Oskar-Maria-Graf-Gymnasium
Neufahrn b. Freising

Abiturjahrgang
2015

S E M I N A R A R B E I T

Rahmenthema des Wissenschaftspropädeutischen Seminars:

Zufall und Nichtdeterminismus

Leitfach: **Informatik**

Thema der Arbeit:

Simulation von genetischen Algorithmen

Zufall am Beispiel von Fechtern

Verfasser: **Jan van Brügge**

Kursleiter: **StR Sven Baumer**

Abgabetermin: 4. November 2014

Bewertung	Note	Notenstufe in Worten	Punkte		Punkte
schriftliche Arbeit				x 3	
Abschlusspräsentation				x 1	
				Summe:	
				Gesamtleistung nach § 61 (7) GSO = Summe:2 (gerundet)	

Datum und Unterschrift der Kursleiterin bzw. des Kursleiters

Inhaltsverzeichnis

1 Einführung **3**
1.1 Einleitung in genetische Algorithmen 3
1.2 Aufbau eines genetischen Algorithmus 3
1.3 Ziel der Arbeit 4

2 Das Optimierungsproblem dieser Arbeit **4**

3 Der Algorithmus **5**
3.1 Erste Ideen 5
3.2 Wichtige Teile des Algorithmus 6
3.2.1 Crossover 6
3.2.2 Trim 7
3.3 Entwicklung 7
3.4 Erste Iteration 7
3.4.1 Allgemeines 7
3.4.2 Probleme und Lösungen 8
3.5 Zweite Iteration 8
3.5.1 Allgemeines 8
3.5.2 Probleme und Lösungen 9

4 Ergebnisse **12**
4.1 Veränderung der Parameter 13
4.2 Ausführung des Algorithmus 14

5 Praktische Anwendung des Algorithmus **14**

6 Ausblick **15**

A Quellenverzeichnis **16**

B Quellcode **16**

1 Einführung

1.1 Einleitung in genetische Algorithmen

Genetische Algorithmen werden heutzutage an vielen Stellen zur Optimierung von Vorgängen in allen Bereichen eingesetzt, zum Beispiel in der Kunst[1] oder der Forschung[2]. Sie dienen zur Verbesserung herkömmlicher Methoden oder zur Erfassung großer Datenmengen[3]. Schon Alan Turing war sich der Wichtigkeit der genetischen Algorithmen bewusst[4]:

> „Man muss mit dem Unterrichten einer Maschine herum experimentieren und schauen, wie gut sie lernt. [...] Es gibt einen offensichtlichen Zusammenhang zwischen diesem Prozess und Evolution. [...] Man darf allerdings hoffen, dass dieser Prozess schneller abläuft."
> *Alan Turing*

1.2 Aufbau eines genetischen Algorithmus

Anhand des breiten Spektrums der Einsatzgebiete von genetischen Algorithmen finden sich viele Forschungsarbeiten zu diesem Thema. Ein genetischer Algorithmus besteht im Grundsatz aus fünf Schritten[5]:

1. Selektion: Auswahl von Individuen für die Rekombination
2. Rekombination: Kombination der ausgewählten Individuen
3. Mutation: Künstliches Verändern der Nachfahren aus der Kombination
4. Evaluation: Zuweisung eines Wertes anhand einer Fitnessfunktion
5. Wiederholung der Schritte Eins bis Vier

Die Fitnessfunktion ist eine fest definierte Methode, die anhand von Parametern, welche die Individuen darstellen, ihren Wert für das zugrunde liegende Problem erhält. Dieser Wert gibt die Qualität des Individuums an. Durch die gezielte Selektion werden Individuen, welche einen deutlich geringeren Wert besitzen als andere, nicht erneut zur Kombination und zur Mutation herangezogen.

[1] *EVOGENIO – Evolutionäre Kunst* 2008
[2] Rudolph und Schwefel 1994
[3] Koh 2006
[4] Turing 1950
[5] *Evolutionärer Algorithmus*

1.3 Ziel der Arbeit

Die Bedeutung genetischer Algorithmen wird sich zukünftig immer weiter steigern, ob in der Industrie, der Forschung oder auch in anderen Bereichen unseres täglichen Lebens. Die verstärkte Nutzung des sog. Data-Minings beispielsweise wird immer häufiger solche genetischen Algorithmen erfordern[6].

Um beispielhaft darzustellen, wie genetische Algorithmen Probleme bearbeiten, wurde für diese Arbeit eine Aufgabe gewählt, die mithilfe eines eigenen, selbst entwickelten genetischen Algorithmus gelöst werden soll. Die Ergebnisse des Algorithmus werden anschließend analysiert und ausgewertet.

2 Das Optimierungsproblem dieser Arbeit

Ein genetischer Algorithmus soll meist ein Optimierungsproblem lösen. Daher war es auch zur Konkretisierung des Seminarthemas nötig, ein konkretes Optimierungsproblem zu finden, welches der Algorithmus lösen soll. Um die Komplexität des Programms möglichst gering zu halten, wurde ein eigenes Optimierungsproblem entwickelt, welches im Folgendem erläutert wird.

Das Optimierungsproblem ist in diesem Falle ein Spiel, in dem zwei wahrscheinlichkeitsgesteuerte Spieler auf einem eindimensionalen Spielbrett gegeneinander antreten. Das Spielbrett besteht aus einer bestimmten Anzahl an Feldern, auf denen die Spieler einmal pro Zug entweder nach vorne oder nach hinten gehen oder stehen bleiben können. Dieselben Optionen stehen auch bei einem Angriff zur Verfügung. Daraus ergeben sich sechs Optionen. Diese werden durch Wahrscheinlichkeiten angegeben, welche in der Summe Eins ergeben. Ein Spieler wählt anhand dieser Wahrscheinlichkeiten zufällig eine Aktion aus, welche anschließend durch Verschieben oder Stehen bleiben auf dem Spielbrett ausgeführt wird. Zusätzlich wird bei einem Angriff die Wahrscheinlichkeit, einen Angriff zu parieren, für die nächste Runde herab gesetzt. Damit wird versucht, ein realistisches Verhalten zu simulieren, da reale Fechter nach einem Angriff in einer offenen Position stehen und schwerer parieren können.

Ein Spiel besteht aus mehreren Partien. Eine Partie gilt dann als gewonnen, wenn entweder der eine Spieler durch zu oftmaliges Rückwärtslaufen aus dem Spielfeld tritt oder der Gegner durch einen Angriff besiegt wird. Um anzugreifen, muss der Spieler direkt vor seinem Gegner stehen, da er sonst diesen nicht trifft. Auch bei Nichttreffen wird die Blockwahrscheinlichkeit herab gesetzt. Falls er den Gegner trifft, wird anhand der aktuellen Blockwahrscheinlichkeit des Gegners überprüft, ob der Angriff erfolgreich war.

[6] Freitas 2003

Nach Beendigung einer Partie, d.h. ein Spieler ist entweder getroffen oder hat das Feld verlassen, werden noch weitere Partien gespielt, um ein repräsentatives Ergebnis zu erhalten. Die Anzahl der Partien ist einstellbar. Die Anzahl muss ungerade sein, da sonst ein Unentschieden als Ergebnis möglich wäre. Der Gewinner des Spiels ist derjenige, der die meisten Partien für sich entscheiden konnte.

Insgesamt spielt jeder Spieler gegen jeden anderen Spieler solch ein Spiel. Die Summe aller Spiele ergeben eine Generation. Bei einer gewählten Anzahl von 200 Spielern und 5 Partien pro Spiel ergibt sich mit der Gauß'schen Formel als Ergebnis 20100 Spiele pro Generation. Daher werden $20100 \cdot 5 = 100500$ einzelne Partien gespielt.

$$\text{Gauß'sche Formel: } \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \text{ für } n = 200\text{: } \sum_{i=1}^{200} i = \frac{200(200+1)}{2} = 20100$$

Die Fitnessfunktion des Algorithmus zählt in diesem Fall die Anzahl der Gewinne der verschiedenen Spieler. Zur Beobachtung der Auswirkungen auf die Wahrscheinlichkeiten der Spieler wurden zusätzlich noch weitere Parameter eingebaut, welche u. a. regeln, wie groß das Spielfeld ist, an welcher Startposition die Spieler stehen und ob die Startwahrscheinlichkeiten zufällig vergeben werden sollen.

