Groker

Aufgabe 5, Runde 1, 34. Bundeswettbewerb Informatik

Marian Dietz, Johannes Heinrich, Erik Fritzsche

1 Lösungsidee

Der Algorithmus basiert darauf, die vergangenen Züge zu analysieren und Zugfolgen zu suchen, die ähnlich zur aktuellen Zugfolge sind. Die Reaktionen des Gegners auf diese Zugfolgen werden dann überprüft und unter bestimmten Bedingungen wird angenommen, dass der Gegner wieder dieselbe Anzahl von Chips setzen wird. Daraus kann dann die optimale Zahl berechnet werden, die die eigene KI legen sollte, damit sie möglichst viele Punkte bekommt.

Im Folgenden werden Abfolgen von Zügen *Situationen* genannt. Die Anzahl der vergangenen Züge ist n. Es existieren die Züge mit den Nummern von 0 bis $n-1$. Die Situation i mit der Länge j ist die Situation, deren Zugfolge bei Zug Nummer $i-j$ beginnt und bei Zug Nummer $i-1$ aufhört. Die *aktuelle Situation* bezeichnet jede Situation, welche beim letzten gesetzten Zug aufhört. e_i bezeichnet die Anzahl der Chips, die die eigene KI im Zug i gelegt hat, g_i ist die Anzahl der Chips, die der Gegner im Zug i gelegt hat.

1.1 Ähnliche Situationen finden

Es werden zunächst zur Berechnung eines Zuges Situationen gesucht, die ähnlich zur aktuellen Situation sind. Bei einer ähnlichen Situation stimmen alle gesetzten Zahlen der ähnlichen mit denen der aktuellen Situation nahezu überein. Jede der gesetzten Zahlen darf sich um eins von der aktuellen Situation unterscheiden.

Um ähnliche Situationen zu finden, müssen bereits mindestens zwei Züge existieren, denn zu ähnlichen Situationen, die gefunden werden, wird immer noch die Reaktion des Gegners danach benötigt, da die Situation sonst nichts aussagt. Gefundene ähnliche Situationen müssen daher *vor* dem letzten Zug aufhören.

i ist der Index des Zuges, mit dem der Gegner auf die Situationen i reagiert. Es wird zur Suche jedes i von 1 bis n getestet. Für jedes i wird gestartet bei einer Situation der Länge $j = 1$. Jetzt wird der Unterschied der Situation i mit der Länge j zur aktuellen Situation berechnet. Dies ist zum einen $d_e = e_{i-j} - e_{n-j}$ für den Unterschied des eigenen Zuges und $d_g = g_{i-j} - g_{n-j}$ für den Unterschied der gegnerischen Züge der Situation i mit der Länge j zur aktuellen Situation. Die Situation i mit der Länge j ist genau dann ähnlich, wenn $|d_e| \leq 1$ und $|g_e| \leq 1$. D. h., der Unterschied zwischen den Zügen der

vergangenen (ähnlichen) Situation darf zwar existieren, jedoch für jeden Zug von jedem Spieler maximal eins sein.

Ist diese Überprüfung abgeschlossen und ist die Situation i zur aktuellen Situation ähnlich, so wird die Länge j der Situation i um eins erhöht, damit die nächstlängere Situation, also mit der Länge $j+1$ auch auf Ähnlichkeit überprüft werden kann. Da bereits im ersten Durchgang der Zug mit dem Index $i-j$ erfolgreich auf Ähnlichkeit getestet wurde, muss nur ein weiterer Zug ebenfalls getestet werden, und zwar $i-(j+1)$. Es werden wieder d_e und d_g ausgerechnet, diesmal mit einem um eins größeren j. Sind auch diese für die Ähnlichkeit klein genug, so ist die Situation i mit der Länge des jetzigen j zur aktuellen Situation ähnlich. Im nächsten Zug wird wieder j um eins erhöht usw., bis irgendwann die Bedingung zur Ähnlichkeit nicht mehr gegeben ist oder wenn $j > i$.

Da der Gesamtunterschied der längsten ähnlichen Situation i später benötigt wird, muss dieser gespeichert werden. Immer, nachdem die Unterschiede erfolgreich getestet wurden, werden zu diesem Gesamtunterschied d (zu Anfang $d = 0$) $|d_e|$ sowie $|d_g|$ hinzuaddiert.

Außerdem wird später der Unterschied benötigt, welcher die Vorzeichen der Unterschiede beachtet. Dies ist d_s. Wie d muss auch d_s (zu Beginn $d_s = 0$) gespeichert werden, zu dem nach jedem erfolgreichem Test d_e sowie d_g hinzuaddiert werden.

Die Länge der längsten Situation i, welche gefunden wurde, ist m.

