation ein[1]
- Nutzen Sie nicht das default package[1]
- Achten Sie auf die korrekte Sichtbarkeit Ihrer Klassen[1]
- Halten Sie auch alle anderen checkstyle Regeln ein[1]

Vier Gewinnt

Die erste Abschlussaufgabe besteht darin, nach der vorgegebenen Anleitung das bekannte Spiel *Vier Gewinnt* zu implementieren. *Vier Gewinnt* ist ein endliches Zwei-Personen-Nullsummenspiel mit perfekter Information.[2] Das bedeutet, dass (bei jedem Zug) jeweils der Verlust des einen Spielers gleich dem Gewinn seines Opponenten ist, und dass zu jedem Zeitpunkt beiden Spielern die gesamte Historie der Partie bekannt ist.

Das Spielfeld von Vier Gewinnt, besteht aus 7 Spalten und 6 Zeilen, wobei abwechselnd zwei Spieler jeweils eine Spalte auswählen, und eine Münze ihrer Farbe einwerfen, mit dem Ziel vier Münzen ihrer Farbe nebeneinander in einer Reihe, Spalte oder Diagonalen zu positionieren.

Über die Benutzerschnittstelle werden die Züge der menschlichen Spieler entgegengenommen und die Spielzustände ausgegeben. Hierfür erhalten Sie von uns die Klasse `Terminal`, die Sie für alle Ein- und Ausgaben Ihres Programms benutzen müssen.[3]

Der grundlegende Such-Algorithmus des Computergegners soll so implementiert werden, dass er für jedes beliebige Zwei-Personen-Nullsummenspiel mit perfekter Information verwendet werden kann.

[1] Der Praktomat wird die Abgabe zurückweisen, falls diese Regel verletzt ist.
[2] Für eine Definition siehe z.B. `http://de.wikipedia.org/wiki/Nullsummenspiel`.
[3] Diese Einschränkung müssen Sie zwecks automatischer Testbarkeit zwingend einhalten.

A Die Benutzerschnittstelle

Ihr Spiel soll genau eine Klasse mit einer `main()`-Methode enthalten, die als Programmargument eine Zahl im Intervall [0, 2] für die Anzahl der menschlichen Spieler entgegennimmt. Gibt es zwei menschliche Spieler, werden abwechselnd beide Spieler nach dem nächsten Zug gefragt, gibt es einen menschlichen Spieler und einen Computergegner, so werden die Züge des Computers durch den unten beschriebenen Algorithmus berechnet, während der menschliche Spieler nach dem jeweils nächsten Zug gefragt wird. Gibt es keine menschlichen Spieler, so kann man den beiden Computergegnern beim Spielen zusehen.

Beachten Sie im Sinne der Testbarkeit die folgenden Hinweise genau:

Es gibt zwei Spieler, `Spieler 1` und `Spieler 2`. `Spieler 1` beginnt immer. Gibt es einen menschlichen Spieler und einen Computergegner, so ist der menschliche Spieler `Spieler 1`.

Vor *jedem* Zug wird der aktuelle Zustand des Spielfeldes auf der Konsole ausgegeben (mit `Terminal.println()`). Nach einer Leerzeile (mit `Terminal.println()`) folgt (mit `Terminal.prompt()`) die Ausgabe "Player␣X:␣" (wobei X die Spielernummer bezeichnet und ␣ für ein Leerzeichen steht). Dann folgt entweder die Eingabeaufforderung für die Spaltennummer (mit `Terminal.readln()`) oder die Ausgabe der berechneten Spaltennummer (mit `Terminal.println()`). Die Spaltennummern beginnen bei 0, das heißt, die Eingabeaufforderung akzeptiert nur Werte im Intervall [0,6]. Falls es bei der Eingabeaufforderung zu Fehleingaben kommt, geben Sie (mit `Terminal.println()`) eine Fehlermeldung aus, die mit "Error,␣" beginnt. Wiederholen Sie dann die Ausgabe von "Player␣X:␣" und die Eingabeaufforderung.

