Tutorial Hase und Igel Software Challenge 2018

Simon Döring

Inhalt

Start zum Schreiben einer KI

Am einfachsten ist es die KI auf Grundlage des SimpleClients zu schreiben:

- Als erstes muss der SimpleClient heruntergeladen werden (http://www.software-challenge.de/downloads/ bitte den SimpleClient als „Quellcode“ downloaden)
- Dieser muss nun in Eclips eingebunden werden: https://cau-kiel-tech-inf.github.io/socha-enduser-docs/#einrichtung-von-eclipse

Wo schreibe ich die KI?

Ist dies getan befindet sich im entsprechenden Projekt das Package „sc.player2018.logic“ mit einer Datei namens **„RandomLogic.java“**. Diese Datei müssen wir nun editieren.

In der Datei gibt es eine Funktion mit der Bezeichnung *public void onRequestAction()*. Diese ist wie Folgt aufgebaut:

```
public void onRequestAction(){
  //optional: Laufzeitmessung
    long startTime = System.nanoTime();


    //Quellcode der KI, welche eine Zug (move) erstellt


  long nowTime = System.nanoTime();

  //sende Move. Danach gibt es kein zurück mehr:
  sendAction(move);

  //optional: LOG-Eintrag mit Zeit:
  log.warn("Time needed for turn: {}", (nowTime - startTime)/1000000);
}
```

Es ist sehr wichtig, dass alle Funktionen, welche die KI verwendet, ihren ursprünglichen Aufruf in der Funktion onRequestAction haben. Dadurch wird sichergestellt, dass alle Membervariablen aktuell sind.

Wichtige Membervariablen

In der Klasse befinden sich auch drei sehr wichtige Membervariablen:

```
//Intern für die Server wichtig. Bitte nicht modifizieren
  private Starter client;
//Gibt u.a Informationen über mögliche Züge und das Spielbrett an
  private GameState gameState;
//Informationen über den aktuellen Spieler, welchen man steuert.
  private Player currentPlayer;
```

In den Folgenden Dokumenten werden Grundlagen zum erstellen einer KI gezeigt. Hierbei werden wir oft die Variablen **gameState** und **currentPlayer** verwenden.

Die **API-Dokumentation** befindet sich im Ordner „doc“ im Verzeichnis des SimpleClients.

Aufgaben

1. Arbeite alle anderen Dokumente durch.
2. Analysiere die Herangehensweise der SimpleClient-KI.
3. Schreibe eine KI.

Spielbrett Informationen

Die Klasse „FieldType“

Bevor wir mit der Klasse Board also dem Spielbrett arbeiten können, müssen wir das Enum „FieldType“ kennenlernen. Dieses Enum definiert die einzelnen Felder des Bretts. Es gibt:

CARROT	Karottenfeld
GOAL	Zielfeld
HARE	Hasenfeld
HEDGEHOG	Igelfeld
SALAD	Salatfeld
START	Start
POSITION_1	Positionsfelder
POSITION_2	
INVALID	Ungültig

Die Dokumentation für FieldType ist in /doc/sc/plugin2018/FieldType.html zu finden.

Funktionen der Klasse „Board“

Das Spielbrett kann man sich wie einer Art modifizieren „Array“ [1: Technisch gesehen hat die Klasse eine private Liste von Feldern. Allerdings ähnelt ein Aufruf der Funktion getTypeAt sehr den Aufruf eines Arrays. Auch wenn der Rückgabewert kein Feld sonder ein FieldType ist.] vorstellen. Mithilfe der FieldTypes können wir die Funktionen des Boards verwenden. Zunächst stellt sich allerdings die Frage, wie wir überhaupt das Spielbrett bekommen können. Das Spielbrett bekommen wir durch den GameState. Mit Hilfe der Methode *gameState.getBoard()*.