Nachdem die Abbruchbedingung erreicht wurde, wird die durchschnittliche Abweichung der Züge der Situation i zur aktuellen Situation ausgerechnet, sofern eine ähnliche Situation i gefunden wurde: $d_{avg} = \frac{d_s}{m \cdot 2}$. Dieser Wert wird nach den Rundungsregeln auf einen ganzzahligen Wert gerundet. Die vorhergesagte Reaktion des Gegners auf die aktuelle Situation ist jetzt $r = max(1, g_i - d_{avg})$. Es wird max benötigt, da mindestens ein Chip gelegt werden muss. Zur Berechnung von r muss der Wert d_{avg} von g_i subtrahiert, statt addiert werden, da der Unterschied, wenn die ähnliche Situation größere Werte als die aktuelle Situation beinhaltet, positiv ist. In diesem Fall sind die Werte in der aktuellen Situation also kleiner und dementsprechend muss auch die vorhergesagte Reaktion des Gegners kleiner sein.

Ein Beispiel dafür: eine ähnliche Situation der Länge zwei wurde gefunden, die einzelnen Unterschiede zur aktuellen Situation sind 0, 1, -1 und 0. Daher ist $d_s = 0$ und $d_{avg} = 0$. Die vorhergesagte Reaktion des Gegners ist dann g_i. Sind die einzelnen Unterschiede jedoch 1, 0, 1 und 1, dann ist $d_s = 3$ und $d_{avg} = 0,75$ bzw. gerundet 1. Daher wäre in diesem Fall die vorhergesagte Reaktion des Gegners $g_i - 1$.

Immer, wenn die Reaktion des Gegners r berechnet wurde, kann die ähnliche Situation einer Liste hinzugefügt werden. Jede Situation besteht aus der Länge, dem Unterschied zur aktuellen Situation, der „Zeit“, wann diese Situation stattfand und der vorhergesagten Reaktion des Gegners. Daher ist die ähnliche Situation (m, d, i, r).

Nachdem die ähnliche Situation, falls vorhanden, hinzugefügt wurde, können die nächsten Situationen überprüft werden, indem i um eins erhöht wird. Dann startet j wieder bei 1, später wird u. U. eine ähnliche Situation zur Liste hinzugefügt usw. Das ganze wird wiederholt, solange $i < n$. Danach ist das Suchen der ähnlichen Situationen abgeschlossen.

Der Pseudocode 1 verdeutlicht diesen Algorithmus.

Algorithm 1 Ähnliche Situationen finden

```
1: function SimilarSituations(e, g)                    ▷ Eigene, Gegnerische Züge
2:     similar ← []                                    ▷ Leere Liste
3:     for i = 1 to n − 1 do
4:         d ← 0
5:         d_s ← 0
6:         m ← −1
7:         for j = 1 to i do
8:             d_e ← e[i − j] − e[n − j]
9:             d_g ← g[i − j] − g[n − j]
10:            if abs(d_e) ≤ 1 and abs(d_g) ≤ 1 then
11:                d ← d + abs(d_e) + abs(d_g)          ▷ Unterschiede klein genug
12:                d_s ← d_s + d_e + d_g
13:                m ← j
14:            else
15:                break                               ▷ Unterschiede zu groß
16:            end if
17:        end for
18:        if m ≠ −1 then
19:            d_avg ← round(d_s/(m · 2))
20:            reaction = max(1, g_i − d_avg)
21:            similar.add(Situation(m, d, i, reaction)) ▷ Situation der Liste hinzufügen
22:        end if
23:    end for
24:    return similar
25: end function
```

Der Algorithmus aus diesem Abschnitt benötigt im schlechtesten Fall $\mathcal{O}(n^2)$ Zeit, da für jedes i höchstens $i-1$ Überprüfungen gemacht werden können, wobei i die Werte von 1 bis $n-1$ annehmen kann.

1.2 Beste zu legende Zahl berechnen

Jetzt soll die beste zu legende Zahl für die eigene KI aus den ähnlichen Situationen berechnet werden. Dafür werden zunächst die zuvor berechneten ähnlichen Situationen nach Priorität sortiert. Welcher Sortieralgorithmus dafür verwendet wird, ist egal, er sollte aber natürlich eine Laufzeit von $\mathcal{O}(n \log n)$ haben. Dabei verliert n nicht seine Bedeutung aus dem vorherigen Abschnitt, da dort höchstens n ähnliche Situationen gefunden werden können (das ist gut im Pseudocode erkennbar). Der Vergleich zweier Situationen funktioniert folgendermaßen: weiter vorne in der sortierten Liste befindet sich die Situation, die die größere Länge hat. Ist diese bei beiden gleich, so wird die Situation mit dem kleineren Unterschied vorgezogen. Sind auch diese identisch, so befindet sich die Situation, die neuer ist, also diejenige, dessen i kleiner ist, vorne.