Die Ausgabe des Spielzustandes erfolgt Spaltenweise von links nach rechts, wobei die einzelnen Spalten durch "|" voneinander getrennt sind. Die Felder enthalten entweder ein Leerzeichen, falls das Feld leer ist oder "1" falls `Spieler 1` bzw. "2" falls `Spieler 2` das Feld besetzt hat.

Wenn das Spiel vorbei ist, also entweder keine Züge mehr möglich sind, oder einer der Spieler gewonnen hat, geben Sie noch ein letztes Mal den Spielstand aus. Falls das Spiel gewonnen ist, geben Sie dann in einer Zeile "Player␣X␣won␣the␣game!" aus, wobei das X durch die jeweilige Spielernummer zu ersetzen ist. Falls keine Züge mehr möglich sind, ohne dass ein Spieler gewonnen hat, geben Sie in einer Zeile "Draw!" aus. Das Programm beendet sich dann jeweils.

A.1 Beispiel: leeres Spielfeld

```
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
```

A.2 Beispiel: Sieg für Spieler zwei

```
| | | | |1| | |
| | | |2|2| | |
| | | |2|2|1| |
| | | |1|2|2| |
| | |1|2|1|1|2|
| |2|1|1|2|1|1|
```

A.3 Beispiel Ablauf (Human vs. Computer)

```
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |

Player 1: 3
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |1| | | |

Player 2: 3
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |2| | | |
| | | |1| | | |

Player 1: 3
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | |1| | | |
| | | |2| | | |
| | | |1| | | |

Player 2: 3
| | | | | | | |
| | | | | | | |
| | | |2| | | |
| | | |1| | | |
| | | |2| | | |
| | | |1| | | |

Player 1: 5
| | | | | | | |
| | | | | | | |
| | | |2| | | |
| | | |1| | | |
| | | |2| | | |
| | | |1| |1| |

Player 2: 4
| | | | | | | |
| | | | | | | |
| | | |2| | | |
| | | |1| | | |
| | | |2| | | |
| | | |1|2|1| |

Player 1:
.
.
.
```

B Hintergrundinformationen zum Computer-Gegner

In diesem Abschnitt wird der Algorithmus für den Computer-Gegner schrittweise entwickelt. Bitte beachten Sie, dass *nicht* sämtliche angegebenen Suchalgorithmen implementiert werden müssen, sondern nur der Algorithmus aus Abschnitt B.3 mit der Optimierung aus Abschnitt B.4 und der lokalen Evaluation aus Abschnitt B.2.1.

Der Algorithmus, der die künstliche Intelligenz des Computergegners simuliert, nutzt die anti-symmetrischen Eigenschaften des Nullsummenspiels. Das heißt, dass die maximale Ausbeute für `Spieler 1` den maximalen Verlust für `Spieler 2` bedeutet und umgekehrt.

B.1 Rekursives Durchsuchen des Spielbaums

Bei einer einfachen rekursiven Formulierung des sogenannten *Min-Max-Algorithmus* (hier die *NegaMax*-Variante), sucht man unter allen möglichen Zügen nach demjenigen Zug, bei dem der Gegner die schlechtesten Optionen hat. Die beste Option für den Gegner wird nach dem gleichen Prinzip gesucht und so lässt sich der rekursive Algorithmus 1 formulieren.

Die Rekursion ist in jedem Fall begrenzt, da es irgendwann keine möglichen Züge mehr gibt (`end position`). Nämlich genau dann, wenn ein Spieler verloren hat, oder (bei einer Patt-Situation) das ganze Spielfeld gefüllt ist.

Beachten Sie, dass die Algorithmen, wie Sie hier vorgestellt werden, der Einfachheit halber jeweils nur die Bewertung der Züge vornehmen. Auf oberster Ebene, muss man sich natürlich "merken" welches jetzt der beste Zug war.

```
double max() {
    best = −∞;

    if end position {
        return −∞ if lost, +∞ if won, 0 if draw
    }

    for each move in list of possible moves {
        perform(move);

        value = -max(); // recursive call

        undo(move);

        if (value > best) {
            best = value;
        }
    }

    return best;
}
```

Algorithmus 1: NegaMax

B.2 Rekursive Suche mit Horizont und lokaler Evaluation

Da beim NegaMax-Algorithmus die Anzahl der zu betrachtenden Spielstände exponentiell mit der Tiefe des Suchbaums wächst, ist dieser Algorithmus so in der Praxis nicht brauchbar.