Unsere Eigene Position können wir mithilfe von **currentPlayer** oder **gameState** herausfinden:

```
currentPlayer.getFieldIndex(); //Es wird immer ein Integer zwischen 0 bis 64 zur
ückgegeben
gameState.getCurrentPlayer().getFieldIndex(); //das Selbe wie oben

if(currentPlayer.getFieldIndex() == gameState.getCurrentPlayer().getFieldIndex()){
    //wird immer ausgeführt
}
```

public final int getNextFieldByType(FieldType type,int pos)

Dieser Funktion übergeben wir eine Position und ein FieldType. Es gibt die Position des nächsten FieldType an, welches **nach** der angegeben Position liegt. Hat das Feld mit der angegeben Position, den gleichen FieldType, so wird dennoch die Position des nächsten Felds angegeben (siehe zweites Beispiel). Gibt es diesen FieldType nicht mehr wird -1 zurückgegeben.

```
gameState.getBoard().getNextFieldByType(FieldType.GOAL, currentPlayer.
getFieldIndex());
```

Dies würde in einem normalen Spiel immer 64 zurückgeben, außer man befindet sich auf dem Ziel. Im diesem Fall wäre das Ergebnis -1.

```
Board b = gameState.getBoard(); //Damit man nicht immer gameState.getBoard() schreiben
muss
final int index = currentPlayer.getFieldIndex(); //Index des Spielers

if(index == b.getNextFieldByType(b.getTypeAt(index), index)){
    System.out.println("Geht nicht");
}
```

Dies liegt daran, dass der verwendete Index nie der Rückgabewert von **getNextFieldByType** sein kann.

```
int next_hedgehog = b.getNextFieldByType(FieldType.HEDGEHOG,
currentPlayer.getFieldIndex());
int next_next_hedgehog = b.getNextFieldByType(FieldType.HEDGEHOG, next_hedgehog);
```

Dies würde die Postion des nächste und vom übernächste Igelfeld bestimmen. Achtung es wird hierbei nicht überprüft, ob es so ein Feld überhaupt gibt. Dies kann zu Fehlern führen. Steht man z.B auf dem letzten Igelfeld, so hat **next_next_hedgehog** den Wert 11, anstelle von -1 (Siehe Übung 2).

public final int getPreviousFieldByType(FieldType type,int pos)

Analog zu getNextFieldByType. Allerdings bezieht sich die Funktion auf das vorherige Feld mit dem entsprechenden FieldType.

public final FieldType getTypeAt(int pos)

Mit dieser Funktionen kann man den FieldType eines bestimmten Feldes ermitteln.

```
FieldType my_field = gameState.getBoard().getTypeAt(currentPlayer.getFieldIndex());
FieldType field_42 = gameState.getBoard().getTypeAt(42);
```

Speichert den aktuelle FieldType auf welchen man steht und den FieldType des Feldes 42.

```
if(gameState.getBoard().getTypeAt(-1) == FieldType.INVALID){
    //Wird immer ausgeführt, da es das Feld an der Position -1 nicht gibt
}
```

Dieses Beispiel zeigt auf, dass es durch getTypeAt nie zu einer IndexOutOfBoundsException kommen kann. Es gibt nur Felder im Intervall von 0 – 64. Sollte man nach einem Feld außerhalb diese Intervalls fragen, so wird immer INVALID zurückgegeben.

Die komplette API-Dokumentation ist in /doc/sc/plugin2018/Board.html zu finden.

Aufgaben

1. Gib die Entfernung des Gegners zum Startfeld aus. Verwende dabei keine Variablen oder Literale. Die Ausnahme ist der Rückgabewert. Dieser darf eine Literale sein (z.B. return 127;). Tipp: Suche in der API von GameState nach einer Funktion, welchen den anderen Spieler zurückgibt oder lese das Dokument „Hase_Igel_Player“.
2. Erkläre warum **next_next_hedgehog** den Wert 11 hätte, wenn wir auf den letzten Igelfeld stehen würden.

```
int next_hedgehog = b.getNextFieldByType(FieldType.HEDGEHOG,
currentPlayer.getFieldIndex());
int next_next_hedgehog = b.getNextFieldByType(FieldType.HEDGEHOG, next_hedgehog);
```

3. Schreibe eine Funktion, welche das übernächste Igelfeld ausgibt. Gibt es solch ein Feld nicht, so soll immer -1 zurückgegeben werden.