Danach werden die vorhergesagten Reaktionen des Gegners der *ersten vier* Situationen mit der höchsten Priorität (daher gehören dazu die vier längsten Situationen) angesehen und deren Minimum r_{min} und Maximum r_{max} der vorhergesagten Reaktionen des Gegners bestimmt. Wenn keine ähnliche Situation existiert oder wenn $r_{max} - r_{min} > 2$ (d. h., die vorhergesagten Werte sind zu unterschiedlich), dann kann durch diese Methode nicht die beste zu legende Zahl bestimmt werden und es wird das Verfahren des Abschnittes 1.3 verwendet.

Die beiden nächsten Abschnitte zeigen zwei verschiedene Fälle, das „sichere“ und das „unsichere“ Legen. Der Algorithmus insgesamt, wozu beide Fälle und dieser Abschnitt gehören, benötigt $\mathcal{O}(n \log n)$ Zeit, da jeder Schritt bis auf das Sortieren konstante Zeit benötigt.

1.2.1 Sicheres Legen

Es muss „sicher“ gelegt werden, wenn die Anzahl der ähnlichen Situationen kleiner als vier ist (es gibt nur wenige ähnliche Situationen) oder wenn $r_{max} - r_{min} > 0$, also $r_{max} \neq r_{min}$. Es gibt in diesem Fall unterschiedliche vorhergesagte Werte. Dann wird folgendes getan.

Wenn $r_{min} > 6$, dann wird $max(1, r_{min} - 6 - z)$ gelegt, wobei z zufällig entweder 0 oder 1 ist. Dies wird getan, da es möglich ist, sechs weniger als der Gegner zu legen. Grundsätzlich wird immer bevorzugt, sechs weniger als der Gegner zu legen, denn wenn der Gegner einige Chips weniger als vorhergesagt legt, kann er höchstens fünf mehr als die eigene KI bekommen. Beim sicheren Legen wird außerdem ein weiterer Zufallsaspekt verwendet: für den Fall, dass der Gegner eins weniger als vorhergesagt legt, wird zufällig eins von der gelegten Anzahl abgezogen (dies zeigt in der Gleichung die Variable z).

Wenn $r_{min} \leq 6$, dann kann nicht einfach sechs weniger gelegt werden. Stattdessen wird von der eigenen KI die Zahl $r_{min} + 3 + z$ gelegt, wobei z wieder zufällig entweder 0 oder 1 ist. Daher werden insgesamt 3 oder 4 Chips mehr als die kleinste Anzahl des

Gegners gelegt. Da der Maximalwert des Gegners nur zwei höher als der Minimalwert sein kann, bekommt die KI auch im schlechtesten Fall (z ist 0 und der Gegner legt r_{max}) einen Chip mehr als der Gegner, sofern dieser die Vorhersage nicht bricht.

1.2.2 Unsicheres Legen

Wenn der Fall oben nicht zutrifft, dann kann „unsicher“ gelegt werden. D. h., die Anzahl der ähnlichen Situationen ist mindestens 4 und $r_{min} = r_{max}$. In diesem Fall wird davon ausgegangen, dass der Gegner auf jeden Fall genau die vorhergesagte Zahl legt.

Wenn $r_{min} > 6$, dann wird $r_{min} - 6$ gelegt. Wenn $r_{min} \leq 6$, dann wird $r_{min} + 5$ gelegt. Hier ist die Argumentation für das Sechs-weniger-legen dieselbe wie aus dem vorherigen Abschnitt.

1.3 Zufall verwenden

Konnte durch die Methode oben keine passende Zahl ermittelt werden, so muss zufällig, aber trotzdem auf den Zahlen des Gegners basierend, gelegt werden.

Ist $n = 0$, dann wird gerade der allererste Zug ausgeführt. In diesem Fall wird einfach sechs gelegt.

Ansonsten werden die letzten *sechs* Züge des Gegners ermittelt. Diese werden sortiert (damit Ausreißer entfernt werden können und Maximum und Minimum einfach bestimmt werden können) und es kann, wie im nächsten Abschnitt beschrieben, ein Ausreißer entfernt werden. Dies ist vor allem von Vorteil, wenn der Gegner hin und wieder eine Zahl legt, die entweder sehr klein oder sehr groß ist und die KI so verwirren möchte.

Danach werden das Minimum s_{min} und das Maximum s_{max} der letzten sechs Züge des Gegners ermittelt. Da die Liste sortiert ist, geschieht dies durch einfache Zugriffe auf das erste und auf das letzte Element der Liste.

Ist $s_{min} > 6$, dann legt die KI $min(20, s_{min} - 6)$, also sechs weniger als der Gegner, jedoch maximal 20. Letzteres hat den Sinn, Gegner zu besiegen, welche die Strategie verfolgen, mehrmals sehr hohe Zahlen und irgendwann dann eine niedrigere Zahl zu legen, durch welche sie aber trotzdem sehr viele Punkte bekommen würden, wenn dieser Fall in der KI nicht berücksichtigt werden würde.