3 Der Algorithmus

Der Algorithmus zur Lösung dieses Optimierungsproblems ist in der Sprache Java geschrieben. Diese Sprache bietet als Hochsprache ausreichende Werkzeuge zur Entwicklung auch komplexerer Algorithmen. Sie ist stark objektorientiert, was die Arbeit mit Spielern und Partien als logisch getrennte Klassen erleichtert. Java ist außerdem einfach zu benutzen, da die virtuelle Maschine häufige Fehler, wie z. B. hängende Zeiger, abfängt. Des Weiteren ist ein Programm, welches in Java geschrieben wurde, leichter zu lesen und zu verstehen, da Java als stark typisierte Sprache für Variablen und Funktionen fest definierte Datentypen benötigt und anders als schwach typisierte Sprachen, wie zum Beispiel Python, die Datentypen nicht zur Laufzeit ermittelt. Auch die hervorragenden Dokumentationswerkzeuge, welche neben normalen Kommentaren auch die Möglichkeit zur Javadoc bieten, prädestinieren die Sprache für diesen Algorithmus.

3.1 Erste Ideen

Neben der Wahl der Programmiersprache musste auch die Art der Programmierweise festgelegt werden. Da sich aufgrund der Art des Algorithmus — verschiedene Spieler, die gegeneinander antreten — eine logische Trennung und Kapselung der verschiedenen Programmteile anbietet, wurde ein objektorientierter Ansatz gewählt. Die Spieler werden

durch eine eigene Klasse dargestellt, welche wiederum eine bestimmte Logik enthält. Immer zwei Spieler befinden sich gleichzeitig auf dem Spielfeld. Gesteuert wird der ganze Wettkampf durch eine zentrale Hauptklasse. Hierdurch wird eine maximale Benutzerfreundlichkeit erreicht, da interne Prozesse, wie z. B. das Auswählen der nächsten Aktion, nach außen hin gekapselt werden. Der Aufrufer muss sich nur die nächste Aktion des Spielers von der Logikklasse geben lassen, und braucht diese nicht jedes Mal selbst zu berechnen. Dies vermeidet Fehler und unnötige Redundanzen.

3.2 Wichtige Teile des Algorithmus

3.2.1 Crossover

Generell dient Crossover, auch Kreuzen genannt, zur Verbindung und Verbesserung eines Individuums durch Adaptierung verschiedener Schemata der beiden Eltern. Diese Eltern sind zwei andere Individuen, welche sich durch ein gutes Ergebnis in der Fitnessfunktion für eine Weitergabe ihrer Fähigkeiten qualifiziert haben.

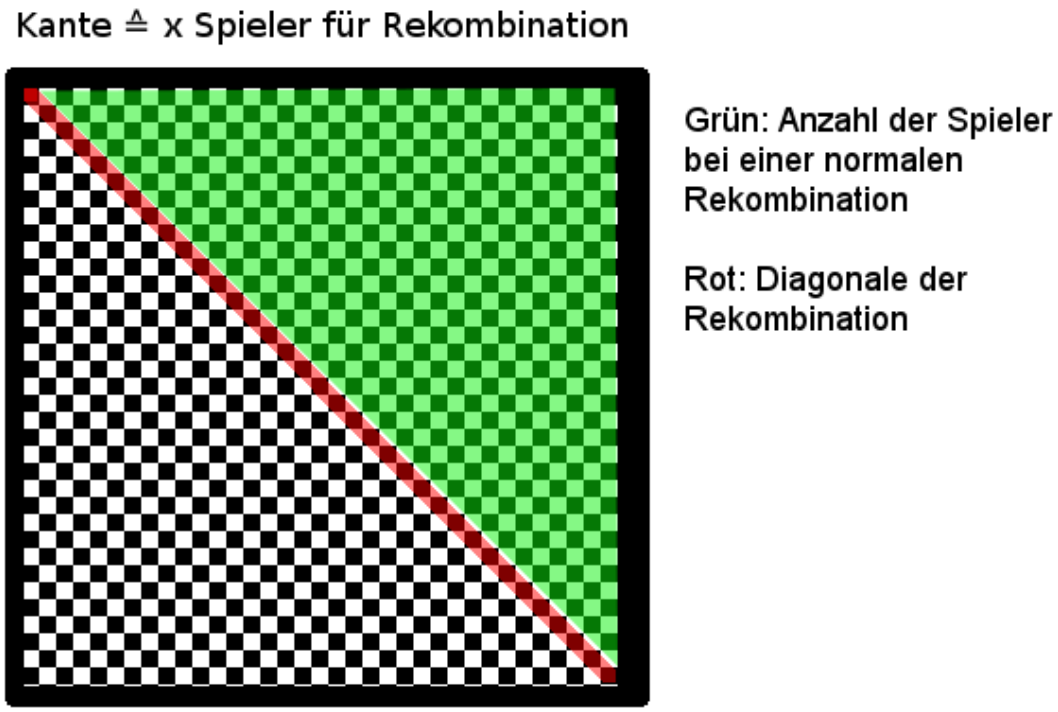

Abbildung 1: Visuelle Darstellung des Crossovers, Quelle: eigene Darstellung

Im beschriebenen Algorithmus wird dieses Prinzip durch die Bildung des Mittelwerts der jeweiligen Wahrscheinlichkeit der Eltern erreicht. Die Anzahl der Individuen, welche für diese Rekombination ausgewählt werden, ergibt sich aus einer Funktion, welche einen bestimmten Prozentsatz der besten Spieler in zwei verschachtelten Schleifen kreuzt. Dieser

Ansatz ist in Abbildung 1 dargestellt. Man sieht auch den Unterschied einer normalen Rekombination, bei der jedes Individuum nur einmal mit jedem anderen gepaart wird, hier grün dargestellt. Des Weiteren sieht man die Diagonale der Rekombination (hier rot dargestellt), welche pro markiertem Kästchen jeweils die simple Vervielfältigung einer einzelnen Logik darstellt.

Zur Vereinfachung der Erstellung neuer Logiken per Crossover wurde ein eigener Konstruktor erstellt, welcher zwei andere Logiken als Parameter benötigt. Dieser Konstruktor bildet von den Einzelwahrscheinlichkeiten der Eltern jeweils den Mittelwert und verwendet anschließend *trim*() (siehe Kapitel 3.2.2), um die Gesamtwahrscheinlichkeit wieder auf Eins zu normalisieren.

3.2.2 Trim

Zur Vereinfachung des Normalisieren, d. h. das Setzen der Gesamtwahrscheinlichkeiten wieder auf Eins, wurde eine Funktion *trim*() in der Klasse "Logic"geschrieben.

$$\text{Formel: } 1/(\sum_{i=0}^{5} Wahrscheinlichkeiten[i])$$

Diese Funktion berechnet den Quotienten von Eins (also 100 %) und der Gesamtwahrscheinlichkeit der einzelnen Aktionen, welcher anschließend mit jeder der Einzelwahrscheinlichkeiten multipliziert wird. Diese Methode wird immer aufgerufen, wenn die Möglichkeit besteht, dass die Gesamtwahrscheinlichkeit ungleich Eins sein kann. Dies ist z. B. der Fall, wenn zwei Logiken gekreuzt werden oder ein Klon verändert wird.

3.3 Entwicklung

Im Zuge der Entwicklung gab es zwei große Iterationen, bei denen das Programm komplett neu geschrieben wurde. Diese werden im Folgenden noch vorgestellt. Die wesentlichen Unterschiede zwischen der ersten und der zweiten Version sind die Möglichkeit zur Nebenläufigkeit, ein verbessertes Zusammenstellen der Spielpartien sowie eine ausgefeiltere Methode zur Kreuzung und Veränderung. Des Weiteren wurden in der zweiten Version einige Fehler behoben, welche in der ersten Version zu einer ungleichen Verteilung der Spiele geführt hatte.