Die Klasse Player

Die Klasse Player repräsentiert einen Spieler. Der eigene Spieler kann in der RandomLogic durch die Variable **currentPlayer** oder mithilfe der GameState Funktion *getCurrentPlayer()* abgefragt werden. Der gegnerische Spieler kann ebenfalls mithilfe einer Methode von GameState bestimmt werden (*gameState.getOtherPlayer()*).

Die Klasse Player besitzt viele Funktionen, welche einen Informationen über den Spieler geben:

java.util.List<CardType>	*getCards()* Gibt die für diesen Spieler verfügbaren Hasenkarten zurück.
java.util.List<CardType>	*getCardsWithout(CardType type)* Gibt Karten ohne bestimmten Typ zurück.
int	*getCarrots()* Die Anzahl an Karotten die der Spieler zur Zeit auf der Hand hat.
int	*getFieldIndex()* Die aktuelle Position der Figur auf dem Spielfeld.
Action	*getLastNonSkipAction* Gibt letzte Aktion des Spielers zurück.
sc.shared.PlayerColor	*getPlayerColor()* Die Farbe dieses Spielers auf dem Spielbrett
int	*getSalads()* Die Anzahl der Salate, die dieser Spieler noch verspeisen muss.
boolean	*inGoal()* Überprüft, ob Spieler im Ziel.
boolean	*ownsCardOfType(CardType type)* Überprüft ob Spieler bestimmte Karte noch besitzt

Die Vollständige API ist in /doc/sc/plugin2018/Player.html zu finden.

Die Meisten dieser Funktionen sind selbsterklärend. Dennoch sind hier einige Beispiele angegeben:

```
System.out.print("Du hast noch folgende Karten: ");
for(CardType c : currentPlayer.getCards()){
    System.out.print(c + " ");
}System.out.println();
```

Dies würde alle Karten ausgeben, welche man noch hat.

```
if(currentPlayer.inGoal() && !currentPlayer.getCards().isEmpty()){
    System.out.println("Was fuer eine Verschwendung.");
}
```

Die If-Bedingung würde dann ausgeführt werden, wenn man im Ziel ist, allerdings noch Karten hat.

Eine weitere wichtige Funktion ist *getFieldIndex*. Diese Funktion wird häufig im Dokument „Hase_Igel_Spielbrett“ verwendet.

Neben diesen Funktionen existieren noch einige „Setter“. Diese sind allerdings hauptsächlich für den Server notwendig und haben einen geringen praktischen Nutzten für uns.

Aufgabe

Schreibe eine If-Bedingung, welche abfragt, ob beide Spieler auf dem selben Feld sind. Da dies nur möglich ist, wenn man auf dem Start- Zielfeld ist, soll zunächst abgefragt werden, ob der aktuelle Spieler auf dem Start- Zielfeld ist. Tipp: Es gibt die Funktion getTypeAt der Klasse Board. Weiter Informationen sind im Dokument „Hase_Igel_Spielbrett“ zu finden.

GameRuleLogic: Hilfreiche Hilfsfunktionen

Die Klasse GameRuleLogic hat viele Hilfsfunktionen, mit welchen man die Regeln des Spieles überprüfen kann. Hierbei sind alle Funktionen static. D.h., dass man sie ohne eine Instanz aufrufen kann.

static int	*calculateCarrots(int moveCount)* Berechnet wie viele Karotten für einen Zug der Länge moveCount benötigt werden.
static int	*calculateMoveableFields(int carrots)* Berechnet, wie weit man mit carrots Karotten gehen kann. Beispiel: Mit 68 Karotten kann man 11 Felder weit gehen.
static boolean	*canPlayCard(GameState state)** Gibt zurück, ob der derzeitige Spieler eine Karte spielen kann.

