Praktische Informatik II
Tutor:in: Tobias Diehl

SoSe 2024
Bearbeiter:in: Altug Uyanik, Öykü Koç

Übungsblatt 4

Lösungsvorschlag

FourInARow.java

Die FourInARow Klasse wurde von **Öykü Koç** verfasst.

Aufgabe 1.1 Implementierung

Die Klasse repräsentiert das Spiel Vier-in-einer-Reihe (Four-in-a-Row) und regelt den Ablauf des Spiels. Das Spielfeld wird als zweidimensionales Feld gehalten, auf dem menschliche und Computer-Spieler ihre Züge machen können. Die Methode `humanMove` erlaubt es dem menschlichen Spieler, einen Zug zu einer bestimmten Position zu machen und dann den Spielzustand auszuwerten. Die Methode `computerMove` erlaubt es dem Computer, den besten Zug auszuwählen und den Spielstand mit Hilfe des NegaMax Alpha-Beta Algorithmus zu bestimmen. Die Methode `getGameResult`, die das Ergebnis des Spiels ermittelt, und die Methode `toString`, die das Spielfeld als Text darstellt, sind weitere wichtige Merkmale der Klasse. Die Klasse `FourInARow` bietet eine umfassende Struktur, die es den Spielern ermöglicht, Züge zu machen, Züge rückgängig zu machen und den Spielzustand zu bewerten.

Die Variablen `field` und `depth` sind private Instanzvariablen der Klasse. Das Feld `field` ist ein zweidimensionales Array vom Typ `Player`, das das Spielfeld repräsentiert. Es muss quadratisch sein, was bedeutet, dass sowohl die Anzahl der Zeilen als auch die Anzahl der Spalten gleich sein müssen. Die Variable `depth` speichert die maximale Suchtiefe für den NegaMax-Algorithmus.

```
12 class FourInARow {
13
14     /**
15      * Spielfeld.
16      */
17     private final Player[][] field;
18     private final int depth;
```

Der Konstruktor `FourInARow` wird verwendet, um eine Instanz der Klasse zu initialisieren. Er akzeptiert zwei Parameter: `field` für das Spielfeld und `depth` für die maximale Suchtiefe. Das Spielfeld wird als zweidimensionales Array übergeben, das den aktuellen Zustand des Spiels darstellt. Die Tiefe gibt an, wie viele Züge im Voraus der Algorithmus analysieren soll.

```
20     /**
21      * Konstruktor.
22      * @param field Spielfeld. Muss quadratisch sein.
23      * @param depth Maximale Suchtiefe.
24      */
25     FourInARow(final Player[][] field, int depth) {
26         this.field = field;
27         this.depth = depth;
28     }
```

Die `humanMove` Methode in der `FourInARow` Klasse ermöglicht es dem menschlichen Spieler, einen Zug auszuführen, indem sie das Symbol des menschlichen Spielers (`HUMAN`) an die angegebene Position im Spielfeld setzt. Danach wird der Spielstatus überprüft, um festzustellen, ob der menschliche Spieler gewonnen hat (`HUMAN_WON`), das Spiel unentschieden steht (`DRAW`) oder fortgesetzt

werden soll (`CONTINUE`). Diese Methode ist entscheidend für die Interaktion des menschlichen Spielers mit dem Spiel und liefert das Ergebnis des Spiels nach seinem Zug.

```
30      /**
31       * Führt den Zug des Menschen aus.
32       * Diese Methode setzt das Symbol des menschlichen Spielers an die angegebene
                Position im Spielfeld
33       * und überprüft danach den Spielstatus, um festzustellen, ob der menschliche Spieler
                 gewonnen hat,
34       * das Spiel unentschieden ist oder das Spiel weitergeht.
35       *
36       * @param row Zeile des Zuges.
37       * @param column Spalte des Zuges.
38       * @return Das Ergebnis des Spiels nach dem Zug des Menschen.
39       */
40      Result humanMove(final int row, final int column) {
41          field[row][column] = HUMAN;
42          return getGameResult(HUMAN);
43      }
```

Die Methode `computerMove()` verwendet den NegaMax-Algorithmus mit Alpha-Beta-Pruning, um den besten Zug für den Computer zu berechnen. Nachdem der Zug berechnet wurde, setzt sie das Symbol des Computers an die berechnete Position im Spielfeld und überprüft anschließend den Spielstatus.

```
45      /**
46       * Führt den Zug des Computers aus.
47       * Diese Methode verwendet den NegaMax-Algorithmus mit Alpha-Beta-Pruning, um den
                besten möglichen Zug für den Computer
48       * zu berechnen. Nachdem der Zug berechnet wurde, wird das Symbol des Computers an
                die berechnete Position gesetzt und
49       * der Spielstatus wird überprüft.
50       *
51       * @return Das Ergebnis des Spiels nach dem Zug des Computers.
52       */
53      Result computerMove() {
54          Move bestMove = negaMaxAlphaBeta(COMPUTER, depth, -Integer.MAX_VALUE, Integer.
                    MAX_VALUE);
55          field[bestMove.getRow()][bestMove.getColumn()] = COMPUTER;
56          return getGameResult(COMPUTER);
57      }
```

Die Methode `toString()` wandelt das Spielfeld in eine textbasierte Darstellung um. Sie durchläuft das Spielfeld Zeile für Zeile und erstellt dabei eine Zeichenfolge, die jede Zelle mit dem Spieler-Symbol in eckigen Klammern darstellt. Diese Darstellung bietet eine visuelle Repräsentation des aktuellen Spielstands in einem leicht lesbaren Format.

```
59      @Override
60      public String toString() {
61          final StringBuilder string = new StringBuilder();
62          String separator = "";
63          for (final Player[] row : field) {
64              string.append(separator);
65              separator = "\n";
66              for (final Player player : row) {
67                  string.append("[").append(player).append("]");
68              }
69          }
70          return string.toString();
71      }
```