3.4 Erste Iteration

3.4.1 Allgemeines

Bei der ersten Iteration wurde vor allem auf den objektorientierten Ansatz geachtet. Deshalb ergab sich eine schöne Trennung der einzelnen Komponenten in eine Struktur

mit insgesamt sieben Klassen. Um die Komplexität des Programms gering zu halten, wurde auf einstellbare Parameter verzichtet. So wurde beispielsweise die Spielerzahl auf 200 eingestellt und die Startfelder sowie die Feldgröße von Beginn an festgelegt.

Auch waren Variablen und Funktionsnamen nicht einheitlich benannt. Sie waren meistens auf Deutsch, allerdings gab es auch Mischformen aus Deutsch und Englisch, was die Lesbarkeit des Programms erheblich erschwerte. Aufgrund von Inkompatibilität mit deutschen Umlauten und der IDE Eclipse für Linux gab es Fehler im Programm, da Umlaute in ASCII-Symbole geändert wurden.

Die meisten Kommentare waren inkonsequent, da Javadoc nicht durchgängig benutzt wurde.

3.4.2 Probleme und Lösungen

Um die Normalisierung der Gesamtwahrscheinlichkeit auf Eins zu gewährleisten, wurde in alle Funktionen, die eine Änderung der Wahrscheinlichkeiten bewirken, ein Codesegment eingesetzt, welches diese Aufgabe übernehmen soll.

$$\text{Formel: } ((\sum_{i=0}^{5} Wahrscheinlichkeiten[i]) - 1)/6$$

Eins wurde von der Summe der Einzelwahrscheinlichkeiten der Aktionen subtrahiert und durch Sechs geteilt (sechs verschiedene Aktionen), um einen Wert pro Wahrscheinlichkeit zu erhalten. Dieser Wert wurde anschließend von jeder Wahrscheinlichkeit abgezogen.

Diese Methode hatte mehrere Nachteile. Der größte Nachteil lag in den vielen Redundanzen im Code, welche eine Wartung sehr schwierig machten. Aus mathematischer Sicht war nachteilig, dass diese Formel den relativen Unterschied zwischen den Wahrscheinlichkeiten veränderte. Außerdem konnte die Wahrscheinlichkeit auch negativ werden, was einem Grundprinzip der Wahrscheinlichkeitsrechnung widerspricht.

Die Lösung lag in der Verwendung eines prozentualen Systems, welches an den jeweiligen Stellen eingesetzt wurde (siehe Kapitel 3.2.2). In der zweiten Iteration wurden die Redundanzen entfernt und durch die Methode $trim()$ ersetzt.

3.5 Zweite Iteration

3.5.1 Allgemeines

Bei der zweiten Iteration wurde vor allem auf die Möglichkeit zur Nebenläufigkeit und auf die Verbesserung der Formeln zur Berechnung der Kreuzungen am Ende jeder Runde Wert gelegt. Es wurden Parameter eingebaut, die eine Flexibilität des Algorithmus gewährleisten. Weiter wurden unnötige Redundanzen sowie Objekterstellung gemindert.

Die zweite Iteration benötigt anstatt eines Zufallsgenerators pro Spieler einen Zufallsgenerator pro Thread. Dies verhindert eine überflüssige Belastung des Arbeitsspeichers, da pro Zeiteinheit nur maximal so viele Zufallsgeneratoren gebraucht werden, wie Threads vorhanden sind. In der ersten Version wurden hier also 199 unbenutzte Zufallsgeneratoren erzeugt.

3.5.2 Probleme und Lösungen

Zu den Problemen der zweiten Version zählten unter anderem der aus der Verwendung von Parametern folgende Zwang zu einer mathematischen Lösung zur Berechnung der zu kreuzenden oder zu verändernden Spieler.

Erste Formel: $(AnzahlSpieler \cdot 0,05)^2$

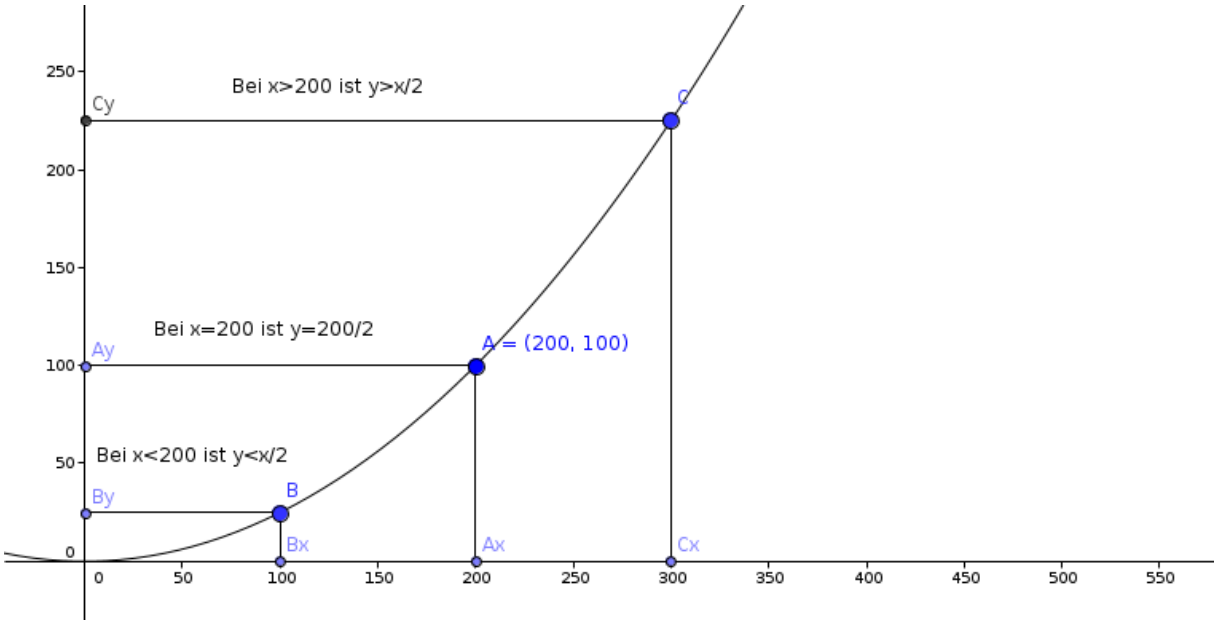

Abbildung 2: Graph der ersten Crossoverformel, Quelle: eigene Darstellung

Das Problem der Formel lag in der Verwendung von zwei geschachtelten Schleifen resultierende polynomielle Vergrößerung des Anteils der zu kreuzenden Spieler an der Gesamtmenge. Daher war der Anteil für große Zahlen nicht mehr der gleiche Anteil an der Gesamtmenge wie für kleine Zahlen, sondern weitaus größer, im Extremfall sogar über 100 %. Wie in Abbildung 2 zu sehen ist, ist der Graph der Funktion keine Ursprungsgerade, sondern eine Parabel, welche den gewünschten Wert $x/2$ nur bei $x = 200$ ergibt. Deswegen wurde die Formel verändert.

Als erster Versuch zur mathematischen Lösung wurde festgelegt, dass 5% der Spieler gepaart werden, um dadurch 50% der neuen Individuen zu erhalten.

$$\text{Zweite Formel: } (\sqrt{AnzahlSpieler \cdot 0,50})^2$$

Die zweite Formel ermöglichte einen anderen Ansatz. Anstatt die gewollten 50 % am Gesamtanteil der neuen Spieler durch eine Quadrierung zu erreichen, wurde von 50 % die Wurzel gezogen, um so die Zahl der zu kreuzenden Spieler zu berechnen.