Wenn die Anzahl der letzten Züge ohne Ausreißer kleiner als sechs ist (es wurden entweder noch nicht sechs Züge gespielt oder es musste ein Ausreißer entfernt werden) und wenn $s_{max} \leq 6$, dann hat der Gegner in den letzten sechs Zügen immer weniger als sieben Chips gelegt. Aufgrund der ersten Bedingung wird nicht zufällig gelegt, die Sicherheit geht jetzt vor und die KI legt genau sechs Chips.

Ansonsten wird der zu legende Wert folgendermaßen durch Zufall berechnet: der kleinste mögliche Wert des Zufalls ist $r_{min} + 5$, der größte mögliche Wert ist $min(20, max(r_{min} + 5, r_{max}))$. Es wird also mindestens der kleinste Wert des Gegners plus fünf gesetzt. Die Obergrenze ist der größte Wert des Gegners, sie muss jedoch mindestens so groß wie die Untergrenze sein (in diesem Fall ist der Zufall eigentlich sinnlos, da es sowieso nur einen möglichen Wert gibt). Außerdem darf auch hier nicht mehr als 20 gelegt werden. Diese Funktionsweise ist an einem einfachen Gegner gut zu sehen:

Legt dieser immer zufällig eine Zahl zwischen 1 und 10 (wie die Qualifikations-KI), dann wird die eigene KI meistens keine ähnlichen Situationen finden. Daher wird der Zufall verwendet. Wird für r_{min} der Wert 1 und für r_{max} der Wert 10 ermittelt, dann ist der kleinste mögliche Wert 6, der größte mögliche Wert 10. Die eigene KI bekommt so in den meisten Fällen mehr Chips als der Gegner, da der eigene Wert meistens größer als der des Gegners, der Abstand aber trotzdem meistens nicht größer als 5 ist.

Der Algorithmus in diesem Abschnitt benötigt $\mathcal{O}(1)$ Zeit, da mit den maximal letzten sechs Zügen des Gegners gearbeitet wird. Daher benötigen das Entfernen der Ausreißer sowie das Sortieren der Liste sowie alle übrigen Schritte nur konstante Zeit.

1.3.1 Ausreißer entfernen

Für den Zufall kann maximal ein Ausreißer aus der Liste mit l Elementen der letzten Züge des Gegners entfernt werden. Dies ist jedoch nur bei mindestens drei Zahlen möglich. Es wird angenommen, dass die Liste bereits sortiert ist. Der Abstand zwischen zwei aufeinanderfolgenden Elementen mit den Indizes i und $i+1$ ist c_i.

Zunächst wird der durchschnittliche Abstand c_{avg} zwischen zwei Zahlen berechnet. Dann wird das kleinste Element als Ausreißer entfernt, wenn der Abstand c_0 zwischen dem ersten und dem zweiten Element größer als der doppelte durchschnittliche Abstand ist: $c_0 > c \cdot 2$.

Das größte Element wird als Ausreißer entfernt, wenn der Abstand c_{l-2} zwischen dem vorletzten und dem letzten Element größer als der doppelte durchschnittliche Abstand ist: $c_{l-2} > c \cdot 2$.

Es kann maximal ein Element aus der Liste entfernt werden, da sie maximal sechs Elemente beinhalten kann. Bei mehr „Ausreißern“ wäre es nicht zwingend notwendig, dass diese wirklich Ausreißer sind.

Der Pseudocode 2 verdeutlicht diesen Algorithmus.

2 Umsetzung

Die KI wurde in der Sprache Java geschrieben. Der gesamte Code ist in der Klasse `AI` enthalten.

Es gibt eine innere Klasse namens `Situation`. Diese Klasse enthält die Variablen `length`, `difference`, `time` und `reaction`, die die Länge der Situation, den Gesamtunterschied zur aktuellen Situation, die „Zeit“ des Zuges und die vorhergesagte Reaktion des Gegners repräsentieren. Die Variable `situationComparator` (außerhalb der Klasse) vom Typ `Comparator<Situation>` kann zwei Situationen durch die Methode `compare()` nach den Kriterien aus Abschnitt „Beste zu legende Zahl berechnen“ vergleichen, indem eine negative Zahl zurückgegeben wird, wenn die erste Situation weiter vorne erscheinen soll, und eine positive Zahl zurückgegeben wird, wenn die zweite Situation weiter vorne erscheinen soll.