Die Methode `evaluate` bewertet den aktuellen Spielzustand aus Sicht des übergebenen Spielers. Sie prüft zuerst, ob der angegebene Spieler gewonnen hat, indem sie die Methode `hasWon(player)` aufruft. Falls der Spieler gewonnen hat, gibt die Methode eine positive Bewertung zurück: `Integer.MAX_VALUE`, wenn es sich um den menschlichen Spieler handelt, andernfalls `Integer.MAX_VALUE`. Wenn der Spieler nicht gewonnen hat oder das Spiel noch nicht beendet ist, wird die Bewertung mit null zurückgegeben. Damit bewertet die `evaluate`-Methode den Spielzustand aus der Perspektive eines bestimmten Spielers und gibt entsprechend eine Bewertung

zurück.

```
73      /**
74       * Bewertet den Spielzustand.
75       * Diese Methode berechnet eine Bewertung für den aktuellen Spielzustand aus Sicht
                 des angegebenen Spielers.
76       * Wenn der angegebene Spieler gewonnen hat, wird eine positive Bewertung zurü
                 ckgegeben, andernfalls null.
77       * Die Bewertung basiert auf der Anzahl der verbleibenden möglichen Züge für den
                 menschlichen Spieler.
78       *
79       * @param player Der Spieler, für den der Spielzustand bewertet wird.
80       * @return Die Bewertung des aktuellen Spielzustands.
81       */
82      public int evaluate(Player player) {
83          boolean wonStatus = hasWon(player);
84          if (wonStatus)
85              return player == HUMAN ? Integer.MAX_VALUE : -Integer.MAX_VALUE;
86          else
87              return 0;
88      }
```

Die Methode `makeMove` aktualisiert das Spielfeld, indem sie das Symbol des angegebenen Spielers (`player`) an der Position platziert, die durch die `move`-Instanz angegeben ist. Diese Methode ändert direkt das `field`-Array, indem sie das Element in der Zeile und Spalte setzt, die durch `move.getRow()` und `move.getColumn()` angegeben sind, auf das Symbol des Spielers.

```
90      /**
91       * Macht einen Zug auf dem Spielfeld.
92       * Diese Methode setzt das Symbol des angegebenen Spielers an die in der Move-Instanz
93       * angegebene Position im Spielfeld.
94       *
95       * @param move Der zu machende Zug.
96       * @param player Der Spieler, der den Zug macht.
97       */
98      public void makeMove(Move move, Player player) {
99          field[move.getRow()][move.getColumn()] = player;
100     }
```

Im Gegensatz dazu macht die Methode `undoMove` einen Zug rückgängig, der zuvor auf dem Spielfeld gemacht wurde. Sie entfernt das Symbol an der Position, die durch die `move`-Instanz angegeben ist, und setzt es auf `EMPTY` zurück. Diese Methode ist nützlich, um das Spielfeld auf seinen vorherigen Zustand zurückzusetzen, wenn verschiedene Spielszenarien durchgespielt werden oder während der Rückgängig-Machung in der Benutzeroberfläche des Spiels.

```
102     /**
103      * Nimmt einen Zug zurück.
104      * Diese Methode entfernt das Symbol an der in der Move-Instanz angegebenen Position
                 im Spielfeld
105      * und setzt es auf leer (EMPTY).
106      *
107      * @param move Der Zug, der zurückgenommen werden soll.
108      */
109     public void undoMove(Move move) {
110         field[move.getRow()][move.getColumn()] = EMPTY;
111     }
```

Die Methode `other` gibt den anderen Spieler zurück, basierend auf dem aktuellen Spieler. Wenn der aktuelle Spieler der Mensch ist, wird der Computer zurückgegeben, und umgekehrt.

```
113     /**
114      * Gibt den anderen Spieler zurück.
115      * Diese Methode gibt den anderen Spieler zurück. Wenn der aktuelle Spieler der
                 Mensch ist, wird der Computer zurückgegeben,
116      * und umgekehrt.
117      *
118      * @param player Der aktuelle Spieler.
119      * @return Der andere Spieler.
120      */
121     public Player other(Player player) {
```

```
122         return (player == HUMAN) ? COMPUTER : HUMAN;
123     }
```

Die Methode `goodScore` berechnet eine Punktzahl basierend auf der aktuellen Suchtiefe (`height`). Diese Punktzahl wird verwendet, um den Wert von Zügen zu erhöhen, die weiter in die Zukunft blicken. Typischerweise wird die Punktzahl durch Multiplikation der Tiefe mit 10 berechnet, was eine einfache Heuristik darstellt, um die Bewertung des Spielzustands zu beeinflussen.

```
125     /**
126      * Berechnet eine gute Punktzahl basierend auf der Tiefe.
127      * Diese Methode berechnet eine Punktzahl basierend auf der aktuellen Tiefe der Suche
                .
128      * Dies kann verwendet werden, um den Wert von Zügen zu erhöhen, die weiter in die
                Zukunft blicken.
129      *
130      * @param height Die aktuelle Tiefe.
131      * @return Die berechnete Punktzahl.
132      */
133     public int goodScore(int height) {
134         return height * 10;
135     }
```

Die Methode `possibleMoves` gibt eine Liste der möglichen Züge für den angegebenen Spieler zurück. Sie durchsucht das Spielfeld nach leeren Positionen (`EMPTY`) und fügt jedes Feld, das dem angegebenen Spieler gehört, der Liste der möglichen Züge hinzu. Jeder Zug wird durch eine Instanz der Klasse `Move` repräsentiert, die die Zeile und Spalte des Zugs sowie eine Bewertung (initialisiert auf 0) enthält. Die Methode gibt eine Liste aller solchen Züge zurück, die der Spieler machen kann.

```
137     /**
138      * Gibt eine Liste der möglichen Züge zurück.
139      * Diese Methode durchsucht das Spielfeld nach leeren Positionen (EMPTY) und erstellt
                eine Liste aller möglichen Züge
140      * für den angegebenen Spieler.
141      *
142      * @param player Der Spieler, für den die möglichen Züge ermittelt werden.
143      * @return Eine Liste der möglichen Züge.
144      */
145     public List<Move> possibleMoves(Player player) {
146         List<Move> emptyMoveList = new ArrayList<>();
147         for (int i = 0; i < field.length; i++) {
148             for (int j = 0; j < field.length; j++) {
149                 if (field[i][j] == player) {
150                     emptyMoveList.add(new Move(i, j, 0));
151                 }
152             }
153         }
154         return emptyMoveList;
155     }
```