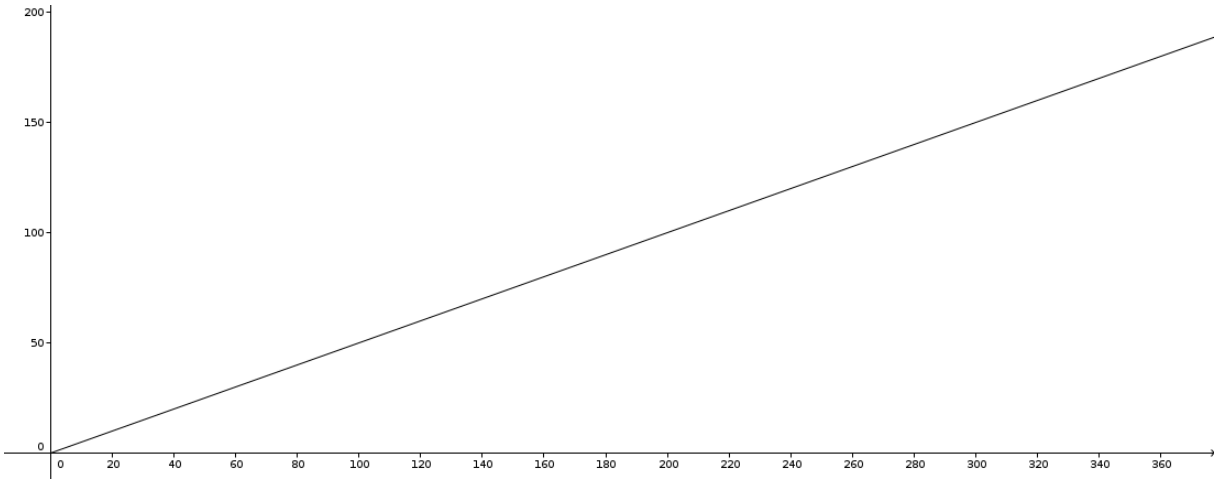

Abbildung 3: Graph der zweiten Crossoverformel, Quelle: eigene Darstellung

Wenn man hierzu den Graphen der Funktion betrachtet (Abbildung 3), stellt man fest, dass die Funktion eine Ursprungsgerade mit der Steigung $m = 0,5$ ergibt, sodass eine Vergrößerung der Anzahl an Spielern auch ein direkt proportionales Wachstum der zu kreuzenden Spieler zur Folge hat.

Da allerdings das Ziehen einer Wurzel positive rationale und nicht nur natürliche Zahlen ergibt, wird das Ergebnis per Konvertierung in eine ganze Zahl abgerundet. Diese Zahl wird anschließend als Zähler der beiden Schleifen benutzt. Die Differenz aus der Hälfte der Spieler und dem Ergebnis der Quadrierung durch die beiden Schleifen bilden den sog. Rest, welcher später durch das Übernehmen von Spielern aus der letzten Runde wett gemacht wird.

$$\text{Rest: } (AnzahlSpieler \cdot 0,50) - \lfloor \sqrt{AnzahlSpieler \cdot 0,50} \rfloor^2$$

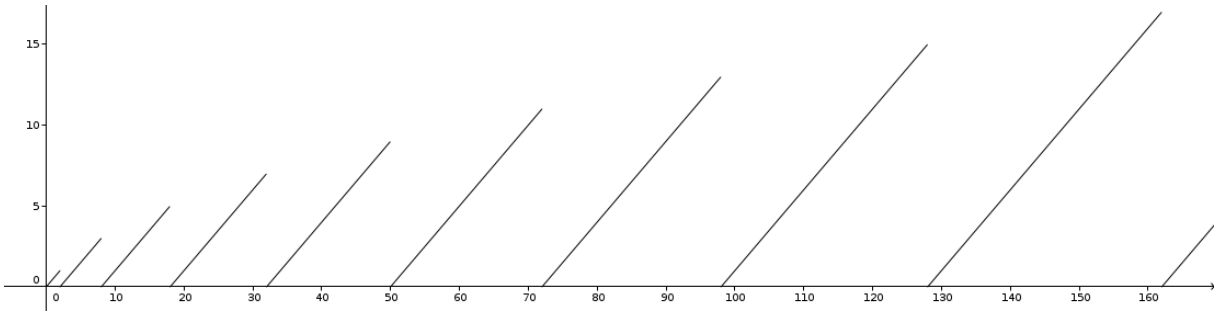

Abbildung 4: Graph der Restfunktion, Quelle: eigene Darstellung

Die Graph der Formel in Abbildung 4 zeigt das lineare Wachstum des Rests bis zu einem bestimmten Punkt, an dem der Rest wieder zu Null springt und erneut anwächst. Dieses Verhalten erklärt sich durch das Erreichen einer Zahl, die ohne Rest radizierbar ist und daraufhin die Funktion des Rests wieder bei Null beginnen lässt.

Ein weiteres Problem der zweiten Version war die Thread-Inkompatibilität durch die Verwendung eines Zufallsgenerators pro Spieler. Da es zwar im Normalfall 200 Spieler gibt, allerdings 20100 Partien, welche gespielt werden müssen, ist es sehr wahrscheinlich, dass ein Spieler von mehreren Threads benutzt wird, da er in mehreren verschiedenen Partien gebraucht wird.

Um das Problem zu lösen, wurden sämtliche eigene Zufallsgeneratoren aus den Spielern entfernt und durch Parameter in den jeweiligen Funktionen ersetzt. Die Zufallsgeneratoren wurden in die BoardMaster-Klassen verlegt, welche selber durch die Implementierung des Callable-Interfaces jeweils einen eigenen Thread darstellen. Durch das Verlegen in diese Klassen ist sichergestellt, dass die einzelnen Threads nicht durch gleichzeitiges Benutzen der Zufallsgeneratoren Fehler verursachten.

Ein weiteres Problem war, dass das Programm innerhalb der Funktion einfror. Aus diesem Grund wurde die kritische Funktion der Logic-Klasse als synchronisiert markiert, um ein gleichzeitiges Benutzen der Funktion bei einem Objekt zu verhindern. Leider hat diese Maßnahme nicht zur Behebung des Problems beigetragen. Als letzter Versuch wurden die einzelnen Zufallsgeneratoren aus den BoardMaster-Klassen durch ThreadLocalRandom-Zufallsgeneratoren ersetzt, welche die Methoden der Zufallsgeneratorklasse überschreibt, um eine bestmögliche Threadkompatibilität zu gewährleisten. Selbst diese Maßnahme hat nicht den gewünschten Erfolg erzielt.

Ein Grund für das Einfrieren der Zufallsgeneratorklasse liegt möglicherweise an den vielen Anfragen an sie. Bei den bereits oben erwähnten 20100 Partien, 5 Spielzügen pro Partie pro Spieler und 5 Spielen in der Sekunde kommt man auf ca. 2,5 Million Anfragen pro Sekunde.

4 Ergebnisse

Um die Ergebnisse des Algorithmus zu analysieren, wurde pro Durchlauf eine Momentaufnahme der Wahrscheinlichkeiten des aktuell besten Spielers erstellt.