Der Zug wird in der Methode `zug()` ausgeführt. Sie fragt zunächst die Spieler-Objekte durch `me()` und `opponent()` sowie deren letzte Züge ab. Hat die eigene KI im letzten Zug nicht 0 gesetzt, dann findet gerade nicht der allererste Zug statt. In diesem Fall

Algorithm 2 Ausreißer entfernen

```
1: function RemoveOutlier(l)                          ▷ Liste mit Zahlen
2:     if l.size < 3 then
3:         return
4:     end if
5:
6:     c_avg ← 0                  ▷ Arithmetisches Mittel der Unterschiede berechnen:
7:     for j = 0 to l.size − 1 do
8:         c_avg ← c_avg + l[j + 1] − l[j]          ▷ Alle Unterschiede zusammenaddieren...
9:     end for
10:    c_avg ← c_avg/(l.size − 1) ▷ ... und dann durch die Anzahl der Unterschiede teilen
11:
12:    if l[1] − l[0] > c_avg · 2 then
13:        l.remove(0)                          ▷ Kleinstes Element ist ein Ausreißer
14:    else if l[l.size − 1] − l[l.size − 2] > c_avg · 2 then
15:        l.remove(l.size - 1)                 ▷ Größtes Element ist ein Ausreißer
16:    end if
17: end function
```

werden die letzten Züge zu den `Lists` `ownMoves` und `opponentMoves` hinzugefügt, beide enthalten Objekte vom Typ `Integer`. Danach wird die Methode `makeMove()` mit dem Zug-Objekt aufgerufen.

`makeMove()` ruft zunächst `bestNumber()` auf, diese Methode berechnet die beste zu legende Zahl. Die zurückgegebene Zahl wird durch das Zug-Objekt gesetzt. Wurde `null` zurückgegeben, so bedeutet dies, dass die Methode mit ähnlichen Situationen zu keinem Ergebnis geführt hat. Dann wird `random()` aufgerufen. Diese Methode verwendet den Zufalls-Algorithmus.

2.1 `bestNumber()` und `similarSituations()`

`bestNumber()` ruft zunächst `similarSolutions()` auf. Diese Methode sucht alle ähnlichen Situationen.

`similarSolutions()` erstellt zunächst eine `ArrayList` mit `Situation`s. Dann wird eine Schleife durchlaufen, welche alle Zahlen `i` von 0 bis exklusiv der Größe von `ownMoves` durchläuft. Bei jedem Durchgang werden die Variablen für den Gesamtunterschied, dem Unterschied, der die Vorzeichen beachtet und der Länge der längsten ähnlichen Situation (zu Anfang -1) erstellt. Dann kommt eine Schleife, welche alle Längen der Situationen von 1 bis inklusiv `i` durchläuft. Darin werden die beiden Unterschiede zur aktuellen Situation berechnet. Sind diese nicht größer als die Konstante `MAX_DIFFERENCE_OLD_CURRENT`, welche den Wert eins besitzt und den maximalen Unterschied zur aktuellen Situation darstellt, dann werden die Unterschiede zu den Gesamtunterschieden hinzuaddiert und die Länge der längsten ähnlichen Situation wird durch die aktuelle Länge ersetzt. Ist diese innere Schleife beendet und wurde die Länge der längsten ähnlichen Situation geändert

(ist nicht mehr -1), dann wird eine Instanz von `Situation` erstellt. Dafür wird unter Verwendung der Methoden `Math.round()` und `Math.max()` die vorhergesagte Reaktion des Gegners durch die Formeln aus „Lösungsidee“ berechnet. Die entstandene Situation wird der Liste hinzugefügt, welche nach Beendigung der äußeren Schleife zurückgegeben wird.

`bestNumber()` sortiert dann den Rückgabewert von `similarSolutions()` durch `Collections.sort()` und `situationComparator`. Maximum und Minimum der möglichen Reaktionen des Gegners, zu Anfang beide `null`, werden durch eine Schleife berechnet, welche die ersten vier Züge der sortierten Liste nimmt. Die kleinste bzw. größte vorhergesagte Reaktion des Gegners wird den Variablen von Maximum und Minimum zugewiesen. Die Zahl vier ist als Konstante namens `LAST_MOVES_TO_COMPARE` gespeichert. Ist das Maximum danach immer noch `null` oder ist das Maximum, von dem das Minimum subtrahiert wird, größer als `MAX_DIFFERENCE_LAST_MOVES`, so wird `null` zurückgegeben, als Zeichen, dass die Methode der ähnlichen Situationen nicht möglich ist. `MAX_DIFFERENCE_LAST_MOVES` ist ebenfalls eine Konstante mit dem Wert zwei und bezeichnet den maximalen Unterschied zwischen dem größten und dem kleinsten vorhergesagten Wert. Durch weitere Überprüfungen und Methoden wie `Math.random()` sowie `Math.max()` werden die verschiedenen Fälle für ähnliche Situationen abgearbeitet.

2.2 `random()`

Diese Methode gibt, wenn `opponentMoves` keine Elemente beinhaltet, direkt sechs zurück. Ansonsten wird eine neue Liste mit den letzten sechs (Konstante `LAST_MOVES_FOR_RANDOM`) Zügen des Gegners erstellt. Dafür wird die Methode `subList()` des Interfaces `List` verwendet und der Rückgabewert davon einer neuen Instanz von `ArrayList<Integer>` übergeben. Danach wird `removeOutlier()` mit der Liste als Argument übergeben, damit ein Ausreißer entfernt werden kann.