Die Methode `hasWon` überprüft, ob der angegebene Spieler das Spiel gewonnen hat, indem sie das Spielfeld horizontal, vertikal und diagonal nach einer Viererreihe des Spielersymbols durchsucht. Zuerst wird das Spieler-Symbol entsprechend des gegebenen Spielers festgelegt. Dann werden zwei Zeichenfolgen `leftSymbol` und `rightSymbol` initialisiert, um die Symbole in den aktuellen Zeilen und Spalten zu speichern.

Die Methode durchläuft zunächst das Spielfeld horizontal und vertikal. Für jede Zeile wird `leftSymbol` und für jede Spalte `rightSymbol` aufgebaut. Wenn entweder `leftSymbol` oder `rightSymbol` eine Zeichenfolge enthält, die aus vier aufeinanderfolgenden Spieler-Symbolen besteht, wird `true` zurückgegeben, was bedeutet, dass der Spieler gewonnen hat.

Danach werden die Diagonalen überprüft. Zuerst die Hauptdiagonalen von links oben nach rechts unten und von rechts oben nach links unten. Dann werden die Nebendiagonalen überprüft. Für jede Richtung wird `leftSymbol` und `rightSymbol` entsprechend aufgebaut. Wenn eine der Diagonalen eine Zeichenfolge enthält, die aus vier aufeinanderfolgenden Spieler-Symbolen besteht,

wird ebenfalls `true` zurückgegeben.

Wenn keine Viererreihe gefunden wird, wird `false` zurückgegeben, was bedeutet, dass der Spieler nicht gewonnen hat.

```
157     /**
158      * Überprüft, ob ein Spieler gewonnen hat.
159      * Diese Methode überprüft, ob der angegebene Spieler gewonnen hat, indem sie das
                Spielfeld horizontal,
160      * vertikal und diagonal nach einer Viererreihe des Symbols des Spielers durchsucht.
161      *
162      * @param player Der Spieler, der überprüft wird.
163      * @return Wahr, wenn der Spieler gewonnen hat, sonst falsch.
164      */
165     private boolean hasWon(Player player) {
166         Player symbol = (player == Player.HUMAN) ? HUMAN : COMPUTER;
167         String leftSymbol = "";
168         String rightSymbol = "";
169 
170         String containsSymbol = player == HUMAN
171                 ? HUMAN.toString() + HUMAN.toString() + HUMAN.toString() + HUMAN.toString
                        ()
172                 : COMPUTER.toString() + COMPUTER.toString() + COMPUTER.toString() +
                        COMPUTER.toString();
173 
174         // Horizontal und vertikal prüfen
175         for (int row = 0; row < field.length; row++) {
176             for (int col = 0; col < field.length; col++) {
177                 leftSymbol += field[row][col];
178                 rightSymbol += field[col][row];
179             }
180             if (leftSymbol.contains(containsSymbol) || rightSymbol.contains(
                        containsSymbol))
181                 return true;
182             leftSymbol = "";
183             rightSymbol = "";
184         }
185 
186         // Diagonale Überprüfung (von links nach rechts und von rechts nach links) -
                Erste Hälfte
187         int curRow = 0;
188         int curCol = 0;
189         int lastCol = field.length - 1;
190 
191         for (int col = 0; col < field.length; col++) {
192             leftSymbol = "";
193             rightSymbol = "";
194 
195             curRow = 0;
196             curCol = col;
197 
198             while (curRow < field.length - col) {
199                 leftSymbol += field[curRow][curCol].toString();
200                 rightSymbol += field[curRow][lastCol - curCol].toString();
201 
202                 curRow++;
203                 curCol++;
204             }
205             if (leftSymbol.contains(containsSymbol) || rightSymbol.contains(
                        containsSymbol))
206                 return true;
207             if (leftSymbol.length() <= 4 && rightSymbol.length() <= 4)
208                 break;
209         }
210 
211         for (int row = 1; row < field.length; row++) {
212             leftSymbol = "";
213             rightSymbol = "";
214 
215             curRow = row;
216             curCol = 0;
217 
218             while (curCol < field.length - row) {
219                 leftSymbol += field[curRow][curCol].toString();
```

```
220                 rightSymbol += field[curRow][lastCol - curCol].toString();
221
222                 curRow++;
223                 curCol++;
224             }
225             if (leftSymbol.contains(containsSymbol) || rightSymbol.contains(
                        containsSymbol))
226                 return true;
227             if (leftSymbol.length() <= 4 && rightSymbol.length() <= 4)
228                 break;
229         }
230         return false;
231     }
```

Die `getGameResult` Methode überprüft den aktuellen Spielstatus, um das Ergebnis des Spiels zu bestimmen. Zuerst wird überprüft, ob der angegebene Spieler (`player`) das Spiel gewonnen hat, indem die `hasWon` Methode aufgerufen wird. Wenn dies der Fall ist, wird das entsprechende Ergebnis zurückgegeben: `HUMAN_WON`, wenn der Spieler ein Mensch ist, andernfalls `COMPUTER_WON`.

Wenn kein Spieler gewonnen hat, wird überprüft, ob es noch mögliche Züge auf dem Spielfeld gibt. Dazu wird die Liste der möglichen Züge für leere Positionen (`EMPTY`) überprüft. Wenn diese Liste leer ist (`possibleMoves(EMPTY).size() == 0`), bedeutet dies, dass das Spiel unentschieden endet, und `DRAW` wird zurückgegeben.