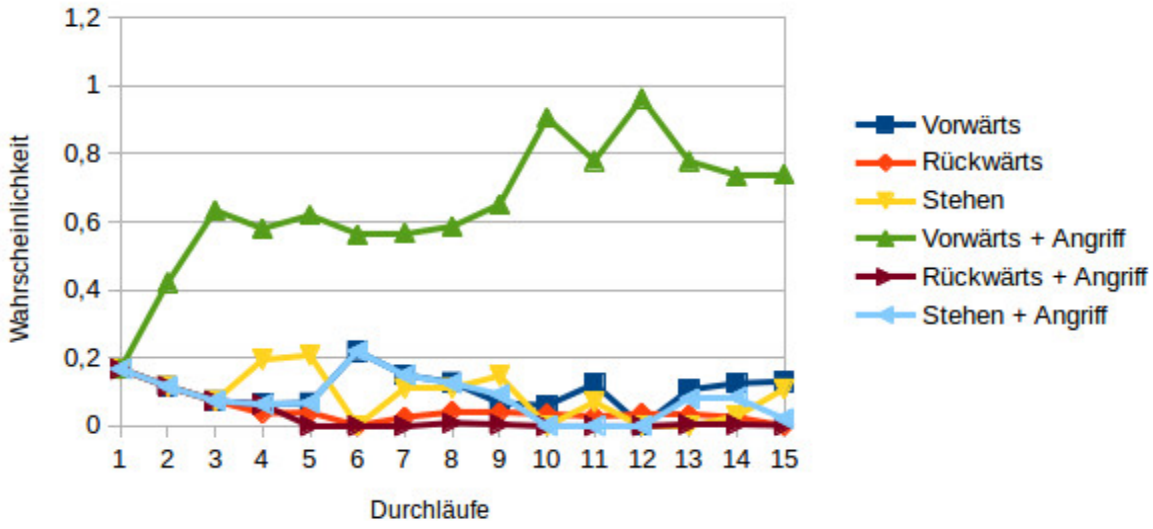

Abbildung 5: Entwicklung der Wahrscheinlichkeiten, Quelle: eigene Darstellung

Dieses wurden mithilfe eines Tabellenkalkulationsprogramms in ein anschauliches Diagramm übersetzt (Abbildung 5). Das Diagramm zeigt eindrucksvoll die rasant in die Höhe schnellende Wahrscheinlichkeit für einen Vorwärtsschritt samt Angriff. Im Gegensatz dazu verlieren die anderen Wahrscheinlichkeiten anfangs gleichermaßen. Später zeigt sich deutlich, dass die Wahrscheinlichkeit für einen Rückwärtsschritt mit folgendem Angriff konstant auf der Nulllinie bleibt. Aufgrund der Tatsache, dass nach einem Schritt zurück ein Angriff auf keinen Fall den Gegner treffen kann, sondern nur die Wahrscheinlichkeit auf einen Block des gegnerischen Angriffs herabgesetzt wird und sich zusätzlich die Gefahr, beim Rückwärtslaufen aus dem Feld zu gehen, erhöht, beweist dieser Nullwert die logische Integrität des Versuchs. Auch zeigt sich, dass sich schon ab der zweiten Runde die Wahrscheinlichkeitswerte annähernd stabilisieren. Von zwei Ausbrüchen abgesehen pendelt sich der Vorwärtsschritt mit Angriff auf einen Wert zwischen 60 % und 80 % ein. Die restlichen Werte bleiben unter der 20 %-Marke.

Interessant sind auch die eben angesprochenen Ausbrüche. Da sie danach sofort wieder auf einen Wert unter 80 % abfallen, zeigt sich deutlich, dass ein Spieler, welcher nur angreift, in der nächsten Runde von anderen Spielern schnell geschlagen wird.

4.1 Veränderung der Parameter

Zur Beobachtung der Auswirkungen beim Verändern der Parameter auf den Algorithmus wurde ein zweiter Versuch gestartet, in dem nur ein Viertel der normalen Anzahl an Spielern benutzt wurde. Wie an dem zugehörigen Diagramm ersichtlich ist (Abbildung 6), schwanken die Werte bei gleichbleibender Anzahl an Generationen viel stärker als mit 200 Spielern. Auch ist interessant, dass die Kurve für ein Vorwärtsgehen mit Angriff zweimal von anderen Kurven überholt wird, was in dem ersten Versuch nicht geschehen ist. Des Weiteren verbleibt auch die Kurve für Rückwärtsgehen und Angriff nicht auf der Nulllinie, sondern verlässt diese zweimal.

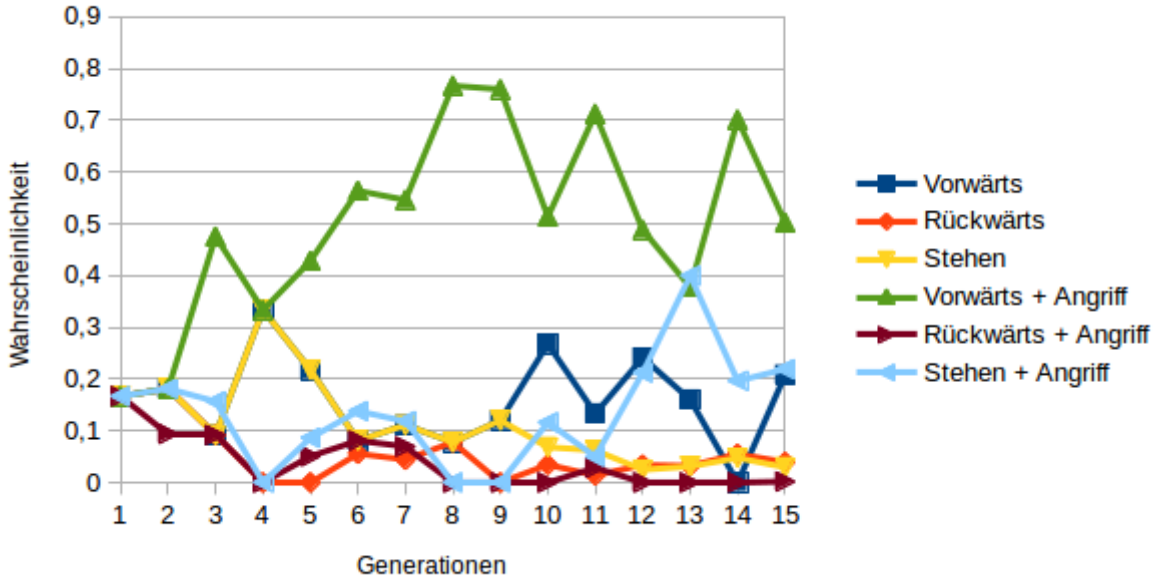

Abbildung 6: Entwicklung der Wahrscheinlichkeiten mit nur 50 Spielern,
Quelle: eigene Darstellung

Diese Entwicklung lässt sich auf die verringerte Anzahl von Individuen zurückführen. Dadurch ist die Wahrscheinlichkeit, dass auch schlechte Logiken übernommen werden, größer.

Für die folgende Rechnung wird angenommen, dass die Wahrscheinlichkeit, dass eine schlechte Logik einen Kampf gewinnt, bei 5% liegt. Mit dieser Annahme lassen sich mithilfe des Binomialkoeffizienten die Wahrscheinlichkeiten für eine Übernahme in die nächste Runde für 50 und für 200 Spieler errechnen.

$$\text{Binomialverteilung allgemein: } B(n;p;k) = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k}$$

Zur Berechnung der Binomialverteilung bei 50 Spielern wird der Binomialkoeffizient

50 über i mit i $\geq$ 25 verwendet, da eigene Tests ergeben haben, dass eine Logik mindestens die Hälfte seiner Kämpfe gewinnen muss, um in die nächste Generation übernommen zu werden.

$$\sum_{i=25}^{50}[\binom{50}{i} \cdot 0,05^i \cdot 0,95^{50-i}] = \sum_{i=25}^{50}[\frac{50!}{i! \cdot (50-i)!} \cdot 0,05^i \cdot (0,95)^{50-i}] = 8,7 \cdot 10^{-14}$$

Zur Berechnung der Wahrscheinlichkeit mit 200 Spielern wird dieselbe Methode angewendet.

$$\sum_{i=100}^{200}[\binom{200}{i} \cdot 0,05^i \cdot (0,95)^{200-i}] = \sum_{i=100}^{200}[\frac{200!}{i! \cdot (200-i)!} \cdot 0,05^i \cdot (0,95)^{200-i}] = 8 \cdot 10^{-15}$$

Diese beiden Rechnungen zeigen den Unterschied der Wahrscheinlichkeiten aufgrund der Anzahl der Spieler. Als Fazit kann man aus diesem Versuch mit anderer Spielerzahl ableiten, dass die Ergebnisse besser werden, je größer die Zahl der Individuen ist.

4.2 Ausführung des Algorithmus

Beim Start des Programms erscheint in der sich öffnenden Konsole die Frage, wie viele Runden gespielt werden sollen. Damit ist die Anzahl an Generationen gemeint, welche von dem Algorithmus der Reihe nach abgearbeitet werden. Danach erscheint die Frage nach der Anzahl der zu benutzenden Threads. Hier ist die Anzahl der gewünschten Prozessorkerne anzugeben. Dieser Wert sollte aus Gründen der Effizienz die Anzahl der physikalisch vorhandenen Kerne nicht überschreiten, da dies eine unnötige Pausierung durch das Betriebssystem zur Folge hat. Das wiederum verlangsamt den Algorithmus. Durch das Drücken der Enter-Taste wird das Programm gestartet und arbeitet die vorher gewählte Anzahl an Durchgängen ab. Sobald der Algorithmus beendet ist, kann man in dem Unterordner *Player* die jeweils besten beiden Spieler der aktuellen Runde abfragen.