Dann werden das Minimum (erstes Element der Liste) und das Maximum (letztes Element) ermittelt. Durch weitere Überprüfungen und Methoden wie `Math.random()`, `Math.min()` und `Math.max()` werden die verschiedenen Fälle für das zufällige Setzen abgearbeitet.

2.2.1 `removeOutlier()`

Ist die Anzahl der Elemente der Liste, die als Argument übergeben wurde, kleiner als drei, so wird die Methode durch `return` direkt beendet. Ansonsten wird durch eine Schleife, die die Elemente vom ersten bis zum vorletzten durchläuft, der durchschnittliche Abstand zwischen zwei Elementen bestimmt. Durch weitere Überprüfungen des Abstandes vom ersten zum zweiten und vom vorletzten zum letzten Element wird u. U. ein Element am Anfang oder am Ende der Liste durch `remove()` entfernt.

3 Beispiele

Die KI kann unter dem Namen „sGroker“ auf dem Turnierserver betrachtet werden: `http://turnier.bundeswettbewerb-informatik.de/ai/profile/3233/`.

Zum Zeitpunkt dieses Schreibens befindet sie sich mit einem Score von 5,85 auf Platz 4.

3.1 Beispiel 1

`http://turnier.bundeswettbewerb-informatik.de/challenge/371661/`

Bei diesem Spiel sieht man leicht, dass der Gegner sehr oft acht Mal hintereinander sieben Chips und danach direkt drei legt. Legt der Gegner irgendwann erneut mehrmals hintereinander sieben Chips, so merkt die KI das, legt währenddessen einen Chip und genau beim neunten Mal so viele, dass sie mehr Chips als der Gegner, der drei legt, bekommt. Dadurch kann der Gegner in keinem dieser Züge die Differenz der Chips zu seinen Gunsten verbessern.

3.2 Beispiel 2

`http://turnier.bundeswettbewerb-informatik.de/challenge/372319/`

Hier gewinnt die KI ebenfalls, jedoch erst nach 160 Zügen. An diesem Spiel kann man gut sehen, wie die Vorhersage des gegnerischen Zuges oft gut klappt, in einigen Fällen jedoch auch etwas Falsches berechnet, sodass der Gegner auch öfters mehr Chips als die eigene KI bekommt.

3.3 Beispiel 3

`http://turnier.bundeswettbewerb-informatik.de/challenge/371568/`

Dieses Spiel gegen einen der besten Spieler endet nach 200 Zügen, da die maximale Länge erreicht ist. Die KI gewinnt dabei nur knapp. Das liegt daran, dass es manchmal funktioniert, die Anzahl der Chips des Gegners zu berechnen, dies jedoch auch oft zu einer falschen Vorhersage führt, wodurch der Gegner ebenfalls öfters Punkte bekommt.

3.4 Beispiel 4

`http://turnier.bundeswettbewerb-informatik.de/challenge/369548/`

Bei diesem Spiel gewinnt der Gegner (die zum Zeitpunkt dieses Schreibens beste KI) mit sechs Chips Vorsprung. Auch hier funktioniert die Vorhersage nicht in allen Fällen, sodass beide Spieler oft Chips bekommen und der Punktestand relativ ausgeglichen ist, der Gegner aber trotzdem zum Schluss einige Punkte mehr hat und gewinnt.

3.5 Beispiel 5

`http://turnier.bundeswettbewerb-informatik.de/challenge/372924/`

An diesem Spiel kann man sehen, dass die KI auch gegen Gegner verlieren kann, die ansonsten oft verlieren. Der Gegner im angegeben Spiel hat zum Zeitpunkt dieses

Schreibens lediglich 1 von 6 Spielen gewonnen, davon alle gegen die höchstplatzierten Spieler. Daran kann man sehen, dass es Gegner mit bestimmten Strategien gibt, die zwar gegen die meisten guten anderen Spieler verlieren, jedoch von unserer KI nicht besiegt werden können.