Wenn weder ein Spieler gewonnen hat noch das Spiel unentschieden ist, wird `CONTINUE` zurückgegeben, was bedeutet, dass das Spiel noch weitergeht und kein endgültiges Ergebnis vorliegt.

```
233     /**
234      * Gibt das Ergebnis des Spiels zurück.
235      * Diese Methode überprüft den aktuellen Spielstatus, um festzustellen, ob ein
                Spieler gewonnen hat,
236      * das Spiel unentschieden ist oder das Spiel weitergeht.
237      *
238      * @param player Der Spieler, der überprüft wird.
239      * @return Das Ergebnis des Spiels.
240      */
241     public Result getGameResult(Player player) {
242         if (hasWon(player))
243             return player == HUMAN ? HUMAN_WON : COMPUTER_WON;
244         if (possibleMoves(EMPTY).size() == 0)
245             return DRAW;
246         return CONTINUE;
247     }
```

Die Methode `negaMax` implementiert den NegaMax-Algorithmus, um den besten Zug für den angegebenen Spieler zu finden. Der NegaMax-Algorithmus ist eine Variante des Minimax-Algorithmus und wird häufig in Spielalgorithmen verwendet, um den bestmöglichen Zug zu ermitteln.

Zu Beginn der Methode wird überprüft, ob die maximale Suchtiefe `height` erreicht ist. Wenn dies der Fall ist, wird eine Bewertung des aktuellen Spielzustands zurückgegeben, die durch die `evaluate` Methode für den angegebenen Spieler berechnet wird.

Ansonsten wird `bestMove` initialisiert, um den besten Zug mit einer sehr niedrigen Bewertung (negative Unendlichkeit) zu beginnen.

Die Methode durchläuft alle möglichen Züge, die noch auf dem Spielfeld gemacht werden können (leere Felder). Für jeden Zug wird `makeMove` aufgerufen, um das Spieler-Symbol an der entsprechenden Position zu setzen. Dann wird rekursiv der Wert des aktuellen Zugs durch Aufruf von `negaMax` für den anderen Spieler und mit verringertem `height` berechnet.

Wenn der Spieler durch den aktuellen Zug gewinnt (`hasWon(player)`), wird die Punktzahl mit `goodScore(height)` positiv bewertet, andernfalls wird der Wert des negierten rekursiven Aufrufs verwendet (`-negaMax(other(player), height - 1).getScore()`).

Der beste Zug wird ausgewählt, indem der höchste Punktwert (`score`) zwischen den

möglichen Zügen gesucht wird. Dies wird durch einfache Maximierung erreicht (`score > bestMove.getScore()`).

Nach der Bewertung wird der Zug mit `undoMove` rückgängig gemacht, um das Spielfeld in den vorherigen Zustand zurückzusetzen, bevor der Zug gemacht wurde.

Schließlich wird der beste Zug zurückgegeben, der den höchsten Bewertungswert (Punktzahl) für den Spieler repräsentiert.

```
249     /**
250      * Implementiert den NegaMax-Algorithmus.
251      * Diese Methode implementiert den NegaMax-Algorithmus, um den besten Zug für den
                angegebenen Spieler zu finden.
252      * Der Algorithmus durchsucht alle möglichen Züge bis zu einer bestimmten Tiefe und
                bewertet die Züge,
253      * um den Zug mit der höchsten Bewertung zurückzugeben.
254      *
255      * @param player Der Spieler, der den Zug macht.
256      * @param height Die aktuelle Tiefe.
257      * @return Der beste Zug.
258      */
259     public Move negaMax(final Player player, int height) {
260         if (height == 0) {
261             return new Move(evaluate(player));
262         }
263 
264         Move bestMove = new Move(-Integer.MAX_VALUE);
265 
266         for (final Move move : possibleMoves(EMPTY)) {
267             makeMove(move, player);
268 
269             final int score = hasWon(player)
270                     ? goodScore(height)
271                     : -negaMax(other(player), height - 1).getScore();
272 
273             if (score > bestMove.getScore()) {
274                 bestMove = new Move(move.getRow(), move.getColumn(), score);
275             }
276 
277             undoMove(move);
278         }
279         return bestMove;
280     }
```

Bonusaufgabe

Die `negaMaxAlphaBeta` Methode implementiert den NegaMax-Algorithmus mit Alpha-Beta-Pruning, um den besten Zug für den angegebenen Spieler zu finden. Alpha-Beta-Pruning ist eine Erweiterung des Minimax-Algorithmus und verbessert die Effizienz, indem unnötige Berechnungen reduziert werden.

Zu Beginn der Methode wird überprüft, ob die maximale Suchtiefe `height` erreicht ist. Wenn dies der Fall ist, wird eine Bewertung des aktuellen Spielzustands durch die `evaluate` Methode für den angegebenen Spieler berechnet und zurückgegeben.

Ansonsten wird `bestMove` initialisiert, um den besten Zug mit einer sehr niedrigen Bewertung (negative Unendlichkeit) zu beginnen.

Die Methode durchläuft alle möglichen Züge, die noch auf dem Spielfeld gemacht werden können (leere Felder). Für jeden Zug wird `makeMove` aufgerufen, um das Spieler-Symbol an der entsprechenden Position zu setzen. Dann wird rekursiv der Wert des aktuellen Zugs durch Aufruf von `negaMaxAlphaBeta` für den anderen Spieler und mit verringertem `height` berechnet.

Wenn der Spieler durch den aktuellen Zug gewinnt (`hasWon(player)`), wird die Punktzahl mit `goodScore(height)` positiv bewertet, andernfalls wird der Wert des negier-

ten rekursiven Aufrufs verwendet (`-negaMaxAlphaBeta(other(player), height - 1, -beta, -alpha).getScore()`).