5 Praktische Anwendung des Algorithmus

Ein mögliches Anwendungsbeispiel für diesen abstrakten Algorithmus ist das Finden eines Mischverhältnisses in der Chemie. Die sechs Wahrscheinlichkeitswerte geben die relative Menge der verschiedenen Chemikalien an, welche eine bestimmte Aufgabe erfüllen sollen. Als Beispiel wurde die Zusammensetzung eines Putzmittels gewählt. Die prozentualen Mengen der Inhaltsstoffe werden am Anfang gleich gesetzt. Diese Inhaltsstoffe könnten Wasser, Säuren, Basen und Seifen sein. Die Fitnessfunktion wäre dann der praktische Test an einer verschmutzten Oberfläche, der anhand seiner Putzleistung bewertet wird. Diese Bewertung ersetzt im Algorithmus die Ermittlung der Anzahl der Gewinne der Fechter.

Danach wird von dem Programm die neuen Mischverhältnisse ausgegeben, mit denen ein neuer Test angesetzt wird.

6 Ausblick

Die Einsatzvielfalt genetischer Algorithmen wird immer mehr zunehmen. So könnten z. B. komplexe genetische Algorithmen in Verbindung mit neuronalen Netzen zur ersten Generation künstlicher Intelligenz führen. Auch zur Optimierung bereits bestehender Abläufe und Aufgabenstellungen werden sie immer mehr verwendet werden.

Sicher ist, dass in der Zukunft genetische Algorithmen aus dem täglichen Abläufen unseres Lebens bald nicht mehr wegzudenken sind.

A Quellenverzeichnis

Monographien & Fachartikel

Freitas, Alex A. (2003): *A Survey of Evolutionary Algorithms for Data Mining and Knowledge Discovery*. Berlin, Heidelberg: Springer Verlag, S. 819–845.

Koh, Hyung-Won (2006): *Evolutionäre Algorithmen für das Biclustering-Problem*. Diplomarbeit. Universität Dortmund.

Rudolph, Dipl.-Inform. G. und Prof. Dr.-Ing. H.-P. Schwefel (1994): „Evolutionäre Algorithmen — ein robustes Optimierkonzept“. *Physikalische Blätter* 50, 236–238.

Turing, A.M. (1950): „Computing machinery and intelligence“. *Mind* 59, S. 433–460.

Internetquellen

EVOGENIO – Evolutionäre Kunst (2008). URL: `http://www.darwin-jahr.de/evo-magazin/evogenio-evolutionaere-kunst` (besucht am 05. 04. 2014).

Evolutionärer Algorithmus. URL: `http://goo.gl/RvU7Cx` (besucht am 05. 04. 2014).