static boolean	*isValidToAdvance(GameState state, int distance)* Überprüft *Advance* Aktionen auf ihre Korrektheit. *Distance* steht für die Distanz die man zurücklegen will.Three
static boolean	*isValidToEat(GameState state)* Überprüft *EatSalad* Züge auf Korrektheit. Diese Funktion bezieht sich **nicht** auf die Karte *TAKE_OR_DROP_CARROTS*
static boolean	*isValidToExchangeCarrots(GameState state, int n)* Überprüft ob der derzeitige Spieler 10 Karotten nehmen oder abgeben kann. „n“ kann entweder 10 oder -10 sein. Je nachdem ob man annehmen oder abgeben will. Diese Funktion bezieht sich **nicht** auf die Karte *EAT_SALAD*.
static boolean	*isValidToFallBack(GameState state)* Überprüft *FallBack* Züge auf Korrektheit
static boolean	*isValidToPlayCard(GameState state, CardType c,int n)** Überprüft ob der derzeitige Spieler die Karte spielen kann. „n“ wird für die *TAKE_OR_DROP_CARROTS* Karte benötigt (s.u). „n“ kann die Wert 0, 20 oder -20 annehmen.
static boolean	*isValidToPlayEatSalad(GameState state)** Überprüft ob der derzeitige Spieler die *EAT_SALAD* Karte spielen darf.
static boolean	*isValidToPlayFallBack(GameState state)** Überprüft ob der derzeitige Spieler die *FALL_BACK* Karte spielen darf.
static boolean	*isValidToPlayHurryAhead(GameState state) ** Überprüft ob der derzeitige Spieler die *HURRY_AHEAD* Karte spielen darf.
static boolean	*isValidToPlayTakeOrDropCarrots(GameState state,int n)** Überprüft ob der derzeitige Spieler die *TAKE_OR_DROP_CARROTS* Karte spielen darf. „n“ kann entweder 0, 20 oder -20 sein.
static boolean	*isValidToSkip(GameState state)* Überprüft, ob der derzeitige Spieler aussetzen darf.
static boolean	*mustEatSalad(GameState state)* Überprüft ob man einen Salat fressen muss. Dies ist immer der Fall, wenn man in der vorherigen Runde ein Salatfeld betreten hat.
static boolean	*mustPlayCard(GameState state) ** Überprüft ob eine Karte gespielt werden muss.

static boolean	*playerMustAdvance(GameState state)* Überprüft ab der derzeitige Spieler im nächsten Zug einen Vorwärtszug machen muss.

*Diese Funktionen werden im Kapitel „Erweiterte Beispiele“ besprochen.

Die komplette API-Dokumentation ist in doc/sc/plugin2018/util/GameRuleLogic.html zu finden.

Simulation des Gegners

Natürlich kann man alle diese Funktionen auch auf den Gegenspieler anwenden. Dafür müssen wir allerdings eine Kopie des GameState erstellen:

```
try {
    GameState otherGameState = gameState.clone();//Deep-Copy
    otherGameState.setTurn(gameState.getTurn()+1); //Rundenanzahl hochsetzten für
switch Befehl
    otherGameState.switchCurrentPlayer(); //Tausche currentPlayer (hängt von der
Rundenanzahl ab)
    //Gib Informationen über den Gegner aus
    System.out.println("CurrentPlayer von otherGameState:" +
otherGameState.getCurrentPlayer());
} catch (CloneNotSupportedException e1) {// Fehlerbehandlung
    e1.printStackTrace();
}
```

Alle Funktionen von GameRuleLogic können wir nun, mithilfe der Variable otherGameState, auf den Gegner anwenden.

Einführende Beispiele

Die meisten Funktionen dieser Klasse sind selbsterklärend. Dennoch werden einige der häufig verwendeten Funktionen mit kleinen Beispielen vorgestellt.

```
if(GameRuleLogic.isValidToEat(gameState) != GameRuleLogic.mustEatSalad(gameState)){
    System.out.println("Unmöglich");
}
```

Nach den Regeln muss man immer ein Salat essen, wenn man im vorherigen Zug ein Salatfeld betreten hat. Außerdem ist dies die einzige Möglichkeit die Aktion EatSalad auszuführen (nicht mit dem Spielen der EAT_SALAD Karte verwechseln).