Der beste Zug wird ausgewählt, indem der höchste Punktwert (`score`) zwischen den möglichen Zügen gesucht wird. Dies wird durch einfache Maximierung erreicht (`score > bestMove.getScore()`).

Nach der Bewertung wird der Zug mit `undoMove` rückgängig gemacht, um das Spielfeld in den vorherigen Zustand zurückzusetzen, bevor der Zug gemacht wurde.

Während des Durchlaufs erfolgt das Alpha-Beta-Pruning, indem die Werte von `alpha` und `beta` aktualisiert werden. Der `alpha`-Wert wird auf den maximalen Wert zwischen `alpha` und `score` gesetzt. Wenn `alpha` größer oder gleich `beta` ist, wird die Schleife unterbrochen, da keine weiteren Züge mehr bewertet werden müssen.

```
282     /**
283      * Implementiert den NegaMax-Algorithmus mit Alpha-Beta-Schnitt.
284      * Diese Methode implementiert den NegaMax-Algorithmus mit Alpha-Beta-Pruning, um den
                besten Zug für den angegebenen Spieler
285      * zu finden. Der Algorithmus durchsucht alle möglichen Züge bis zu einer bestimmten
               Tiefe und verwendet Alpha-Beta-Pruning,
286      * um die Anzahl der zu bewertenden Züge zu reduzieren und den effizientesten Zug zu
               finden.
287      *
288      * @param player Der Spieler, der den Zug macht.
289      * @param height Die aktuelle Tiefe.
290      * @param alpha Der Alpha-Wert für das Pruning.
291      * @param beta Der Beta-Wert für das Pruning.
292      * @return Der beste Zug.
293      */
294     public Move negaMaxAlphaBeta(final Player player, int height, int alpha, int beta) {
295         if (height == 0)
296             return new Move(evaluate(player));
297 
298         Move bestMove = new Move(-Integer.MAX_VALUE);
299 
300         for (final Move move : possibleMoves(EMPTY)) {
301             makeMove(move, player);
302 
303             final int score = hasWon(player)
304                     ? goodScore(height)
305                     : -negaMaxAlphaBeta(other(player), height - 1, -beta, -alpha).
                               getScore();
306 
307             if (score > bestMove.getScore()) {
308                 bestMove = new Move(move.getRow(), move.getColumn(), score);
309             }
310 
311             undoMove(move);
312 
313             // Alpha-Beta-Pruning
314             alpha = Math.max(alpha, score);
315             if (alpha >= beta) {
316                 break;
317             }
318         }
319         return bestMove;
320     }
321 }
```

Aufgabe 1.2 Tests

FourInARowTest.java

Die FourInARowTest Klasse wurde von **Altug Uyanik** verfasst.

Die Methode `asField` nimmt eine Zeichenketten-Eingabe und wandelt sie in ein zweidimensionales Spielfeld-Array um. Diese Methode verarbeitet die Eingabe zeilenweise und spaltenweise und füllt jede Zelle mit dem entsprechenden Spieler- oder Leerwert. Zunächst wird die Eingabezeichenkette durch das Zeichen `\n`, das die Zeilen darstellt, getrennt, und jede Zeile wird als ein Array-Element übernommen. Dann wird eine Schleife über jede dieser Zeilen gelegt und jedes Zeichen überprüft. Das Zeichen `'X'` stellt den menschlichen Spieler dar und wird als `HUMAN` markiert. Das Zeichen `'O'` stellt den Computer dar und wird als `COMPUTER` markiert. Alle anderen Zeichen, also `'.'`, stellen leere Zellen dar und werden als `EMPTY` markiert. Als Ergebnis wird die Zeichenketten-Eingabe in ein zweidimensionales Array umgewandelt, das mit den entsprechenden `Player` Werten gefüllt ist, und zurückgegeben. Diese Methode nimmt die Zeichenketten-Darstellung des Spielfelds und wandelt sie in eine Datenstruktur um, die das Programm verwenden kann.

```
13 public class FourInARowTest {
14
15     /**
16      * Erzeuge ein Spielfeld aus einem String. Zeilen werden durch '\n' getrennt,
17      * wobei am Ende kein '\n' steht. '.' repräsentiert leeres Feld, 'X' Steine
18      * der menschlichen Spieler*in und 'O' Steine des Computers.
19      *
20      * @param string Eine Zeichenkette in demselben Format, in dem auch die
21      *               toString-Methode des Spiels das Spielfeld darstellt.
22      * @return Ein Spielfeld mit den zur Eingabe passenden Belegungen.
23      */
24     private Player[][] asField(final String string) {
25         String[] rows = string.split("\n");
26         Player[][] field = new Player[rows.length][rows[0].length()];
27
28         for (int i = 0; i < rows.length; i++) {
29             for (int j = 0; j < rows[i].length(); j++) {
30                 switch (rows[i].charAt(j)) {
31                     case 'X':
32                         field[i][j] = HUMAN;
33                         break;
34                     case 'O':
35                         field[i][j] = COMPUTER;
36                         break;
37                     default:
38                         field[i][j] = EMPTY;
39                         break;
40                 }
41             }
42         }
43         return field;
44     }
```

testHumanMove

Diese Methode testet den Zug des menschlichen Spielers. Das Spielfeld wird initialisiert und ein Zug des menschlichen Spielers (HUMAN) wird durchgeführt. Danach wird überprüft, ob das Spielfeld korrekt aktualisiert wurde, indem geprüft wird, ob das entsprechende Feld tatsächlich den Wert `HUMAN` enthält.

```
46     /**
47      * Testet den Zug des menschlichen Spielers.
48      * Überprüft, ob das Feld nach dem Zug des Menschen korrekt aktualisiert wird.
49      */
50     @Test
51     void testHumanMove() {
52         Player[][] field = asField(
53                 "....\n" +
54                         "....\n" +
55                         "....\n" +
56                         "...."
57         );
58         FourInARow game = new FourInARow(field, 4);
```

```
59          game.humanMove(0, 0);
60          assertEquals(HUMAN, field[0][0]);
61      }
```