4 Quelltext

Listing 1: AI.java

```
import java.util.*;

public class AI {

    // Maximaler Unterschied zwischen der aktuellen und der anderen Situation,
        mit der verglichen wird:
    private static final int MAX_DIFFERENCE_OLD_CURRENT = 1;
    // Maximaler Unterschied zwischen der Vorhersage der letzten ähnlichen
        Situationen:
    private static final int MAX_DIFFERENCE_LAST_MOVES = 2;
    // Die Anzahl der letzten ähnlichen Situationen, deren Vorhersagen
        verglichen werden sollen:
    private static final int LAST_MOVES_TO_COMPARE = 4;
    // Die Anzahl der letzten Züge, die für den Zufall verwendet werden sollen:
    private static final int LAST_MOVES_FOR_RANDOM = 6;
    // Der maximale Bereich der Zahlen, die beim Zufall gesetzt werden können:
    private static final int MAX_RANDOM = 20;

    // Die eigenen Züge bzw. die des Gegners:
    private List<Integer> ownMoves = new ArrayList<Integer>();
    private List<Integer> opponentMoves = new ArrayList<Integer>();

    /**
     * Führt einen Zug aus.
     */
    public void zug(int id, Spiel.Zustand zustand, Spiel.Zug zug) {
        Spiel.Zustand.Spieler me = me(id, zustand);
        Spiel.Zustand.Spieler opponent = opponent(id, zustand);

        // Die letzten Züge (wenn vorhanden) hinzufügen:
        if (me.letzterZug() != 0) {
            ownMoves.add(me.letzterZug());
            opponentMoves.add(opponent.letzterZug());
        }

        makeMove(zug);
    }

    // private Spiel.Zustand.Spieler me(int id, Spiel.Zustand zustand) und
```

```
// private Spiel.Zustand.Spieler opponent(int id, Spiel.Zustand zustand)

/**
 * Eine Situation mit der Ähnlichkeit zur aktuellen Situation.
 */
public class Situation {

    int length;    // Länge der Situation
    int difference; // Unterschied zur aktuellen Situation
    int time;      // Gibt an, beim wievielten Zug die Situation stattfand
    int reaction;  // Gibt die danach gesetzte Zahl des Gegners an

    public Situation(int length, int difference, int time, int reaction) {
        this.length = length;
        this.difference = difference;
        this.time = time;
        this.reaction = reaction;
    }

}

/**
 * Sortiert Situationen nach deren Wichtigkeit. Die längste Situation
 * befindet sich ganz vorne. Bei gleicher Länge unterscheidet der
 * Unterschied zur aktullen Situation. Bei gleichem Unterschied wird die
 * neueste Situation bevorzugt.
 */
private Comparator<Situation> situationComparator = new
    Comparator<Situation>() {
    public int compare(Situation left, Situation right) {
        if (left.length != right.length) {
            return right.length - left.length;
        } if (left.difference != right.difference) {
            return left.difference - right.difference;
        } else {
            return right.time - left.time;
        }
    }
};

/**
 * Setzt den Zug.
 */
private void makeMove(Spiel.Zug zug) {
    Integer move = bestNumber();

    if (move != null) {
        // Zug konnte berechnet werden
        zug.setzen(move);
```

```
            return;
        }

        // Zufällig setzen
        zug.setzen(random(zug));
    }

    /**
     * Berechnet die beste zu setzende Zahl aus den zu der aktuellen Situation
     * möglichst ähnlichen Situationen.
     */
    private Integer bestNumber() {
        // Ähnliche Situationen zu der aktuellen:
        List<Situation> situations = similarSituations();

        // Die niedrigste bzw. höchste Reaktion, die der Gegner machen wird:
        Integer min = null, max = null;

        // Situationen sortieren, damit die wichtigsten davon ganz am Anfang
        // stehen. Danach die kleinste und die größte Reaktion des Gegners
            suchen.
        Collections.sort(situations, situationComparator);
        for (int i = 0; i < Math.min(LAST_MOVES_TO_COMPARE, situations.size());
            i++) {
            Situation situation = situations.get(i);

                if (min == null || situation.reaction < min)
                    min = situation.reaction;
                if (max == null || situation.reaction > max)
                    max = situation.reaction;
        }

        if (max == null || max - min > MAX_DIFFERENCE_LAST_MOVES) {
            // Bei zu unterschiedlichen vorhergesagten Werten kann nichts
                genaues angegeben werden:
            return null;
        }

        if (situations.size() < LAST_MOVES_TO_COMPARE || max - min > 0) {
            // Nur wenige ähnliche Situationen existieren oder die
            // vorhergesagten Werte sind unterschiedlich. Daher vorschtiger
                setzen.

            if (min > 6) {
                // Sechs weniger als die kleinste vorhergesagte Zahl und zur
                // Sicherheit noch 1-2 weniger setzen, mindestens jedoch 1.
                return Math.max(1, min - 6 - ((int) (Math.random() * 2)));
            } else {
                // 4-5 mehr als die kleinste vorgergesagte Zahl legen.
```

```
            return min + 3 + ((int) (Math.random() * 2));
        }
    } else {
        // Es gibt genug ähnliche Situationen, welche alle dasselbe
            voraussagen.

        if (min > 6)