Dadurch wird auch die Unerfüllbarkeit des folgenden Ausdrucks impliziert:

```
if(GameRuleLogic.isValidToEat(gameState) && (GameRuleLogic.
isValidToFallBack(gameState) || GameRuleLogic.canMove(gameState))){
    System.out.println("Unmöglich");
}
```

Eine weiter hilfreiche Funktion ist *isValidToAdvance*. Mit dieser Funktion wird überprüft, ob ein Vorwärtszug mit der übergeben Distanz überhaupt möglich ist:

```
if(GameRuleLogic.isValidToAdvance(gameState, GameRuleLogic.
calculateMoveableFields(currentPlayer.getCarrots())+1)){
    System.out.println("Unmöglich");
}
```

Die Funktion *calculateMoveableFields* gibt hierbei die maximale Entfernung zurück, welche man mit den übergebenen Karotten laufen darf. Diese maximale Entfernung wird immer um 1 erhöht, was dazu führt, dass der Zug immer unmöglich ist.

Erweitertes Beispiel

Alle Funktionen die mit einem * markiert wurden (s.o) haben eine Gemeinsamkeit. Sie beziehen sich auf das Spielen von Karten. Das Spielen von Karten ist allerdings nur erlaubt, wenn man das entsprechende Hasenfeld in der selben Zug betreten hat. Deshalb müssen wir GameState bearbeiten, damit diese Funktionen überhaupt Sinn haben. Das folgende Beispiel gibt eine Möglichkeit an, wie man diese Funktionen einsetzten kann:

```
if(gameState.getNextFieldByType(FieldType.HARE, currentPlayer.getFieldIndex())>0
){//wenn es ein nächstes Hasenfeld gibt
try {

    GameState gameHare = gameState.clone(); //erstelle Deep-Copy
    //setzte den aktuellen Spieler auf ein Hasenfeld
    gameHare.getCurrentPlayer().setFieldIndex(gameState.getNextFieldByType
(FieldType.HARE, currentPlayer.getFieldIndex()));
    System.out.println(gameHare.getCurrentPlayer().getCards()); //gib alle Karten aus

    //Welche Karten kann man spielen?
    System.out.println("Play EatSalad: " + GameRuleLogic.
isValidToPlayEatSalad(gameHare));
    System.out.println("Play TakeOrDropCarrots: " +
GameRuleLogic.isValidToPlayTakeOrDropCarrots(gameHare,20));
    System.out.println("Play HurryAhead: " + GameRuleLogic.
isValidToPlayHurryAhead(gameHare));
    System.out.println("Play FallBack: " + GameRuleLogic.
isValidToPlayFallBack(gameHare));
} catch (CloneNotSupportedException e1) {
    e1.printStackTrace();
}
}//ende if
```

Hierfür müssen wir den Spieler einfach nur auf das Hasenfeld setzten. Allerdings wird nicht überprüft, ob der Spieler überhaupt bis zum nächsten Hasenfeld laufen kann.

Aufgaben

1. Ist das Ausführen der inneren If-Bedingung wirklich unmöglich. Erkläre warum oder gib ein Gegenbeispiel an:

```
int dif = gameState.getOtherPlayer().getFieldIndex() - gameState.getCurrentPlayer
().getFieldIndex();

if(dif >= 0 && dif <= GameRuleLogic.calculateMoveableFields(currentPlayer.
getCarrots())){
    if(GameRuleLogic.isValidToAdvance(gameState, dif)){
        System.out.println("Unmöglich");
    }
}
```

2. Erweiterte das Beispiel aus dem Kapitel „Erweitertes Beispiel“ so, dass sicher gestellt wird, dass der aktuelle Spieler auf ein Hasenfeld gesetzt wird, welches er wirklich erreichen kann.