testComputerMove

Diese Methode testet den Zug des Computers. Das Spielfeld wird initialisiert und ein Zug des Computers wird durchgeführt. Danach wird überprüft, ob das Spiel korrekt fortgesetzt wird, indem der Rückgabewert der Methode `computerMove` überprüft wird. Es wird erwartet, dass das Spiel mit `CONTINUE` fortgesetzt wird.

```
63      /**
64       * Testet den Zug des Computers.
65       * Überprüft, ob das Spiel nach dem Zug des Computers korrekt fortgesetzt wird.
66       */
67      @Test
68      void testComputerMove() {
69          Player[][] field = asField(
70                  "X...\n" +
71                          "....\n" +
72                          "....\n" +
73                          "...."
74          );
75          FourInARow game = new FourInARow(field, 4);
76          assertEquals(CONTINUE, game.computerMove());
77      }
```

testHumanWinHorizontal

Diese Methode testet, ob der menschliche Spieler das Spiel gewinnt, wenn er vier Steine horizontal in einer Reihe hat. Das Spielfeld wird initialisiert und ein weiterer Zug des menschlichen Spielers wird durchgeführt. Danach wird überprüft, ob der Spieler gewonnen hat, indem der Rückgabewert der Methode `humanMove` überprüft wird.

```
79      /**
80       * Testet, ob der Mensch horizontal gewinnt.
81       * Überprüft, ob der Mensch das Spiel gewinnt, wenn er vier Steine horizontal in
                 einer Reihe hat.
82       */
83      @Test
84      void testHumanWinHorizontal() {
85          Player[][] field = asField(
86                  "OOO.\n" +
87                          "....\n" +
88                          "XXX.\n" +
89                          "...."
90          );
91          FourInARow game = new FourInARow(field, 4);
92          assertEquals(HUMAN_WON, game.humanMove(2, 3));
93      }
```

testHumanWinVertical

Diese Methode testet, ob der menschliche Spieler das Spiel gewinnt, wenn er vier Steine vertikal in einer Reihe hat. Das Spielfeld wird initialisiert und ein weiterer Zug des menschlichen Spielers wird durchgeführt. Danach wird überprüft, ob der Spieler gewonnen hat, indem der Rückgabewert der Methode `humanMove` überprüft wird.

```
95      /**
96       * Testet, ob der Mensch vertikal gewinnt.
97       * Überprüft, ob der Mensch das Spiel gewinnt, wenn er vier Steine vertikal in einer
                 Reihe hat.
```

```
98       */
99      @Test
100     void testHumanWinVertical() {
101         Player[][] field = asField(
102                 "OOOX\n" +
103                         "...X\n" +
104                         "OXOX\n" +
105                         "...."
106         );
107         FourInARow game = new FourInARow(field, 4);
108         assertEquals(HUMAN_WON, game.humanMove(3, 3));
109     }
```

testHumanWinDiagonal

Diese Methode testet, ob der menschliche Spieler das Spiel gewinnt, wenn er vier Steine diagonal in einer Reihe hat. Das Spielfeld wird initialisiert und ein weiterer Zug des menschlichen Spielers wird durchgeführt. Danach wird überprüft, ob der Spieler gewonnen hat, indem der Rückgabewert der Methode `humanMove` überprüft wird.

```
111     /**
112      * Testet, ob der Mensch diagonal gewinnt.
113      * Überprüft, ob der Mensch das Spiel gewinnt, wenn er vier Steine diagonal in einer
               Reihe hat.
114      */
115     @Test
116     void testHumanWinDiagonal() {
117         Player[][] field = asField(
118                 "OOOX\n" +
119                         "..X.\n" +
120                         "OXO.\n" +
121                         "...."
122         );
123         FourInARow game = new FourInARow(field, 4);
124         assertEquals(HUMAN_WON, game.humanMove(3,0 ));
125     }
```

testComputerWinHorizontal

Diese Methode testet, ob der Computer das Spiel gewinnt, wenn er vier Steine horizontal in einer Reihe hat. Das Spielfeld wird initialisiert und ein Zug des Computers wird durchgeführt. Danach wird überprüft, ob der Computer gewonnen hat, indem der Rückgabewert der Methode `computerMove` überprüft wird.

```
127     /**
128      * Testet, ob der Computer horizontal gewinnt.
129      * Überprüft, ob der Computer das Spiel gewinnt, wenn er vier Steine horizontal in
               einer Reihe hat.
130      */
131     @Test
132     void testComputerWinHorizontal() {
133         Player[][] field = asField(
134                 "XX..\n" +
135                         "X..X\n" +
136                         "OOOO\n" +
137                         "...."
138         );
139         FourInARow game = new FourInARow(field, 4);
140         assertEquals(COMPUTER_WON, game.computerMove());
141     }
```

testComputerWinVertical

Diese Methode testet, ob der Computer das Spiel gewinnt, wenn er vier Steine vertikal in einer Reihe hat. Das Spielfeld wird initialisiert und ein Zug des Computers wird durchgeführt. Danach wird überprüft, ob der Computer gewonnen hat, indem der Rückgabewert der Methode `computerMove` überprüft wird.

```
143     /**
144      * Testet, ob der Computer vertikal gewinnt.
145      * Überprüft, ob der Computer das Spiel gewinnt, wenn er vier Steine vertikal in
                einer Reihe hat.
146      */
147     @Test
148     void testComputerWinVertical() {
149         Player[][] field = asField(
150                 "OX..\n" +
151                         "O..X\n" +
152                         "OXOX\n" +
153                         "O..."
154         );
155         FourInARow game = new FourInARow(field, 4);
156         assertEquals(COMPUTER_WON, game.computerMove());
157     }
```