B Quellcode

```
import java.util.InputMismatchException;
import java.util.Scanner;


public class Starter
{
 public static void main(String[] args)
 {
  Scanner scanner = new Scanner(System.in);
  System.out.println("Wie viele Runden soll gespielt werden?");
  int passes = 0, threadCount = 0;
  boolean flawless = true;

  //Anzahl an Durchläufen ermitteln
  do
  {
   try
   {
    passes = scanner.nextInt();
    if(passes <= 0)
    {
     System.err.println("Fehlerhafte Eingabe: nur positive Zahl erlaubt. Bitte erneut
         eingeben");
     flawless = false;
    }
   }
   catch(InputMismatchException e)
   {
```

```
      System.err.println("Fehlerhafte Eingabe: nur positive Zahl erlaubt. Bitte erneut
          eingeben");
      flawless = false;
     }
    }while(!flawless);

    System.out.println("Wie viele Threads sollen genutzt werden?");

    //Anzahl an Threads ermitteln
    do
    {
     try
     {
      threadCount = scanner.nextInt();
      if(threadCount <= 0)
      {
       System.err.println("Fehlerhafte Eingabe: nur positive Zahl erlaubt. Bitte erneut
           eingeben");
       flawless = false;
      }
     }
     catch(InputMismatchException e)
     {
      System.err.println("Fehlerhafte Eingabe: nur positive Zahl erlaubt. Bitte erneut
          eingeben");
      flawless = false;
     }
    }while(!flawless);

    scanner.close(); //Systemresourcen freigeben

    System.out.println("Starting Game");

    new Game(false, false, 10, 2, threadCount, 5, passes, 200);
   }
  }


  public enum Action
  {
   FORWARD(0),
   BACKWARD(1),
   STAND(2),
   ATTACKFWD(3),
   ATTACKBWD(4),
   ATTACKSTAND(5);

   private int value;

   private Action(int value)
   {
    this.value = value;
   }

   public int getValue()
   {
    return value;
   }
```

```
 public static Action getAction(int number)
 {
  switch(number)
  {
  case 0: return FORWARD;
  case 1: return BACKWARD;
  case 2: return STAND;
  case 3: return ATTACKFWD;
  case 4: return ATTACKBWD;
  case 5: return ATTACKSTAND;
  default: return null;
  }
 }
}

import java.util.concurrent.ThreadLocalRandom;

public class Board
{
 private Field[] fields;

 private Player player1;
 private Player player2;

 private int passes;
 private int player1Wins = 0;
 private int player2Wins = 0;

 /**
  * Konstruktor von Board, bereitet eine Partie vor
  * @param logic1 Logik von Spieler 1
  * @param logic2 Logik von Spieler 2
  * @param fieldLength Länge des Spielbretts
  * @param startField Startfeld der Spieler (vom Rand aus). 1 bedeutet ganz am Rand. Darf
      nicht größer as fieldLength/2 sein.
  * @param passes Anzahl der Partien, muss ungerade sein
  */
 public Board(Logic logic1, Logic logic2, int fieldLength, int startField, int passes)
 {
  this.passes = passes;

  if(startField > fieldLength/2) //Nicht mehr auf einer Hälfte
  {
   System.err.println("startField ist größer als fieldLength/2, Aufstellung nicht möglich
       ");
   System.exit(1);
  }

  if(passes%2 != 1) //kann auch unentschieden geben
  {
   System.err.println("passes muss ungerade sein");
   System.exit(1);
  }

  fields = new Field[fieldLength];
  for(int i = 0; i < fields.length; i++)
  {
```

```
   fields[i] = new Field(i);
  }

  player1 = new Player(logic1, fields[startField-1], 1);
  player2 = new Player(logic2, fields[fields.length-startField], -1);

  fields[startField-1].setPlayer(player1);
  fields[fields.length - startField].setPlayer(player2);
 }

 /**
  * Startet die Partie
  */
 public Logic playGame(ThreadLocalRandom rand)
 {
  for(int i = 0; i < passes; i++)
  {
   Player actualPlayer = getStartPlayer(i);
   while(true)
   {
    actualPlayer.resetBlock();
    if(performAction(actualPlayer, rand)) break;
    actualPlayer = getNextPlayer(actualPlayer);
   }
  }

  if(player1Wins > player2Wins) return player1.getLogic();

  return player2.getLogic(); //Gleichstand ist nicht möglich
 }

 /**
  * Führt eine Aktion aus
  * @param player aktueller Spieler
  * @return true, wenn Spiel beendet
  */
 private boolean performAction(Player player, ThreadLocalRandom rand)
 {
  switch(player.getAction(rand).getValue())
  {
  case 0: return forward(player, false);
  case 1: return backward(player, false);
  case 3: return forward(player, true);
  case 4: return backward(player, true);
  case 5: return attack(player);
  default: return false; //Nur bei STAND
  }
 }

 /**
  * Führt einen Angriff auf das Feld vor dem Spieler durch
  * @param player aktueller Spieler
  * @return true, bei Treffer
  */
 private boolean attack(Player player)
 {
  player.attack();
  if(player.getField().getNumber() + player.getOrientation() >= 0 && player.getField().
```

```
      getNumber() + player.getOrientation() < fields.length)
  {
   if(!fields[player.getField().getNumber() + player.getOrientation()].isFree())
   {
    if(Math.random() >= getNextPlayer(player).getBlockProbability())
    {
     if(player == player1) player1Wins++;
     else player2Wins++;
     return true;
    }
   }
  }
  return false;
 }

 /**
  * Geht ein Feld zurück
  * @param player aktueller Spieler
  * @param attack true, wenn der Spieler auch angreift
  * @return true, wenn Feld verlassen
  */
 private boolean backward(Player player, boolean attack)
 {
  int i = player.getField().getNumber() - player.getOrientation();

  if(i >= 0 && i < fields.length)
  {
   fields[player.getField().getNumber()].setPlayer(null);
   fields[i].setPlayer(player);
   player.setField(fields[i]);
  }
  else
  {
   if(player == player1) player2Wins++;
   else player1Wins++;

   return true;
  }

  if(attack) player.attack(); //Kann niemanden treffen, daher nur Blockreduktion

  return false;
 }

 /**
  * Geht ein Feld vor
  * @param player aktueller Spieler
  * @param attack true, wenn der Spieler angreift
  * @return true, wenn anderen Spieler getroffen
  */
 private boolean forward(Player player, boolean attack)
 {
  int i = player.getField().getNumber() + player.getOrientation();

  if(fields[i].isFree()) //Aus dem Feld laufen ist nicht möglich, da anderer Spieler im
      Weg
  {
   fields[player.getField().getNumber()].setPlayer(null);
```

```
    fields[i].setPlayer(player);
    player.setField(fields[i]);
   }

   if(attack)
   {
    return attack(player);
   }

   return false;
  }

  //+++++++++++++++++Getter & Setter++++++++++++++++++++//

  /**
   * Gibt den nächsten Spieler zurück
   * @param actualPlayer aktueller Spieler
   * @return anderer Spieler
   */
  private Player getNextPlayer(Player actualPlayer)
  {
   if(actualPlayer == player1) return player2;
   return player1;
  }

  /**
   * Gibt je nach Runde den Startspieler zurück
   * @param i aktuelle Runde
   * @return player
   */
  private Player getStartPlayer(int i)
  {
   if(i%2 == 0) return player1;
   return player2;
  }
}

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ThreadLocalRandom;


public class BoardMaster implements Callable<List<Logic>>
{
 private List<Board> boards;
 private ThreadLocalRandom rand;

 public BoardMaster(List<Board> boards)
 {
  this.boards = boards;
  this.rand = ThreadLocalRandom.current(); //ein Zufallsgenerator pro Thread
 }

 /**
  * Callable Methode für Nebenläufigkeit
  * @return Liste der Gewinner der Partien
  */
```

```
 @Override
 public List<Logic> call() throws Exception
 {
  List<Logic> winner = new ArrayList<Logic>();

  for(Board board : boards)
  {
   winner.add(board.playGame(rand));
  }
  return winner;
 }
}


public class Field
{
 private Player player; //Spieler, der auf dem Feld steht
 private int number; //Zahl im Array

 /**
  * Konstruktor: Erstellt ein leeres Feld
  * @param number Index im Array
  */
 public Field(int number)
 {
  this.number = number;
  player = null;
 }

 /**
  * Setzt eine Referenz des Spielers auf das Feld
  * @param player Spieler, der gesetzt wird
  */
 public void setPlayer(Player player)
 {
  this.player = player;
 }

 /**
  * Gibt den Spieler auf dem Feld zurück
  * @return Spieler
  */
 public Player getPlayer()
 {
  return player;
 }

 /**
  * Feld belegt?
  * @return isFree
  */
 public boolean isFree()
 {
  return player == null ? true : false;
 }

 /**
  * Gibt die Nummer im Array zurück
```

```
  * @return number
  */
 public int getNumber()
 {
  return number;
 }
}

import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Hashtable;
import java.util.List;
import java.util.Map.Entry;
import java.util.Random;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.concurrent.ThreadLocalRandom;


public class Game
{
 private Hashtable<Logic, Integer> logics;
 private int fieldLength, startField, passesPerPlay, passes, logicCount, threadCount;
 private ExecutorService exec;
 private Logic[] bestPlayers;
 private ThreadLocalRandom random;

 /**
  * Konstruktor von Game, Zentrum des Algorithmus
  * @param randomStartValues true, wenn Startwerte der Logiken zufällig sein sollen
  * @param oneRandom nur bei randomStartValue==false, verändert eine Wahrscheinlichkeit
      zufällig
  * @param fieldLength Länge des Spielbretts
  * @param startField Startfeld der Spieler, gemmessen vom äußeren Rand, muss kleiner/
      gleich fieldLength/2 sein und größer 0 sein
  * @param threadCount Anzahl der Threads, die für die Berechnung benutzt werden sollen
  * @param passesPerPlay Anzahl der Spielrunden pro Partie (muss ungerade sein)
  * @param passes Anzahl der Durchläufe (Generationen)
  * @param logicCount Anzahl der Logiken (muss gerade sein)
  */
 public Game(boolean randomStartValues, boolean oneRandom, int fieldLength, int
     startField, int threadCount, int passesPerPlay, int passes, int logicCount)
 {
  this.random = ThreadLocalRandom.current();
  this.passes = passes;
  this.passesPerPlay = passesPerPlay;
  this.startField = startField;
  this.fieldLength = fieldLength;
  this.logicCount = logicCount;
  this.bestPlayers = new Logic[2];