Daher wird die Suche üblicherweise in der Tiefe begrenzt, das heißt es gibt einen sogenannten *Horizont* (das ist der Spielstand, an dem die Rekursion abgebrochen wird), an dem der Spielstand lokal evaluiert, d.h. "abgeschätzt" wird. Die lokale Evaluation liefert einen positiven Wert, wenn der Spieler, der gerade an der Reihe ist, einen Vorteil gegenüber seinem Gegner hat. Je höher dieser Wert ist, um so größer ist sein Vorteil. Ist der Wert negativ, so ist der Gegner im Vorteil.

In Abschnitt B.2.1 ist ein Verfahren angegeben, wie eine solche lokale Evaluation berechnet werden kann.

Algorithmus 2 beschreibt den NegaMax-Algorithmus mit begrenztem Horizont.

```
double max(depth) {
    if end position or depth = 0 {
        return localEvaluation();
    }

    best = −∞;

    for each move in list of possible moves {
        perform(move);

        value = -max(depth - 1);

        undo(move);

        if (value > best) {
            best = value;
        }
    }

    return best;
}
```

Algorithmus 2: NegaMax mit Horizont

B.2.1 Lokale Evaluation

Wir beschäftigen uns nun mit der Frage, wie eine lokale Evaluation eines Spielstandes vorgenommen werden kann. Die Bewertung, die wir betrachten, soll auf dem Begriff der "Drohungen" basieren. Dabei ist eine Drohung das Vorhandensein von drei Spielsteinen in einer Reihe mit einem freien Feld im Anschluss **oder dazwischen**, d.h. eine potentielle Möglichkeit vier Spielsteine in einer Reihe zu erreichen. Wir wollen den Spielstand um so höher bewerten, je mehr eigene Drohungen auf dem Spielfeld existieren und je schneller diese erreichbar sind.

Eine Drohung definieren wir daher als ein freies Feld, das bei Besetzen durch einen Spieler zu vier Spielsteinen in einer Reihe führt. Der Erreichbarkeitswert $r(x)$ einer Drohung x ist die Anzahl der Spielsteine, die benötigt werden, um aus dem aktuellen Spielstand heraus die Drohung auf Feld x zu realisieren.

Eine Drohung x, die im nächsten Zug bereits belegt werden kann (d.h. mit $r(x) = 1$), nennt man eine *akute Drohung*.

Beispiele für Drohungen und deren Erreichbarkeitswert sind in Tabelle 1 zu finden.

	0	**1**	**2**	**3**	**4**	**5**	**6**
F	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0
C	0	0	2	2	1	0	0
B	0	1	1	1	2	0	0
A	0	2	1	2	1	2	0

Tabelle 1: Im Beispiel gibt es eine akute Drohung x von `Spieler 2` ($r(x) = 1$) auf Feld D-2, eine Drohung y von `Spieler 1` ($r(y) = 2$) auf Feld B-0, sowie eine weitere Drohung z von `Spieler 1` ($r(z) = 3$) auf Feld D-5.

Wir definieren nun eine lokale Evaluationsfunktion basierend auf Drohungen. Dazu bestimmen wir zunächst den Begriff der Gesamtdrohung eines Spielers. Sei $r : D \to \mathbb{N}$ die Funktion, die jeder Drohung $d \in D$ ihre Erreichbarkeit zuordnet, und seien $D_1 \subseteq D$ die Drohungen von Spieler 1 und $D_2 \subseteq D$ die Drohungen von Spieler 2. Die Gesamtdrohung a_i eines Spielers $i \in \{1, 2\}$ berechnet man wie folgt:

$$a_i = \sum_{d \in D_i} \frac{1}{\ln(r(d)+1)}$$

Die Evaluationsfunktion aus Algorithmus 3 liefert die Differenz der Gesamtdrohungen beider Spieler, falls noch kein Spieler gewonnen hat. In einer Pattsituation liefert sie den Wert 0, da $D = \emptyset$. Falls ein Spieler gewonnen hat, wird je nach Lage $+\infty$ oder $-\infty$ zurückgegeben.

```
double localEvaluation() {
    int i = active player
    int j = opponent

    if (player i lost the game) {
      return −∞;
    } else if (player j lost the game) {
      return +∞;
    } else {
      return a_i − a_j;
    }
}
```

Algorithmus 3: Lokale Evaluation als Differenz der Gesamtdrohungen

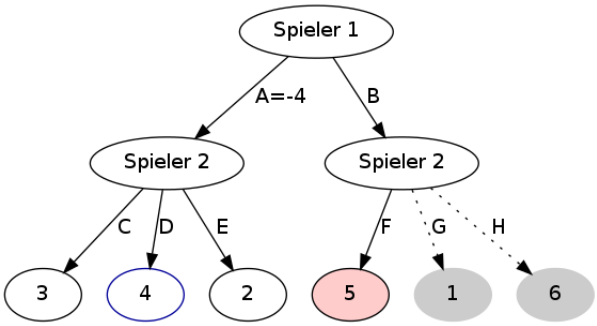
B.3 Rekursive Suche mit Beta-Pruning

Da auch die Suche mit Horizont noch relativ ineffizient ist, benutzen wir noch eine weitere gängige Optimierung, das sogenannte Beta-Pruning. Hier wird die Suche geschickt in der Breite begrenzt, indem man jedem untersuchten Teilbaum den besten Wert der bereits untersuchten Nachbar-Teilbäume mitgibt. Wenn der Algorithmus jetzt den Zweig des Gegners durchsucht und merkt, dass der Gegner hier besser abschneiden kann, als in einem der bereits untersuchten Teilbäume, kann an der Stelle die Suche abgebrochen werden, weil bereits klar ist, dass der Spieler diesen Zug niemals wählen würde, um dem Gegner diese Chance nicht zu geben.

Da Gegner und Spieler während der Rekursion jeweils die Rollen tauschen, werden zwei Werte `alpha` und `beta` mitgeführt, wobei `alpha` an der Wurzel mit dem Wert $-\infty$ und `beta` mit $+\infty$ initialisiert wird. Diese Überlegungen führen uns zu Algorithmus 4, dem sogenannten Alpha-Beta-Algorithmus. Beachten Sie, dass `alpha` hier die Rolle der Variablen `best` aus Algorithmus 2 mit übernimmt.

B.3.1 Beta-Pruning

Die Grafik rechts soll das Beta-Pruning veranschaulichen. In der dargestellten Suche ist der Teilbaum für Zug A von `Spieler 1` schon evaluiert und hat den Wert -4. Jetzt ist bei der Evaluation des Teilbaums für Zug B bereits nach dem ersten Knoten mit Wertung 5 klar, dass `Spieler 1` diesen Knoten niemals wählen wird, egal was noch kommt. Das liegt daran, dass schon ein besserer Zug (Zug A, beta=4) bekannt ist, und die Annahme gilt, dass jeder Spieler den besten Zug wählt.



Die Suche wird daher unterbrochen und der Wert 5 wird nach oben weitergereicht. Für `Spieler 1` erhält Zug B die Wertung -5. Beachten Sie, dass bei einer vollständigen Suche die Wertung -6 vorgenommen werden würde. Der Wert -5 reicht aber bereits aus, um diesen Zug niemals zu wählen.

```
double max(depth, alpha, beta) {
    if end position or depth = 0 {
        return localEvaluation();
    }

    for each move in list of possible moves {
        perform(move);

        value = -max(depth - 1, -beta, -alpha);

        undo(move);

        if (value > alpha) {
            alpha = value;

            if (value >= beta) {
                break; // beta-pruning
            }
        }
    }

    return alpha;
}
```

Algorithmus 4: Alpha-Beta-Algorithmus

B.4 Optimierung: Sortierung der Move-Liste

Um die Vorteile des Beta-Pruning besser zu nutzen, ist man darauf bedacht, dass für jeden Spieler seine aussichtsreichsten Spielzüge zuerst untersucht werden. Das bedeutet, dass man an jedem Punkt in der Rekursion die Liste der möglichen Züge (`list of possible moves`) sortiert.