testComputerWinDiagonal

Diese Methode testet, ob der Computer das Spiel gewinnt, wenn er vier Steine diagonal in einer Reihe hat. Das Spielfeld wird initialisiert und ein Zug des Computers wird durchgeführt. Danach wird überprüft, ob der Computer gewonnen hat, indem der Rückgabewert der Methode `computerMove` überprüft wird.

```
159     /**
160      * Testet, ob der Computer diagonal gewinnt.
161      * Überprüft, ob der Computer das Spiel gewinnt, wenn er vier Steine diagonal in
                einer Reihe hat.
162      */
163     @Test
164     void testComputerWinDiagonal() {
165         Player[][] field = asField(
166                 "XX.O\n" +
167                         "X.OX\n" +
168                         ".O..\n" +
169                         "O..."
170         );
171         FourInARow game = new FourInARow(field, 4);
172         assertEquals(COMPUTER_WON, game.computerMove());
173     }
```

testDraw

Diese Methode testet, ob das Spiel unentschieden endet. Das Spielfeld wird initialisiert, sodass keine weiteren Züge mehr möglich sind und kein Spieler gewonnen hat. Danach wird überprüft, ob das Spiel als unentschieden erkannt wird, indem der Rückgabewert der Methode `getGameResult` überprüft wird.

```
175     /**
176      * Testet, ob das Spiel unentschieden endet.
177      * Überprüft, ob das Spiel korrekt als Unentschieden erkannt wird.
178      */
179     @Test
180     void testDraw() {
181         Player[][] field = asField(
182                 "XXXO\n" +
183                         "OOOX\n" +
```

```
                        "XXXO\n" +
                        "OOOX"
        );
        FourInARow game = new FourInARow(field, 4);
        assertEquals(DRAW, game.getGameResult(HUMAN));
    }
```

testToString

Diese Methode testet die `toString`-Methode des Spiels. Das Spielfeld wird initialisiert und die `toString`-Methode wird aufgerufen. Danach wird überprüft, ob die Rückgabe der `toString`-Methode mit der erwarteten Zeichenketten-Darstellung des Spielfelds übereinstimmt.

```
    /**
     * Testet die toString-Methode.
     * Überprüft, ob die toString-Methode das Spielfeld korrekt darstellt.
     */
    @Test
    void testToString() {
        Player[][] field = asField(
                "XOXO\n" +
                        "OXOX\n" +
                        "XOXO\n" +
                        ".XO."
        );
        FourInARow game = new FourInARow(field, 4);
        assertEquals("[X][O][X][O]\n[O][X][O][X]\n[X][O][X][O]\n[.][X][O][.]", game.
                toString());
    }
```

testEvaluate

Diese Methode testet die Bewertungsfunktion des Spiels. Das Spielfeld wird initialisiert und die `evaluate`-Methode wird aufgerufen. Danach wird überprüft, ob die Rückgabe der `evaluate`-Methode mit dem erwarteten Wert übereinstimmt, abhängig davon, ob ein Spieler gewonnen hat oder nicht.

```
     * Testet die Bewertungsfunktion.
     * Überprüft, ob die evaluate-Methode den korrekten Wert für den gegebenen
            Spielzustand zurückgibt.
     */
    @Test
    void testEvaluate() {
        Player[][] field = asField(
                "OOO.\n" +
                        "....\n" +
                        "XXXX\n" +
                        "...."
        );
        FourInARow game = new FourInARow(field, 4);
        assertEquals(Integer.MAX_VALUE, game.evaluate(HUMAN));
    }
```

testPossibleMoves

Diese Methode testet die Methode, die die möglichen Züge zurückgibt. Das Spielfeld wird initialisiert und die Methode `possibleMoves` wird aufgerufen. Danach wird überprüft, ob die Anzahl der zurückgegebenen möglichen Züge korrekt ist.

```
    /**
     * Testet die Methode für mögliche Züge.
     * Überprüft, ob die possibleMoves-Methode die korrekte Anzahl von möglichen Zügen
            zurückgibt.
```

```
226      */
227     @Test
228     void testPossibleMoves() {
229         Player[][] field = asField(
230                 "....\n" +
231                         "....\n" +
232                         "....\n" +
233                         "...."
234         );
235         FourInARow game = new FourInARow(field, 4);
236         assertEquals(16, game.possibleMoves(EMPTY).size());
237     }
```

testMakeMove

Diese Methode testet das Ausführen eines Zugs. Das Spielfeld wird initialisiert und ein Zug wird mit der Methode `makeMove` durchgeführt. Danach wird überprüft, ob das Spielfeld korrekt aktualisiert wurde, indem geprüft wird, ob das entsprechende Feld tatsächlich den Wert `HUMAN` enthält.

```
239     /**
240      * Testet das Ausführen eines Zugs.
241      * Überprüft, ob die makeMove-Methode den Zug korrekt ausführt.
242      */
243     @Test
244     void testMakeMove() {
245         Player[][] field = asField(
246                 "....\n" +
247                         "....\n" +
248                         "....\n" +
249                         "...."
250         );
251         FourInARow game = new FourInARow(field, 4);
252         Move move = new Move(0, 0, 0);
253         game.makeMove(move, HUMAN);
254         assertEquals(HUMAN, field[0][0]);
255     }
```

testUndoMove

Diese Methode testet das Zurücknehmen eines Zugs. Das Spielfeld wird initialisiert und ein Zug wird durchgeführt. Danach wird der Zug mit der Methode `undoMove` zurückgenommen und überprüft, ob das Spielfeld korrekt aktualisiert wurde, indem geprüft wird, ob das entsprechende Feld wieder den Wert `EMPTY` enthält.

```
257     /**
258      * Testet das Zurücknehmen eines Zugs.
259      * Überprüft, ob die undoMove-Methode den Zug korrekt zurücknimmt.
260      */
261     @Test
262     void testUndoMove() {
263         Player[][] field = asField(
264                 "X...\n" +
265                         "....\n" +
266                         "....\n" +
267                         "...."
268         );
269         FourInARow game = new FourInARow(field, 4);
270         Move move = new Move(0, 0, 0);
271         game.undoMove(move);
272         assertEquals(EMPTY, field[0][0]);
273     }
```