  logics = new Hashtable<Logic, Integer>();
  exec = Executors.newFixedThreadPool(threadCount);

  if(logicCount%2 == 1)
```

```
 {
  System.err.println("logicCount muss gerade sein");
  System.exit(1);
 }

 for(int i = 0; i < logicCount; i++)
 {
  logics.put(new Logic(random, randomStartValues, oneRandom), 0);
 }

 this.threadCount = threadCount;

 run();
}

/**
 * Hauptschleife (Loop) arbeitet alle Durchgänge ab
 */
private void run()
{
 for(int i = 0; i < passes; i++) //Hauptschleife
 {
  List<List<Board>> boards = new ArrayList<List<Board>>();
  List<BoardMaster> boardMasters = new ArrayList<BoardMaster>();
  List<Board> allMatches = makeMatches();
  int counter = 0;
  int logicSizeDivThreads = allMatches.size()/threadCount; //Wird öfter gebraucht -
      keine Mehrfachberechnung

  for(int j = 0; j < threadCount; j++) //Matchmaking
  {
   boards.add(new ArrayList<Board>());
   int masterCounter = 0;
   do
   {
    boards.get(j).add(allMatches.get(counter));
    masterCounter++;
    counter++;
   }while(masterCounter < logicSizeDivThreads || (j == threadCount-1 && counter <
       allMatches.size()-1)); //Überzählige auch hinzufügen
  }

  for(int j = 0; j < threadCount; j++)
  {
   boardMasters.add(new BoardMaster(boards.get(j)));
  }

  List<Future<List<Logic>>> futures = null;

  try
  {
   futures = exec.invokeAll(boardMasters);
  }
  catch (InterruptedException e) {e.printStackTrace();}

  for(Future<List<Logic>> future : futures)
  {
   try
```

```
 {
  for(Logic logic : future.get())
  {
   logics.put(logic, logics.get(logic)+1);
  }
  future.cancel(false);
 }
 catch (InterruptedException e) {e.printStackTrace();}
 catch (ExecutionException e) {e.printStackTrace();}
}

List<Logic> logicsList = sortLogics(); //Fitnessfunktion
logics.clear();

bestPlayers[0] = logicsList.get(0);
bestPlayers[1] = logicsList.get(1);

saveBestPlayers("" + i);

if(i == passes - 1) break; //Unnötige Neupaarung verhindern

//Paarungszeit ;)
int crossoverCount = (int) (Math.sqrt(logicCount/2)); //~50% der neuen Logiken durch
    int-Konvertierung
for(int a = 0; a < crossoverCount; a++)
{
 for(int b = 0; b < crossoverCount; b++)
 {
  logics.put(new Logic(logicsList.get(a), logicsList.get(b)), 0);
 }
}

for(int j = 0; j < crossoverCount; j++)
{
 logicsList.remove(0);
}

//Ganz gefährliche Strahlung
int radiationCount = (int) (((float)logicCount) * 0.25f); //25% werden geklont und ver
    ändert -> 50% der neuen Logiken
for(int a = 0; a < radiationCount; a++)
{
 logics.put(logicsList.get(a), 0);
 logics.put(new Logic(random, logicsList.get(a), true), 0);
}

for(int a = 0; a < radiationCount; a++)
{
 logicsList.remove(0);
}

//Die restlichen Überlebenden (füllen die Plätze der fehlenden Paarungskandidaten)
int rest = (int) (logicCount - logics.size());
for(int a = 0; a < rest; a++)
{
 logics.put(logicsList.get(0), 0);
 logicsList.remove(0);
}
```

```
  System.out.println("Finished Pass: " + (i+1));
 }
 System.out.println("Finished Pass: " + passes);
 exec.shutdownNow();
 //saveBestPlayers("" + passes);
}

/**
 * Speichert die besten beiden Spieler in einer Datei
 */
private void saveBestPlayers(String prefix)
{
 PrintWriter player1 = null, player2 = null;
 try
 {
  player1 = new PrintWriter("Player/" + prefix + "player1.player");
  //player2 = new PrintWriter("Player/" + prefix + "player2.player");
 }
 catch (FileNotFoundException e)
 {
  e.printStackTrace();
 }

 bestPlayers[0].print(player1);
 //bestPlayers[1].print(player2);

 player1.close();
 //player2.close();
}

/**
 * Erstellt die Spiele der Logiken - jede spielt einmal gegen jede andere (Formel: (
     logicCount + 1) * logicCount/2 (Gauß))
 * @return List<Board> alle Partien der Runde
 */
private List<Board> makeMatches()
{
 List<Board> all = new ArrayList<Board>();
 Logic[] tmp = new Logic[logics.size()];
 tmp = logics.keySet().toArray(tmp);
 List<Logic> l = new ArrayList<Logic>(); //Einfache Handhabung
 for(Logic m : tmp)
 {
  l.add(m);
 }

 Logic actualLogic = null;

 while(l.size() > 1)
 {
  actualLogic = l.get(0);

  for(Logic ls : l)
  {
   all.add(new Board(actualLogic, ls, fieldLength, startField, passesPerPlay));
  }
  l.remove(0);
 }
```

```
  return all;
 }

 /**
  * Sortiert die Logiken nach Anzahl der Gewinne
  * @return sortierte ArrayList mit den Logiken
  */
 private List<Logic> sortLogics()
 {
  List<Logic> sorted = new ArrayList<Logic>();
  int winnerCount = 0; //Minimalwert, man kann minimal 0 mal gewinnen

  while(sorted.size() < logics.size())
  {
   if(logics.containsValue(winnerCount))
   {
    for(Entry<Logic, Integer> entry : logics.entrySet())
    {
     if(entry.getValue().intValue() == winnerCount)
     {
      //System.out.println(winnerCount);
      sorted.add(entry.getKey()); //Einsortieren
     }
    }
   }
   winnerCount++;
  }

  Collections.reverse(sorted); //Beste Logiken nach oben
  return sorted;
 }
}

import java.io.PrintWriter;
import java.util.concurrent.ThreadLocalRandom;


public class Logic
{
 private float[] probabilities;

 /**
  * Konstruktor zur einmaligen Erzeugung der Wahrscheinlichkeiten
  * @param randomStartValues false, wenn startWerte 1/6 sein sollen
  * @param oneRandom nur bei randomStartValues==false, verändert einen der Werte
  */
 public Logic(ThreadLocalRandom rand, boolean randomStartValues, boolean oneRandom)
 {
  probabilities = new float[6];

  if(randomStartValues)
  {
   for(int i = 0; i < probabilities.length; i++)
   {
    probabilities[i] = rand.nextFloat();
   }
   trim();
```

```
 }
 else
 {
  for(int i = 0; i < probabilities.length; i++)
  {
   probabilities[i] = 1f/probabilities.length;
  }

  if(oneRandom)
  {
   changeOneRandom(rand);
  }
 }
}

/**
 * Paarungskonstruktor ;)
 * @param father logik 1
 * @param mother logik 2
 */
public Logic(Logic father, Logic mother)
{
 probabilities = new float[6];

 float[] fatherProbs = father.getProbabilities();
 float[] motherProbs = mother.getProbabilities();

 for(int i = 0; i < probabilities.length; i++)
 {
  probabilities[i] = (fatherProbs[i] + motherProbs[i])/2f;
 }

 trim();
}

/**
 * Kopierkonstruktor, mit Option zum Ver'ndern
 * @param clone Logik, welche geklont werden soll
 * @param changeRandom soll ein Wert verändert werden?
 */
public Logic(ThreadLocalRandom rand, Logic clone, boolean changeRandom)
{
 probabilities = clone.getProbabilities();

 if(changeRandom)
 {
  changeOneRandom(rand);
 }
}

private void changeOneRandom(ThreadLocalRandom rand)
{
 if(rand.nextBoolean())
 {
  int index = rand.nextInt(6);
  probabilities[index] += rand.nextFloat();
  if(probabilities[index] > 1) probabilities[index] = 1;
 }
```

```
 else
 {
  int index = rand.nextInt(6);
  probabilities[index] -= rand.nextFloat();
  if(probabilities[index] < 0) probabilities[index] = 0;
 }
 trim();
}

/**
 * Trimmt die Werte wieder auf insgesammt 100%
 */
public void trim()
{
 float all = 0f;

 for(float prob : probabilities)
 {
  all += prob;
 }

 if(all != 1f)
 {
  float factor = 1f/all;

  for(int i = 0; i < probabilities.length; i++)
  {
   probabilities[i] *= factor;
  }
 }
}

/**
 * Gibt die Wahrscheinlichkeiten aus, praktisch für debugging o.ä.
 */
@Override
public String toString()
{
 return "Logic: {" + probabilities[0]+", " + probabilities[1]+", " + probabilities[2]+",
     " + probabilities[3]+", " + probabilities[4]+", " + probabilities[5]+"}";
}

public void print(PrintWriter stream)
{
 for(int i = 0; i < probabilities.length; i++)
 {
  stream.println(probabilities[i]);
 }
}

//+++++++++++++++++Getter & Setter++++++++++++++++++++//
/**
 * Gibt die nächste Aktion des Spielers zurück
 * @param rand Referenz auf Random des BoardMasters
 * @return Action
 */
public Action getAction(ThreadLocalRandom rand)
{
```

```
  float v = rand.nextFloat();

  if(v < probabilities[0]) return Action.getAction(0);
  else if(v < probabilities[0] + probabilities[1]) return Action.getAction(1);
  else if(v < probabilities[0] + probabilities[1] + probabilities[2]) return Action.
      getAction(2);
  else if(v < probabilities[0] + probabilities[1] + probabilities[2] + probabilities[3])
      return Action.getAction(3);
  else if(v < probabilities[0] + probabilities[1] + probabilities[2] + probabilities[3] +
       probabilities[4]) return Action.getAction(4);
  else  return Action.getAction(5);
 }

 /**
  * Gibt die float-Warscheinlichkeiten des Spielers zurück
  * @return probabilities
  */
 public float[] getProbabilities()
 {
  return probabilities;
 }
}

import java.io.PrintWriter;
import java.util.concurrent.ThreadLocalRandom;


public class Player
{
 private int orientation; //1 fuer nach rechts, -1 fuer nach links
 private float blockProb;

 private Logic logic;
 private Field field;

 /**
  * Erstellt einen neuen Spieler
  * @param logic Logik des Spielers
  * @param field Feld, auf dem er steht
  * @param orientation 1 fuer nach rechts, -1 fuer nach links
  */
 public Player(Logic logic, Field field, int orientation)
 {
  this.blockProb = 0.8f;
  this.logic = logic;
  this.field = field;
  this.orientation = orientation;
 }

 //+++++++++++++++++Getter & Setter++++++++++++++++++++//

 public Action getAction(ThreadLocalRandom rand)
 {
  return logic.getAction(rand);
 }

 public void print(PrintWriter stream)
 {
```

```
  logic.print(stream);
 }

 public Logic getLogic()
 {
  return logic;
 }

 public Field getField()
 {
  return field;
 }

 public void setField(Field field)
 {
  this.field = field;
 }

 public int getOrientation()
 {
  return orientation;
 }

 public float getBlockProbability()
 {
  return blockProb;
 }

 public void attack()
 {
  blockProb = 0.1f;
 }

 public void resetBlock()
 {
  blockProb = 0.8f;
 }
}
```

Ich erkläre hiermit, dass ich die Seminararbeit ohne fremde Hilfe angefertigt und nur die im Literaturverzeichnis angeführten Quellen und Hilfsmittel benutzt habe.

…....., den …....
Ort Datum

…....
Unterschrift des Verfassers