Verwenden Sie zur Sortierung drei Kriterien, die Sie der Reihe nach prüfen. Ein weiteres Kriterium wird nur zu Rate gezogen, wenn alle vorherigen keine Unterscheidung ermöglichen, d.h. die drei Kriterien sind priorisiert (lexikographische Sortierung).

Das erste Kriterium der Sortierung prüft Spalten, die *eigene* Drohungen enthalten. Solche Spalten werden immer bevorzugt, das heißt jeder Spieler betrachtet zunächst die Spalten, in denen eigene Vierer-Reihen erreicht werden können. Dabei spielt die Erreichbarkeit der Drohung **keine** Rolle.

Das zweite Kriterium berechnet für das nächste zu besetzende Feld jeder Spalte, die Anzahl der Viererreihen, die unter Einbeziehung dieses Feldes prinzipiell möglich sind. Hierzu kann man die Werte aus Tabelle 2 statisch ins Programm übernehmen und den passenden Wert für das zweite Kriterium nachschlagen.

Beachten Sie, dass Sie sowohl die Spalten mit als auch die Spalten ohne eigene Drohungen untereinander nach diesem Kriterium sortieren müssen.

Zuletzt, d.h. wenn zwei Spalten nach den zuvor betrachteten Kriterien gleich sind, wird von links nach rechts sortiert, d.h. kleinere Spaltennummern werden bevorzugt.

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Tabelle 2: Anzahl möglicher Vierer pro Feld im Vier-Gewinnt Spiel.

C Aufgabenstellung

Implementieren Sie ein *Vier-Gewinnt*-Spiel, welches für den Computergegner den Alpha-Beta-Algorithmus aus Abschnitt B.3 mit Sortierung der Zugliste nach Abschnitt B.4 und lokaler Evaluation nach Abschnitt B.2.1 umsetzt.

Gehen Sie dazu wie folgt vor:

1. Überlegen Sie sich eine Repräsentation des Spielfeldes und implementieren Sie die vorgegebenen generischen Schnittstellen `Game` und `Move` in eigenen Klassen für das *Vier-Gewinnt*-Spiel.
2. Implementieren Sie den Alpha-Beta-Algorithmus für den Computergegner in einer generischen Klasse mit folgender Signatur:

   ```
   public class AlphaBetaAlgorithm<G extends Game<M>, M extends Move>
        implements AIPlayer<M>
   ```

   Ihre Implementierung soll nicht nur für das *Vier-Gewinnt*-Spiel verwendbar sein, sondern auch für andere Nullsummenspiele (wie z.B. Schach).
3. Schreiben Sie eine Hauptklasse `ConnectFour` mit einer `main`-Methode, über die das Spiel gestartet werden kann. Verwenden Sie einen Kommandozeilenparameter, über den die Anzahl der menschlichen Spieler angegeben werden kann (vgl. Abschnitt A).

Verwenden Sie in Ihrer Implementierung die von uns definierten Schnittstellen `Game`, `Move` und `AIPlayer` und verändern Sie diese *nicht*. Verwenden Sie für die Modellierung eines Spielzuges das Interface `Move`. Verwenden Sie für Ihre *Vier Gewinnt*-Implementierung das Interface `Game`.

Verwenden Sie für Ihren Computergegner einen Horizont von 11 Halbzügen (`depth` = 11).

Prüfen Sie sämtliche Benutzereingaben auf zulässige Werte.

Testen Sie Ihre Implementierung intensiv.

Nachtrag: Verwenden Sie für die Double-Vergleiche ein EPSILON von $1E-14$.