testOtherComputer

Diese Methode testet die Methode, die den gegnerischen Spieler zurückgibt. Es wird überprüft, ob die Methode `other` für den menschlichen Spieler (HUMAN) den Computer (COMPUTER) zurückgibt.

```
275     /**
276      * Testet die Methode other für den Computer.
277      * Überprüft, ob die other-Methode den korrekten gegnerischen Spieler zurückgibt.
278      */
279     @Test
280     void testOtherComputer() {
281         FourInARow game = new FourInARow(new Player[4][4], 4);
282         assertEquals(COMPUTER, game.other(HUMAN));
283     }
```

testOtherHuman

Diese Methode testet die Methode, die den gegnerischen Spieler zurückgibt. Es wird überprüft, ob die Methode `other` für den Computer (COMPUTER) den menschlichen Spieler (HUMAN) zurückgibt.

```
285     /**
286      * Testet die Methode other für den Menschen.
287      * Überprüft, ob die other-Methode den korrekten gegnerischen Spieler zurückgibt.
288      */
289     @Test
290     void testOtherHuman() {
291         FourInARow game = new FourInARow(new Player[6][6], 6);
292         assertEquals(HUMAN, game.other(COMPUTER));
293     }
```

testGoodScore

Diese Methode testet die Bewertungsfunktion für die Tiefe des Minimax-Algorithmus. Es wird überprüft, ob die Methode `goodScore` den korrekten Wert für die gegebene Tiefe zurückgibt.

```
295     /**
296      * Testet die goodScore-Methode.
297      * Überprüft, ob die goodScore-Methode den korrekten Wert für die gegebene Höhe zurü
               ckgibt.
298      */
299     @Test
300     void testGoodScore() {
301         FourInARow game = new FourInARow(new Player[6][6], 6);
302         assertEquals(60, game.goodScore(6));
303         assertEquals(50, game.goodScore(5));
304         assertEquals(40, game.goodScore(4));
305         assertEquals(30, game.goodScore(3));
306         assertEquals(20, game.goodScore(2));
307         assertEquals(10, game.goodScore(1));
308     }
```

testGetGameResultHumanWon

Diese Methode testet das Spielergebnis, wenn der menschliche Spieler gewonnen hat. Das Spielfeld wird initialisiert und die Methode `getGameResult` wird aufgerufen. Danach wird überprüft, ob das Ergebnis korrekt als `HUMAN_WON` erkannt wird.

```
310     /**
311      * Testet das Spielergebnis, wenn der Mensch gewinnt.
312      * Überprüft, ob getGameResult den korrekten Wert zurückgibt, wenn der Mensch gewinnt
               .
```

```
313      */
314     @Test
315     void testGetGameResultHumanWon() {
316         Player[][] field = asField(
317                 "XXXX\n" +
318                         "OOOX\n" +
319                         "OOXX\n" +
320                         "XXOO"
321         );
322         FourInARow game = new FourInARow(field, 4);
323         assertEquals(HUMAN_WON, game.getGameResult(HUMAN));
324     }
```

testGetGameResultComputerWon

Diese Methode testet das Spielergebnis, wenn der Computer gewonnen hat. Das Spielfeld wird initialisiert und die Methode `getGameResult` wird aufgerufen. Danach wird überprüft, ob das Ergebnis korrekt als `COMPUTER_WON` erkannt wird.

```
326     /**
327      * Testet das Spielergebnis, wenn der Computer gewinnt.
328      * Überprüft, ob getGameResult den korrekten Wert zurückgibt, wenn der Computer
             gewinnt.
329      */
330     @Test
331     void testGetGameResultComputerWon() {
332         Player[][] field = asField(
333                 "OOOO\n" +
334                         "XXXO\n" +
335                         "XXOO\n" +
336                         "OOXX"
337         );
338         FourInARow game = new FourInARow(field, 4);
339         assertEquals(COMPUTER_WON, game.getGameResult(COMPUTER));
340     }
```

testGetGameResultDraw

Diese Methode testet das Spielergebnis bei einem Unentschieden. Das Spielfeld wird initialisiert und die Methode `getGameResult` wird aufgerufen. Danach wird überprüft, ob das Ergebnis korrekt als `DRAW` erkannt wird.

```
342     /**
343      * Testet das Spielergebnis bei einem Unentschieden.
344      * Überprüft, ob getGameResult den korrekten Wert für ein Unentschieden zurückgibt.
345      */
346     @Test
347     void testGetGameResultDraw() {
348         Player[][] field = asField(
349                 "XOXO\n" +
350                         "XXXO\n" +
351                         "XXOO\n" +
352                         "OOXX"
353         );
354         FourInARow game = new FourInARow(field, 4);
355         assertEquals(DRAW, game.getGameResult(HUMAN));
356     }
```

testNegaMaxAlphaBeta

Diese Methode testet den NegaMax-Algorithmus mit Alpha-Beta-Schnitt. Das Spielfeld wird initialisiert und die Methode `negaMax` wird aufgerufen. Danach wird überprüft, ob der Algorithmus den besten Zug korrekt berechnet hat, indem geprüft wird, ob der beste Zug nicht null ist.

```
358     /**
359      * Testet den NegaMax-Algorithmus.
360      * Überprüft, ob der NegaMax-Algorithmus den besten Zug korrekt berechnet.
361      */
362     @Test
363     void testNegaMaxAlphaBeta() {
364         Player[][] field = asField(
365                 "....\n" +
366                         "....\n" +
367                         "....\n" +
368                         "...."
369         );
370         FourInARow game = new FourInARow(field, 4);
371         Move bestMove = game.negaMax(COMPUTER, 4);
372         assertNotNull(bestMove);
373     }
374 }
```