            return min - 6; // 6 weniger als der Gegner legen
        else
            return min + 5; // 5 mehr als der Gegner legen
    }
}

/**
 * Sucht zur aktuellen Situation ähnliche Situationen.
 */
private List<Situation> similarSituations() {
    List<Situation> similar = new ArrayList<Situation>();

    // i ist der Index des Zuges mit der Reaktion auf die vorangegangene
        Situation. Die Schleife fängt
    // mit 1 an, denn die vorangegangene Situation muss mindestens ein Zug
        lang sein.
    for (int i = 1; i < ownMoves.size(); i++) {
        int totalDifference = 0; // Der Unterschied zwischen der Situation
            und der aktuellen Situation
        int signedDifference = 0; // Der Unterschied, wobei Vorzeichen
            beachtet werden
        int maxLength = -1; // Länge der längsten ähnlichen gefundenen
            Situation

        // j ist die Länge der ähnlichen Situation. Bei jedem Durchgang
            wird die nächstgrößere Situation
        // überprüft, wobei der Gesamtunterschied jedes Mal größer werden
            kann.
        for (int j = 1; j <= i; j++) {
            // difference1 und difference2 sind die Unterschiede zwischen
                den gesetzten Zahlen der aktuellen
            // Situation und den gesetzten Zahlen der (möglicherweise)
                ähnlichen Situation.
            int difference1 = ownMoves.get(i - j) -
                ownMoves.get(ownMoves.size() - j);
            int difference2 = opponentMoves.get(i - j) -
                opponentMoves.get(opponentMoves.size() - j);

            if (Math.abs(difference1) <= MAX_DIFFERENCE_OLD_CURRENT &&
                Math.abs(difference2) <= MAX_DIFFERENCE_OLD_CURRENT) {
                // Unterschiede sind klein genug, sodass die Situation
                    ähnlich ist
```

```
                totalDifference += Math.abs(difference1) +
                    Math.abs(difference2);
                signedDifference += difference1 + difference2;

                maxLength = j;
            } else {
                // Unterschiede zu groß, daher ist auch die nächstgrößere
                    Situation nicht ähnlich genug
                break;
            }
        }

        if (maxLength != -1) {
            // Es wurde eine ähnliche Situation gefunden

            // Die durchschnittliche Abweichung:
            int avgSignedDifference = (int) Math.round(((double)
                signedDifference) / (maxLength * 2));
            // Die Reaktion des Gegners, wobei die Abweichung einberechnet
                wird:
            int reaction = Math.max(1, opponentMoves.get(i) -
                avgSignedDifference);

            similar.add(new Situation(maxLength, totalDifference, i,
                reaction));
        }
    }

    return similar;
}

/**
 * Wählt zufällig eine Zahl aus dem Bereich der vom Gegner zuletzt
     gesetzten Zahlen aus.
 */
private int random(Spiel.Zug zug) {
    if (opponentMoves.isEmpty())
        return 6;

    // Letzte Züge des Gegners raussuchen, sortieren und Ausreißer
        entfernen:
    int subListStart = Math.max(0, opponentMoves.size() -
        LAST_MOVES_FOR_RANDOM);
    List<Integer> moves = new
        ArrayList<Integer>(opponentMoves.subList(subListStart,
        opponentMoves.size()));
    Collections.sort(moves);
    removeOutlier(moves);
```

```
        // Min und Max der Züge raussuchen:
        int min = moves.get(0);
        int max = moves.get(moves.size() - 1);

        if (min > 6) {
            // 6 weniger als das Minimum setzen (jedoch maximal MAX_RANDOM):
            return Math.min(MAX_RANDOM, min - 6);
        }
        if (moves.size() < LAST_MOVES_FOR_RANDOM && max <= 6) {
            // Kein "sicherer" Zufall, da es nicht mehr genug Züge gibt.
            // min und max sind kleiner als 7, also 6 verwenden:
            return 6;
        }

        // Der kleinste mögliche Zufallswert ist fünf mehr als das Minimum, der
            Bereich geht
        // bis zum Maximum der gegnerischen Züge, kann jedoch höchstens
            MAX_RANDOM lang sein:
        int startValue = min + 5;
        int endValue = Math.min(MAX_RANDOM, Math.max(startValue, max));
        return ((int) (Math.random() * (endValue - startValue + 1))) +
            startValue;
    }

    /**
     * Entfernt maximal einen Ausreißer aus der angegebenen sortierten Liste.
     */
    private void removeOutlier(List<Integer> list) {
        if (list.size() < 3) return; // Nur bei min. 3 Elementen möglich

        // Durchschnittlicher Abstand zwischen den jeweiligen Elementen
            berechnen:
        double avgDifference = 0;
        for (int j = 0; j < list.size() - 1; j++)
            avgDifference += list.get(j + 1) - list.get(j);
        avgDifference /= list.size() - 1;

        if (list.get(1) - list.get(0) > avgDifference * 2) {
            // Kleinstes Element hat den doppelten durchschnittlichen Abstand
            list.remove(0);
        } else if (list.get(list.size() - 1) - list.get(list.size() - 2) >
            avgDifference * 2) {
            // Größtes Element hat den doppelten durchschnittlichen Abstand
            list.remove(list.size() - 1);
        }
    }

}
```